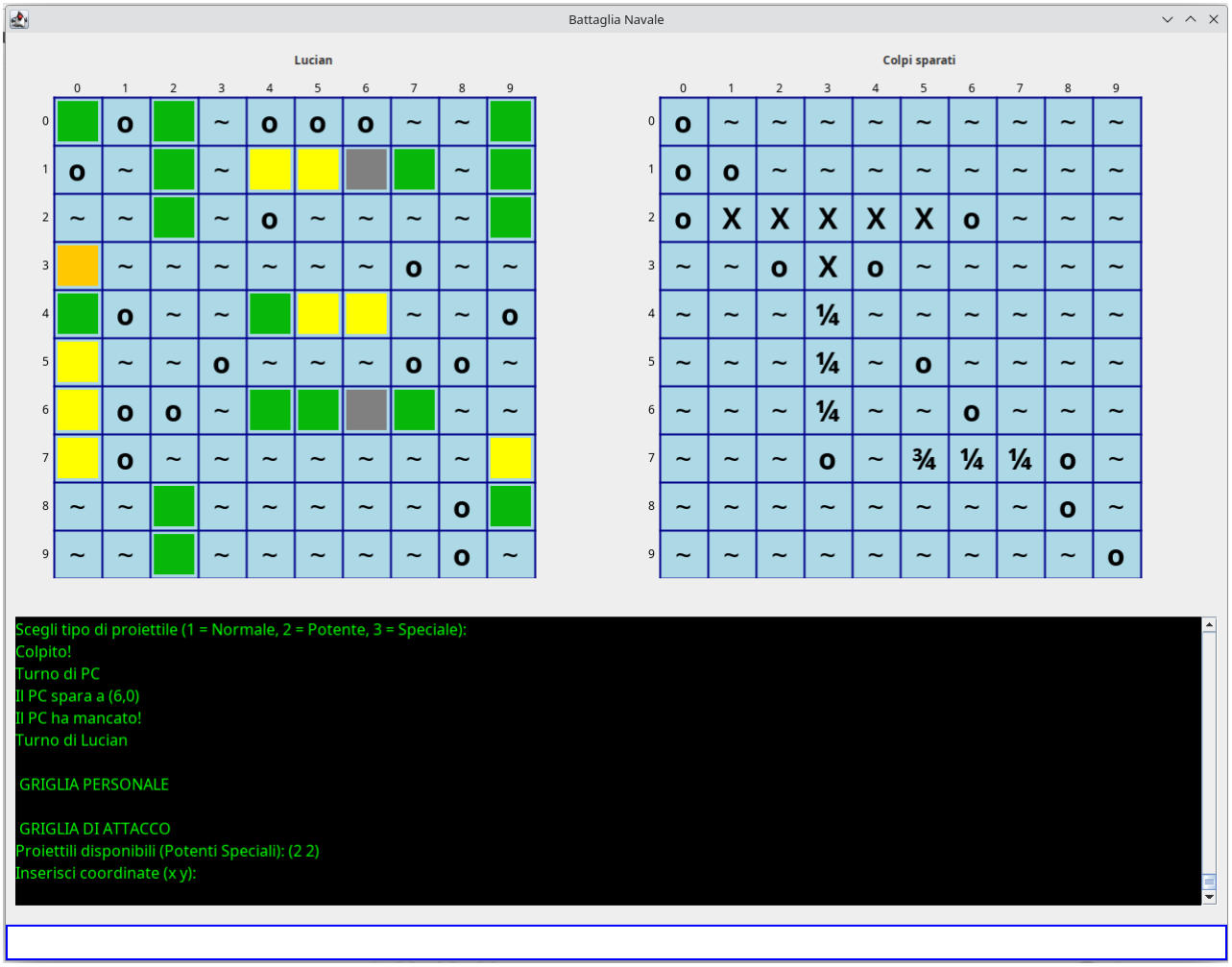
Progetto “Battleship AI”

Relazione per il progetto del Corso di PMO

Papadopol Lucian-Ioan

22 marzo 2025



Indice

1 ANALISI

1.1 REQUISITI

L'obiettivo dell'applicazione è creare un videogioco che implementi il gioco Battaglia Navale.

Il giocatore deve avere come avversario un'intelligenza artificiale basilare pseudo-AI; il gioco deve rispettare delle regole specifiche che si discostano leggermente da quelle classiche garantendo anche un'esperienza chiara e coinvolgente per l'utente.

1.1.1 Requisiti Funzionali

L'applicazione dovrà consentire ai giocatori di:

- Avviare una nuova partita.
- Selezionare il livello di difficoltà dell'IA.
- Posizionare le proprie navi su una griglia, rispettando le regole del gioco:
  - Le navi non devono sovrapporsi.
  - Devono essere posizionate interamente dentro la griglia.
  - Possono avere orientamento orizzontale o verticale.
- Effettuare attacchi sulla griglia avversaria scegliendo le coordinate del colpo.
- Utilizzare diverse tipologie di proiettili, con effetti differenti sul gioco.
- Visualizzare lo stato della griglia, distinguendo:
  - Le proprie navi e gli attacchi subiti.
  - Gli attacchi effettuati contro l'avversario.
- Gestire la fine della partita, determinando il vincitore quando tutte le navi di un giocatore sono affondate.
- Mostrare messaggi informativi e di gioco, per guidare il giocatore durante la partita.

