Relazione Battleships

Angelo Tinti
Bruno Esposito
Marco Biagini
Riccardo Gennari

29 giugno 2020

Indice

Capitolo 1

Analisi

1.1 Requisiti

Si intende realizzare una versione alternativa del classico gioco Battaglia navale in cui ci sono due giocatori.

Requisiti funzionali

- Possibilità di creare e gestire diversi profili giocatore.
- Vedere le statistiche riguardanti le proprie precedenti partite.
- Possibilità di giocare contro un altro giocatore umano o contro un'intelligenza artificiale.
- Possibilità di scegliere fra due modalità di gioco:
  - **Classica:** La partita verrà conclusa nel momento in cui uno dei due giocatori avrà distrutto tutte le navi nemiche.
  - **A punti:** Vincerà la partita il primo giocatore ad aver raggiunto il punteggio prestabilito ad inizio partita.
- Durante il proprio turno, il giocatore avrà a disposizione un numero di attacchi pari al numero di navi in proprio possesso non ancora affondate.

Requisiti non funzionali

- L'applicazione dovrà avere una grafica minimale per garantire sia la semplicità di utilizzo da parte dell'utente e sia la fluidità anche nei sistemi meno moderni.

1.2 Analisi e modello del dominio

Battleships dovrà fornire la possibilità di gestire diversi profili giocatore, con relative credenziali di login, e di visualizzare le loro statistiche basate sulle partite precedenti.
All'inizio di una partita, si scelgono quali giocatori parteciperanno: due giocatori umani, oppure un giocatore umano e uno controllato dall'intelligenza artificiale. Ciascun giocatore, dopo aver posizionato le proprie navi, inizierà ad infliggere tanti attacchi al proprio avversario quante sono le proprie navi non ancora affondate. Il software, infine, terminerà la partita in base alla modalità di gioco scelta. Nello schema UML "Figura 1.1", vediamo le relazioni che intercorrono fra le entità principali. La complessità nella realizzazione di questo dominio sarà gestire in maniera efficace l'alternarsi dei turni e il relativo scambio di informazioni generate.

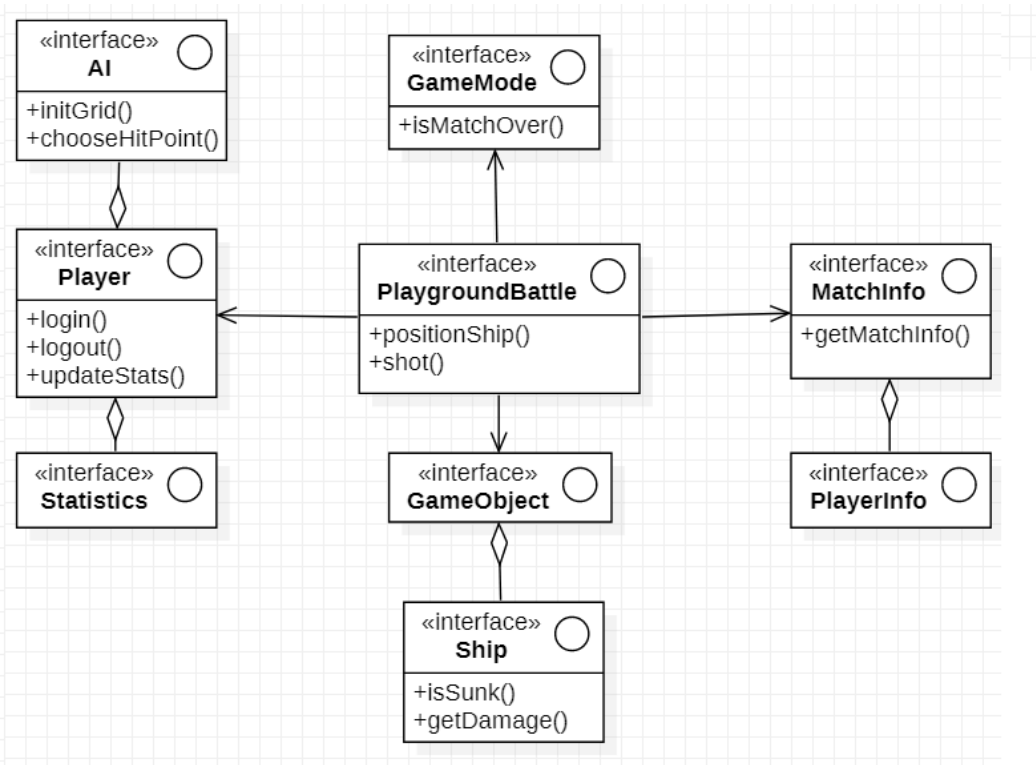

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

Per garantire indipendenza logica tra aspetti di grafica, di dominio applicativo e di controllo, abbiamo scelto di utilizzare il pattern MVC. Ogni componente del pattern dispone di una interfaccia e di una relativa classe implementativa. La seguente figura descrive in maggior dettaglio l'architettura.

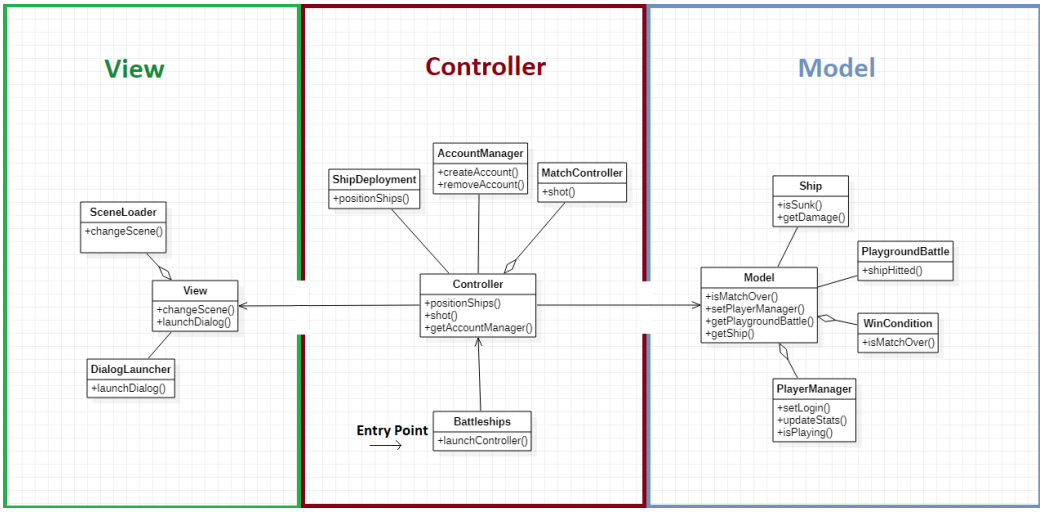

Figura 2.1: Schema UML architetturale di Battleships, raffigurante le principali interfacce dell'applicazione

La classe Battleships è il punto di ingresso del software. Essa istanzia l'interfaccia Controller, che a sua volta contiene l'istanza dell'interfaccia di Model e di View. Queste interfacce costituiscono il punto di ingresso dei tre

componenti del pattern MVC. Il flusso di controllo è determinato dall'input dell'utente sull'interfaccia grafica. Le classi che gestiscono l'input (che sono parte del controller) potranno interagire con le funzionalità del Model e della View tramite l'interfaccia principale Controller. In questa architettura, la View fornisce metodi "generici" per la gestione della grafica, la cui effettiva implementazione viene delegata a sotto-parti indipendenti della View stessa. In questo modo, il Controller non necessiterebbe di alcun adattamento in caso di sostituzione delle librerie grafiche. Quest'ultimo si occupa di aggiornare le informazioni del Model e, in base alla sua risposta, modificherà la View. L'interfaccia Model definisce la logica di dominio dell'applicazione, le entità e le interazioni tra loro.

2.2 Design dettagliato

Bruno Esposito

AccountManager e PlayerManager

La classe AccountManager rappresenta l'interfaccia per la gestione degli account utente: in particolare permette sia di salvare in maniera persistente i dati di ciascun utente, sia di gestire i loro dati durante il gioco. Questa classe viene invocata dal Controller e gestisce il salvataggio e la cancellazione di un account utente su disco tramite due metodi fondamentali che sono createAccount() e removeAccount(). Ho scelto di non far restituire uno specifico valore da ciascuno di questi due metodi, ma di far lanciare eccezioni che dovranno poi essere intercettate a livello di View dal ProfileController. La gestione dei dati, invece, riguarda i punteggi e lo stato di login/logout del giocatore. I primi sono gestiti dai metodi setWinner() e setLoser(); i secondi invece da logInAccount() e logOutAccount().

