Battleship

Fabio Urbini

Indice

1 **Analisi**
- 1.1 Requisiti
- 1.2 Problema

2 **Design**
- 2.1 Architettura
- 2.2 Design Dettagliato

3 **Sviluppo**
- 3.1 Testing automatizzato
- 3.2 Divisione dei compiti e metodologia di lavoro
- 3.3 Note di sviluppo

4 **Commenti finali**
- 4.1 Conclusioni e lavori futuri
- 4.2 Difficoltà incontrate e commenti per i docenti

Capitolo 1

Analisi

1.1 Requisiti

L'applicazione dovrà render possibile far una partita di battaglia navale, potendo variare in modo molto semplice le regole di gioco, come poter creare nuovi tipi di nave, di grandezze diverse, cambiare la dimensione della mappa, tutto in modo trasparente e facile per l'utente, o il programmatore che intende estendere funzionalità o far uso di quelle già predisposte.

Requisiti funzionali

- Consentire di creare le flotte dei due giocatori (umani) e sparare alle rispettive flotte nemiche
- Giocare 1 contro 1, umano contro umano
- GIocare 1 contro 1, umano contro Intelligenza Artificiale

Requisiti non funzionali

- Modifica del set di regole
- Salvataggio e caricamento della partita
- Modifica del tipo di sparo
- Giocabilità via rete
- Modifica della difficoltà di gioco contro IA

1.2 Problema

Il programma dovrà permettere di poter creare due flotte, una per giocatore, composte da navi di diverso tipo, mantenendo una struttura estendibile e che possa permettere il maggior numero di possibilità possibili per future estensioni. Ogni flotta andrà posizionata in una mappa legata al giocatore. Ogni giocatore può sparare alla flotta nemica, attraverso la mappa, che gestirà gli spari effettuati e lo stato della flotta.
Essendo possibile una partita con un'intelligenza artificiale, a partire da un tipo primitivo, la difficoltà primaria starà nel riuscire a posizionare una flotta correttamente e nel creare degli spari rivolti alla flotta del giocatore.
Una versione futura disporrà di un'intelligenza artificiale che valuterà tramite un algoritmo di maggiore complessità, come sparare alla flotta del giocatore di modo da

minimizzare gli spari effettuati, non prevista nel monte ore. Inoltre dovrà esser predisposto il programma in modo tale da permettere la giocabilità via rete e una facile estensione del gioco. Un esempio tratta l'uso di spari di vario genere, più complessi del semplice sparo su una singola casella.

La creazione di spari da parte dell'intelligenza artificiale richiede algoritmi performanti, che mantengano il sistema reattivo.

