UNIVERSITÀ DEGLI STUDI DI URBINO CARLO BO

Dipartimento di Scienze Pure e Applicate
Corso di Laurea in Informatica Applicata
Programmazione e Modellazione a Oggetti, a.a. 2023/2024

Progetto per Esame

BATTLESHIP

Professore:
Sara Montagni

Studente:
Luca Grasso

Matricola:
294612

Anno Accademico 2023-2024

Indice

Elenco delle figure

Capitolo 1

Analisi

1.1 Analisi dei requisiti

Il gioco **Battleship** si basa su una serie di elementi essenziali che devono essere implementati per garantire il corretto svolgimento della partita.

1.1.1 Requisiti Funzionali

Il gioco deve implementare i seguenti requisiti funzionali:

- **Creazione della griglia di gioco**: Il programma deve generare due griglie di gioco, una per il giocatore e una per l'intelligenza artificiale (IA), ognuna delle quali rappresenta il campo di battaglia con dimensioni standard di 10x10.
- **Posizionamento delle navi**: Le navi, con lunghezze diverse, devono essere posizionate sulla griglia in modo casuale per l'IA, mentre il giocatore può scegliere se posizionarle manualmente o automaticamente.
- **Turni di gioco**: Il gameplay deve alternarsi tra il giocatore e l'IA per permettere a ciascuno di sparare in una casella della griglia avversaria.
- **Feedback sui colpi**: Il programma deve fornire feedback immediato sui colpi, indicando se si è colpita o mancata una nave.
- **Condizioni di vittoria**: Il gioco termina quando tutte le navi di una flotta sono affondate, e il vincitore viene annunciato.
- **Intelligenza artificiale (IA)**: L'IA deve poter sparare in modo casuale o seguire strategie avanzate dopo aver colpito una nave.
- **Interfaccia utente**: L'utente deve poter interagire con la griglia tramite input chiari e avere una visione grafica o testuale dello stato di gioco.

1.1.2 Requisiti Non Funzionali

I requisiti non funzionali assicurano che l'esperienza del gioco sia fluida e priva di problemi tecnici:

- **Usabilità**: L'interfaccia deve essere intuitiva, con comandi semplici e chiare indicazioni per il giocatore.
- **Prestazioni**: L'applicazione deve rispondere rapidamente ai comandi e alle azioni del giocatore e dell'IA, senza ritardi percepibili.
- **Portabilità**: Il gioco dovrebbe essere eseguibile su diverse piattaforme, garantendo un'ampia accessibilità.
- **Affidabilità**: L'applicazione deve essere robusta, gestendo eventuali errori in modo efficiente senza causare arresti anomali.
- **Manutenibilità**: Il codice deve essere ben documentato e organizzato per facilitare eventuali modifiche o miglioramenti futuri.

1.1.3 Requisiti di Qualità

Gli aspetti qualitativi del gioco includono:

- **Esperienza utente**: Il gioco deve fornire una sfida equilibrata, divertente e coinvolgente per il giocatore.
- **Interfaccia grafica opzionale**: Se il gioco prevede una modalità grafica, questa deve essere ben progettata e facile da usare.

1.2 Modello del Dominio e Organizzazione

L'architettura del sistema è suddivisa in varie componenti. Negli schemi ho proposto il Diagramma d'Uso (Figura 1.1) e il Diagramma del Dominio (Figura 1.2).