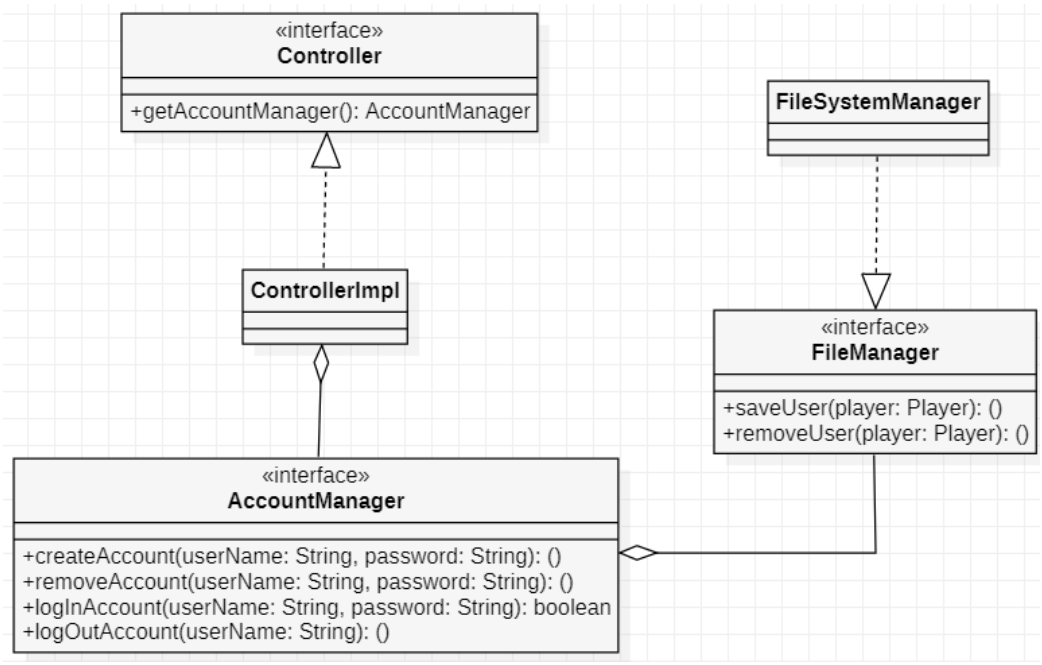

Figura 2.2: Schema UML del gestore degli account giocatore

La classe PlayerManager incapsula la logica che è dietro alla gestione dei profili giocatore. Essa infatti fa parte del Model e permette di gestire tutte le operazioni che si possono effettuare su un profilo giocatore. In particolare, quando occorre modificare o aggiornare qualsiasi informazione di uno o più giocatori è l'AccountManager che, oltre ad occuparsi del salvataggio effettivo su file di ciascuno di essi, notifica l'operazione da effettuare al PlayerManager. In futuro l'estensione di tali interfacce può permettere di aggiungere o modificare tali operazioni sui giocatori: in questo caso ho ritenuto che queste fossero le operazioni principali da prendere in considerazione.

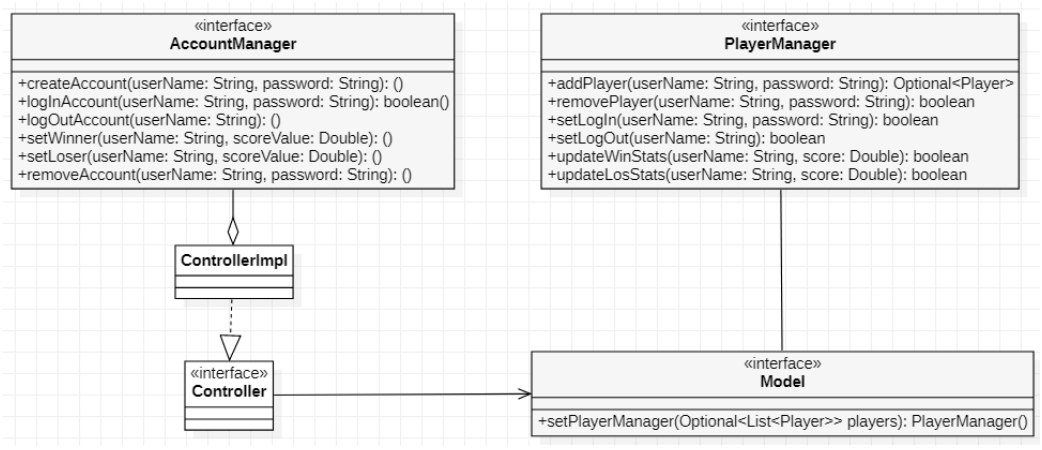

Figura 2.3: Schema UML della gestione logica del profilo giocatore

Player e Statistics

Nello sviluppo della suddetta logica relativa alle operazioni sui giocatori occorreva innanzitutto definire l'oggetto Player: tale interfaccia è quella che racchiude tutte le informazioni che un qualsiasi tipo di giocatore deve avere. In questo caso, per fare in modo che fosse possibile definire anche altri tipi di giocatore ho usato un AbstractPlayer per l'implementazione di base del Player. Infatti, così facendo ho potuto sia considerarne i due diversi tipi (HumanPlayer ed ArtificialPlayer) che interessavano al nostro dominio, sia differenziare la loro relativa implementazione garantendo anche delle possibili aggiunte di ulteriori tipi di giocatore.

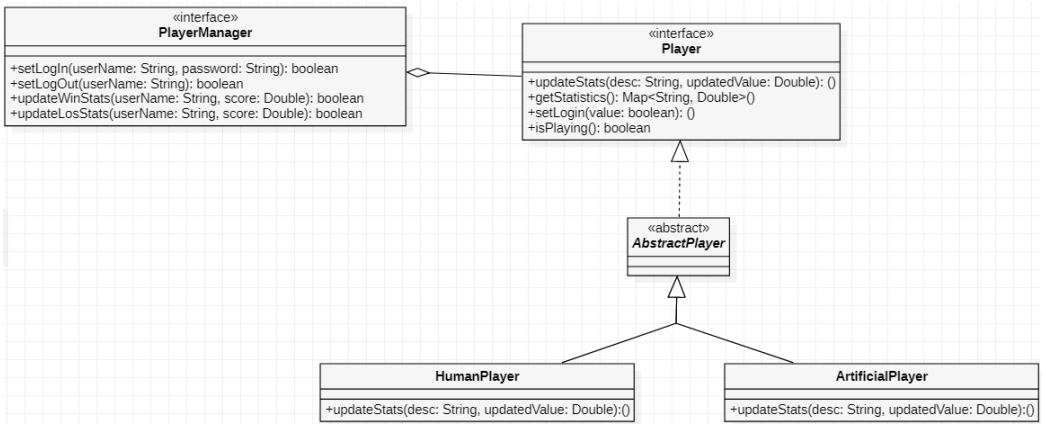

Figura 2.4: Schema UML del profilo giocatore

Un altro elemento molto importante nel nostro dominio riguarda le statistiche: in particolare ogni giocatore deve avere delle specifiche informazioni relative agli esiti delle partite giocate. Per il calcolo di valori come 'percentuale di vittorie', 'totali partite giocate' ed altri, ho introdotto Statistics che si basa sul pattern Strategy. In questo modo ho implementato le due diverse strategie di calcolo delle statistiche, ovvero: LoserStatsCalculator e WinnerStatsCalculator.

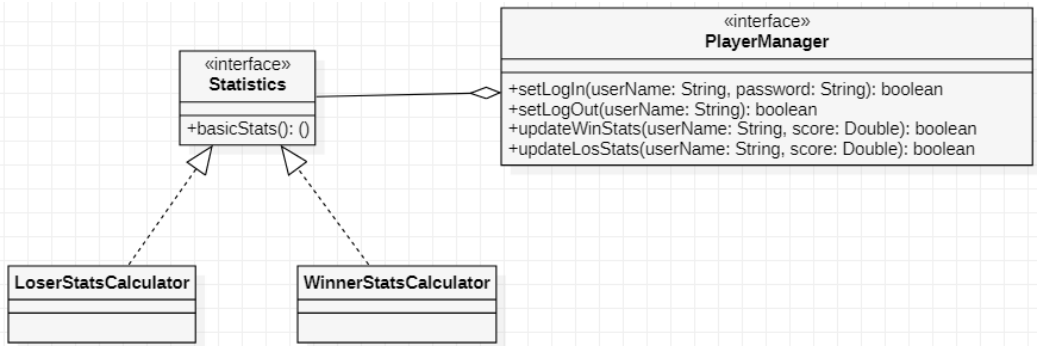

Figura 2.5: Schema UML delle statistiche del giocatore

Artificial Intelligence

Nel nostro dominio applicativo abbiamo voluto implementare anche un avversario virtuale di base, in modo tale da consentire anche partite del tipo "giocatore vs giocatore artificiale". Nel realizzare il comportamento di quest'ultimo (ArtificialPlayer) ho pensato di separare due parti diverse ma strettamente collegate: l'astrazione di ArtificialIntelligence e la relativa implementazione (IntelligenceComputation). In questo modo ho gestito l'interazione tra ArtificialIntelligence e IntelligenceComputation utilizzando il pattern Bridge, legando direttamente l'interfaccia con la classe astratta per far si che ci fosse compatibilità tra tutte le specializzazioni. Ho fatto questa scelta proprio per facilitare l'inserimento di ulteriori tipi di intelligenza artificiale e delle relative implementazioni, in modo da poter variare le difficoltà di gioco contro il giocatore virtuale.

