Battaglia Navale

Relazione di progetto
Programmazione e Modellazione ad Oggetti
Università degli Studi di Urbino Carlo Bo
Corso di Laurea in Informatica - Scienza e Tecnologia

Elisa Troiani 330031
Mohamedelhadi Bdeoui 330988

20 Febbraio 2026

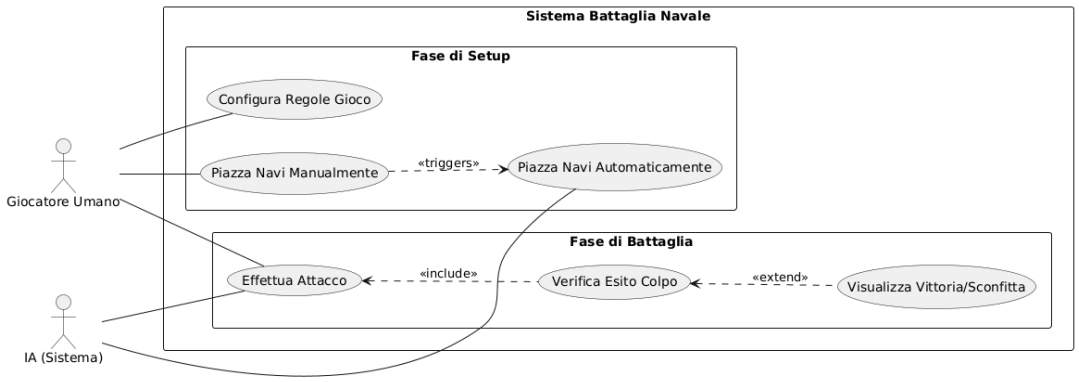
1. Analisi

L'obiettivo del progetto è la realizzazione di un sistema software che permetta a un utente di giocare a **Battaglia Navale** contro un avversario gestito dal computer.
Il sistema deve gestire le regole classiche del gioco, arricchite da vincoli di posizionamento realistici, e fornire diversi livelli di sfida basati sul livello dell'intelligenza artificiale.

1.1 Requisiti

I requisiti funzionali identificati per l'applicazione sono:

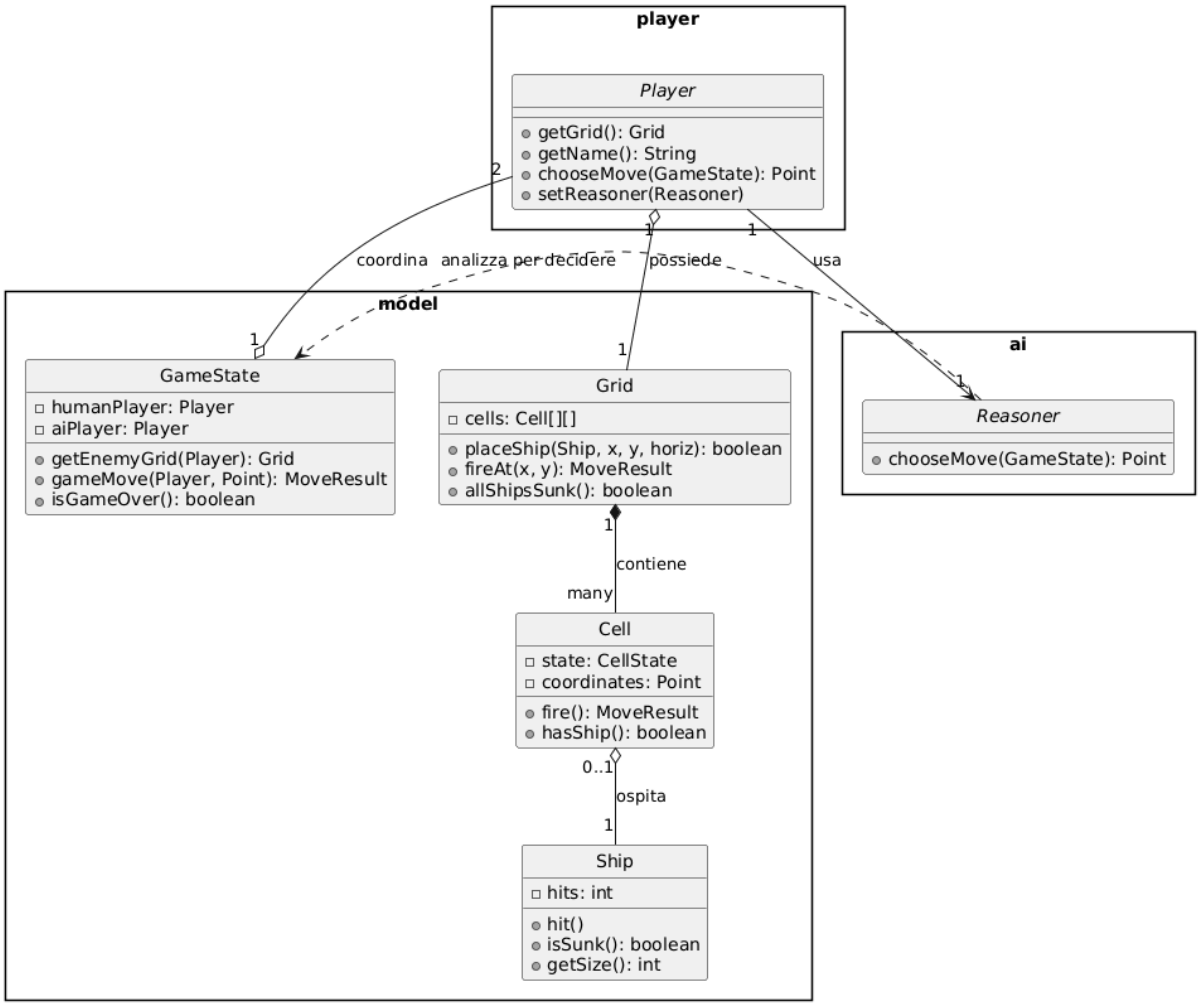
- **Configurazione del Gioco:** Il sistema deve permettere la definizione di una griglia di gioco e di una flotta composta da diverse tipologie di unità navali, ognuna con una propria dimensione.
- **Selezione Difficoltà:** L'utente deve poter scegliere tra diversi livelli di sfida prima dell'inizio della partita.
- **Posizionamento delle Navi:** Il sistema deve supportare il posizionamento delle navi sia per il giocatore umano che per l'intelligenza artificiale, garantendo il rispetto dei confini e della distanza minima tra le unità, impedendo il posizionamento di navi in celle già occupate, fuori dai bordi o in celle immediatamente adiacenti ad altre unità
- **Gestione del Turno:** Il software deve coordinare un sistema a turni alternati. Deve validare la mossa del giocatore, impedendo la ripetizione di colpi su celle già esplorate e permettendo l'esecuzione di un altro turno nel caso si sia colpita la cella di una nave avversaria, e successivamente processare la risposta dell'avversario.
- **Rilevamento Esito:** Per ogni interazione, il sistema deve restituire uno stato atomico: Acqua, Colpito, o Affondato.
- **Determinazione del Vincitore:** Il sistema deve monitorare costantemente lo stato di integrità di entrambe le flotte, dichiarando la terminazione del match nel momento in cui l'ultima sezione dell'ultima nave di uno schieramento viene distrutta.