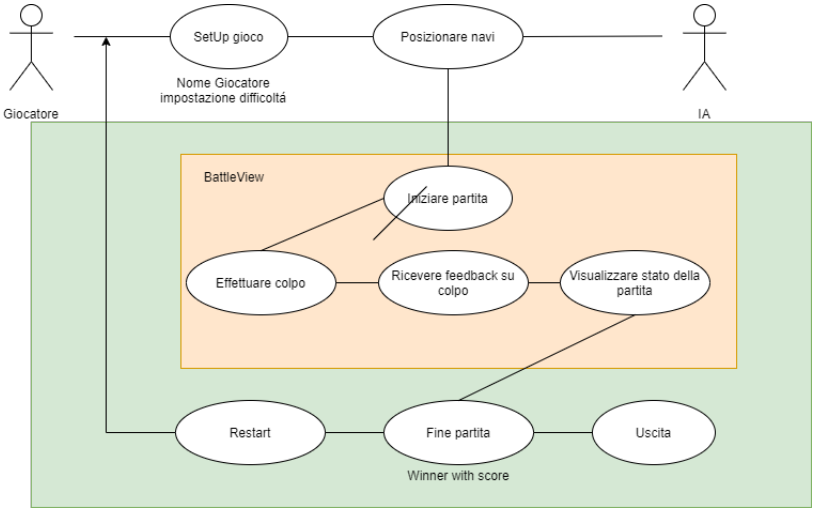

Figura 1.1: UML Diagramma d'uso

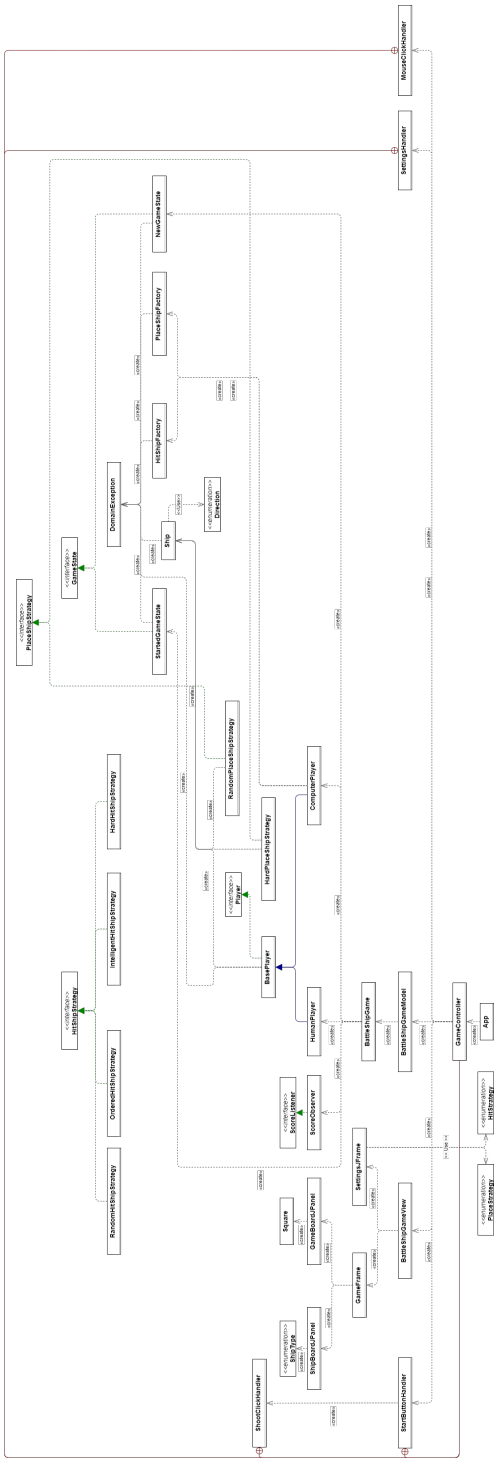

Figura 1.2: UML del Modello del Dominio Completo

Capitolo 2

Design

2.1 Architettura

L'architettura dell'applicazione **Battleship** è stata progettata per garantire una separazione chiara tra le diverse componenti del sistema e per facilitare la gestione delle interazioni tra di esse e la comprensione. Abbiamo adottato il pattern architetturale **Model-View-Controller (MVC)** per raggiungere questi obiettivi.

Il pattern **MVC** è stato utilizzato per suddividere l'applicazione in tre componenti principali: **Model**, **View** e **Controller**. Questa suddivisione permette una gestione modulare e una facile manutenzione dell'applicazione.

2.1.1 Model (Modello)

Il **Model** rappresenta la logica del programma, responsabile della gestione delle regole del gioco, della disposizione delle navi, della gestione dei colpi e del controllo della vittoria o sconfitta. Il Model contiene classi chiave come:

- **BattleShipGameModel**: Contiene la logica di business, gestisce il flusso del gioco, la posizione delle navi e la gestione dei turni.
- **Ship**: Rappresenta una nave all'interno della griglia. Include informazioni come la lunghezza della nave e il suo stato (affondata o non affondata).
- **Grid**: Gestisce la griglia di gioco, una matrice di celle che possono contenere navi o essere colpite.
- **Player e ComputerPlayer**: Gestiscono le azioni dei giocatori, sia umani che computer, compreso il posizionamento delle navi e la scelta delle mosse.

Questa separazione consente una gestione ordinata delle regole del gioco, semplificando eventuali modifiche alla logica di gioco senza influenzare la vista o il controller.

2.1.2 View (Vista)

La **View** è responsabile della presentazione grafica del gioco al giocatore. Essa riceve informazioni dal Model e visualizza lo stato attuale del gioco, come la griglia, le navi posizionate e i colpi effettuati. Alcune delle classi principali nella View includono:

- **GameBoardJPanel**: Rappresenta la griglia di gioco visibile al giocatore, permettendo di vedere le navi e i colpi.
- **GameFrame**: Gestisce la finestra principale del gioco, compresi i pulsanti per avviare una nuova partita o per impostare la difficoltà.

2.1.3 Controller (Controllore)

Il **Controller** funge da intermediario tra il Model e la View. Riceve input dall'utente, come clic o scelte di mosse, e invia queste informazioni al Model per aggiornare lo stato del gioco. Le principali classi del Controller sono:

- **GameController**: Gestisce l'interazione tra l'interfaccia grafica e il Model, incluso il controllo dei turni e la gestione della logica di gioco.
- **SettingsHandler, StartButtonHandler**: Gestiscono le impostazioni di gioco e l'avvio di una nuova partita.

Il Controller riceve input dall'utente e li traduce in azioni che influenzano il Model, mantenendo la separazione tra logica e interfaccia.

2.2 Design Dettagliato

Nella progettazione dettagliata del programma ho adottato diversi aspetti di design che contribuiscono ad una struttura modulare ben organizzata. Di seguito approfondisco alcuni elementi chiave del design, illustrando i problemi e le soluzioni proposte.

2.2.1 Singleton

Descrizione del problema

Quando si ha la necessità di assicurarsi che una determinata classe abbia una sola istanza durante il ciclo di vita dell'applicazione, è utile adottare il pattern Singleton. Questo è particolarmente importante in contesti dove la condivisione di stato o configurazioni tra più componenti è cruciale. Nel tuo caso, le classi *BattleShipGameView* e *SettingsJFrame* potrebbero necessitare di una singola istanza per coordinare le operazioni e mantenere la consistenza dello stato tra diverse parti dell'applicazione.(Figura 2.1)

Soluzione proposta

Utilizzare il pattern Singleton per garantire che sia sempre presente una sola istanza delle classi *BattleShipGameView* e *SettingsJFrame*. Implemento questo pattern utilizzando un costruttore privato e un metodo pubblico statico per fornire l'accesso all'istanza unica.

Motivazione

- **Controllo Unico**: Garantire una sola istanza delle classi evita problemi di consistenza e garantisce un unico punto di controllo e gestione.
- **Gestione dello Stato**: Mantenere stato o configurazioni tra più componenti dell'applicazione in modo centralizzato e coerente.
- **Facilità di Accesso**: Fornire un accesso globale controllato all'istanza dell'oggetto, utile per risolvere condivisioni complesse tra diverse parti dell'applicazione.
- **Prestazioni**: Evitare l'overhead creato da molteplici istanze della stessa classe, specialmente se queste richiedono risorse ingenti o inizializzazioni complesse.

Considerazioni

Nota Bene: l'Utilizzo del Singleton in questo ambito, specialmente per il Setting mi ha permesso di poter riutilizzare la Dialog di di iterazione. Senza questo Pattern non riuscivo a gestire il JButton del setting correttamente (rimaneva disabilitato).