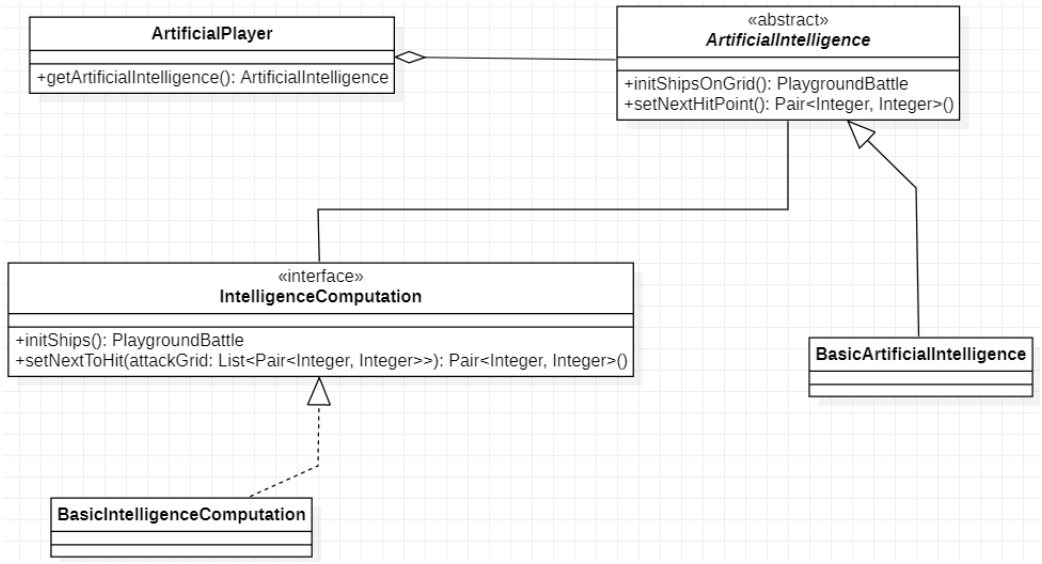

Figura 2.6: Schema UML dell'intelligenza artificiale

Riccardo Gennari

DialogLauncher

L'interfaccia DialogLauncher fornisce un metodo statico per il caricamento a video di finestre di dialogo con l'utente.
Questo metodo delega il compito di lanciare la finestra di dialogo ad una classe che estenda AbstractDialog. La classe in questione viene fornita dall'enum DialogType, che conosce quale classe sia l'implementazione concreta di ogni tipo di dialogo disponibile (ovvero di ogni valore dell'enum stessa).
In questo modo, utilizzando l'interfaccia DialogLauncher, è sufficiente indicare qualitativamente quale tipo di dialogo lanciare (ad esempio conferma, login, errore) senza doverne conoscerne la classe. E' inoltre semplice fornire supporto a nuovi tipi di dialoghi, in quanto è sufficiente che essi estendano AbstractDialog e che abbiano un valore associato in DialogType.

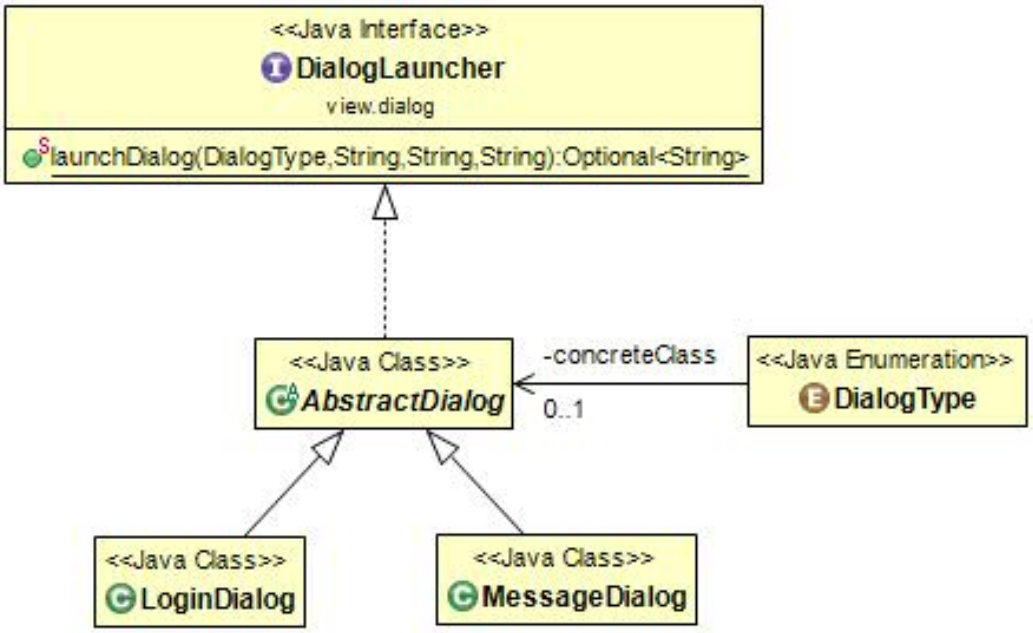

Figura 2.7: Rappresentazione UML del sistema per il lancio di dialoghi

SceneLoader

L'interfaccia SceneLoader fornisce un metodo per il cambio della scena (schermata) attiva sullo schermo. Il valore dell'enum SceneName passato come parametro determina quale schermata verrà caricata. La sua implementazione concreta, SceneLoaderImpl, richiede a LayoutLoader di caricare la nuova schermata. Utilizzando un approccio Strategy, l'implementazione di LayoutLoader utilizzata determina il modo in cui la schermata viene caricata. Con questo sistema è semplice aggiungere nuove schermate: è sufficiente aggiungere un valore all'enum SceneName. E' anche possibile aggiungere nuovi modi di caricare schermate aggiungendo implementazioni di LayoutLoader. Il modo in cui SceneLoader viene chiamato dal Controller non usa alcun riferimento alla effettiva implementazione delle schermate o alla strategia usata per caricarle. Per questo motivo è possibile rivedere totalmente l'implementazione di questa parte di view senza impattare minimamente il controller (o il model).

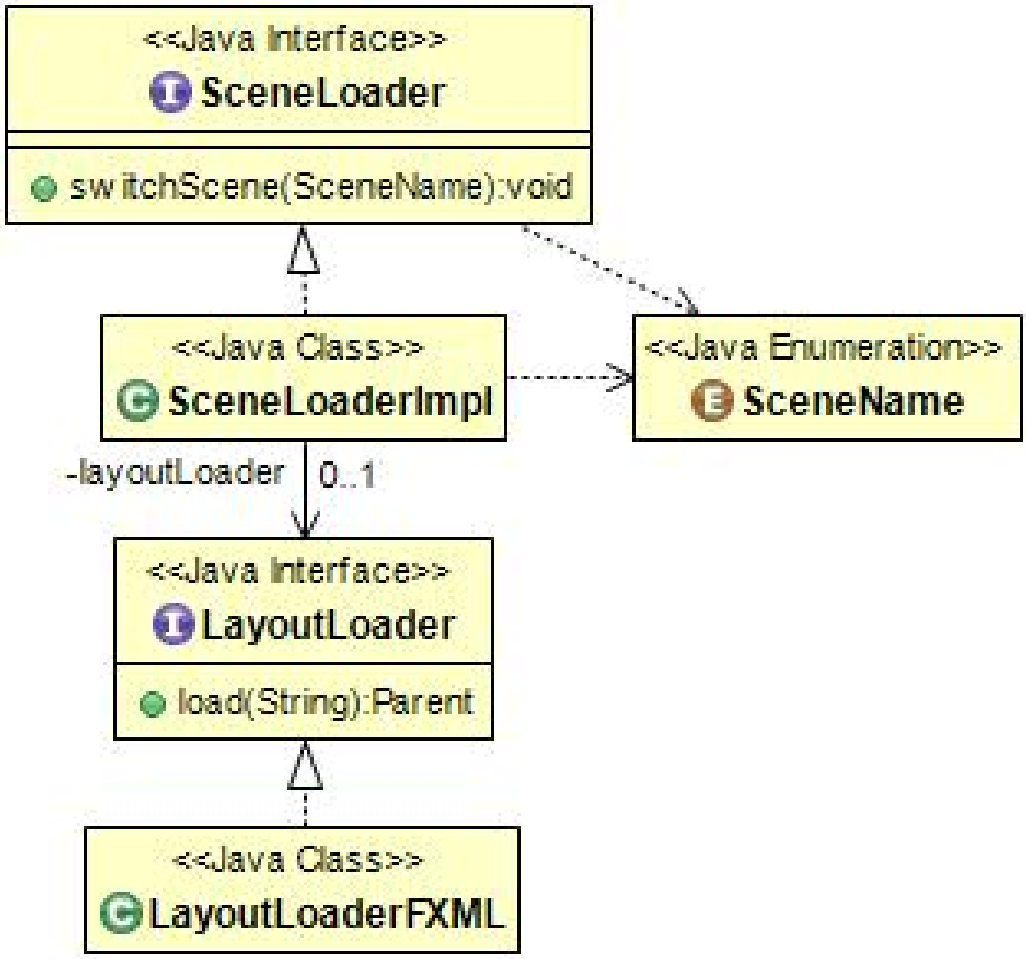

Figura 2.8: Rappresentazione UML del sistema per il cambio di scena

MatchInitializer

L'interfaccia MatchInitializer fornisce un metodo per dare il via ad una nuova partita (match).
Per farlo si avvale della classe MatchSettings (che funge da controller per javafx) per ricevere input dell'utente, quindi svolge i passi necessari ad avviare una partita. L'interfaccia PlayerChecker fornisce a MatchSettings metodi di controllo dell'input indipendenti dall'interfaccia grafica utilizzata.
L'interfaccia MatchInitializer è quindi indipendente dalle interazioni con l'utente: gli input vengono acquisiti da MatchSettings, che può essere sostituita senza conseguenze se necessario (ad esempio se il progetto decidesse di non usare più javafx). Inoltre nulla vieta la modifica/aggiunta di precedure per

inizializzare una partita (è sufficiente, con un approccio Strategy, aggiungere classi che implementino MatchInitializer).