1.2 Modello del dominio

Il dominio applicativo è composto dalle seguenti entità e relazioni:

- **Campo di Battaglia (Griglia):** Uno spazio bidimensionale composto da unità discrete chiamate "Celle".
- **Cella:** L'atomo del dominio. Ogni cella ha una coordinata univoca (x,y) e può trovarsi in uno stato di "occupazione" (presenza di una sezione navale) e uno stato di "esplorazione" (colpita o non colpita).
- **Nave:** Un'entità logica composta da un’insieme di celle correlate. Ha una dimensione che varia da 2 a 5 e un orientamento, che può essere orizzontale o verticale. Una nave è considerata affondata quando tutte le sue sezioni sono state colpite.
- **Regola di Distanza:** Un vincolo del dominio che impone che ogni nave sia circondata da almeno un'unità di spazio vuoto. Due navi non possono mai essere adiacenti, nemmeno diagonalmente.
- **Giocatore:** Un'entità capace di prendere decisioni tattiche sulla base dello stato della griglia avversaria. Può essere Umano o Artificiale.
- **Strategia di Tiro:** Il processo logico con cui un giocatore decide quale cella attaccare.



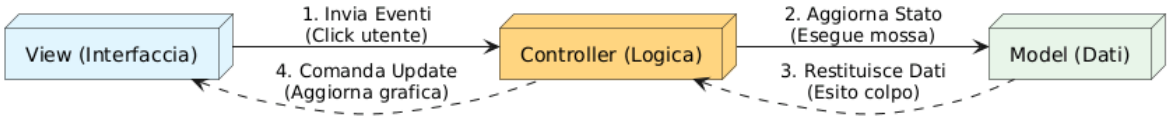
Schema UML dell’analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

2. Design

In questa fase viene delineata la struttura del software. L'obiettivo è creare un sistema in cui la logica di business (le regole della Battaglia Navale) sia completamente indipendente dall'interfaccia grafica e dalle diverse implementazioni delle strategie di gioco.

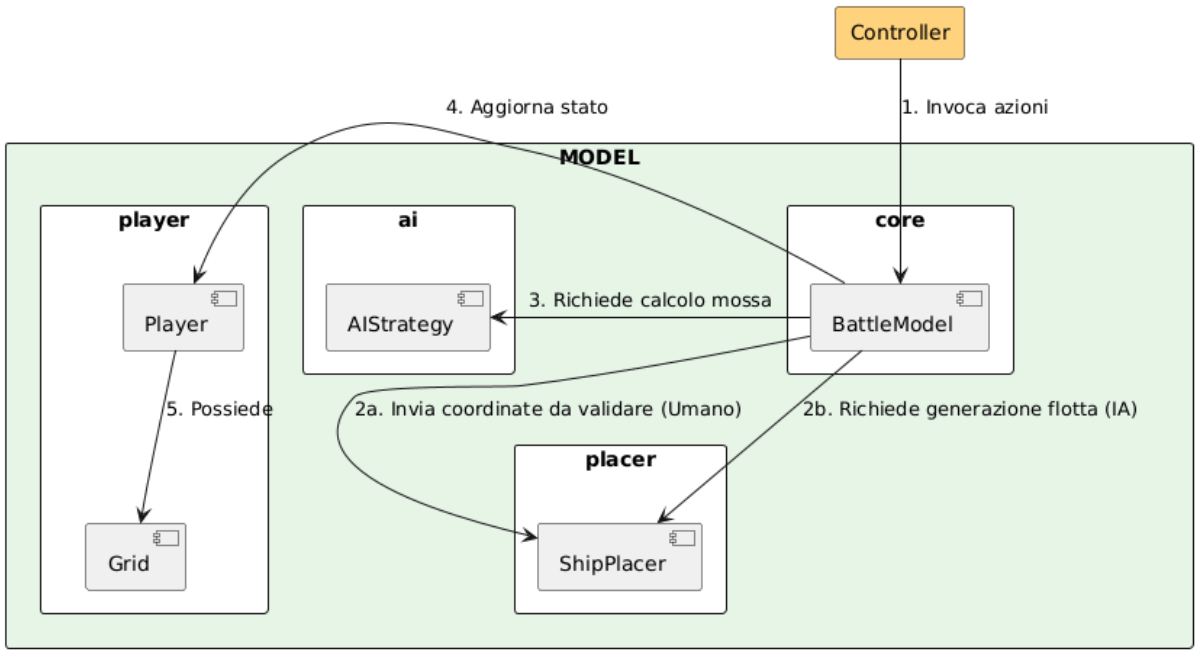
2.1 Architettura

L'architettura del sistema si basa sul pattern **Model-View-Controller (MVC)**. Questa scelta è fondamentale per garantire che modifiche alla rappresentazione grafica non influenzino la logica di gioco e viceversa.



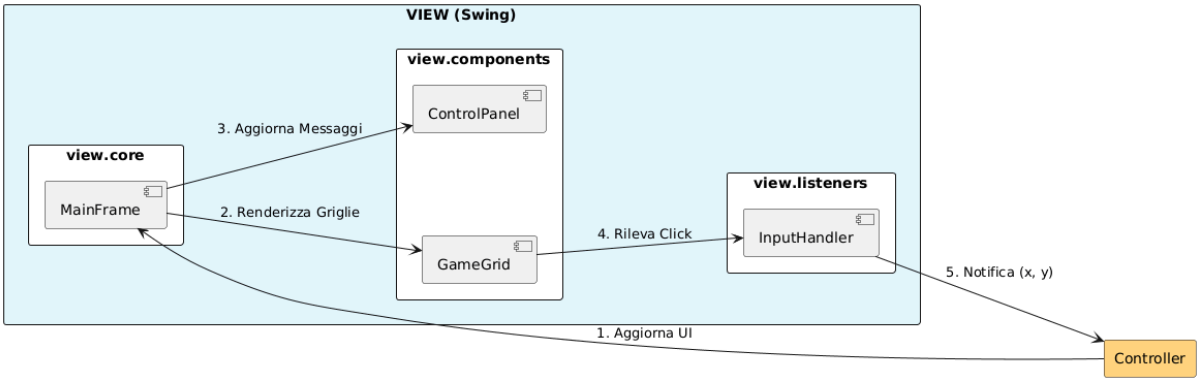
Model

Il Model rappresenta il nucleo logico e informativo dell'applicazione. Esso gestisce lo stato interno del gioco (posizionamento delle navi, griglie di gioco, turnazione) e ne garantisce la coerenza applicando le regole del dominio. Seguendo il principio dell'**incapsulamento**, il Model espone i dati solo attraverso interfacce controllate, impedendo manipolazioni dirette dall'esterno.



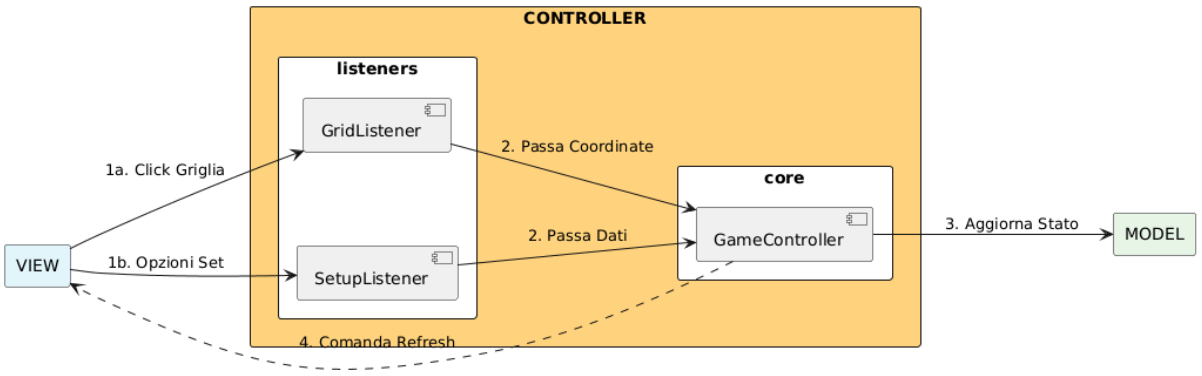
View

La View è il componente responsabile della presentazione visiva dei dati all'utente finale. Il suo compito primario è tradurre lo stato del Model in una rappresentazione grafica comprensibile e interattiva. Nel nostro progetto, la View è stata realizzata tramite la libreria **Java Swing**, non contiene logica decisionale né conserva dati di gioco, limitandosi a notificare le interazioni dell'utente (come il click su una coordinata). Questo approccio assicura che la View sia facilmente sostituibile senza impatti sulla stabilità funzionale del programma.