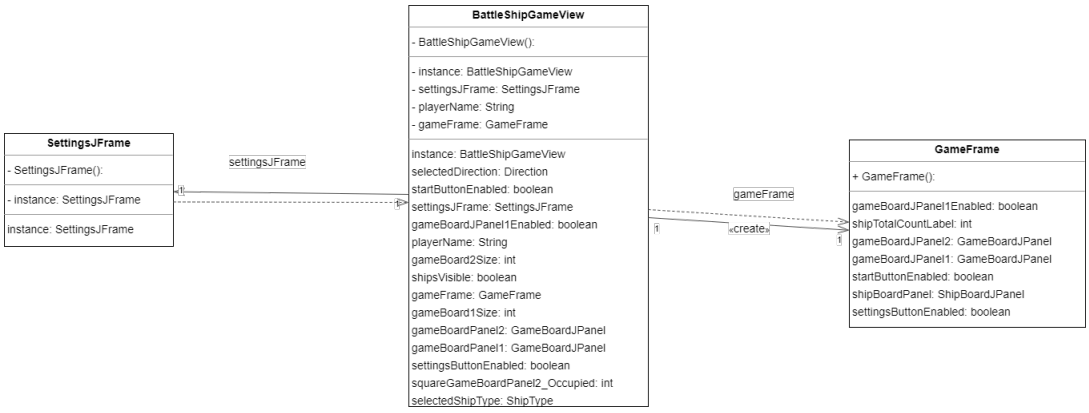

Figura 2.1: UML Singleton nella View

2.2.2 Pattern Strategy

Descrizione del problema

Nel progetto *BattleShipGame*, è necessario gestire diverse strategie di posizionamento delle navi e strategie di attacco in base a differenti configurazioni per il **ComputerPlayer**. Attualmente, le decisioni su come posizionare o attaccare potrebbero essere gestite in modo rigido all'interno del codice principale del gioco, creando un elevato accoppiamento tra la logica del gioco e l'implementazione delle singole strategie. Questo approccio rende difficile aggiungere nuove strategie o modificare quelle esistenti senza cambiare il codice della logica principale del gioco, aumentando il rischio di introdurre errori e riducendo la manutenibilità.(Figura 2.2)

Soluzione proposta

Per risolvere questo problema, propongo l'adozione del **Pattern Strategy**, che permette di separare le diverse strategie di gioco dalla logica principale, rendendo il codice più modulare e flessibile. Di seguito, i dettagli della soluzione proposta:

1. **Interfaccia comune per le strategie**: Le classi che rappresentano strategie di posizionamento o di attacco devono implementare un'interfaccia comune. Ad esempio:

   - *PlaceStrategy* per le strategie di posizionamento delle navi.
   - *HitStrategy* per le strategie di attacco.

   Questo consente al gioco di trattare le diverse strategie in modo uniforme, senza conoscere i dettagli concreti delle loro implementazioni.

2. **Classi concrete delle strategie**: Diverse classi implementano queste interfacce, come ad esempio:

   - Strategie di posizionamento: *RandomPlaceShipStrategy*, *HardPlaceShipStrategy*.
   - Strategie di attacco: *RandomHitShipStrategy*, *OrderedHitShipStrategy*, *IntelligentHitShipStrategy*, *HardHitShipStrategy*.

   Queste classi concrete contengono la logica specifica per ciascuna strategia. Ad esempio, la classe *RandomPlaceShipStrategy* implementerà il posizionamento casuale delle navi, mentre *IntelligentHitShipStrategy* implementerà un algoritmo più sofisticato per colpire le navi nemiche.

3. **Selezione dinamica delle strategie**: Il file di configurazione, come *strategyProperties.properties*, potrebbe specificare quale strategia utilizzare

per ogni fase del gioco. La classe principale del gioco caricherà la configurazione e istanzierà dinamicamente la strategia appropriata utilizzando una factory o semplicemente attraverso l'analisi della configurazione.

4. **Integrazione nel gioco**: Il gioco non avrà bisogno di conoscere i dettagli delle strategie concrete. La logica principale può chiamare i metodi definiti nelle interfacce (*PlaceStrategy* e *HitStrategy*) senza preoccuparsi di quale specifica implementazione è stata utilizzata. Questo rende il sistema altamente flessibile e facilmente estendibile.

Conclusione

Adottare il **Pattern Strategy** nel progetto *BattleShipGame* porta numerosi vantaggi:

- **Flessibilità**: È facile aggiungere nuove strategie di gioco o modificare quelle esistenti senza toccare il codice della logica principale.
- **Manutenibilità**: Separare la logica delle strategie rende il codice più modulare e quindi più facile da mantenere e testare.
- **Riduzione dell'accoppiamento**: Il gioco non dipende più direttamente dalle implementazioni concrete delle strategie, migliorando l'architettura generale.
- **Configurabilità**: Utilizzando un file di configurazione come *strategyProperties.properties*, è possibile cambiare le strategie senza modificare il codice sorgente, rendendo il sistema adattabile a diversi scenari di gioco.

Questa soluzione garantisce un'applicazione più modulare e scalabile, migliorando la qualità del codice e facilitando l'implementazione di future estensioni.