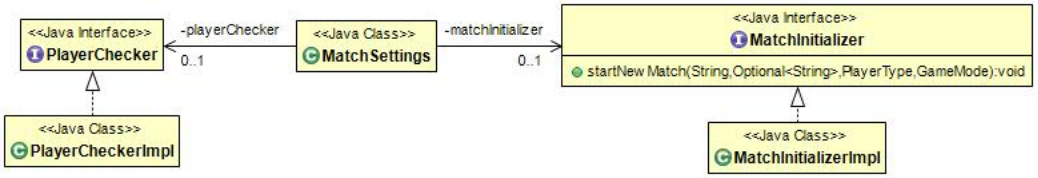

Figura 2.9: Rappresentazione UML del sistema per l'inizializzazione di un match

WinCondition

L'interfaccia WinCondition fornisce un metodo per verificare se un giocatore (ad un qualunque punto della partita) soddisfa le condizioni per essere il vincitore. Ricevuti i dati di un giocatore, l'implementazione WinConditionImpl si appoggia alla enum GameMode (che specifica le condizioni di fine partita per ogni modalità di gioco) per determinare se il giocatore abbia vinto o meno. Questo approccio rende l'interfaccia WinCondition indipendente da aggiunte e rimozioni delle modalità di gioco disponibili.

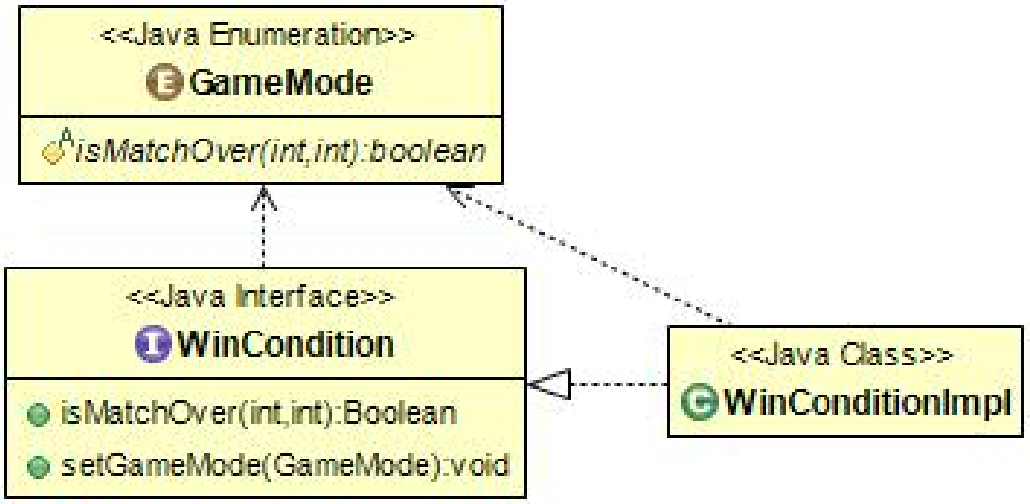

Figura 2.10: Rappresentazione UML del sistema per il controllo di fine partita

CurrentPlayer

L'interfaccia CurrentPlayer fornisce metodi per tenere traccia di quale giocatore (durante una partita) sia il giocatore attivo (cioè di chi è il turno

corrente). I giocatori sono rappresentati in modo astratto come valori della enum PlayerNumber.
E' possibile settare arbitrariamente quale sia il giocatore corrente, oppure passare il turno al giocatore successivo senza dover sapere chi sia. Le funzionalità di questa interfaccia sono di supporto alla parte del software che gestirà i turni dei giocatori.

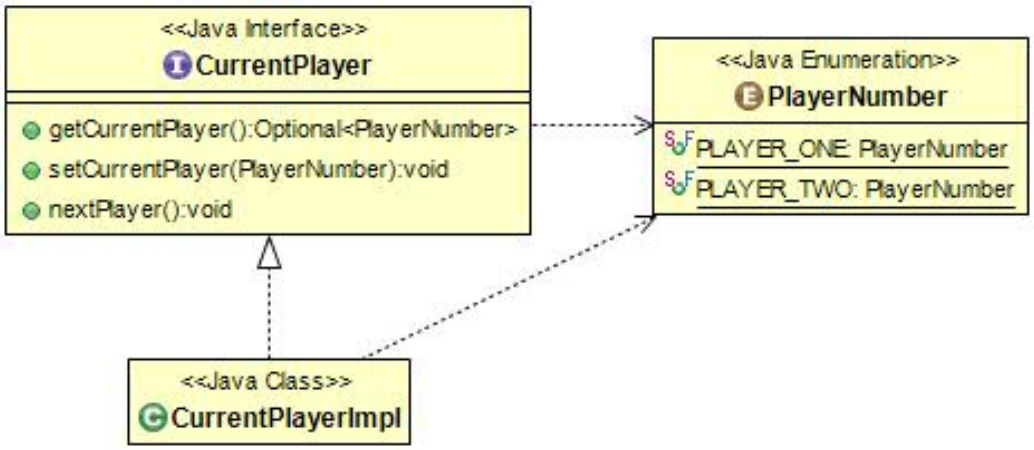

Figura 2.11: Rappresentazione UML del sistema per il tracciamento del giocatore corrente

MatchInfo

L'interfaccia MatchInfo fornisce metodi per il recupero di dati riguardanti la partita in corso.
In particolare contiene un'istanza dell'interfaccia PlayerInfo per ogni giocatore in partita, che può essere recuperata tramite getter. In questo modo si separano i dati "semplici" della partita (composti ad esempio da singoli campi) dalle informazioni dei giocatori, la cui composizione è strettamente legata alla implementazione dei giocatori stessi. Il cambiamento della gestione dei giocatori lascerebbe quindi l'interfaccia MatchInfo inalterata.

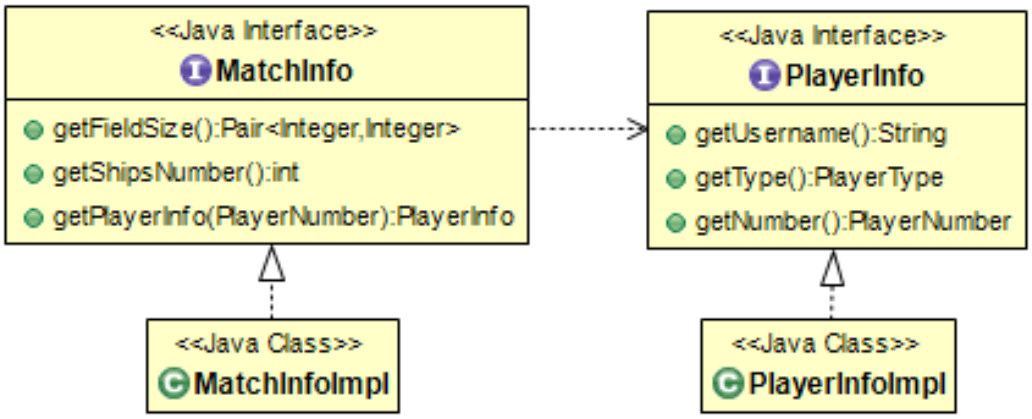

Figura 2.12: Rappresentazione UML del sistema per la memorizzazione dei dati della partita in corso

Marco Biagini

Ship

La classe Ship rappresenta e gestisce l'entità nave all'interno del progetto. Questa classe estende l'interfaccia GameObject, e tiene traccia in particolare del tipo di nave di cui si tratta, della sua dimensione e della sua orientazione (orizzontale o verticale).
La tipologia di nave viene fornita dall'enum ShipType, che tiene traccia di tutte le categorie di navi esistenti all'interno del gioco. In particolare, ogni tipologia di nave ha una sua dimensione fissa che non può essere modificata. Per questo design non ho utilizzato pattern, ma le scelte progettuali che ho realizzato sono aperte alla possibilità di estensioni future. In particolare, potrà essere gestita l'aggiunta di nuove tipologie di navi o di ulteriori oggetti all'interno del gioco.

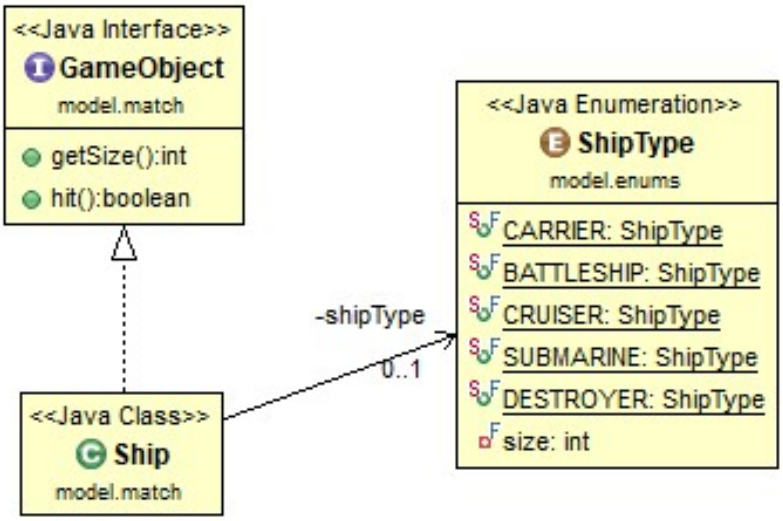

Figura 2.13: Rappresentazione UML delle componenti principali riguardanti le navi

Posizionamento navi

Questa parte riguarda il posizionamento delle navi all'interno della griglia di gioco.
La classe ShipDeployment (che funge da controller per javafx), per collocare correttamente le navi sulla griglia, si avvale della tecnica del drag-and-drop, gestita internamente dai metodi dragImage e dropImage. L'utente ha la possibilità di trascinare le navi dalla loro locazione originaria in una cella qualsiasi della griglia, a patto che non vada a collidere con altre navi già presenti e che non fuoriesca dai confini. In questo caso il "drop" non andrà a buon fine. E' stata implementata inoltre la possibilità di ruotare le navi di 90 gradi.
La classe ManageDeployment fornisce funzionalità necessarie al corretto funzionamento di ShipDeployment. Inoltre, è presente un riferimento all'Interfaccia PlaygroundBattle, utile per salvare le coordinate delle navi che sono state posizionate sulla griglia.