1.1.2 Requisiti Non Funzionali

Oltre alle funzionalità principali, l'applicazione dovrà:

- Essere accessibile e di facile uso per l'utente.
- Garantire tempi di risposta rapidi per una fluida esperienza di gioco.
- Fornire un'interfaccia chiara e leggibile, con feedback sugli eventi di gioco.
- Offrire un'esperienza equa e bilanciata, assicurando che l'IA non sia né troppo prevedibile né imbattibile.
- Essere estensibile, permettendo future modifiche o aggiunte di nuove modalità di gioco.
- Non richiedere connessione a Internet, essendo pensata per il gioco in locale.

1.1.3 Diagramma dei casi d'uso

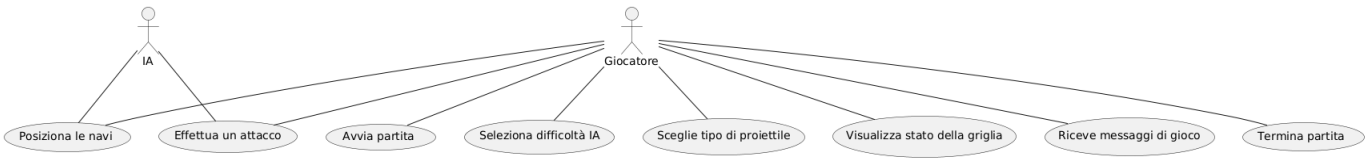


Figura 1: Diagramma dei casi d'uso

Il diagramma dei casi d'uso rappresenta le principali interazioni tra il giocatore e il sistema di gioco Battaglia Navale.

- **Attori Coinvolti**
  - *Giocatore.* L'utente umano che interagisce con l'applicazione.
  - *IA.* L'intelligenza artificiale che funge da avversario automatico.
- **Casi d'Uso Principali**
  - *Avvia partita.* Il giocatore inizia una nuova partita.
  - *Seleziona difficoltà IA.* Il giocatore sceglie il livello di sfida dell'intelligenza artificiale.
  - *Posiziona le navi.* Il giocatore e l'IA dispongono le proprie navi sulla griglia, rispettando le regole di posizionamento.
  - *Effettua un attacco.* Durante il proprio turno, il giocatore o l'IA scelgono una casella sulla griglia avversaria per tentare di colpire una nave nemica.
  - *Sceglie tipo di proiettile.* Il giocatore può decidere quale proiettile utilizzare tra le opzioni disponibili.
  - *Visualizza stato della griglia.* Il giocatore può controllare la propria griglia e quella d'attacco per analizzare l'andamento della partita.
  - *Riceve messaggi di gioco.* Il sistema fornisce notifiche sullo stato della partita (es. colpi andati a segno, navi affondate).
  - *Termina partita.* Il gioco rileva automaticamente il vincitore quando tutte le navi di un giocatore sono affondate.
- **Relazioni tra Attori e Casi d'Uso**
  - Il Giocatore interagisce con tutte le funzionalità del sistema, poiché prende decisioni attive durante la partita.
  - L'IA è responsabile solo di due azioni:
    1. Posizionare le proprie navi in modo automatico.
    2. Effettuare attacchi contro il giocatore, secondo una logica variabile in base al livello di difficoltà selezionato.

1.2 MODELLO DEL DOMINIO

1.2.1 Descrizione del dominio

L'applicazione Battaglia Navale si svolge in un campo di gioco diviso in due griglie:

1. *La griglia personale*, dove il giocatore posiziona le proprie navi e vengono indicati i colpi ricevuti oltre al livello di danno subito per ogni nave.
2. *La griglia d'attacco*, dove vengono registrati i colpi sparati contro l'avversario.

Le navi vengono posizionate all'inizio del gioco e devono rispettare regole specifiche: non sovrapporsi, essere posizionate interamente nella griglia.

Durante il gioco, il giocatore e l'IA si alternano per effettuare attacchi, cercando di affondare le navi avversarie. Un colpo può colpire una nave o finire in acqua.

Ogni nave è composta da caselle della griglia, e ogni casella può avere diversi stati:

- Vuota e non colpita (acqua).
- Vuota, colpita ma senza danno (acqua).
- Occupata da una nave e non colpita.
- Occupata da una nave, colpita e con associato danno progressivo.
- Affondata, se tutte le caselle di una nave sono colpite.

Esistono diverse tipologie di proiettili, che possono infliggere più o meno danni a una casella.

Il gioco termina quando tutte le navi di un giocatore sono affondate.

1.2.2 Identificazione delle Entità

Analizzando il problema, emergono le seguenti entità principali:

- **Giocatore**. Partecipa alla partita e possiede una griglia.
- **IA**. Gioca contro il giocatore con una strategia variabile.
- **Griglia**. Campo di gioco su cui vengono posizionate le navi e registrati i colpi.
- **Casella**. Unità base della griglia, può contenere o meno una parte di una nave.
- **Nave**. Composta da più caselle, può essere colpita e affondata.
- **Proiettile**. Può avere effetti diversi sulla griglia (normale, potente, speciale).
- **Partita**. Gestisce il flusso di gioco e le condizioni di vittoria.