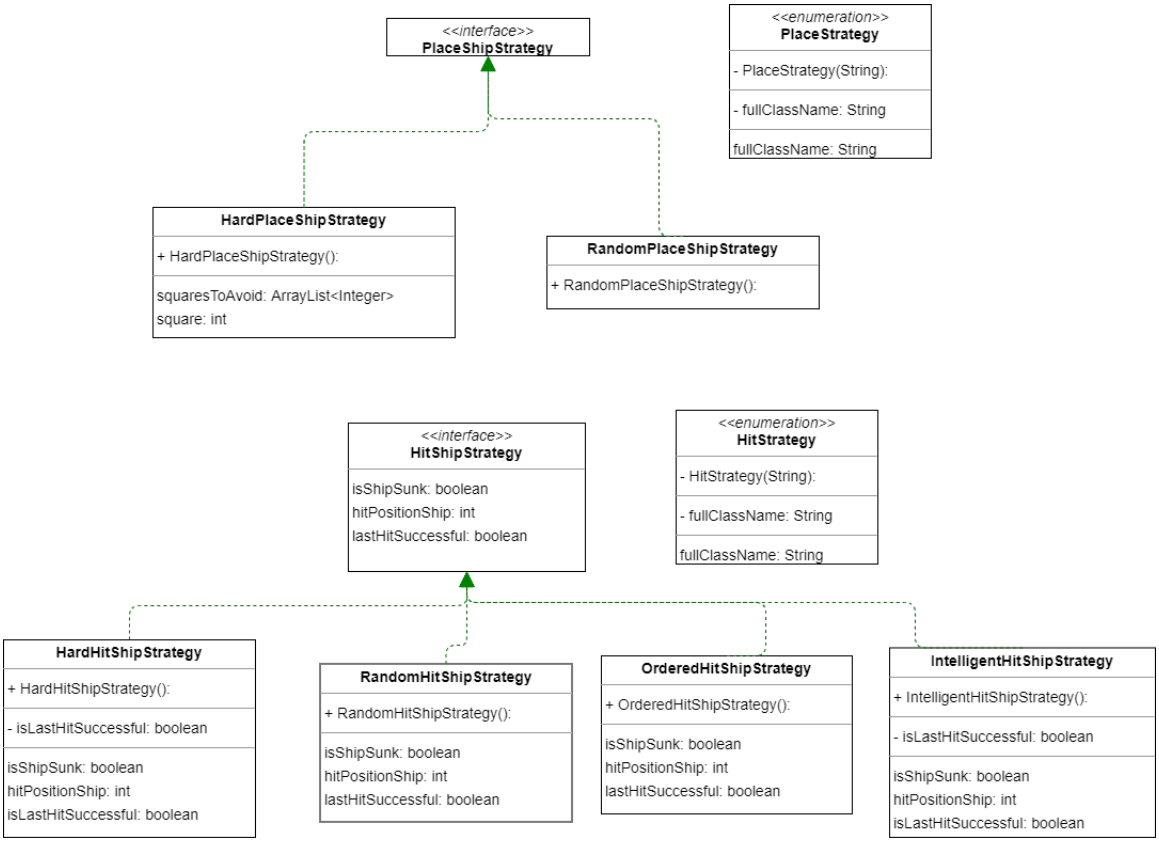

Figura 2.2: UML Strategy

2.2.3 Pattern Factory

Descrizione del problema

Nel progetto *BattleShipGame*, oltre alla necessità di avere differenti strategie per il posizionamento delle navi e l'attacco (già gestito dal *Pattern Strategy*), esiste anche la necessità di creare dinamicamente queste strategie in base a diverse condizioni o configurazioni di gioco.(Figura 2.3)

Creare direttamente le istanze delle strategie (come *RandomHitShipStrategy* o *HardPlaceShipStrategy*) all'interno della logica del gioco aumenterebbe il grado di accoppiamento e renderebbe il codice più difficile da mantenere. Inoltre, quando c'è la necessità di introdurre nuove strategie, sarebbe necessario modificare varie parti del codice esistente, infrangendo il principio di *apertura/chiusura* (Open/Closed Principle).

Soluzione proposta

Per risolvere questo problema e disaccoppiare la logica di creazione delle strategie dalla logica principale del gioco, propongo l'adozione del **Pattern Factory**. Questo pattern consente di centralizzare la creazione delle strategie in una classe dedicata, separando ulteriormente la responsabilità e migliorando la manutenibilità del codice.

1. **Interfacce Factory**: Le interfacce *IHitShipFactory* e *IPlaceShipFactory* definiscono i contratti per le factory che creeranno le varie strategie di attacco (*HitStrategy*) e di posizionamento delle navi (*PlaceStrategy*).

2. **Classi Factory concrete**: Le classi come *HitShipFactory* e *PlaceShipFactory* sono le implementazioni concrete delle interfacce factory. Queste classi contengono la logica per decidere quale strategia restituire, basandosi su configurazioni o logiche definite (come il contenuto di *strategyProperties.properties*).

3. **Integrazione con la logica di gioco**: La classe *ComputerPlayer*, invece di istanziare direttamente le strategie, si affida alle factory per ottenere la strategia appropriata.

Vantaggi del Pattern Factory

L'adozione del **Pattern Factory** per creare dinamicamente le strategie di gioco offre numerosi vantaggi:

- **Disaccoppiamento**: La logica di creazione delle strategie è separata dalla logica principale del gioco.

- **Facilità di manutenzione**: Aggiungere nuove strategie non richiede modifiche al codice della logica di gioco, ma solo alle factory.

- **Flessibilità**: La creazione delle strategie può essere facilmente modificata o estesa.

- **Riutilizzabilità**: Le factory possono essere riutilizzate in diverse parti dell'applicazione.

Conclusione

Combinando il **Pattern Factory** con il **Pattern Strategy**, il progetto *BattleShipGame* diventa altamente flessibile e modulare, migliorando la manutenibilità, riducendo il rischio di errori e consentendo un'estensione del sistema per futuri miglioramenti.