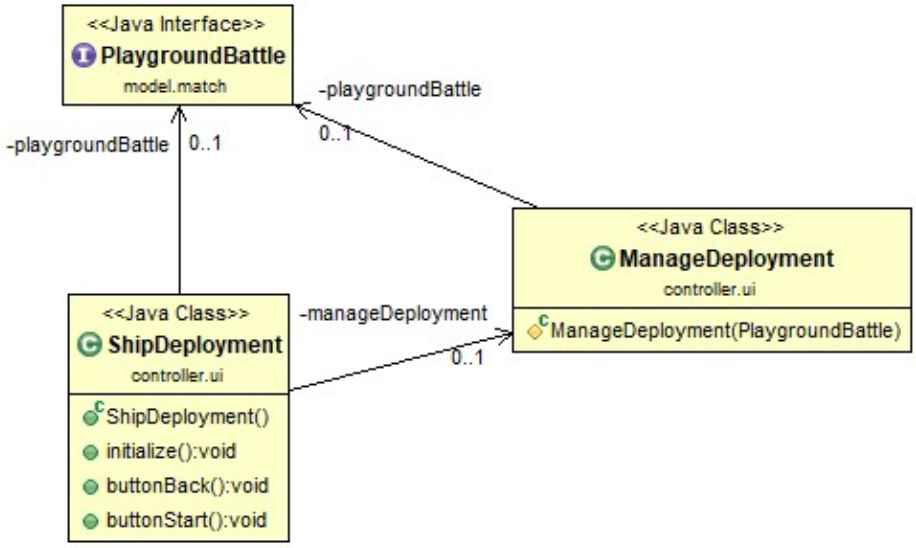

Figura 2.14: Rappresentazione UML del sistema di posizionamento delle navi

Angelo Tinti

Griglia di gioco

La classe Playground modella la griglia utilizzata durante una partita. Su di essa è possibile fare diverse operazioni proprie del match tra le quali posizionare le navi e colpirle, inoltre fornisce anche metodi per effettuare controlli per ottenere informazioni necessarie per il corretto utilizzo. Vi sono metodi che hanno stesso comportamento ma una modalità di utilizzo differente, questo perché volevo dare la possibilità di avere più libertà di scelta a chi usufruirà di tale interfaccia.

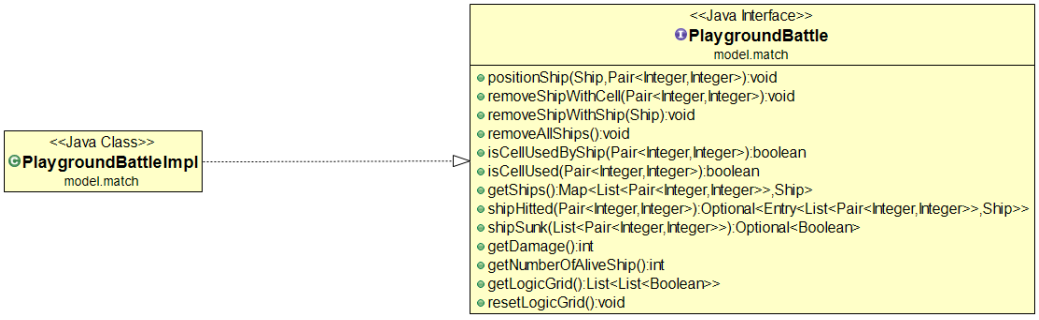

Figura 2.15: Rappresentazione UML dell'intefaccia playground

Controller partita

La classe MatchController si occupa di elaborare i dati passatigli dalla view (BattleView) e modificare a sua volta la griglia (PlaygroundBattle).
Utilizzando i metodi presenti nel campo da gioco va a modificare lo stato del sistema ad ogni nuova interazione. Ancor prima dell'inizio della partita si occupa di ricevere il risultato del posizionamento delle navi e riportare nel nel model tali dati.

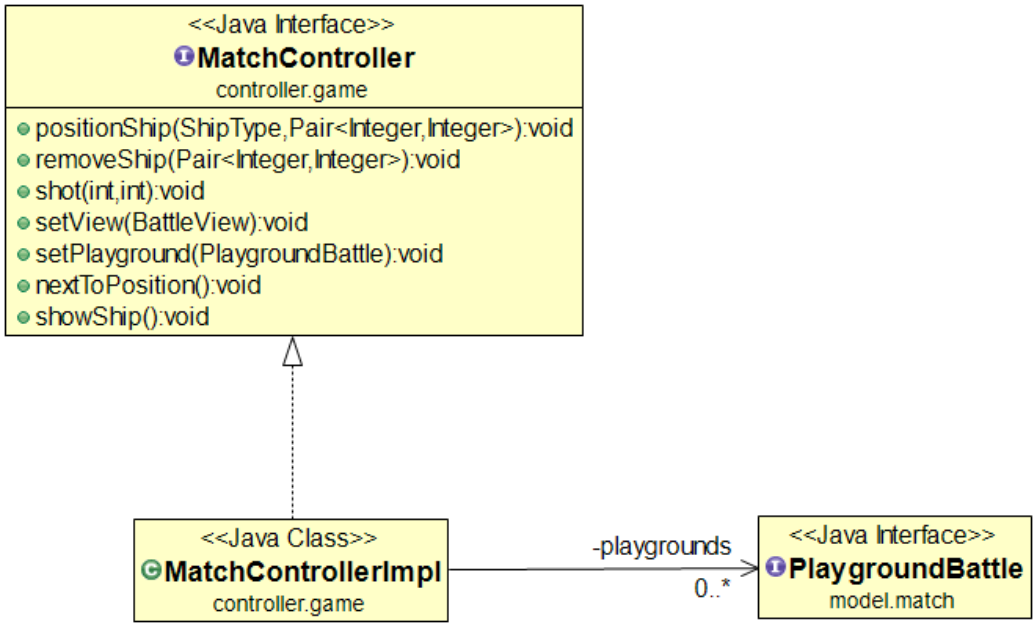

Figura 2.16: Rappresentazione UML dell'entità MathcController e delle funzionalità offerte.

View partita

La classe BattleView gestisce la parte grafica dell'applicazione in fase di battaglia fra i due giocatori. Essa fornisce metodi richiamati dal MatchController in base alla situazione che si viene a presentare (colpo andato a segno, vincita ecc...). Essa verrà caricata al termine del posizionamento delle navi dal controller del match.

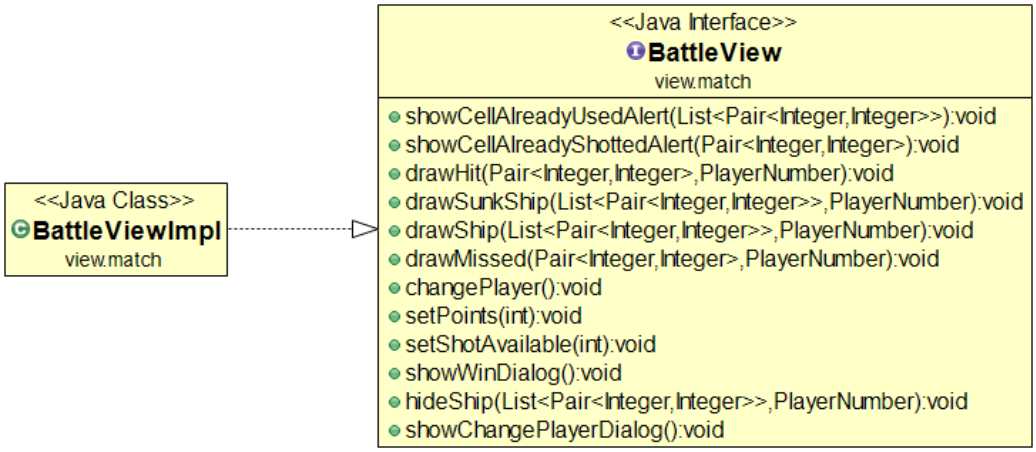

Figura 2.17: Rappresentazione UML dell'entità BattleView e delle funzionalità offerte.

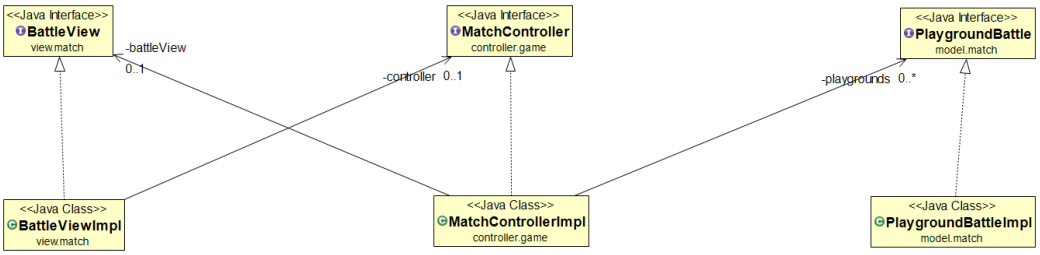

Figura 2.18: Rappresentazione UML delle relazioni che intercorrono tra le entità esposte precedentemente.

Per quanto riguarda le partite contro l'intelligenza artificiale, il controller, constato di essere in tale condizione, chiederà al controller principale di chiamare la shot su di se con i parametri che esso otterrà dal model dell'IA.