1.2.3 Relazioni tra le entità

1. Un giocatore *possiede* una griglia su cui posiziona le proprie navi.
2. Una griglia è *composta* da caselle che possono essere vuote o occupate da una nave.
3. Una nave è *composta* da più caselle e può essere affondata.
4. Un giocatore o l'IA possono sparare un proiettile, che colpisce una casella della griglia avversaria.
5. La partita coordina i turni e determina la vittoria.

Entità	Descrizione	Relazione con altre entità
Partita	Rappresenta l'intero flusso del gioco	• Controlla le griglie dei giocatori • Gestisce i turni tra il giocatore e l'IA
Griglia	Campo di gioco del giocatore	È composta da più caselle
Casella	Singola unità della griglia	Può contenere una nave
Nave	Unità da affondare	È composta da più caselle
Proiettile	Oggetto che infligge danno in un attacco	Colpisce una casella

Figura 2: Riepilogo entità principali e loro relazioni

1.2.4 Schema UML del dominio

Questo schema è una rappresentazione concettuale che descrive le entità principali del dominio applicativo e le loro relazioni.

Serve a comprendere e modellare il problema prima di iniziare l'implementazione del software.

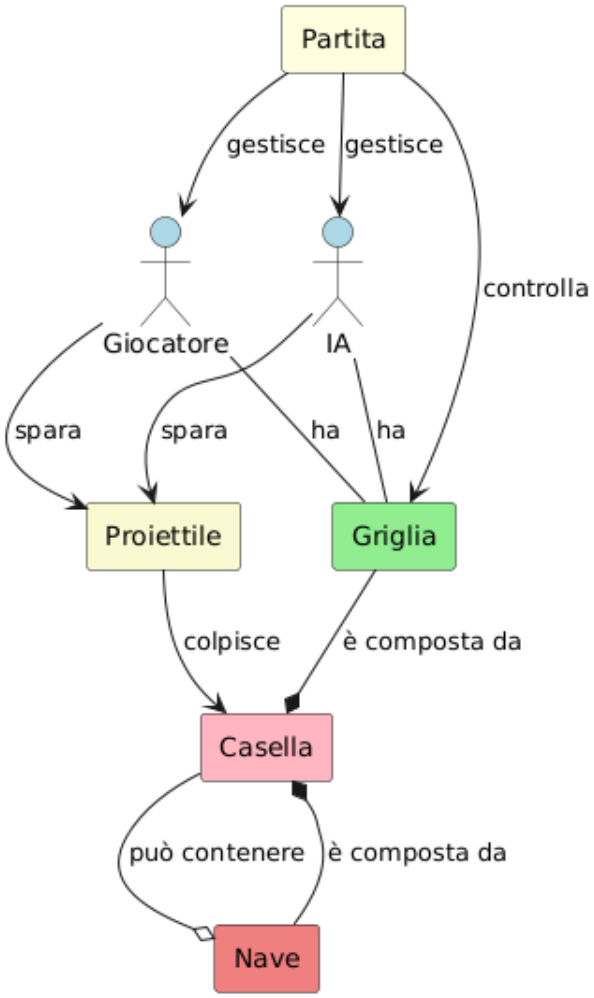


Figura 3: Schema ULM del dominio

2 DESIGN

2.1 ARCHITETTURA

L'architettura dell'applicazione Battleship-AI è progettata seguendo il pattern Model-View-Controller (MVC), un design pattern che permette di separare la logica di business (Model), la gestione dell'interfaccia utente (View) e il controllo del flusso dell'applicazione (Controller).

L'uso di MVC permette di mantenere il codice modulare, facilitando la manutenzione, l'estensibilità e la possibilità di sostituire le interfacce utente (passaggio tra GUI e TUI) senza modificare il modello o il controller.

2.1.1 Panoramica dell'architettura MVC

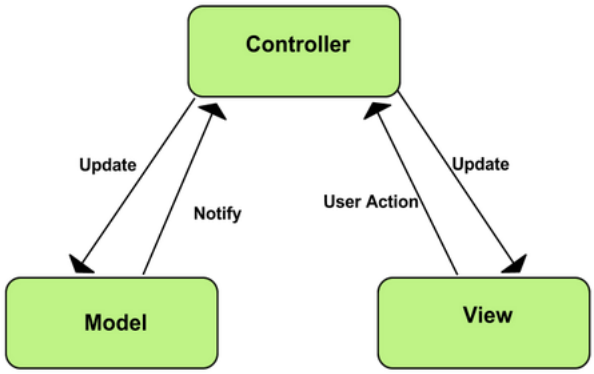


Figura 4: MVC design pattern

L'architettura si suddivide in tre componenti principali:

Model

- Definisce la logica di gioco e le regole.
- Contiene le classi che gestiscono griglie, navi, caselle, proiettili e logica di battaglia.
- Non conosce né interagisce direttamente con la View.

View

- Si occupa della rappresentazione dell'interfaccia utente.
- Può essere implementata in modalità grafica (GUI) o testuale (TUI).
- Riceve aggiornamenti dallo stato del Model e li mostra all'utente.