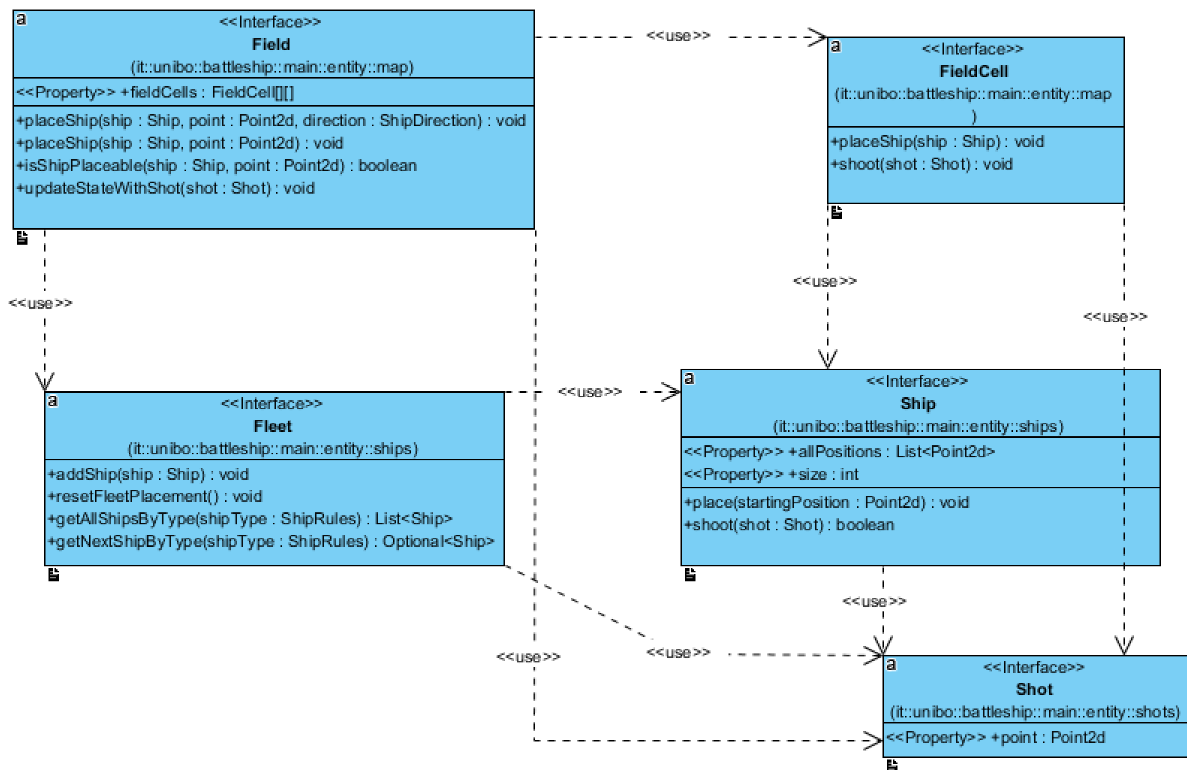

Figura 1.1 UML Analisi.
Nell'UML in figura vengono esposte le principali interfacce utilizzate per il funzionamento del programma e i loro principali metodi.

Capitolo 1

Design

2.1 Architettura

Per il programma è stato scelto l'utilizzo del pattern architetturale Entity Control Boundary, invece del classico Model-View-Controller, per permettere una flessibilità maggiore nella parte Boundary/View. Viene previsto l'utilizzo della console oltre che una GUI. L'idea è di poter dividere la parte Boundary in più classi che possano gestire il problema (il posizionamento della flotta, l'effettivo inizio della partita con gli spari). Il Control si dovrà occupare dell'interazione dell'utente attraverso il Boundary, mascherando quel che avviene nella parte Entity. Nella fase implementativa non si è riuscito a implementare correttamente il pattern ECB, per errori iniziali che hanno messo in pausa l'effettiva realizzazione con refactoring previsti in futuro.

Dovrebbe comunque esser possibile sostituire il modulo Boundary senza grossi effetti collaterali sul Control.