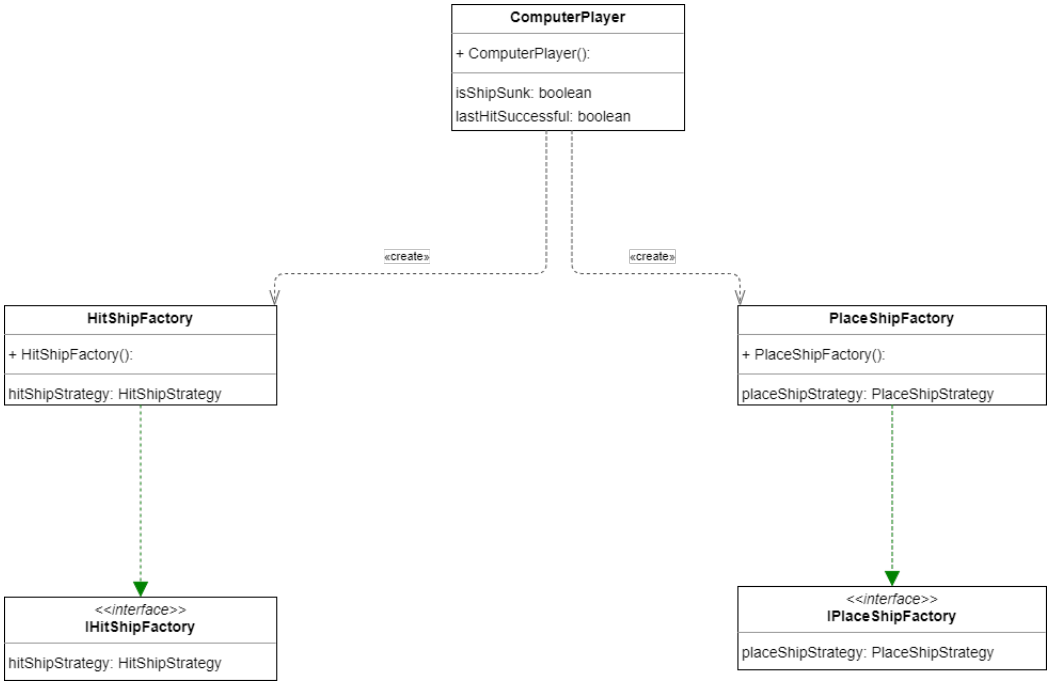

Figura 2.3: UML Factory

2.2.4 Pattern State

Descrizione del problema

Nel progetto di un gioco si può trovare vari stati (come *Nuovo Gioco*, *Gioco Iniziato*, *Gioco Pausato*, ecc.). Gestire questi stati utilizzando una serie di istruzioni condizionali nella classe principale renderebbe il codice difficile da mantenere e poco flessibile. Ogni volta che si vuole aggiungere un nuovo stato o modificare un comportamento esistente, sarebbe necessario modificare molte parti del codice, aumentando il rischio di errori.(Figura 2.4)

Soluzione proposta

Per risolvere questo problema, propongo l'uso del **Pattern State**, che consente di modellare i diversi stati del gioco come classi separate, ognuna delle quali implementa un comportamento specifico. Questo riduce il grado di accoppiamento e rende il sistema più flessibile e facile da estendere.

1. **Interfaccia *GameState***: L'interfaccia *GameState* definisce i metodi comuni a tutti gli stati del gioco, come ad esempio *startGame()*, *pauseGame()*, *endGame()*, ecc.

2. **Stati concreti**: Gli stati concreti, come *NewGameState* e *StartedGameState*, implementano l'interfaccia *GameState* e contengono il comportamento specifico per ogni stato.

3. **Classe *BattleShipGame***: La classe principale del gioco contiene un riferimento all'oggetto *GameState* corrente. Quando lo stato del gioco cambia, l'oggetto *GameState* viene aggiornato con una nuova istanza dello stato appropriato.

Vantaggi del Pattern State

- **Disaccoppiamento**: Ogni stato è incapsulato in una classe separata, rendendo il codice della classe principale più pulito e facile da mantenere.
- **Flessibilità**: Aggiungere nuovi stati o modificare quelli esistenti è semplice e non richiede modifiche alla logica principale del gioco.
- **Estendibilità**: È facile aggiungere nuovi stati senza dover alterare il codice esistente.

Conclusione

Adottare il **Pattern State** nel progetto *BattleShipGame* permette di modellare in modo flessibile e manutenibile i diversi stati in cui il gioco può trovarsi, migliorando la modularità del codice e facilitando l'estensione del sistema con nuovi stati o comportamenti. Visto la natura del gioco ho avuto solo la necessità di implementare un StartedGameState e NewGameState.

2.2.5 Pattern Observer

Descrizione del problema

Nel progetto *BattleShipGame*, ci sono diversi elementi, come il punteggio o lo stato del giocatore, che devono essere monitorati da altre parti del sistema (ad esempio, un'interfaccia utente o un modulo di gestione del punteggio). Senza un meccanismo strutturato, questa notifica dei cambiamenti potrebbe essere gestita tramite chiamate dirette ai metodi degli altri componenti, creando un forte accoppiamento e una struttura rigida che sarebbe difficile da mantenere ed estendere.(Figura 2.5)

Soluzione proposta

Per risolvere questo problema, adotto il **Pattern Observer**, che consente agli oggetti di registrarsi come osservatori di un soggetto (ad esempio, il punteggio o lo stato del giocatore). Quando il soggetto cambia il suo stato, notificherà automaticamente tutti gli osservatori registrati, che potranno quindi reagire di conseguenza.

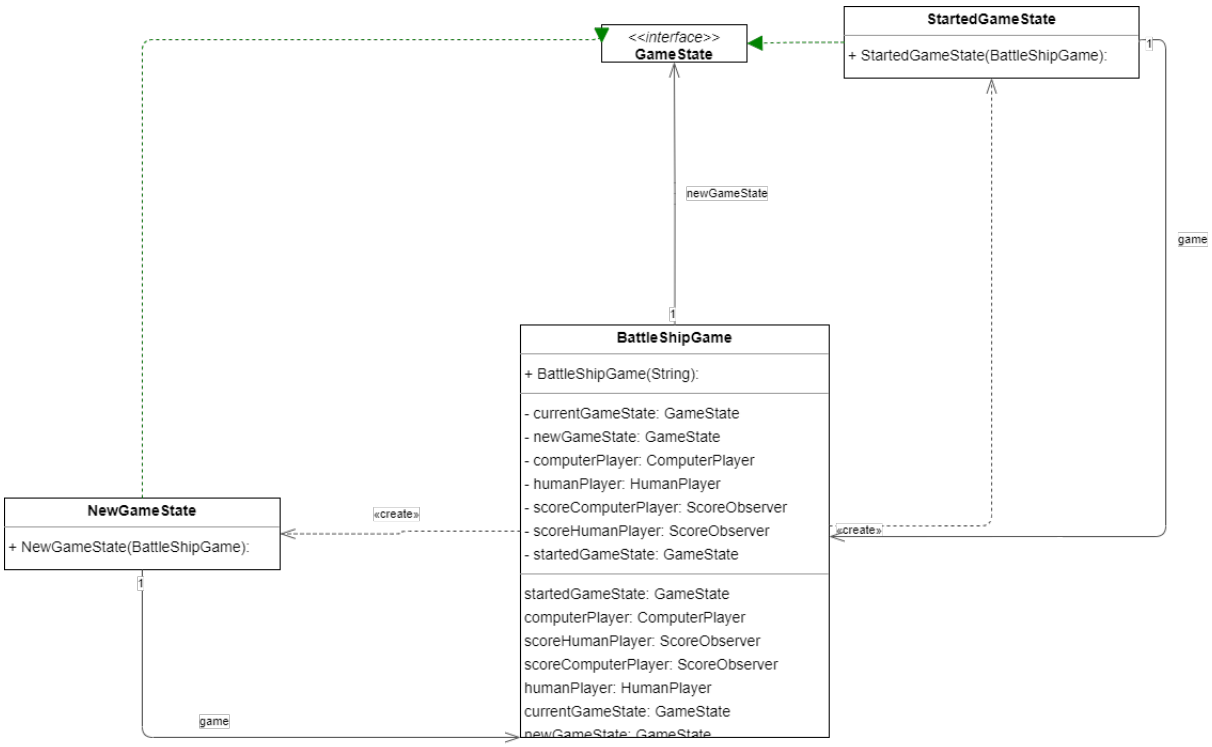

Figura 2.4: UML State

1. **Interfaccia *ScoreObserver***: L'interfaccia *ScoreObserver* rappresenta gli oggetti che vogliono essere notificati quando lo stato del punteggio cambia. Ogni classe che implementa questa interfaccia deve fornire un'implementazione del metodo *update()*.