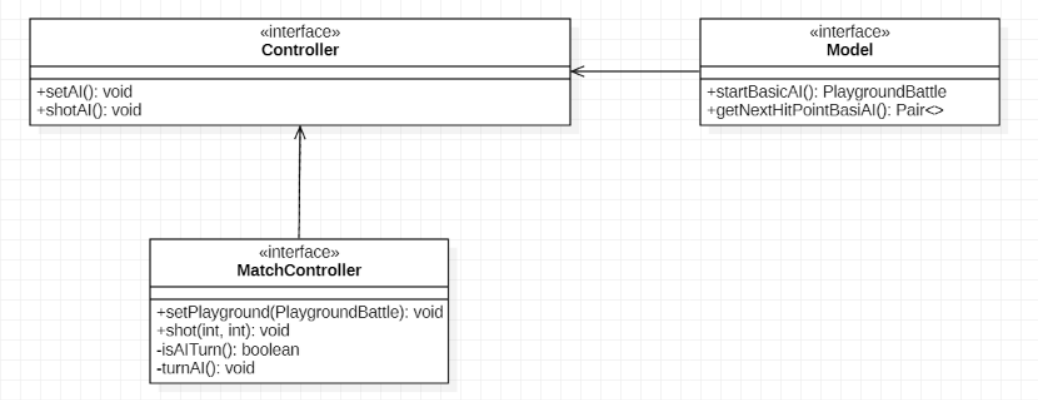

Figura 2.19: Rappresentazione UML del legame MatchController e scelte dell'intelligenza artificiale.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per il testing automatizzato del software abbiamo scelto di utilizzare JUnit, abbiamo inoltre effettuato manualmente i test della GUI, in quanto ci interessava non solo verificarne il corretto funzionamento, ma anche il "look and feel" e la semplicità d'uso da parte dell'utente finale.
Sono stati realizzati test automatizzati per le seguenti entità:

- *CurrentPlayer*: verifica del corretto tracciamento del giocatore corrente, in particolare nel caso di passaggio di valori null.
- *WinCondition*: verifica del corretto comportamento dei metodi, in particolare che vengano lanciate eccezioni se vengono passati parametri non validi nel dominio del problema.
- *Ship*: verifica del corretto comportamento delle navi, in particolare per quanto riguarda la dimensione e i colpi subiti.
- *PlayGroundBattle*: verifica del corretto posizionamento delle navi sulla griglia e relativa rimozione. Inoltre, sono stati testati gli spari e se i cambiamenti portati da essi sul campo di gioco fossero inerenti a quanto aspettato.
- *StatsCalculator*: verifica il corretto funzionamento del calcolo dei valori statistici sia di colui che vince la partita, sia di colui che è stato sconfitto.
- *PlayerManagerStatistics*: verifica il corretto funzionamento dei metodi che aggiornano le statistiche dei giocatori.

- *BasicIntelligenceComputation*: verifica il corretto funzionamento dell'implementazione dell'intelligenza artificiale nel posizionare tutte le navi possibili sulla griglia di gioco.

3.2 Metodologia di lavoro

Per lo sviluppo del progetto, abbiamo cercato di mantenere la suddivisione prestabilita dei ruoli, assegnando ad ognuno compiti distinti, in modo da poter effettuare delle scelte di design in autonomia. Per coordinare il nostro lavoro, abbiamo usato Git come DVCS. In particolare, abbiamo usato "develop" come branch di sviluppo in cui ognuno condivideva i propri commit, dopo aver lavorato sulla propria copia locale del repository. Una volta implementate tutte le funzionalità minime dell'applicazione, abbiamo unito il nostro branch di sviluppo al branch master, che abbiamo usato come branch di release. Fatto questo abbiamo continuato a lavorare sul branch di sviluppo, applicando le nuove modifiche su master una volta raggiunto un funzionamento stabile e soddisfacente. Non si è rivelato necessario l'uso di feature branches. Dopo aver analizzato il dominio applicativo abbiamo cominciato a sviluppare il progetto individualmente, per cui ognuno ha realizzato la propria parte di design. Successivamente, una volta definito il design delle singole parti, ci siamo accordati su come integrare il nostro lavoro. Di seguito un riassunto dei compiti svolti dai componenti del gruppo:

- Angelo Tinti: realizzazione grafica e comportamento delle griglie di gioco.
- Bruno Esposito: creazione e gestione dei profili giocatore e delle relative operazioni su di essi; gestione delle statistiche giocatore; implementazione di base dell'intelligenza artificiale.
- Marco Biagini: gestione delle navi e del loro posizionamento all'interno della griglia di gioco.
- Riccardo Gennari: menù principale, schermata statistiche (solo gui e relativo controller), selezione giocatore (fase di pre-partita) e gestione modalità di gioco.

3.3 Note di sviluppo

Abbiamo fatto uso di Gradle, limitandoci ad utilizzarlo come gestore di dipendenze per il progetto. Nello sviluppo dell'applicazione, abbiamo cercato

di utilizzare dove possibile alcuni elementi avanzati del linguaggio Java. Di seguito un resoconto più dettagliato:

- Angelo Tinti
  - Lambda: utilizzare per ridurre la quantità di codice e fornire una maggior comprensibilità.
  - Stream: così come per le lambda expressions, ho utilizzato gli stream per rendere il codice più compatto e ridurne la complessità.
  - Optional: utilizzati come wrapper per dati che, in base alla situazione, potrebbero non essere presenti.
  - JAVAFX + FXML: utilizzato per la realizzazione della grafica della partita in fase di scontro.
- Bruno Esposito
  - Serializable: per poter caricare e salvare su file binario ogni profilo giocatore in modo da garantire la persistenza;
  - Lambdas: per rendere il codice più compatto e semplice da comprendere;
  - Stream: per implementare il salvataggio su file dei giocatori e per effettuare operazioni o controlli su determinate strutture dati;
  - Optional: per il controllo di eventuali oggetti di valore null;
  - JavaFX + FXML: per realizzare l'interfaccia grafica relativa alla creazione ed alla cancellazione di account, ed il relativo controller;
  - Inoltre, nell'implementazione dell'intelligenza artificiale, ho sviluppato due semplici algoritmi: uno per il posizionamento delle navi e l'altro per il calcolo del prossimo punto di attacco. Per entrambi ho fatto riferimento a numeri interi generati in maniera pseudo-random tramite la libreria Math, in modo da rendere meno prevedibile l'implementazione di base dell'IA.
- Marco Biagini
  - Lambdas: per compattare il codice e renderlo più leggibile;
  - Optional: per evitare problematiche relativi a valori null;
  - Stream: usati principalmente per le iterazioni sulle collezioni;
  - JavaFX + FXML: per realizzare l'interfaccia grafica.

- Riccardo Gennari
  - Lambdas: per rendere il codice più leggibile e compatto;
  - Optional: per rappresentare tipi di valore "vuoti" evitando problematici valori null;
  - JavaFx + FXML: per realizzare l'interfaccia grafica;
  - Generici: parametri di metodo generici per rendere il codice più flessibile e riutilizzabile *(ex. view.util.ChoiceboxInitializer)*;

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Angelo Tinti

La creazione della nostra versione di Battleships è stata un'esperienza interessante. E' stata una prima volta dato che non avevo mai sviluppato un progetto di tale dimensione ed utilizzando strumenti avanzati come ad esempio i DVCS. Sinceramente, per quanto riguarda la mia parte, non sono molto soddisfatto del risultato raggiunto, ci sono degli aspetti che sarebbe stato opportuno gestire ma a causa di una mal gestione del tempo non sono riuscito a coprire. Inoltre mi sono ritrovato a far fronte a difficoltà nello sviluppare codice utilizzando strumenti di cui ho compreso non semplicemente l'utilizzo: purtroppo ho speso molto tempo nella comprensione di javafx e relazione con i file FXML. Se avessi fin da subito richiesto un consiglio probabilmente avrei utilizzato molto meno tempo. Un altro errore lo ho commesso in fase di design, non son riuscito in tale momento a capire le giuste relazioni che si sarebbero venute a creare fra le componenti del sistema, infatti più tardi, quando ero già in fase di sviluppo, mi son ritrovato a modificare parti precedenti e a ripensare la struttura delle mie classi. Comunque penso che tutto ciò mi abbia permesso di capire meglio come lavorare per progetti futuri, saprò cosa non fare in base agli errori commessi e che tipo di approccio avere con un sistema di questo tipo.

Bruno Esposito

Ho trovato istruttivo ed utile lavorare su questo progetto in quanto mi ha permesso di cimentarmi per la prima volta con l'ideazione e la realizzazione di un software. La progettazione riguardante il mio ruolo è stata molto sti-

molante, in particolare la fase di analisi e la strutturazione.
Per quanto riguarda lo sviluppo posso dire di aver lavorato bene con il gruppo. Nonostante le varie difficoltà relative al progetto e nonostante non ci conoscessimo siamo comunque riusciti ad ultimare il tutto, anche se un po' in ritardo rispetto alla scadenza. Inoltre, anche se il periodo di emergenza in parte ha potuto ostacolarci, siamo comunque riusciti a lavorare in parallelo e ad integrare le varie parti ogni volta che si è presentata l'occasione: non sono mancate le diverse chiamate online di gruppo. Organizzandoci in questo modo, non ho dovuto ricoprire un ruolo specifico ma ho cercato di confrontarmi quanto più possibile con gli altri. Sicuramente non tutto è stato fatto nel migliore dei modi, anche soprattutto per una questione di inesperienza.
Mi ritengo abbastanza soddisfatto di come ho svolto il mio ruolo e mi piacerebbe continuare, appena possibile, nello sviluppo di questo progetto inserendo ulteriori aspetti.