Controller

- Gestisce il flusso di gioco e l'interazione tra Model e View.
- Contiene la logica di gestione dei turni, attacchi, scelta dei proiettili e gestione dell'IA.
- Non implementa logica di gioco, ma chiama il Model per eseguire azioni e aggiorna la View di conseguenza.

2.1.2 Ruoli e Interazione tra i Componenti

Model
Ruoli principali:

- Gestisce la logica di gioco, comprese le regole e lo stato della partita.
- Contiene e aggiorna i dati relativi alla griglia, alle navi e ai colpi sparati.
- Fornisce informazioni sullo stato del gioco (es. chi ha vinto, quali navi sono affondate).

Interazione con gli altri componenti:

- Riceve comandi dal Controller per aggiornare lo stato della partita (es. eseguire un turno, applicare danni a una nave).
- Restituisce informazioni al Controller, come il risultato di un attacco o lo stato del gioco.

View
Ruoli principali:

- Mostra all'utente lo stato della partita (interfaccia testuale o grafica).
- Riceve input dal giocatore e lo inoltra al Controller.
- Aggiorna la visualizzazione in base agli eventi di gioco.

Interazione con gli altri componenti:

- Riceve dati dal Controller per aggiornare l'interfaccia utente.
- Invia input utente al Controller, che poi decide le azioni da eseguire.

Controller
Ruoli principali:

- Coordina il flusso di gioco, decidendo quando e come aggiornare il Model o la View.
- Gestisce la logica dell'intelligenza artificiale.
- Garantisce la separazione tra Model e View, agendo come mediatore.

Interazione con gli altri componenti:

- Riceve input dalla View e lo traduce in comandi per il Model.
- Interroga il Model per ottenere lo stato della partita.
- Aggiorna la View in base ai cambiamenti avvenuti nel Model.

2.1.3 Identificazione delle Classi Chiave

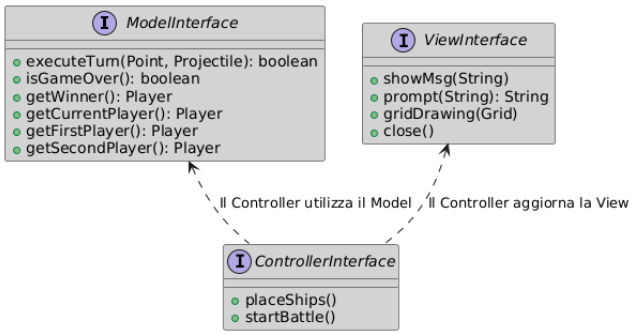


Figura 5: Interfacce del MVC

Model
Punto di ingresso: *ModelInterface.java* Definisce l'insieme di operazioni eseguibili sulla logica di gioco e permette al Controller di interagire con il Model senza dipendere dall'implementazione concreta.
Classi che implementano il Model:

- *Battle.java* Contiene la logica della partita e implementa ModelInterface.

View
Punto di ingresso: *ViewInterface.java* Definisce un'interfaccia comune per la gestione dell'interfaccia utente. Consente di gestire sia l'interfaccia testuale (TUI) che quella grafica (GUI) senza modificare il Controller.
Classi che implementano la View:

- *Gui.java* Implementa ViewInterface, fornendo un'interfaccia grafica.
- *Tui.java* Implementa ViewInterface, fornendo un'interfaccia testuale.

Controller
Punto di ingresso: *ControllerInterface.java* Definisce i metodi fondamentali per la gestione della partita e il posizionamento delle navi. Separa la logica del gioco dalla gestione dell'interfaccia.
Classi che implementano il Controller:

- *GameController.java* Implementa ControllerInterface, gestisce l'interazione tra Model e View.

2.2 DESIGN DETTAGLIATO

2.2.1 Gestione flessibile dei proiettili con il design pattern Factory Method

Il problema
Nel gioco della battaglia navale, ogni turno prevede la scelta e l'uso di un proiettile da parte del giocatore o dell'IA.
Sono disponibili più tipi di proiettili (standard, potente, speciale), ognuno con caratteristiche differenti.
Il problema nasce nel momento in cui la logica di creazione dei proiettili viene ripetuta o dispersa nel codice, portando a duplicazioni, mancanza di coerenza e difficoltà di manutenzione.

La soluzione
Per risolvere questo problema si è adottato il **Factory Method**, che consente di incapsulare la logica di creazione dei proiettili all'interno di una classe dedicata *ProjectileHandler*, restituendo un oggetto polimorfico di tipo Projectile.

Alternative considerate

- Creazione diretta nei controller
  - Pro: semplice da scrivere
  - Contro: ripetitivo, difficile da manutenere, rende difficile introdurre nuovi tipi di proiettile

Implementazione
Classi coinvolte

- *Projectile.java* classe astratta che rappresenta un generico proiettile.
- *StandardProjectile.java*, *PowerProjectile.java*, *SpecialProjectile.java* sottoclassi concrete di Projectile.
- *ProjectileHandler.java* classe che contiene il metodo factory *makeProjectile(int type, Player player).*

Relazioni

- Il controller, sia umano che IA, chiama ProjectileHandler.makeProjectile() per creare un proiettile.