Controller

Il Controller funge da mediatore tra la View e il Model, gestendo il flusso dei dati e le reazioni agli eventi. Il suo ruolo consiste nell'osservare le interazioni dell'utente sulla View e tradurle in comandi per il Model. Abbiamo adottato una strategia basata su molteplici **Listener** specializzati: ogni elemento interattivo della View (pulsanti della griglia, selettori di difficoltà) invia pacchetti di informazioni o eventi al Controller. Quest'ultimo, dopo aver processato l'input e aggiornato il Model, coordina il refresh della View per riflettere i cambiamenti di stato. Questa separazione permette di gestire la complessità dell'input in modo modulare e ordinato.



2.2 Design Dettagliato

Elisa Troiani

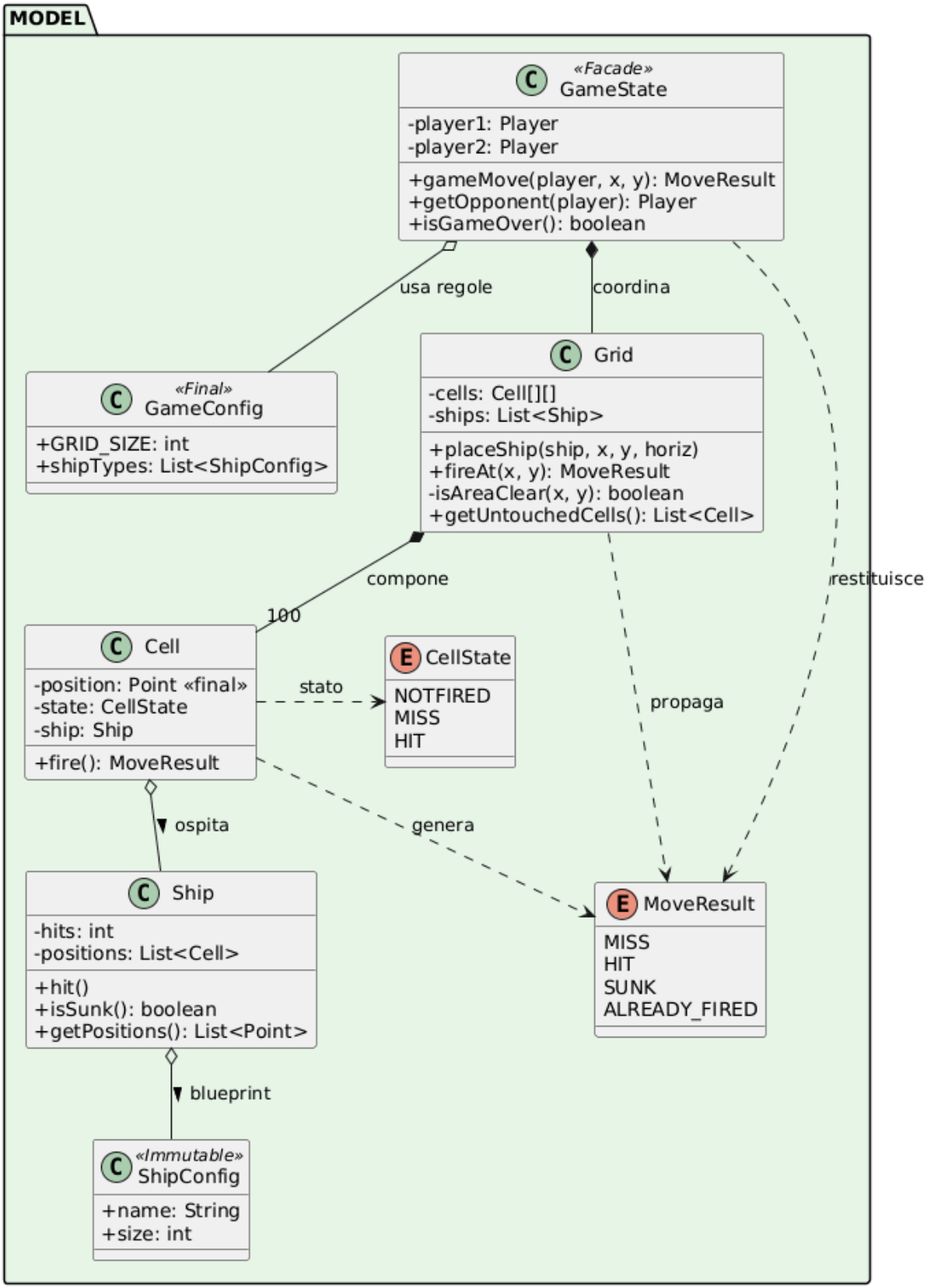
Model

Nella Battaglia Navale, il problema principale del Model non è solo memorizzare la posizione delle navi, ma garantire che le regole del gioco (non sovrapposizione, confini della griglia, alternanza dei turni) siano rispettate e che l'utente non possa "barare" accedendo a informazioni riservate come la posizione delle navi nemiche.

Architettura e Logica del Model

Il Model è stato progettato seguendo il principio della **delega delle responsabilità**: ogni componente, dalla singola cella al gestore della partita, possiede l'autonomia necessaria per gestire il proprio stato. Questo approccio evita classi "tuttofare" e garantisce che il Controller interagisca con un sistema robusto e modulare.

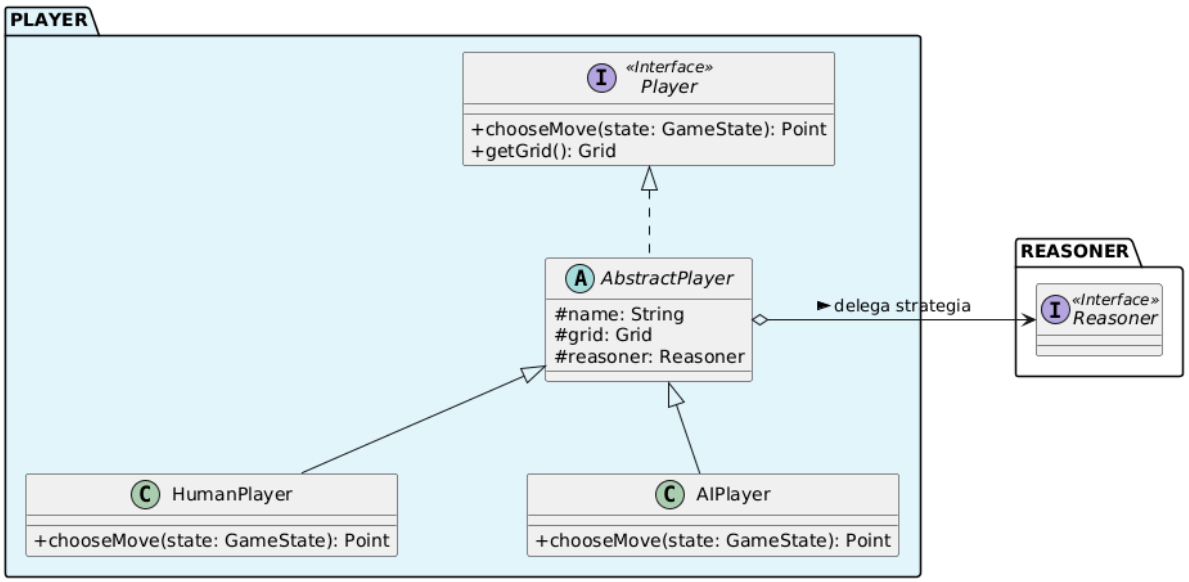
- **Entità Atomiche (Cell & Ship)** La classe `Cell` non è un semplice contenitore di dati, ma un'entità attiva. Grazie all'enum `CellState` e al metodo `fire()`, la cella gestisce autonomamente la transizione di stato e notifica l'eventuale nave ospitata, restituendo al Controller un `MoveResult` che disaccoppia la logica interna dall'esito pubblico. Parallelamente,ho distinto la definizione statica (`ShipConfig`, immutabile) dall'istanza dinamica (`Ship`), la quale sfrutta le **Stream API** di Java per manipolare le coordinate in modo dichiarativo e conciso.
- **L'Arbitro Spaziale (Grid)** La `Grid` incapsula la matrice di oggetti `Cell` e agisce come garante delle regole geometriche.
    - **Integrità:** Il piazzamento delle navi segue una validazione a due fasi (`canPlace` seguito da `place`) per garantire l'atomicità dell'operazione.
    - **Cuscinetto di Sicurezza:** L'algoritmo di posizionamento implementa una scansione perimetrale 3x3 (metodo `isAreaClear`), impedendo alle navi di toccarsi anche solo diagonalmente.
    - **Efficienza:** Metodi come `getUntouchedCells()` sostituiscono i cicli annidati con flussi di dati (Streams), migliorando leggibilità e performance.
- **Gestione della Partita (GameState & Config)** L'architettura si completa con il **Pattern Facade** implementato in `GameState`. Questa classe astrae la complessità del sistema (due giocatori, due griglie) offrendo al Controller un'interfaccia semplificata. La logica di gioco segue una catena di deleghe (*Controller → GameState → Grid → Cell*), mentre `GameConfig` centralizza le costanti di gioco, facilitando future modifiche al bilanciamento senza alterare il codice logico.