2. **Classe *ScoreListener***: *ScoreListener* è la classe che contiene la lista degli osservatori e notifica tutti gli osservatori ogni volta che cambia il punteggio.

3. **Classe *BasePlayer* o *Player***: Questa classe può rappresentare i giocatori del gioco e potrebbe essere uno degli osservatori del punteggio. Quando il punteggio cambia, il giocatore può essere notificato per aggiornare la sua interfaccia o il suo stato.

Vantaggi del Pattern Observer

- **Disaccoppiamento**: Il soggetto e gli osservatori sono disaccoppiati, rendendo il sistema molto flessibile.

- **Estendibilità**: È facile aggiungere nuovi osservatori in futuro senza dover modificare il codice esistente del soggetto.

- **Modularità**: Gli osservatori possono essere aggiunti e rimossi dinamicamente durante l'esecuzione dell'applicazione.

Conclusione

L'adozione del **Pattern Observer** nel progetto *BattleShipGame* consente di costruire un sistema di notifiche flessibile ed estensibile, migliorando la manutenibilità e la modularità del sistema. Ho utilizzato il pattern per gestire il punteggio in modo flessibile.

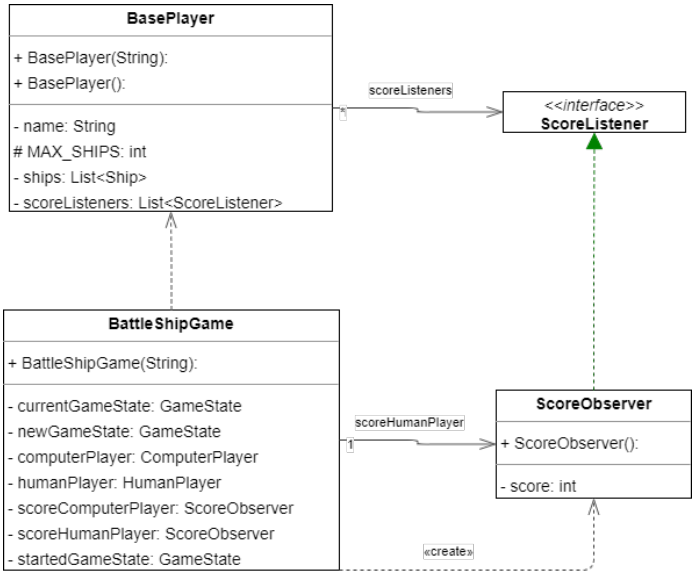

Figura 2.5: UML ScoreObserver

2.2.6 Pattern Factory Method

Descrizione

Il *Pattern Factory Method* consente di delegare la creazione di oggetti a una sottoclasse, che decide quale oggetto istanziare. Questo pattern è utile per ridurre il grado di accoppiamento tra le classi e migliorare la flessibilità del codice, specialmente quando si ha la necessità di creare oggetti complessi o appartenenti a una gerarchia di classi.(Figura 2.6)

Applicazione nel progetto

Nel progetto *BattleShipGame*, questo pattern è applicato per la creazione dinamica dei giocatori (*Player*). Le classi *HumanPlayer* e *ComputerPlayer* rappresentano implementazioni concrete di *Player*, e una factory potrebbe essere utilizzata per gestire la creazione di queste istanze in modo flessibile.

Implementazione

1. **Interfaccia PlayerFactory**: Definisce il metodo per creare un giocatore.
2. **ConcretePlayerFactory**: Implementa la logica per creare dinamicamente un *HumanPlayer* o un *ComputerPlayer*, in base al parametro passato.
3. **Uso del Factory Method**: La logica principale del gioco può delegare alla factory la creazione dei giocatori, disaccoppiando la logica di creazione dal resto del codice.

Vantaggi del Pattern Factory Method

- **Flessibilità**: La factory permette di creare dinamicamente oggetti diversi in modo flessibile, permettendo l'estensione del sistema con nuovi tipi di giocatori senza modificare il codice esistente.
- **Disaccoppiamento**: Il codice che utilizza i giocatori non ha bisogno di conoscere i dettagli della loro creazione, rendendo il sistema più modulare e mantenibile.
- **Estendibilità**: Nuovi tipi di giocatori, come *AIPlayer* o altri *Player*, possono essere facilmente aggiunti estendendo la factory.

2.2.7 Pattern Composite

Descrizione del problema

Nel progetto *BattleShipGame*, esiste la necessità di gestire in modo efficiente la rappresentazione e l'interazione con la griglia di gioco e i vari componenti visivi, come le navi e le caselle. La griglia è composta da singole celle che rappresentano spazi vuoti o parti di navi. Gestire queste caselle singolarmente e poi come parte di una griglia completa può portare a un codice difficile da mantenere e aumentare il rischio di errori.(Figura 2.7)

Senza una struttura coerente, la gestione di queste componenti visive rischierebbe di diventare ingombrante e ripetitiva.

Soluzione proposta

Per risolvere questo problema, propongo l'adozione del **Pattern Composite**, che permette di trattare sia le singole caselle (*Square*) che interi pannelli di gioco (*GameBoardJPanel*, *ShipBoardJPanel*) attraverso una gerarchia comune.

1. **Componente foglia**: La classe *Square* rappresenta una singola cella della griglia. Questa componente foglia gestisce lo stato e il comportamento di ogni cella individualmente.

2. **Componente composito**: Le classi *GameBoardJPanel* e *ShipBoardJPanel* raggruppano diverse caselle (*Square*) e rappresentano l'intero tabellone di gioco. Questi compositi possono aggiungere, rimuovere o iterare sui singoli componenti e gestire il loro stato come un'unica entità.