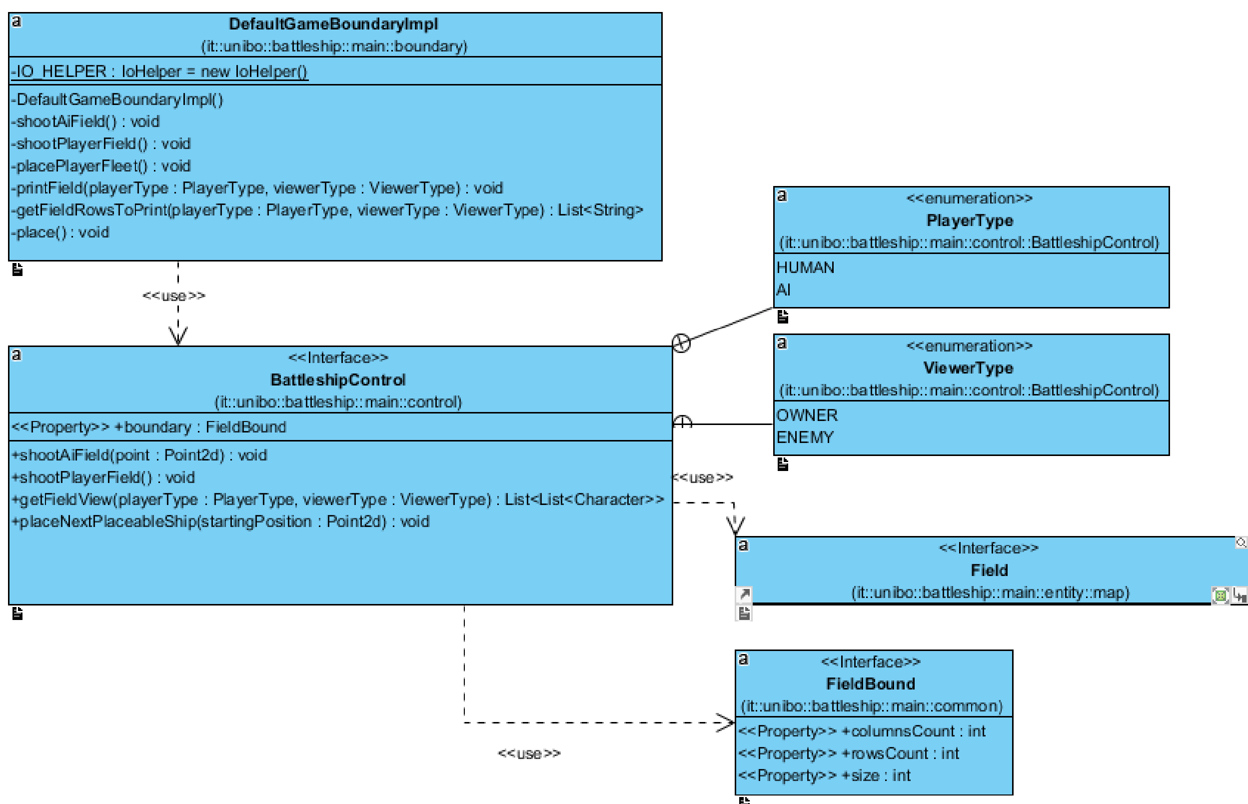

Figura 2.1 Architettura
Nella figura viene rappresentata la relazione tra le varie classi del modello ECB. Si prevede un futuro refactoring per implementare correttamente il pattern. Attualmente il boundary fa uso dei metodi del control. La principale entity legata al control è Field.

2.2 Design dettagliato

Nell'implementazione del programma è stato cercato di seguire il più possibile principi conosciuti della programmazione e design pattern. Tra i principi di base, DRY e KISS, oltre al tentativo di seguire i principi SOLID, in particolare il DIP e LSP.

L'utilizzo frequente del pattern Factory (Static Factory Method, Abstract Factory e Simple Factory) servono per il raggiungimento dello scopo, per avere maggior controllo sulla costruzione effettiva degli oggetti in gioco e un maggiore disaccoppiamento. Inoltre è stato utilizzato il pattern Strategy in modo massivo, per mantenere l'aspetto implementativo slegato dalla struttura prevista del gioco.