MODEL
«Facade»
GameState
-player1: Player
-player2: Player
+gameMove(player, x, y): MoveResult
+getOpponent(player): Player
+isGameOver(): boolean
usa regole
coordina
restituisce
«Final»
GameConfig
+GRID_SIZE: int
+shipTypes: List<ShipConfig>
Grid
-cells: Cell[][]
-ships: List<Ship>
+placeShip(ship, x, y, horiz)
+fireAt(x, y): MoveResult
-isAreaClear(x, y): boolean
+getUntouchedCells(): List<Cell>
compone
100
propaga
Cell
-position: Point «final»
-state: CellState
-ship: Ship
+fire(): MoveResult
stato
CellState
NOTFIRED
MISS
HIT
ospita
genera
Ship
-hits: int
-positions: List<Cell>
+hit()
+isSunk(): boolean
+getPositions(): List<Point>
MoveResult
MISS
HIT
SUNK
ALREADY_FIRED
blueprint
«Immutable»
ShipConfig
+name: String
+size: int

Player

Il package `player` è stato progettato per unificare la gestione di due entità profondamente diverse: l'utente umano (lento, asincrono, guidato dalla UI) e l'intelligenza artificiale (veloce, sincrona, guidata da algoritmi). L'obiettivo è stato disaccoppiare la logica del turno dalla natura dello sfidante.

- **Player e AbstractPlayer (Polimorfismo e DRY)** L'interfaccia `Player` definisce il contratto essenziale, incentrato sul metodo `chooseMove(GameState)`. Per evitare duplicazione di codice, ho introdotto la classe intermedia `AbstractPlayer`, applicando il principio **DRY (Don't Repeat Yourself)**. Questa classe centralizza la gestione dei dati comuni (come la griglia e il nome), dichiarandoli `protected` per consentire alle sottoclassi concrete di accedervi direttamente senza violare l'incapsulamento verso l'esterno.
- **AIPlayer (Pattern Strategy)** Questa classe è l'applicazione pratica del **Pattern Strategy**. `AIPlayer` non contiene logica decisionale, ma agisce come un contenitore che delega la scelta della mossa a un oggetto `Reasoner`.
    - **Flessibilità a Runtime:** Questa architettura permette di cambiare la difficoltà dell'IA (es. da "Random" a "Smart") a partita in corso, semplicemente iniettando un nuovo `Reasoner` senza dover ricreare l'oggetto giocatore.
    - **Separazione:** La complessità algoritmica è isolata nel `Reasoner`, mantenendo `AIPlayer` leggero e focalizzato sulla gestione dell'identità.
- **HumanPlayer (Gestione Asincrona)** Rappresenta l'utente reale e gestisce la natura asincrona dell'input. A differenza dell'IA, un umano non può "restituire" una mossa istantanea alla chiamata del metodo.
    - **Segnalazione al Controller:** L'implementazione di `chooseMove()` solleva una UnsupportedOperationException (o restituisce un segnale specifico). Questo è un meccanismo architetturale che notifica al Controller di interrompere il flusso logico automatico e mettersi in attesa attiva degli eventi della **GUI** (come i click del mouse), colmando il divario tra i tempi della CPU e quelli dell'utente.

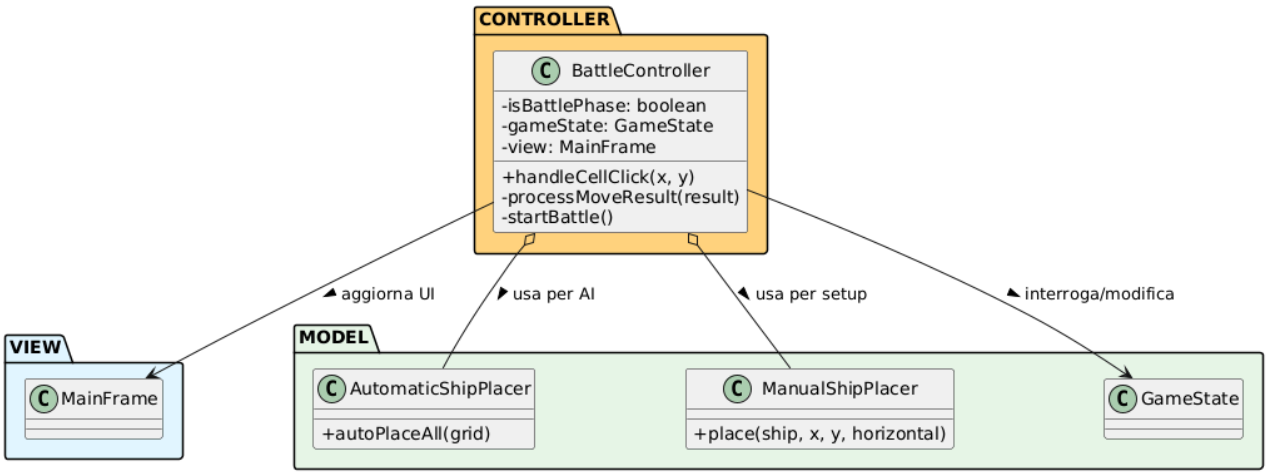


Controller

BattleController: Il Cuore Operativo

Il `BattleController` è l'elemento centrale che orchestra l'intero flusso dell'applicazione. Non si limita a smistare dati tra View e Model, ma agisce come un gestore di stato intelligente, garantendo che le regole del gioco siano applicate correttamente in ogni fase del match.

- **Gestione Dinamica delle Fasi**: Attraverso la variabile `isBattlePhase`, il controller distingue nettamente tra il setup e il combattimento.
    - **Fase di Setup**: Coordina il piazzamento manuale del giocatore tramite il `ManualShipPlacer` e automatizza quello dell'IA con l'`AutomaticShipPlacer`, garantendo l'atomicità della configurazione iniziale prima di sbloccare la battaglia.
- **Logica di Assistenza Tattica**: Nel metodo `processMoveResult()`, il controller non si limita ad aggiornare la grafica, ma implementa una funzione di "Smart UI". Quando una nave viene affondata, il controller identifica e disabilita automaticamente le celle del perimetro circostante. Sfruttando la regola del "cuscinetto di sicurezza" del modello, questa automazione riduce il carico cognitivo del giocatore, eliminando bersagli inutili e rendendo l'interfaccia più pulita e intuitiva.
- **Interfaccia MVC Attiva**: Il controller traduce i click sulla griglia in comandi logici per il `GameState`. La sua natura proattiva si manifesta nella capacità di filtrare l'input dell'utente in base allo stato del gioco, impedendo mosse non valide (come colpire celle già cliccate o piazzare navi sovrapposte) e fornendo feedback visivi immediati.