3. **Interfaccia comune**: Le classi foglia e composito condividono una stessa interfaccia (o classe astratta), consentendo di gestire in modo uniforme sia gli elementi individuali che i gruppi di elementi.

Vantaggi

- **Uniformità**: Trattare oggetti individuali e compositi nello stesso modo semplifica la gestione del codice e riduce la duplicazione.

- **Manutenibilità**: Il codice diventa più modulare e facile da estendere, permettendo l'aggiunta di nuove componenti senza alterare il comportamento esistente.

- **Flessibilità**: Si possono creare strutture gerarchiche complesse senza aumentare la complessità della logica di gestione.

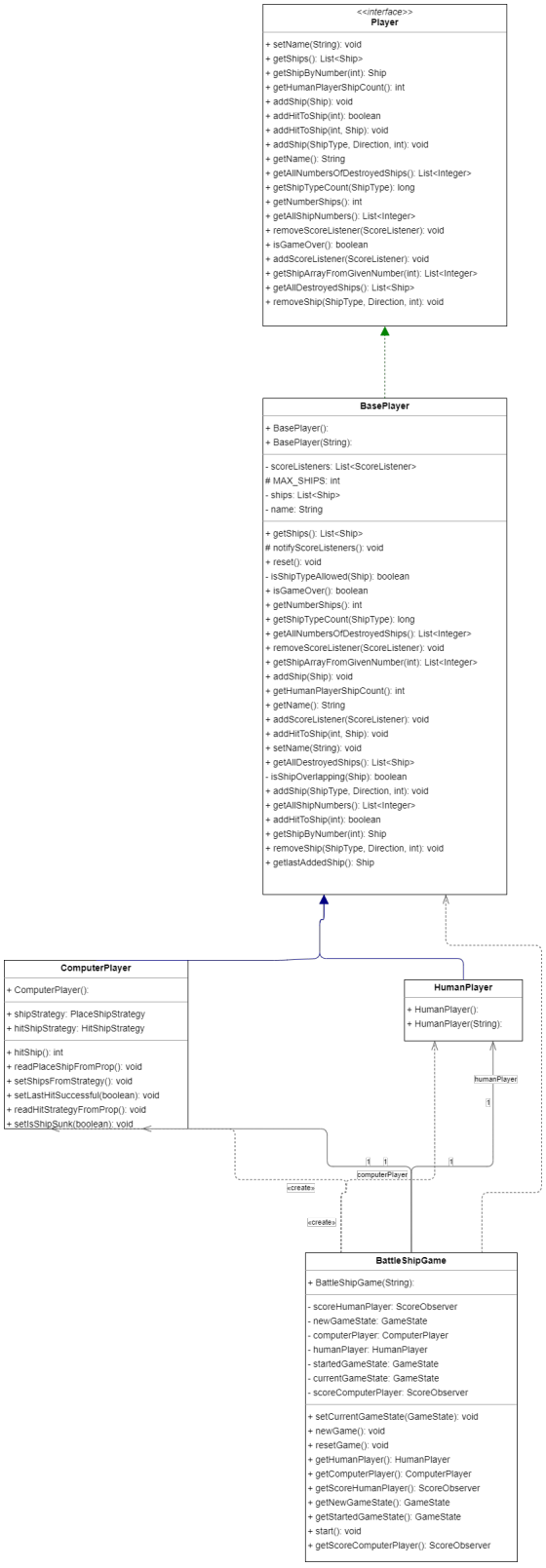

Figura 2.6: UML Method

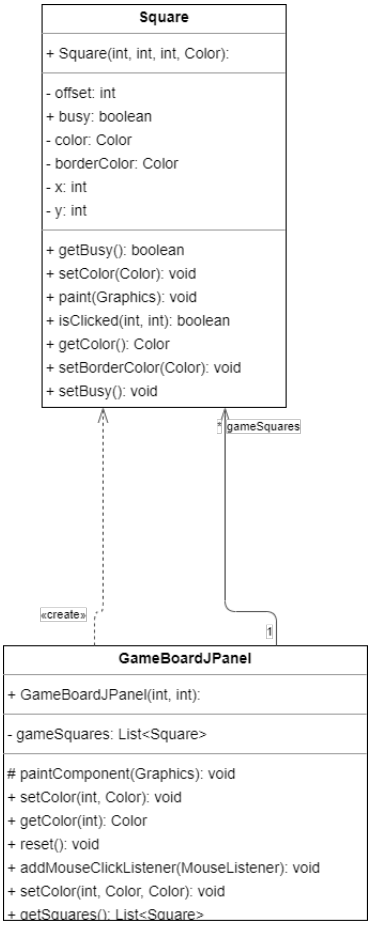

Figura 2.7: UML Composite

Capitolo 3

Sviluppo

3.1 Testing

Per il testing del progetto *BattleShipGame*, è stato utilizzato il framework di unit testing **JUnit 5**, seguendo i principi del *Test Driven Development (TDD)* e dell'automatizzazione dei test. I test sono stati scritti con l'obiettivo di verificare il corretto funzionamento del modello di gioco, cercando di "stressare" il software per individuare problemi e bug che altrimenti sarebbero risultati difficili da rilevare. Ogni nuova funzionalità implementata nel modello veniva subito sottoposta a test unitari per garantirne la correttezza.

Il ciclo di sviluppo seguiva la regola per cui, nel caso in cui un test fallisse, lo sviluppo si interrompeva immediatamente per indagare sul problema riscontrato e risolverlo prima di proseguire. Questa metodologia ha permesso di assicurarsi che ogni parte del progetto funzionasse correttamente prima di aggiungere nuove feature, riducendo così il tempo di debug e aumentando la stabilità del codice.

L'uso di **JUnit 5** è stato particolarmente utile per testare i metodi del modello logico, come la gestione delle navi, la verifica dei colpi e l'affondamento delle unità. Questo ha reso possibile verificare immediatamente se un metodo appena aggiunto si comportasse come previsto. In particolare, i test sono stati scritti per verificare:

- Il corretto posizionamento delle navi sulla griglia di gioco.
- Il calcolo degli attacchi e delle collisioni tra i colpi e le navi del giocatore o del computer.
- L'affondamento delle navi e la gestione delle varie fasi di gioco.