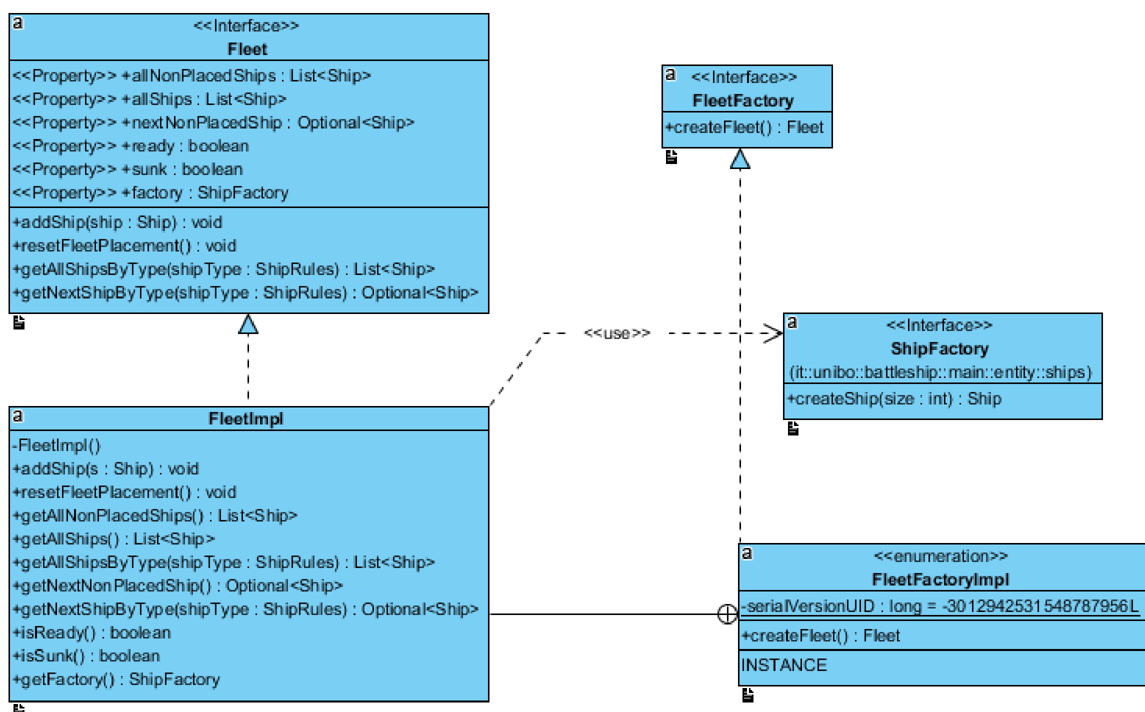

Figura 2.2

Nella figura viene mostrata una parte di design che riguarda la creazione di Fleet, Ship, attraverso l'uso del design pattern Simple Factory, utilizzato da Fleet per la costruzione di oggetti Ship. Nella stessa figura viene utilizzato il pattern Factory per la creazione dell'oggetto Fleet, insieme all'uso di un Singleton per l'implementazione effettiva di FleetFactory. Non è necessario avere più di una Factory di Fleet dello stesso tipo.