Mohamedelhadi Bdeoui

AI

Durante l'analisi si è deciso di dividere il Player dalla sua parte decisionale (`Reasoner`) in modo da **isolare lo stato dei dati dalla logica algoritmica**. Questa separazione segue il design pattern **Strategy**, permettendo al sistema di gestire diverse tipologie di intelligenza artificiale o input umano in modo intercambiabile.
Il `Reasoner` si occupa della fase analitica, ricevendo il **GameState** e processando la mossa migliore basandosi sulla cronologia dei colpi e sulle regole di gioco.

Difficoltà Facile (Easy Reasoner)

- **Algoritmo:** *Random Search*.
- **Logica:** L'IA sceglie le coordinate in modo completamente casuale tra le celle con stato `NOTFIRED`. Non ha "memoria" dei colpi andati a segno e non cerca di affondare una nave dopo averla colpita.

Difficoltà Media (Medium Reasoner)

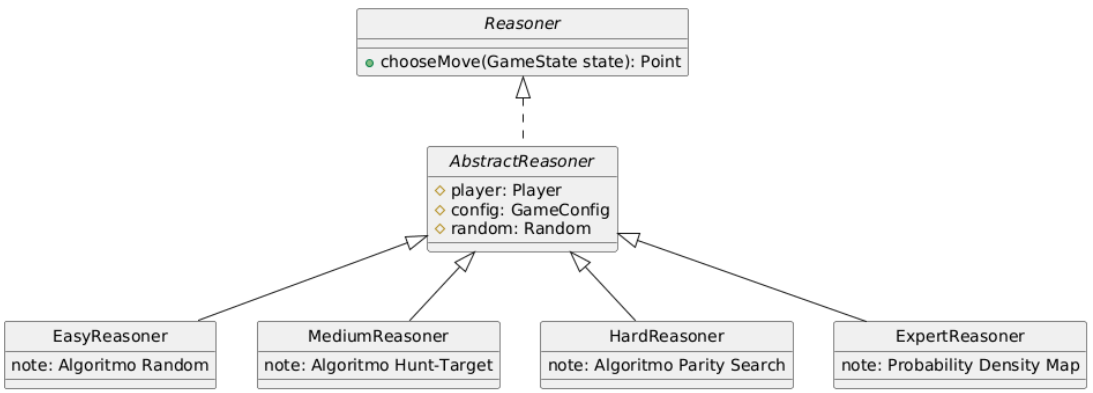
- **Algoritmo:** *Hunt-Target (Caccia e Bersaglio)*.
- **Logica:**
    - **Fase Hunt:** Spara casualmente (spesso seguendo un pattern a scacchiera per efficienza).
    - **Fase Target:** Quando colpisce una nave (`HIT`), entra in modalità bersaglio e attacca sistematicamente le 4 celle adiacenti (Nord, Sud, Est, Ovest) finché la nave non viene affondata.

Difficoltà Difficile (Hard Reasoner)

- **Algoritmo:** *Parity Search + Heuristic Targeting*.
- **Logica:** Ottimizza la caccia sparando "a scacchiera". Questo riduce del 50% i colpi necessari per trovare le navi.

Difficoltà Esperto (Expert Reasoner)

- **Algoritmo:** *Probability Density Map (PDM)*.
- **Logica:** Non spara a caso. Per ogni turno, simula migliaia di possibili posizionamenti per ogni nave nemica rimasta sulla griglia attuale. Le celle che compaiono più frequentemente in queste simulazioni ricevono un punteggio più alto. L'IA spara nella cella con la densità di probabilità massima.



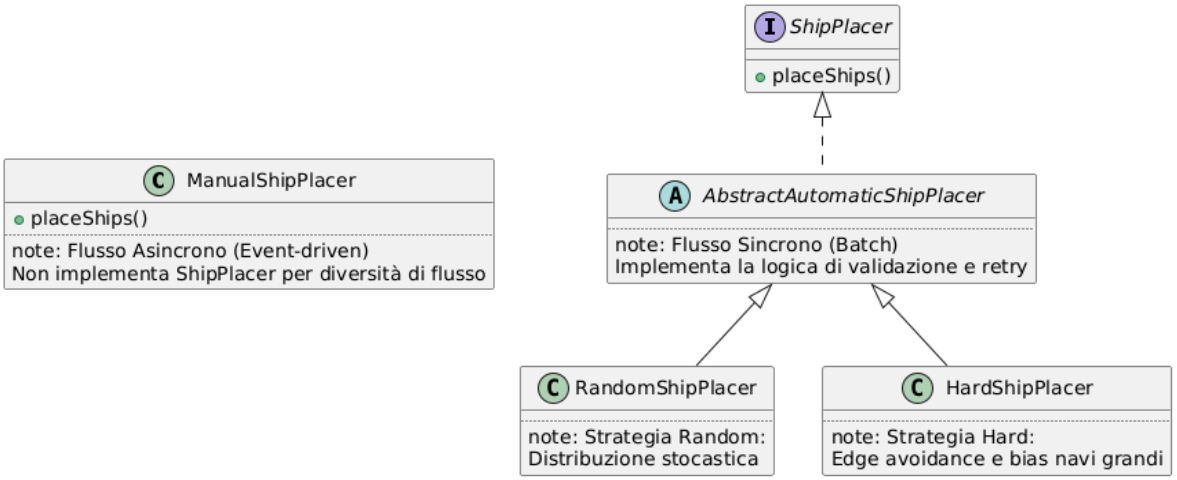
Placer

La scelta di separare la logica del **posizionamento delle navi** segue esattamente la stessa filosofia di design adottata per il `Reasoner`: il principio di **Inversione delle Dipendenze** e la creazione di componenti intercambiabili.

Placer manuale e automatico

Internamente, per un motivo funzionale e di gestione dei flussi, si è deciso di separare i posizionatori perché rappresentano due **stati logici** differenti del ciclo di vita dell'applicazione:

- **Manuale (Flusso Asincrono/Event-driven):**Il sistema rimane in uno stato di attesa (IDLE). Il flusso è guidato dall'utente: ogni clic è un evento isolato che il controller deve catturare, processare e validare.
- **Automatico (Flusso Sincrono/Batch):** È un processo atomico. Una volta avviato, viene eseguito interamente fino al completamento. Non c'è interazione esterna: l'algoritmo gira in un loop finché la griglia non è valida. All'interno di questa categoria si distinguono due strategie:
    - **Strategia Random:** si limita a una distribuzione puramente stocastica e uniforme sulla griglia, dove ogni coordinata e orientamento hanno la stessa probabilità di essere scelti per garantire una disposizione caotica e imprevedibile.
    - **Strategia Hard:** agisce con una logica difensiva evoluta, proteggendo le navi più grandi attraverso un orientamento forzato e nascondendo l'intera flotta dai pattern di ricerca tipici dei giocatori umani tramite l'esclusione sistematica dei bordi esterni del campo di gioco.