Sebbene il progetto contenga anche un'interfaccia grafica e una componente di casualità nelle scelte del computer, il focus principale dei test è stato sulla parte logica del modello, lasciando la parte interattiva e visiva per un controllo manuale tramite debug visivo.

3.1.1 Debug visivo e log dei messaggi

Per facilitare il debug durante lo sviluppo, nel progetto è stato integrato un sistema di debug visivo attivabile dalle impostazioni del gioco. Questa modalità consente di mostrare tutte le navi sulla griglia, comprese quelle nascoste del computer, rendendo più facile monitorare lo stato della partita e verificare il corretto funzionamento degli attacchi e delle collisioni.

Inoltre, il core del gioco è stato progettato per fornire feedback tramite messaggi di `System.out` durante l'esecuzione, che aiutano a monitorare il flusso di gioco e a identificare eventuali anomalie nel comportamento del software. Questi messaggi sono utili non solo per il debugging manuale, ma anche per fornire informazioni durante l'esecuzione dei test.

3.1.2 Unit Test per il Model

I test automatizzati sono stati concentrati sulla parte logica del gioco, implementata nel modello. In particolare, è stato creato un set di **unit test** specifici per la classe `BattleShipGameModel`, che è responsabile della gestione del gioco. Questi test verificano:

- Il corretto posizionamento delle navi sulla griglia.
- La gestione dei colpi.
- Il corretto affondamento delle navi.

I test sono stati strutturati per coprire vari scenari, inclusi i casi limite, come colpi ai margini della griglia o colpi che coinvolgono navi sovrapposte. Grazie a questi test, è stato possibile garantire che la logica del gioco funzionasse correttamente in tutti i casi previsti, riducendo il rischio di bug durante l'implementazione delle feature principali.

3.2 Metodologia di lavoro

Il progetto "BattleShipGame" è stato interamente sviluppato in maniera individuale, il che significa che tutte le fasi di progettazione e implementazione sono state realizzate da me. Sebbene non vi sia stato alcun lavoro di integrazione con altri sviluppatori, ho deciso di adottare comunque pratiche tipiche di un ambiente collaborativo, come l'utilizzo di un sistema di controllo di versione distribuito (DVCS), ovvero Git, per mantenere traccia dei progressi.

Durante lo sviluppo, ho eseguito commit frequenti ogni volta che apportavo modifiche significative o aggiungevo nuove funzionalità al codice. I commit venivano regolarmente pushati su un repository GitHub, garantendo così che tutte le fasi del progetto fossero documentate e versionate in modo appropriato. Il ramo principale del progetto è stato denominato `master`, e solo quando il

codice ha raggiunto uno stato stabile e definitivo, ho effettuato la merge su tale branch per preparare la release.

Durante lo sviluppo, sono state eseguite verifiche periodiche per assicurarsi che i requisiti inizialmente definiti fossero rispettati. Queste verifiche mi hanno permesso di valutare l'opportunità di apportare modifiche, implementare nuove funzionalità o riorganizzare il codice per migliorare la struttura generale del progetto. Una delle attività fondamentali durante la fase di progettazione è stata la creazione di una chiara suddivisione in package per contenere i vari file sorgente. Questa organizzazione non solo ha migliorato la leggibilità del progetto, ma ha anche facilitato la gestione e la manutenibilità del codice nel tempo.

La fase successiva dello sviluppo ha riguardato l'integrazione degli asset grafici per il gioco. È stato scelto con cura un set di immagini per rappresentare navi, colpi, aree di mare, ed elementi visivi dell'interfaccia. Dopo aver selezionato questi asset, ho apportato alcune modifiche alla logica della `View` per garantire che tutte le immagini fossero ben integrate e che il layout risultasse armonioso. Ho anche selezionato i colori e il font da utilizzare per i vari elementi grafici, come menu, messaggi a schermo.

Una volta completata la prima iterazione del progetto, ho confrontato i risultati con i requisiti iniziali per verificare che tutti gli obiettivi fossero stati raggiunti. Eventuali funzionalità aggiuntive, come l'aggiunta di nuovi tipi di navi o strategie di attacco avanzate, sono state valutate e implementate per migliorare il gameplay e aumentare la varietà delle situazioni di gioco. In questo contesto, ho deciso di introdurre funzionalità avanzate come nuove modalità di gioco e potenziamenti (ad esempio, l'introduzione di strategie di attacco intelligenti per l'IA), bilanciando la difficoltà del gioco per rendere l'esperienza coinvolgente ma equa.

3.3 Feature avanzate del linguaggio Java

Nel corso dello sviluppo del progetto *BattleShipGame*, sono state utilizzate diverse feature avanzate del linguaggio Java e dell'ecosistema. Queste funzionalità hanno contribuito a migliorare l'efficienza e la manutenibilità del codice, oltre a semplificare la gestione delle complessità legate alla logica del gioco. Di seguito sono elencate le principali feature avanzate utilizzate, con riferimenti alle classi in cui tali caratteristiche sono state implementate.

3.3.1 Feature avanzate del linguaggio Java

- **Uso di espressioni lambda**: Le espressioni lambda sono state utilizzate per semplificare l'implementazione di funzioni e metodi che richiedono callback o listener. Nella classe `SettingsJFrame`, le lambda expressions sono impiegate per gestire gli eventi della GUI, migliorando la leggibilità

del codice e riducendo la necessità di classi anonime per gestire gli eventi dell'interfaccia utente.

3.3.2 Algoritmi sviluppati

- **Logica dell'intelligenza artificiale (IA) del computer**: Nella classe `ComputerPlayer`, è stato sviluppato un algoritmo che gestisce l'intelligenza artificiale del computer. Questo algoritmo decide in modo strategico o casuale dove sparare, basandosi su strategie differenti a seconda del livello di difficoltà impostato. L'IA può memorizzare i colpi precedenti e adattare il proprio comportamento in base agli esiti.
- **Gestione delle collisioni e affondamento**: Un algoritmo nella classe `BattleShipGameModel` gestisce le collisioni tra colpi e navi, determinando se una nave è stata colpita o affondata. Questo sistema verifica la correttezza del colpo basandosi sulle coordinate della griglia e aggiorna lo stato del gioco di conseguenza.