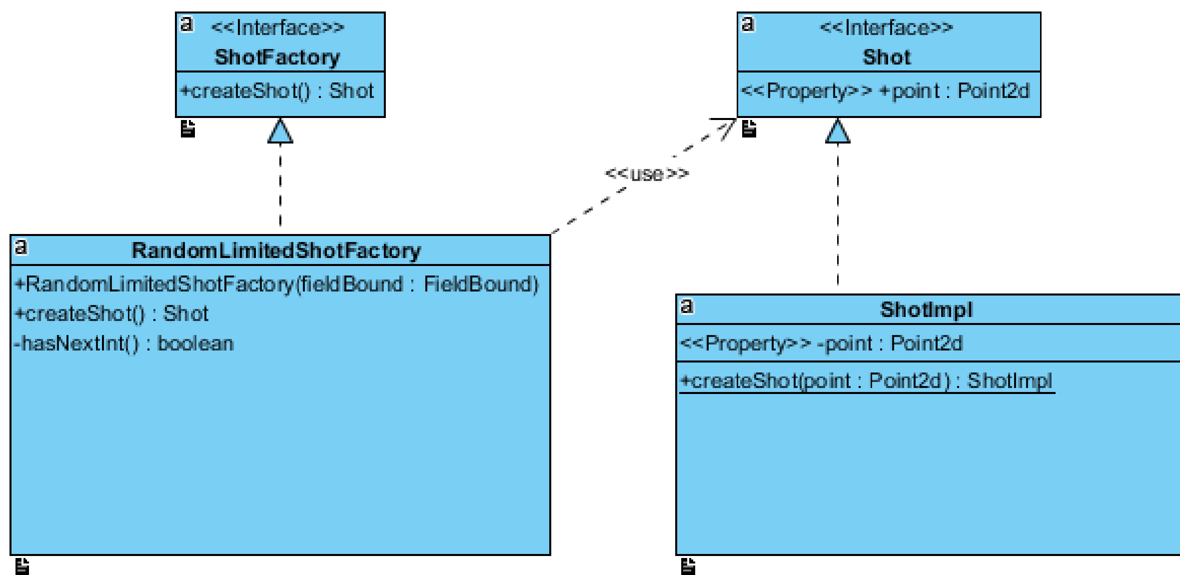

Figura 2.3

In questa figura viene mostrato l'uso del design pattern Static Factory Method per la creazione di Shot, per avere un maggior controllo sull'effettiva creazione (come precondizioni sul punto dato in ingresso). Viene utilizzato il Factory pattern per la creazione di un oggetto Shot e RandomLimitedShotFactory è una sua effettiva implementazione, che servirebbe per creare un oggetto sparo in modo casuale, ma solo per un numero limitato di valori.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Durante lo sviluppo del programma è stato preparato un modulo (relativamente piccolo) di test automatizzati, utilizzando la suite JUnit (4). I test effettuati sono parziali e fini a verificare il continuo funzionamento di parti specifiche del programma. In particolare sono stati effettuati vari test sulla creazione della flotta e delle navi e uno per la creazione di spari per colpire le navi. Si prevede di estendere i test automatici in futuro, in previsione di un futuro refactoring del progetto.

3.2 Divisione dei compiti e metodologia di lavoro

Il progetto è stato svolto singolarmente, utilizzando il DVCS Mercurial come repository di appoggio. In particolare è stato utile per l'utilizzo da diversi computer o postazioni, mantenendo allineato tutto il lavoro. Principalmente è stato utilizzato con semplici push/pull. Non è mai stato necessario tornare ad una versione precedente, pur avendo passato vari stadi di refactoring del progetto (alcuni piuttosto ampi, mettendo in discussione alcune parti viste in fase di analisi).
In uno stadio iniziale è stato dedicato molto tempo ad un'analisi approfondita del sistema che si voleva sviluppare, fino a giungere ad un prototipo funzionante del gioco.

Da quel momento in poi è stata cercata di utilizzare una metodologia orientata all'AGILE, cercando di implementare nuove funzioni (o modificare quelle esistenti) e poi rivalutare il sistema attraverso analisi successive. Come modello di sviluppo inoltre si è cercato di seguire il TDD (Test Driven Development), che è risultato utile nel scovare bug introdotti con modifiche anche lievi.

3.3 Note di sviluppo

Per lo sviluppo del software è stata utilizzata la libreria JUnit (4), per la creazione di test automatici e la libreria Google Guava per la creazione dei metodi hashCode, equals e toString.

È stato trovato un algoritmo interessante per la sequenza di spari da effettuare durante la partita, da studiare ed applicare in futuro, visibile al link (http://io9.gizmodo.com/5910188/an-algorithm-to-help-you-play-the-perfect-game-of-battleship).

Non sono stati rilevati problemi di performance eccetto che per l’uso effettivo dell’intelligenza artificiale, nel posizionare la flotta e dare punti su cui sparare e si prevede che in futuro, nel caso di un’estensione del software, siano i primi punti su cui metter mano.

Capitolo 4

Commenti finali

4.1 Conclusioni e lavori futuri

Durante lo svolgimento del progetto, c'è stata una forte propensione a rimanere sulla parte dell'analisi, il che ha rallentato di molto qualsiasi implementazione, con continue discussioni su quel che di volta in volta è stato prodotto o si sarebbe potuto produrre. Il fatto di cercare la perfezione in fase di analisi è stato tra i primi problemi da risolvere.

È stato cercato di rivalutare il codice con refactoring di volta in volta, senza dare nulla per scontato, o senza etichettare delle parti di programma come intoccabili.

Si prevede in futuro di implementare correttamente il pattern ECB, utilizzando altri design pattern conosciuti, per poter rendere il software più estendibile e robusto. L'idea, oltre che di migliorare la qualità del software, è di estendere le funzionalità, potendo giocare via rete o con un'intelligenza artificiale di livello avanzato (con l'algoritmo trovato descritto in precedenza).