- ProjectileHandler crea il tipo di proiettile richiesto e restituisce l'oggetto di tipo Projectile corrispondente;
  il metodo può adattarsi alla disponibilità del giocatore, ad esempio se ha ancora proiettili speciali e decidere dinamicamente il tipo di proiettile da creare.
- Il chiamante non conosce la classe concreta: utilizza l'oggetto restituito in modo polimorfico.

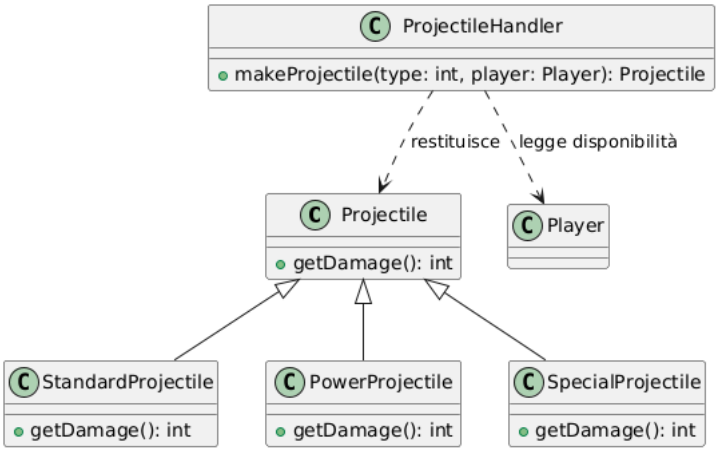


Figura 6: Implementazione del design pattern Factory

2.2.2 Gestione flessibile delle strategie di attacco dell'IA con il design pattern Strategy

Il problema
Nel gioco della Battaglia Navale, il comportamento dell'IA durante il proprio turno varia in base al livello di difficoltà scelto dal giocatore (facile, medio, difficile).
Inizialmente, tutta la logica delle diverse modalità era inserita direttamente nella classe PcTurnHandler, tramite una struttura switch-case che selezionava tra i metodi playEasy(), playMedium() e playHard().
Questo approccio presentava alcuni problemi:

- Il codice diventava difficile da leggere e mantenere, a causa della crescita della complessità interna.
- Aggiungere nuove strategie richiedeva modificare la classe PcTurnHandler.
- Il comportamento dell'IA era rigidamente accoppiato a una singola classe.

La soluzione
Per risolvere questi problemi è stato adottato il Strategy Pattern, un pattern comportamentale che permette di incapsulare algoritmi alternativi (strategie) all'interno di classi distinte, intercambiabili a runtime.

Struttura della soluzione

- È stata introdotta un'interfaccia AITurnStrategy, che rappresenta una generica strategia di attacco dell'IA.
- Ogni strategia concreta (EasyStrategy, MediumStrategy, HardStrategy) implementa questa interfaccia e incapsula il proprio algoritmo.
- La classe PcTurnHandler seleziona la strategia in fase di costruzione, in base al livello di difficoltà scelto, e la usa senza conoscerne i dettagli.

Alternative considerate

- Switch-case centralizzato in PcTurnHandler:
  - Pro: semplice da implementare

– Contro: codice poco leggibile, difficile da estendere

Implementazione
Classi coinvolte

- *AITurnStrategy.java* interfaccia che definisce il metodo executeTurn(Player currentPlayer).
- *EasyStrategy.java*, *MediumStrategy.java*, *HardStrategy.java* → classi concrete che implementano AITurnStrategy, ognuna con una strategia di attacco diversa.
- *PcTurnHandler.java* seleziona la strategia in base alla difficoltà e la utilizza per eseguire il turno del PC.

Relazioni

- PcTurnHandler dipende dall'interfaccia AITurnStrategy, non dalle implementazioni specifiche.
- Ogni strategia può utilizzare ModelInterface, ViewInterface e ProjectileHandler per eseguire il comportamento specifico.

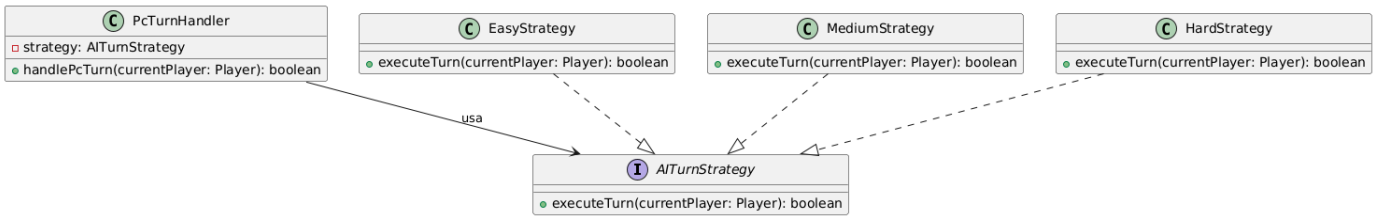


Figura 7: Design pattern Strategy

2.2.3 Gestione del livello di difficoltà dell'intelligenza artificiale

Il problema
Nel contesto del gioco della Battaglia Navale, il giocatore umano si confronta con un avversario gestito dall'intelligenza artificiale (IA).
Tuttavia, un'IA che gioca sempre nello stesso modo ad esempio tirando a caso, rischia di risultare troppo prevedibile e noiosa per un utente esperto, oppure troppo difficile se gioca sempre in modo ottimizzato.
Il problema risiede quindi nella necessità di offrire un'esperienza di gioco equilibrata e personalizzabile, capace di adattarsi al livello del giocatore.
In particolare, si volevano soddisfare i seguenti requisiti:

- Offrire una sfida più accessibile ai principianti.
- Consentire una partita più stimolante per giocatori esperti.
- Evitare che l'IA si comporti in modo completamente casuale o totalmente invincibile in tutte le situazioni.