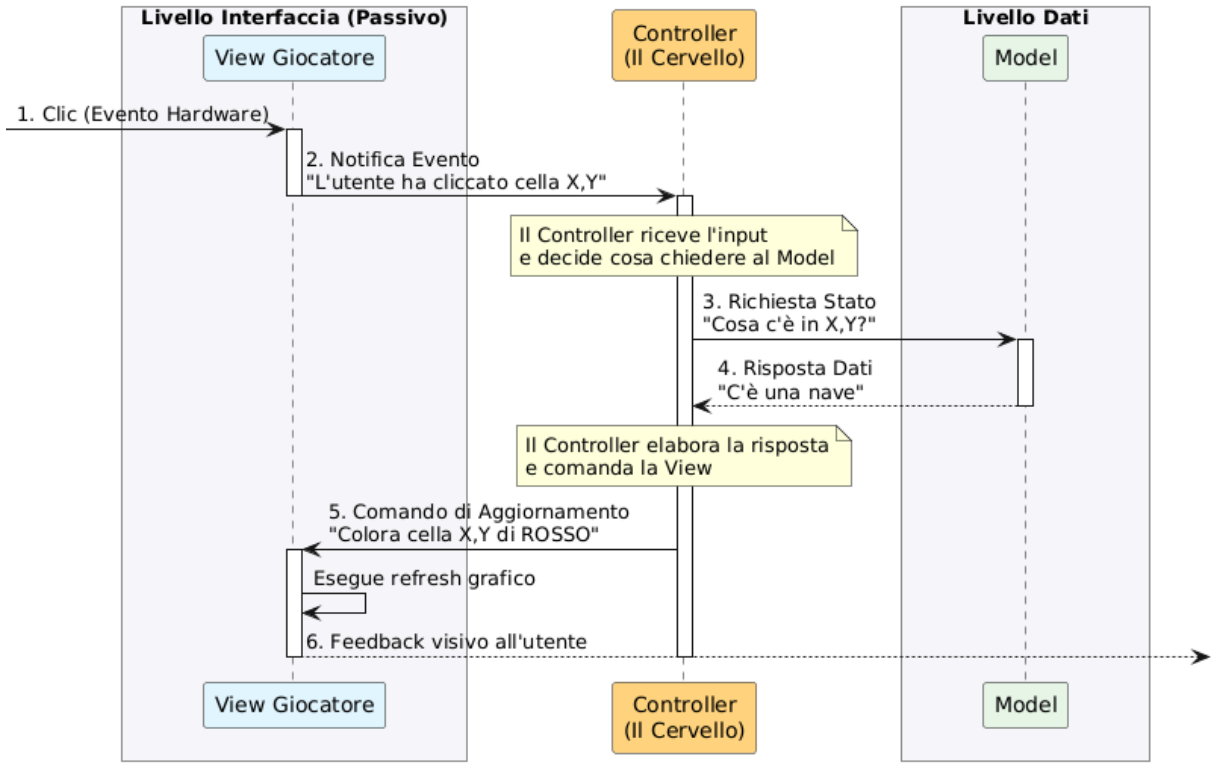


View

Si è optato per un'architettura **MVC** con **Passive View**. In questo modello, il **Controller** assume il ruolo di mediatore esclusivo: il **Modello** è sollevato dall'onere di notificare i cambiamenti e la **View** è privata di qualsiasi logica di interpretazione dei dati. Questo garantisce un **disaccoppiamento totale** tra i due, rendendo il sistema estremamente deterministico e facilitando il debugging, poiché ogni variazione grafica è l'esito diretto di un comando esplicito del Controller.

Nello specifico, la View è stata strutturata per assolvere alle seguenti funzioni fondamentali:

- **Rappresentazione del Modello:** Traduce le matrici logiche del gioco (griglie) in elementi grafici, aggiornandosi in modo sincrono ogni volta che il Controller impartisce un comando di refresh. Non limita la sua azione alla semplice visualizzazione, ma gestisce la **"Nebbia di Guerra"**, nascondendo o rivelando le informazioni in base alla prospettiva del giocatore corrente e allo stato della partita.
- **Punto di Ingresso Eventi:** Cattura le interazioni dell'utente (clic sulle celle, pressione tasti per rotazione) e le inoltra al Controller. In questa fase, la View funge da interfaccia per il flusso **Manuale**, trasformando le coordinate dei pixel della GUI in coordinate logiche (**x, y**) per la griglia, agendo come sensore passivo del sistema.
- **Separazione delle Responsabilità:** La View è totalmente agnostica rispetto alle regole (non sa se un colpo è legale o se una nave è affondata); si limita a riflettere i parametri ricevuti. Questa indipendenza permette di modificare l'intero aspetto estetico del gioco senza alterare la logica di battaglia o gli algoritmi di posizionamento.
- **Feedback Visivo:** Oltre alla griglia, la View gestisce componenti ausiliari come pannelli di messaggistica per le notifiche di gioco (es. *"Nave Colpita!"*) e la barra di stato della flotta, fornendo un riscontro immediato che guida l'esperienza dell'utente durante il passaggio tra le diverse fasi di gioco.



3. Sviluppo

L'intero ciclo di vita del software è stato improntato su un approccio iterativo e incrementale. La separazione dei compiti offerta dall'architettura MVC ha permesso di sviluppare e testare i singoli moduli (Model, View, Controller) in isolamento, garantendo una maggiore manutenibilità del codice.

3.1 Strategia di Testing Automatizzato

Il testing non è stato considerato un'attività finale, ma una parte integrante dello sviluppo.

In linea con le migliori pratiche dello sviluppo software, è stata adottata una politica di tolleranza zero verso i fallimenti dei test: ogni volta che un test non andava a buon fine, la priorità assoluta veniva assegnata all'analisi della causa radice e alla successiva risoluzione del problema. Questo approccio ha garantito che lo sviluppo non procedesse con "debiti tecnici" o bug irrisolti; l'aggiunta di nuove funzionalità veniva sospesa sistematicamente fino a quando non veniva ripristinata la completa integrità del codice, attestata dal superamento dell'intera suite di test.

Test del package Model

- `testShipPlacementSuccess;`
- `testBoundaryViolation;`
- `testProximityRuleViolation;`
- `testSinkingLogic;`
- `testFireAtSameCellTwice;`
- `testAllShipsSunk.`

Test del package Controller

- `testInitialStatus;`
- `testPlacementAndTransition.`

Test del package Placer

- `testRandomPlacementSuccess.`

Test del package Reasoner

- `testFirstMoveIsRandomAndValid;`

- testAvoidsAlreadyHitCells;
- testEndGameHunt.

3.2 Metodologia di Lavoro

L'approccio adottato dal gruppo ha mirato a coniugare l'autonomia operativa individuale con una costante sinergia collettiva. La struttura del lavoro è stata suddivisa in due macro-fasi ben distinte: una di progettazione corale e una di sviluppo indipendente.

La fase iniziale è stata dedicata interamente all'analisi e alla modellazione. In questa fase abbiamo lavorato in stretta collaborazione per definire l'architettura del sistema attraverso la stesura di diagrammi UML e il confronto continuo sulle scelte tecnologiche. Investire un tempo considerevole in questa attività preliminare si è rivelato determinante: la solidità della progettazione ha permesso una transizione fluida verso la scrittura del codice, riducendo al minimo imprevisti strutturali o necessità di refactoring radicali durante l'implementazione.

Lo sviluppo è stato supportato da strumenti moderni che hanno garantito la coerenza del codice e la gestione delle versioni:

- **Versionamento (Git):** L'uso di Git ha permesso una gestione granulare delle funzionalità;
- **Gestione delle Dipendenze:** L'integrazione di librerie esterne o framework (come **JUnit**) è stata gestita in modo centralizzato, assicurando che l'ambiente di sviluppo fosse facilmente replicabile;
- **Refactoring Costante:** Grazie al disaccoppiamento dei componenti, è stato possibile applicare tecniche di refactoring (come l'estrazione della logica del Reasoner dal Player) senza compromettere la stabilità complessiva del sistema.