Marco Biagini

Lo sviluppo di questo progetto è stata senza dubbio un'esperienza molto formativa, poiché non avevo mai lavorato in team prima d'ora, e soprattutto non avevo mai realizzato un'applicazione di questo genere. Il mio team si è rivelato molto di aiuto nelle situazioni di maggior difficoltà, e questa la ritengo una cosa importante. Non ho ricoperto un ruolo particolare all'interno del team, in quanto abbiamo lavorato in modo coordinato ma comunque indipendente, comunicandoci eventuali problemi da risolvere tutti insieme. Per quanto riguarda il risultato finale, sicuramente sarebbe potuto essere migliore, ma ritengo che sia comunque soddisfacente. Grazie a questo progetto ho avuto la possibilità di conoscere alcuni aspetti avanzati di JavaFX, che torneranno senz'altro utili in progetti futuri.

Riccardo Gennari

Sviluppare questa applicazione è stato interessante, non avevo mai lavorato ad un progetto come parte di un team, nè utilizzato git per coordinarmi con gli altri. L'esperienza è stata positiva, anche se è stato molto impegnativo portare avanti il lavoro al progetto in concorrenza con il resto delle lezione ed esami. Il tempo è stato decisamente un problema. Avremmo sicuramente potuto fare un lavoro migliore, ma date le circostanze sono soddisfatto del risultato. Senz'altro è stato formativo. Non direi di aver coperto un ruolo particolare nel team, in quanto abbiamo lavorato tutti abbastanza indipendentemente, contattandoci quando necessario per aggiornarci e risolvere problemi. Sulla carta dovrei essere stato il repo maintainer, ma in pratica non

è stato quasi mai necessario. Non credo che porterei avanti il progetto, ma qualcuna delle parti fatte meglio la riutilizzerò sicuro per altre applicazioni.

Appendice A

Guida utente

Al lancio dell'applicazione, ci si trova nel menu principale (figura A.1). Per cominciare una partita è necessario dapprima creare un profilo giocatore dal menu profiles (figura A.2) se non lo si è già fatto, e quindi cliccare su new game per accedere alla schermata di selezione giocatore (figura A.4).
Qui, selezionare un profilo da utilizzare per la partita, scegliendo tra quelli precedentemente registrati, ed inserire le credenziali quando richiesto. Si può barrare l'opzione AI per fare sì che il giocatore2 sia controllato dal computer. Dal menù a tendina a destra è possibile scegliere la modalità di gioco.
Quando si è pronti, cliccare il pulsante start. Verrà caricata la schermata di posizionamento navi (figura A.5). Qui ogni giocatore trascinerà le navi visibili in fondo alla schermata sulla griglia di gioco, per deciderne il posizionamento.
Per procedere, cliccare su start game. Il match vero e proprio avrà inizio (figura A.6): ogni giocatore, durante il proprio turno cliccherà in una casella della griglia avversaria per cercare di colpire le navi nemiche. Verrà visualizzata un'icona diversa a seconda dell'esito del colpo. La partita finirà quando saranno soddisfatte le condizioni definite dalla modalità di gioco scelta.
A questo punto si tornerà al menu principale: è possibile visualizzare le statistiche dei giocatori dal menu statistiche (figura A.3).

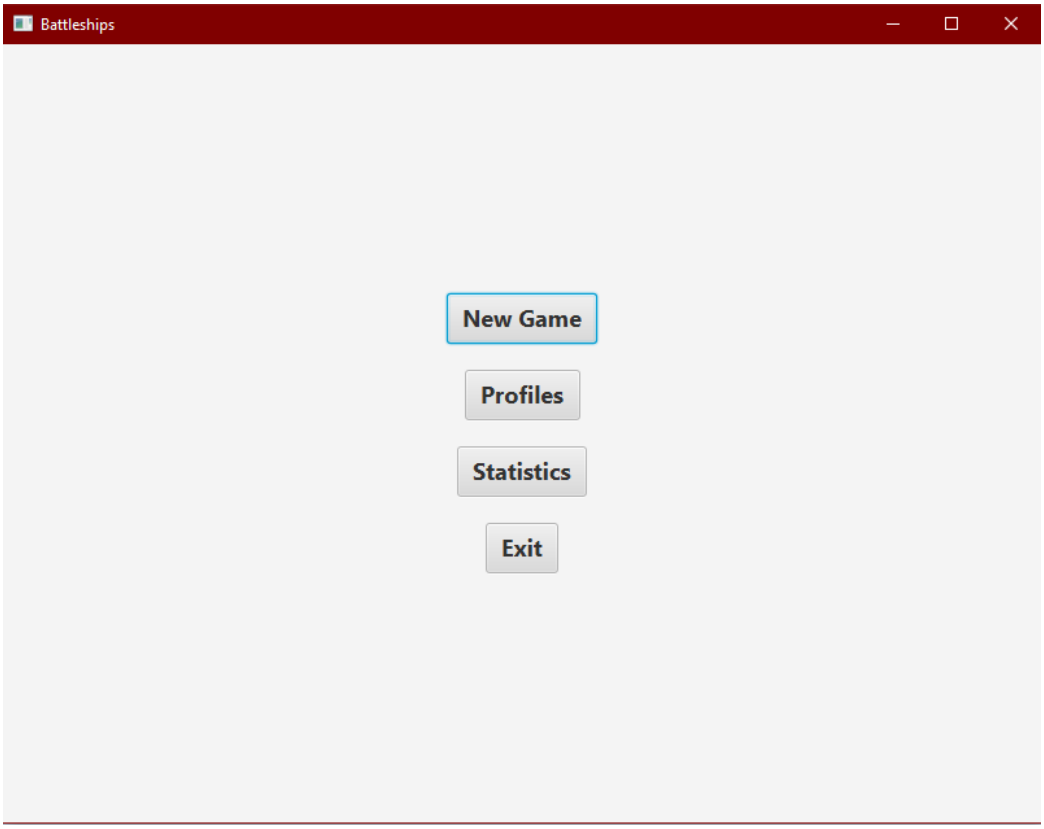

Figura A.1: Menu principale.