La soluzione
Per risolvere il problema si è deciso di introdurre tre livelli di difficoltà per l'IA, ognuno con un comportamento di attacco differente:

- Livello 1 facile: l'IA sceglie casualmente una cella da colpire.
- Livello 2 medio: l'IA utilizza una strategia "hunt and target": se colpisce una nave, cerca di colpire le celle adiacenti.
- Livello 3 difficile: l'IA usa un approccio pseudo-intelligente, analizzando la griglia per stimare la probabilità che una cella contenga una nave, e colpendo di conseguenza.

Alternative considerate

- IA unica con comportamento fisso:
  - Pro: implementazione più semplice.
  - Contro: esperienza di gioco statica, poco scalabile, adatta solo a un certo tipo di giocatore.

Considerazioni sulla scelta
La soluzione scelta ha il vantaggio di:

- Rendere il gioco accessibile a tutti, dai principianti agli utenti esperti.
- Migliorare la longevità del gioco, permettendo di ripetere le partite con livelli di sfida differenti.
- Facilitare la manutenzione e l'espandibilità, grazie all'uso del pattern Strategy che incapsula ogni comportamento in una classe separata.

Implementazione

Modalità Facile – Tiro completamente casuale

- L'IA sceglie casualmente una cella della griglia del giocatore, senza tenere conto di colpi precedenti o della disposizione delle navi.
- Anche la scelta del tipo di proiettile è casuale.
- Non viene applicata alcuna logica predittiva o di ricerca.
- Obiettivo: simulare un comportamento semplice, simile a un principiante, rendendo la sfida molto accessibile.

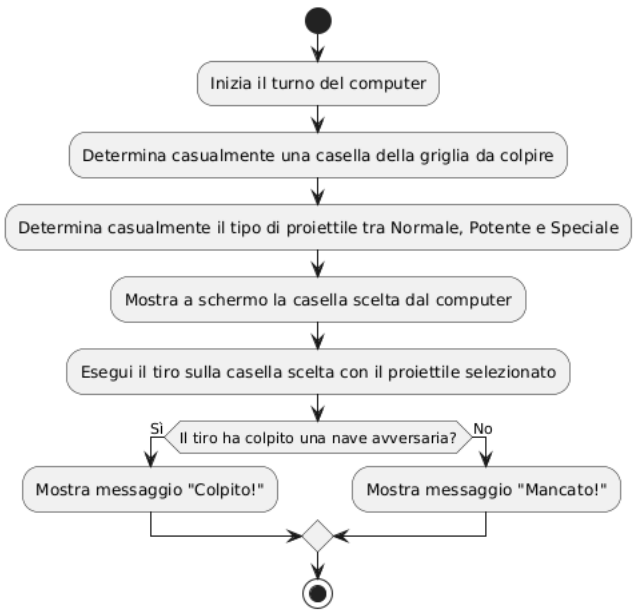


Figura 8: Flowchart algoritmo di attacco casuale - livello semplice

Modalità Media – "Hunt and Target"

- L'IA effettua un tiro casuale finché non colpisce una nave (modalità "hunt").
- Una volta ottenuto un colpo a segno, memorizza la posizione e inizia a colpire le celle adiacenti (modalità "target"), nel tentativo di affondare la nave.

- Usa anche un proiettile più potente se individua una nave non ancora completamente danneggiata.
- Obiettivo: simulare un giocatore più attento, in grado di ragionare in modo elementare sulla disposizione delle navi.