Elisa Troiani

Il design è stato sviluppato partendo dalla componente atomica del sistema: la classe Cell, unità base sia per la Grid (campo da gioco) che per la Ship (entità logica della nave).
Per garantire una gestione robusta degli eventi di gioco, ho poi introdotto due elementi chiave: CellState (Enum) e MoveResult (Enum).
Per una questione di separazione delle responsabilità tra la definizione dei parametri e la logica di runtime, ho deciso di separare ShipConfig, ovvero la struttura base della nave, da Ship, classe che tiene traccia dello stato della nave.
La stessa cosa è stata fatta con GameConfig, che definisce le configurazioni del gioco, e GameState, che fornisce metodi per la gestione dei turni, delle vittorie o sconfitte e l'accesso alla griglia.

Sono passata poi alla realizzazione del GameController.

Per la realizzazione dei giocatori sono partita dall'interfaccia Player, seguendo poi con la classe Astratta AbstractPlayer e le sue classi figlie HumanPlayer e AIPlayer.

Mohamedelhadi Bdeoui

Per la gestione della CPU, ho deciso di puntare su un'architettura flessibile utilizzando **classi astratte** e **interfacce**. Nel pacchetto **placer**, la classe astratta funge da base comune: gestisce in automatico i controlli sui bordi della griglia e i tentativi di inserimento in caso di collisioni. Partendo da questo schema, ho implementato diverse strategie, da quella puramente casuale a quella più tattica.

Per quanto riguarda le strategie di attacco, l'uso di un'**interfaccia** dedicata mi ha permesso di rendere i vari livelli di difficoltà facilmente intercambiabili. Il sistema può così scalare da una ricerca ottimizzata a "scacchiera" a una modalità di puntamento che utilizza una Mappa di Densità di Probabilità.

Per quanto riguarda l'interfaccia grafica, ho utilizzato le librerie **Java Swing e AWT** organizzando la visualizzazione secondo una struttura gerarchica di pannelli e pulsanti personalizzati. Grazie all'uso dei **Layout Manager**, come **BorderLayout** e **GridLayout**, sono riuscito a ottenere una plancia di gioco ordinata e responsive, capace di adattarsi visivamente alle diverse fasi del match, dal setup iniziale alla battaglia finale.

3.3 Note di Sviluppo

Elisa Troiani

Nello sviluppo dei moduli model, controller e player, sono state utilizzate le seguenti feature del linguaggio Java:

- **Lambda Expressions**: utilizzate ampiamente all'interno del BattleController per la gestione degli eventi della GUI (ActionListener) e per la definizione delle azioni di callback (es. `exitAction`), permettendo una sintassi più concisa rispetto alle *anonymous inner classes*.
- **Stream API**: impiegate in Ship e in `Grid`, per operazioni di aggregazione e filtraggio, come il controllo dello stato di affondamento di tutte le navi.
- **Optional**: utilizzato come tipo di ritorno per i metodi di ricerca (es. recupero di una nave in una cella), garantendo una gestione sicura dei valori nulli e forzando il chiamante a gestire esplicitamente l'eventuale assenza dell'oggetto.

Ho sviluppato con il metodo `canPlaceShipAt` un algoritmo di collisione spaziale all'interno della classe `Grid`. Prima di confermare l'inserimento di una `Ship`, l'algoritmo effettua una doppia verifica:

- Calcola l'ingombro della nave in base all'orientamento (orizzontale/verticale) e verifica che non ecceda le dimensioni `width` e `height` definite in `GameConfig`.
- Oltre a controllare se le celle sono libere, scansiona il perimetro circostante per assicurarsi che venga rispettata la distanza minima di una cella tra le navi, prevenendo il posizionamento adiacente.

All'interno del `BattleController`, in sinergia con il `model` e la `view`, ho implementato un algoritmo di scansione che si attiva al momento dell'affondamento di una nave.
Secondo le regole, le navi non possono essere adiacenti. Quando una nave viene affondata, infatti, le celle che la circondano diventano automaticamente zone in cui è impossibile trovare altre navi. L'algoritmo implementato esegue i seguenti passaggi:

1. Identifica tutte le coordinate occupate dalla nave appena affondata.
2. Per ogni coordinata, calcola l'intorno di Moore (le 8 celle adiacenti).
3. Verifica che tali celle siano all'interno dei confini della griglia e che non siano già state colpite.
4. Marca queste celle come "non bersagliabili" (invalidazione del target).

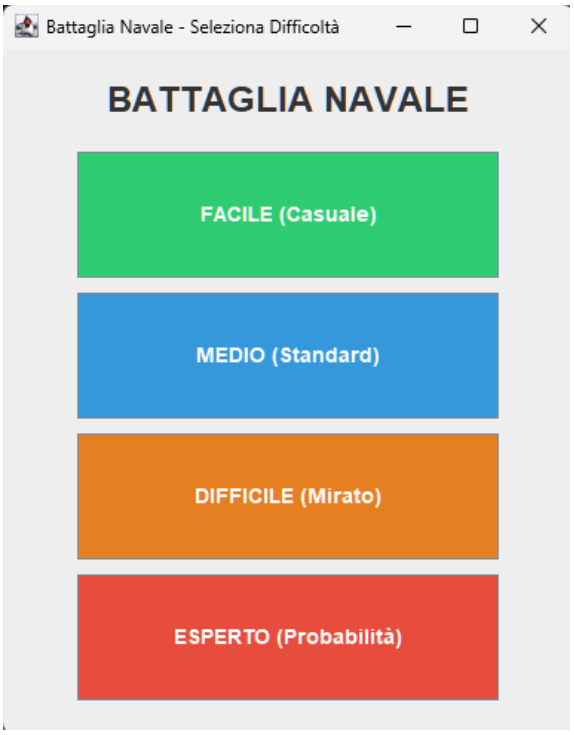
Mohamedelhadi Bdeoui

Nello sviluppo dei moduli **reasoner**, **placer** e **view** sono state utilizzate le seguenti feature del linguaggio Java:

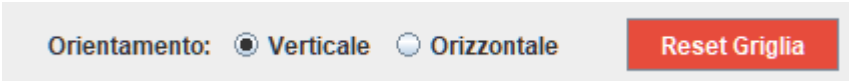
- **Lambda Expressions**: utilizzate per mantenere il codice leggibile, snello e moderno.
- **Switch Expressions**: adottate per gestire flussi decisionali a più rami in modo compatto. Questa feature evita lunghe concatenazioni di **if-else** o blocchi **switch** tradizionali, riducendo la complessità visiva e il rischio di errori logici nel controllo delle direzioni.
- **Stream API**: impiegate come strumento principale per la manipolazione dei dati. Consentono di processare collezioni di oggetti in modo dichiarativo, eliminando cicli iterativi pesanti e variabili d'appoggio superflue. Il risultato è un codice intuitivo che utilizza metodi simili al linguaggio naturale. Decisamente il mio strumento preferito.
- **Java Swing & AWT:** queste librerie sono state utilizzate per costruire l'intera interfaccia grafica, organizzando i componenti in una gerarchia di **JFrame**, **JPanel** e **JButton**.
    - **Layout Managers:** l'uso di **BorderLayout** e **GridLayout** è stato fondamentale per mantenere le griglie di gioco perfettamente allineate e responsive, evitando il posizionamento manuale delle coordinate dei tasti.
    - **Personalizzazione Estetica:** tramite le classi **Color**, **Font** e **ImageIcon**, ho trasformato i componenti standard in elementi tematici (es. il blu per il mare, icone per le navi), migliorando l'esperienza dell'utente.
    - **Gestione Eventi:** il sistema sfrutta gli **ActionListener** di **AWT** per catturare i click del giocatore, permettendo al controller di reagire in tempo reale alle mosse sulla plancia.