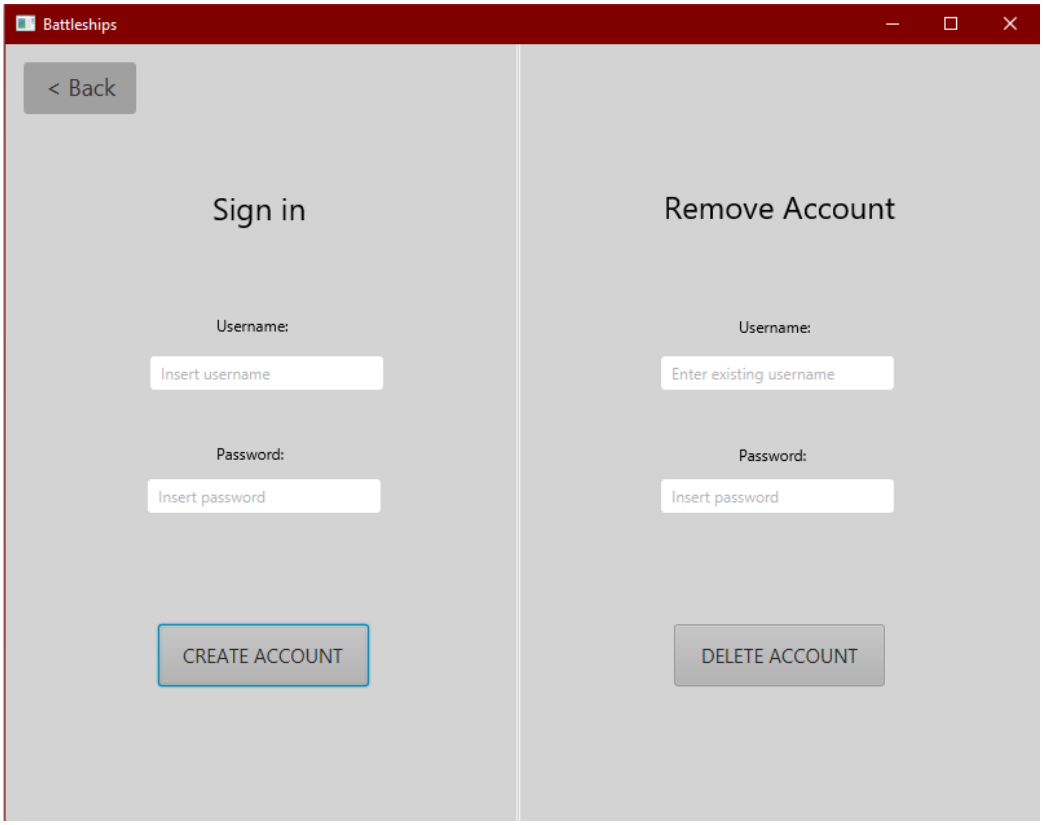

Figura A.2: Schermata profilo giocatore

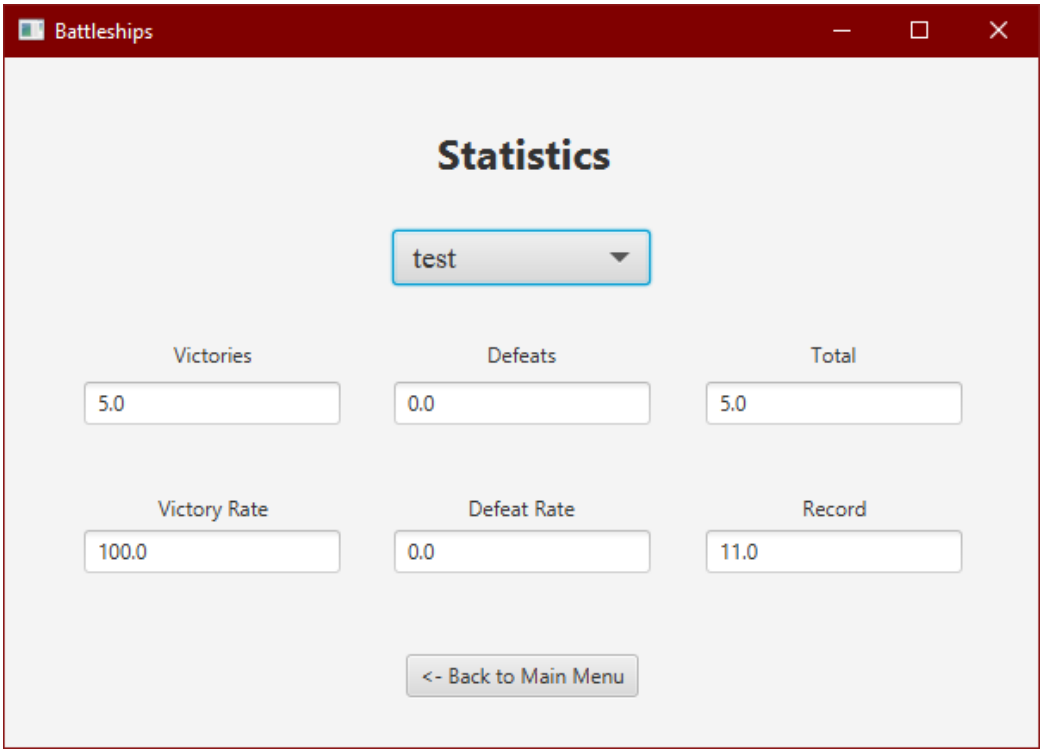

Figura A.3: Menu statistiche.

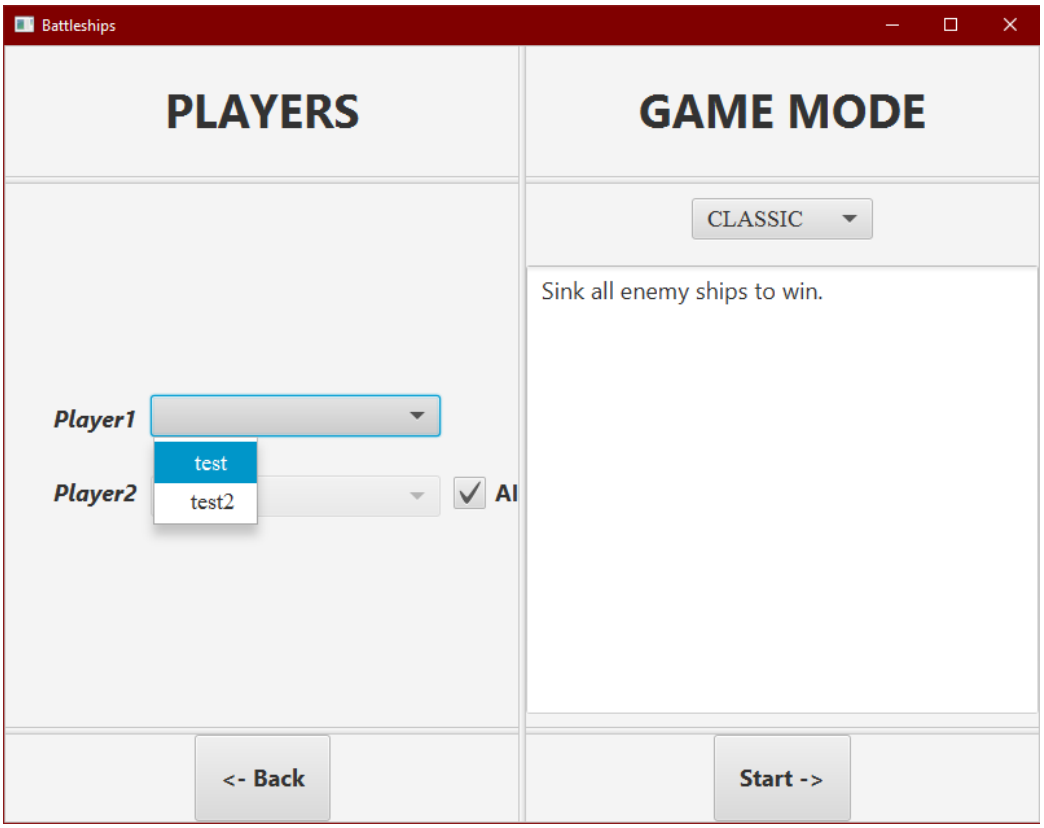

Figura A.4: Menu selezione giocatore.

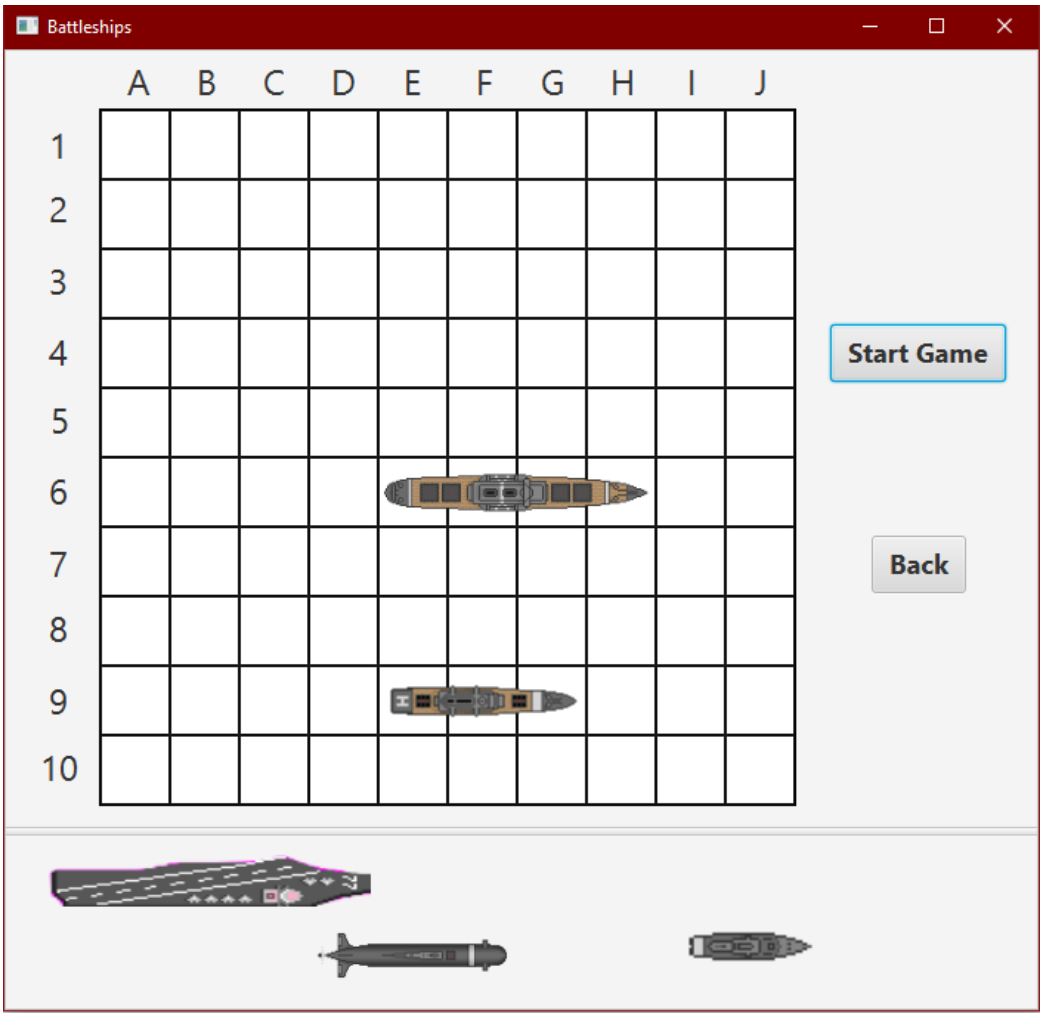

Figura A.5: Schermata posizionamento navi.

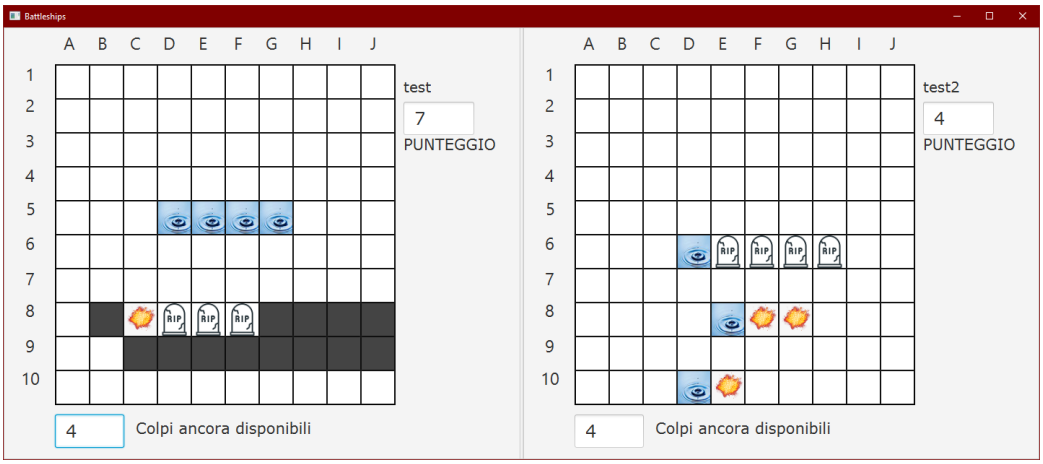

Figura A.6: Schermata del match in corso.