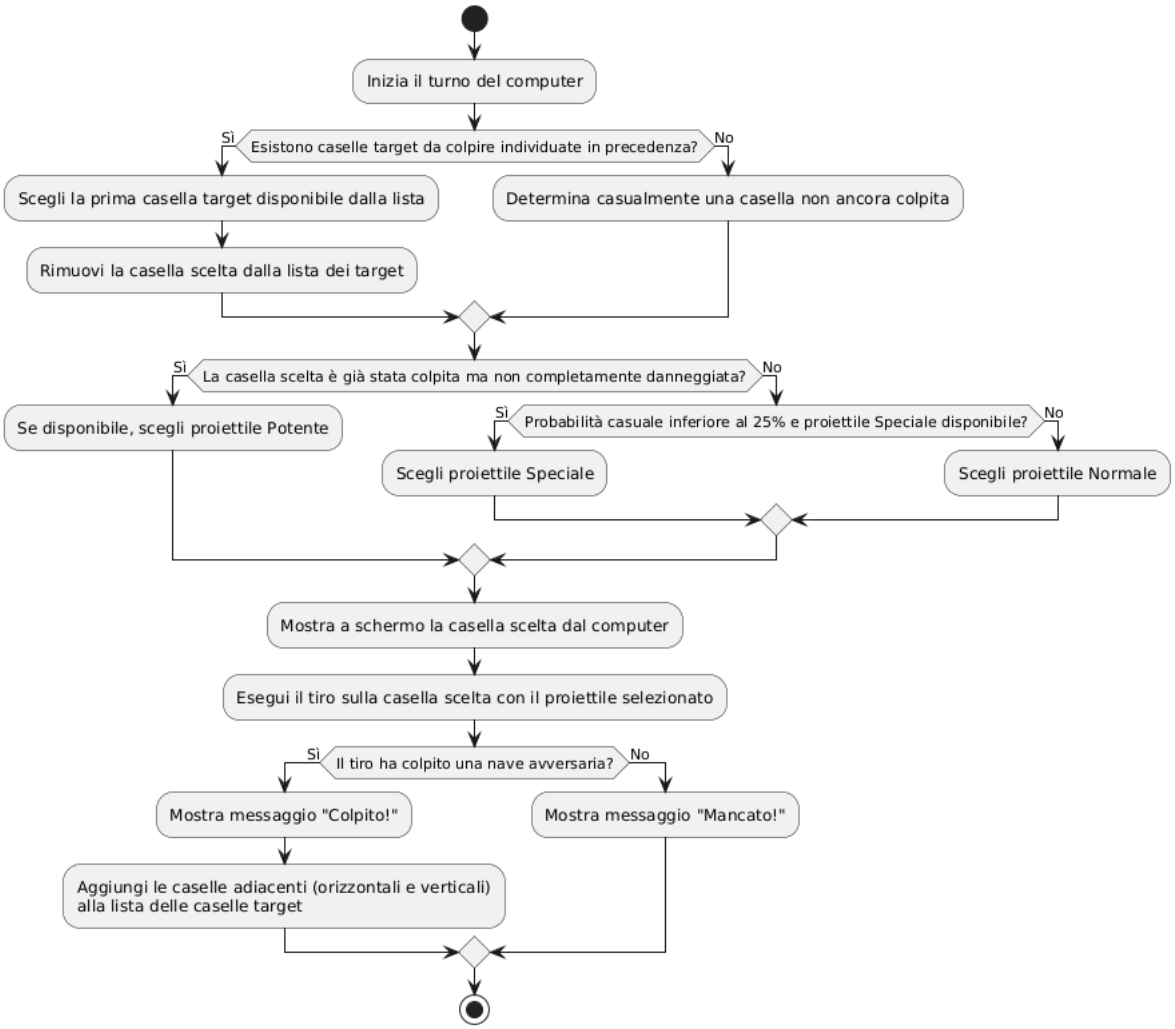


Figura 9: Flowchart algoritmo di attacco ottimizzato - livello medio

Modalità Difficile – Pseudo-AI con valutazione probabilistica

- L'IA analizza l'intera griglia nemica e assegna un punteggio a ciascuna cella in base alla probabilità che contenga parte di una nave.
- Se trova celle già colpite ma non affondate, passa in modalità "target" intelligente e colpisce le celle più promettenti attorno al bersaglio.
- La scelta del tipo di proiettile è ottimizzata in base alla probabilità stimata di successo.
- L'approccio adottato trae ispirazione dai primi sistemi esperti, che spesso combinavano regole decisionali con valutazioni probabilistiche per gestire l'incertezza e migliorare l'efficacia delle decisioni.
- Obiettivo: simulare un comportamento "intelligente", che anticipa le mosse del giocatore e aumenta significativamente la difficoltà.

Figura 10: Flowchart algoritmo di attacco pseudo-IA - livello difficile

3 SVILUPPO

3.1 TESTING AUTOMATIZZATO

Il testing automatizzato è stato implementato nel package **test**.
Ho scelto di testare "il core" della logica di gioco ovvero l'insieme delle entità e delle regole cui il buon funzionamento impatta maggiormente sulla funzionalità complessiva dell'applicazione.

Vi sono quattro test JUnit:

1. **GridSquareTest.java** Testa il comportamento della classe *GridSquare.java*, che rappresenta una singola casella della griglia.
   a) *gridSquareCreationTest()* Verifica che una casella venga creata correttamente con coordinate e valori di default.
   b) *customGridSquareCreationTest()* Testa la creazione di una casella con valori personalizzati.
   c) *validDamageTest()* Verifica che una casella possa ricevere danno e memorizzarlo correttamente.
   d) *invalidDamageTest()* Testa il comportamento quando si cerca di impostare un livello di danno negativo o oltre il limite massimo.
   e) *isOccupiedTest()* Controlla che si possa impostare una casella come occupata e che il valore venga aggiornato correttamente.
   f) *isHitTest()* Controlla che si possa impostare una casella come colpita e che il valore venga aggiornato correttamente.
2. **GridTest.java** Testa il comportamento della classe *Grid.java*, che gestisce l'intera griglia del gioco.
   a) *validShipPositioningTest()* Verifica che una nave possa essere posizionata correttamente in una zona libera della griglia.
   b) *outOfGridPositioningTest()* Tenta di posizionare una nave fuori dai limiti della griglia e verifica che venga lanciata un eccezione.
   c) *placeOverOccupiedTest()* Verifica che non sia possibile sovrapporre due navi e che quindi venga lanciata una eccezione.
   d) *everythingIsSunkTest()* Controlla che la griglia abbia tutte le navi affondate.
3. **ShipTest.java** Testa il comportamento della classe *Ship.java*, che rappresenta una nave composta da più caselle.
   a) *partialDamageTest()* Crea una nave composta da tre caselle, alcune con danno parziale e controlla che la nave non venga considerata affondata.
   b) *isSunkTest()* Crea una nave con tutte le caselle completamente danneggiate e controlla che il metodo isSunk() restituisca true.
   c) *fullDamageSunkTest()* Simula un attacco progressivo:
      i. Colpisce la nave più volte fino a farla affondare completamente.
      ii. Dopo ogni colpo, controlla che isSunk() restituisca false fino al colpo finale.
      iii. Dopo il colpo finale, isSunk() deve restituire true.
4. **BattleTest.java** Testa il comportamento della classe *Battle.java*, che gestisce l'intera partita.
   a) *notEndingTurnTest()* Simula un turno in cui il giocatore manca il bersaglio (colpo a vuoto). Controlla che:
      i. Il turno passi all'avversario.
      ii. Il gioco non sia terminato (isGameOver() deve restituire false).
   b) *endingTurnTest()* Simula un colpo che affonda l'ultima nave dell'avversario. Controlla che:

i. Il gioco venga dichiarato terminato (isGameOver() restituisce true).

ii. Il vincitore sia il giocatore che ha colpito (getWinner() restituisce il vincitore corretto).

c) *executeTurnAfterGameOverTest()* Simula il caso in cui un giocatore tenta di sparare dopo la fine della partita e verifica che venga lanciata una eccezione.

3.2 METODOLOGIA DI LAVORO

Lo sviluppo del progetto è stato svolto interamente in autonomia, senza attività di gruppo. La scelta di lavorare da solo ha richiesto un'organizzazione personale efficace e una pianificazione attenta, al fine di gestire in modo equilibrato il tempo dedicato all'analisi, alla progettazione, allo sviluppo e alla fase di test e rifinitura del codice.
Ho adottato una mia personale metodologia di lavoro ispirata a pratiche agili, modellata sulle esigenze del mio tempo disponibile. In particolare, ho preferito suddividere l'attività settimanale in due momenti distinti:

- Durante la settimana, quando avevo a disposizione blocchi di tempo più brevi (in media un'ora), mi sono concentrato sullo sviluppo e test di piccole unità funzionali (classi, metodi, strategie specifiche), lavorando in modo incrementale e isolato. Questa fase è stata utile per sperimentare, identificare errori in modo tempestivo e mantenere costante il contatto con il progetto.
- Durante il fine settimana, ho dedicato sessioni più estese al refactoring, all'integrazione dei moduli sviluppati nei giorni precedenti e alla pulizia del codice, consolidando la coerenza architetturale ad esempio applicando pattern come MVC, Strategy, Factory.

Nel corso del progetto, ho seguito un processo iterativo così strutturato:

- Analisi del problema e dei requisiti – individuazione delle funzionalità attese e delle entità del dominio.
- Progettazione modulare e architetturale – separazione chiara dei ruoli, uso di interfacce per disaccoppiamento, riflessioni sull'applicazione di design pattern.
- Sviluppo incrementale – realizzazione dei package:
  - model.* per la logica di gioco e le entità;
  - view.* per la GUI/TUI e i componenti di output;
  - controller.* per la gestione del flusso di gioco;
- Testing e validazione – uso di test JUnit per verificare le funzionalità "core" fondamentali.
- Refactoring e documentazione – pulizia del codice, miglioramento della leggibilità e commenti, stesura della relazione.

3.3 NOTE DI SVILUPPO

Feature avanzate del linguaggio che sono state utilizzate

- *Uso di lambda expressions*
  Utilizzate in modo puntuale, ad esempio nella scrittura di test unitari per la verifica di eccezioni nel test *GridSquareTest.java*.

```
Exception exception = assertThrows(IllegalArgumentException.class, () -> { ... });
```

- *Uso di Stream API combinate con lambda expressions*
  Impiego stream() in *Ship.java* su tutte le caselle che compongono una nave per verificarne l'affondamento completo.

```
return gridSquares.stream()
        .allMatch(gridSquare -> gridSquare.getDamageLevel()
                                >= gridSquare.getMaxResistance());
```

  Impiego di stream() in Grid.java per verificare che tutte le navi siano affondate.

```
return ships.stream().allMatch(ship -> ship.isSunk());
```

Sviluppo di algoritmi particolarmente interessanti
I seguenti algoritmi sono stati trattati in dettaglio nella sezione 2.2 “Design dettagliato” sottosezione 2.2.3, si elencano per completezza.

- *Modalità "Hunt and Target" (livello IA medio)*: implementazione di una strategia dinamica che, a seguito di un colpo a segno, seleziona le celle adiacenti come target, evitando quelle già colpite.
- *Modalità "Pseudo-AI" (livello IA difficile)*: calcolo di un punteggio euristico per ogni cella della griglia in base alla disposizione e agli spazi disponibili, simulando un comportamento intelligente. Viene poi stimata una probabilità per guidare la scelta del tipo di proiettile.