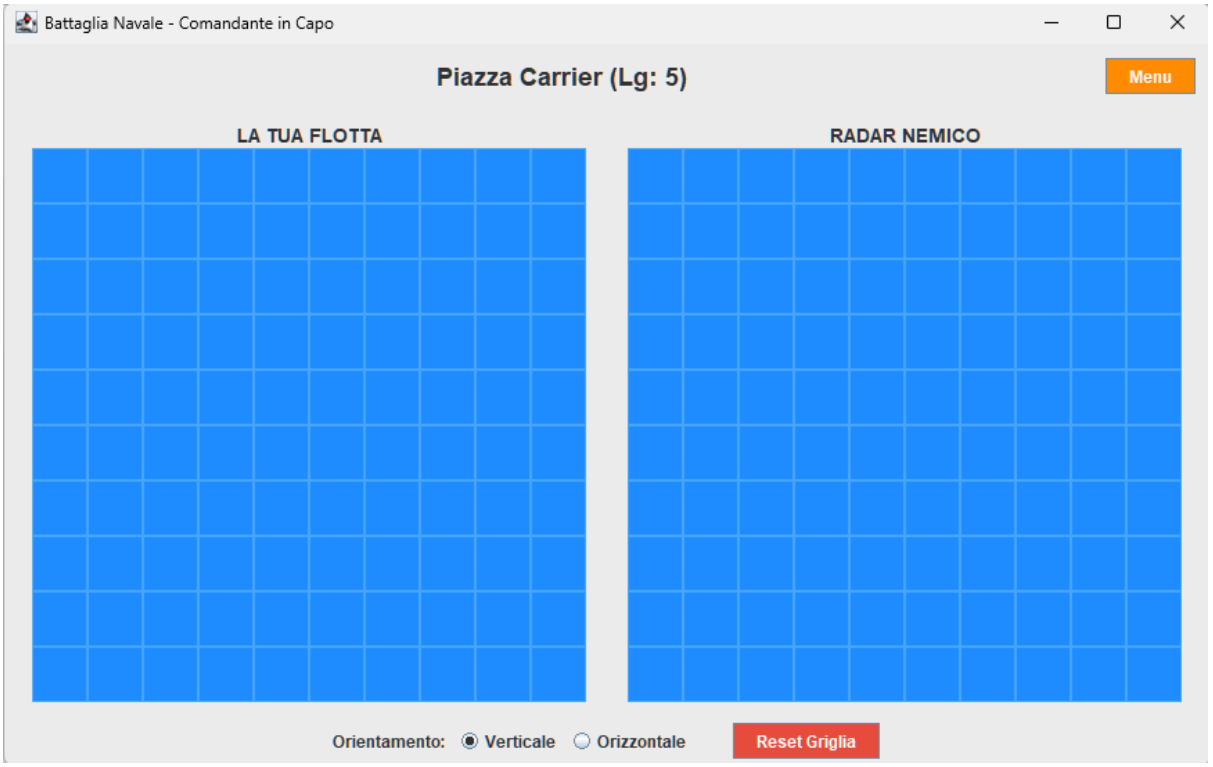
4. Guida all’utilizzo

All’avvio dell’applicazione verrà chiesto all’utente quale difficoltà affrontare:



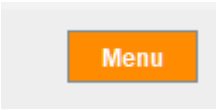
Dopo la scelta bisognerà posizionare le navi a partire dalla più grande scegliendo la direzione con i bottoni appositi nel pannello inferiore in cui, se non si è ancora finito il posizionamento, è possibile premere un bottone per resettare il posizionamento.

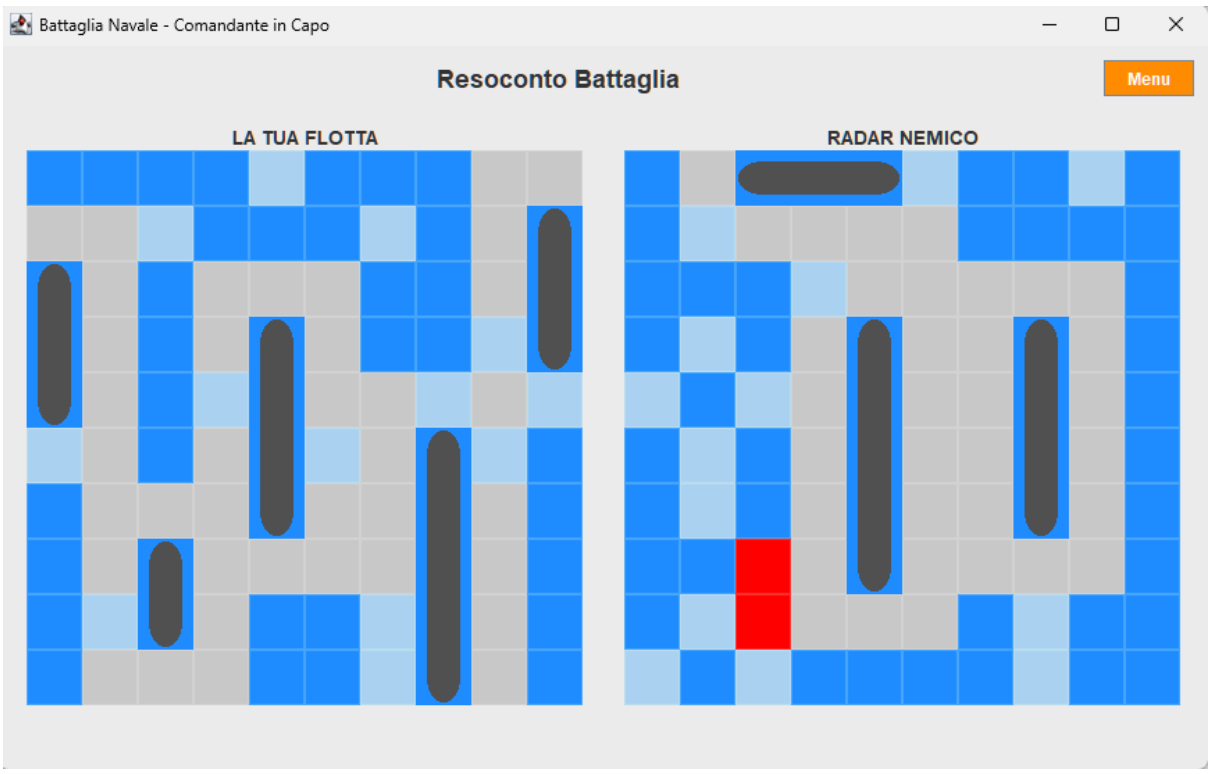




Finita la fase di setup spariranno i tasti di setup e si potrà iniziare a giocare a battaglia navale.

Per ricominciare durante o dopo aver completato la partita basterà premere il tasto menu in alto a destra.





5. Commenti Finali

Elisa Troiani
Il progetto ha portato alla realizzazione di un'applicazione solida e funzionale, rispettando tutti i requisiti prefissati in fase di analisi. L'architettura adottata (MCV), ha dimostrato la sua efficacia specialmente durante la fase di integrazione. Grazie all'uso dei Design Pattern, il software non è solo un prodotto finito, ma una base estensibile: l'attuale struttura permetterebbe, ad esempio, di aggiungere una modalità multiplayer online o nuove tipologie di armi con modifiche minime al codice esistente.

Mohamedelhadi Bdeoui
Anche se non perfetto esteticamente, sono molto soddisfatto del risultato finale, soprattutto per la struttura articolata in package specializzati, che trovo facilmente espandibili e sostituibili. Inizialmente ero titubante all'idea di creare 'troppi package', ma alla fine ho trovato molto comodo questo approccio. Lo sviluppo mi ha portato a discutere varie soluzioni, scoprendo nuovi pattern e strategie che hanno migliorato la mia adattabilità alle richieste del progetto, invece di basarmi solo su preferenze personali dettate dalla 'coerenza'.