1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

DISPEA
DIPARTIMENTO DI
SCIENZE PURE E APPLICATE

WALK OF WAR

Relazione di progetto
Programmazione ad Oggetti

Baffioni William 300536
Galli Thomas 301594

Indice

Capitolo 1
Analisi

WALK of WAR è una rivisitazione strategica e dinamica del classico *Gioco dell'Oca*. Mentre l'obiettivo principale resta invariato — raggiungere il traguardo per primi — il percorso per arrivarci si arricchisce di nuove sfide e scelte strategiche, trasformando il gioco da una semplice corsa di fortuna a un campo di battaglia dove ogni mossa è cruciale. Il gioco introduce infatti diverse nuove meccaniche per ridurre al minimo l'elemento aleatorio tipico del gioco originale e richiede ai giocatori di sviluppare una strategia precisa per vincere. Tra queste innovazioni spiccano il sistema di *duello* e il *negozio*, elementi pensati per aggiungere profondità e complessità al gameplay.

Il *negozio* permette ai giocatori di acquistare equipaggiamenti — come armi, armature e scudi — di diverse rarità, alcuni oggetti avranno anche effetti e abilità uniche, come l'immunità al furto (meccanica che vedremo in seguito) o la possibilità di riposizionarsi sulla mappa nel caso il giocatore sia capitato su una casella malus e non voglia subirne gli effetti. Il negozio si rigenera ad ogni apertura, offrendo una nuova selezione di oggetti, spingendo i giocatori a una gestione ottimale delle loro monete, che si guadagnano raggiungendo le apposite caselle bonus, ma ad ogni visita il giocatore è costretto a retrocedere di un numero di caselle costante (imposto dalla visita al negozio) sommato ad un numero di caselle variabile che dipende dalla rarità dell'oggetto acquistato, questo per aumentare le possibili strategie che un giocatore può decidere di intraprendere.

Questi oggetti sono fondamentali per prepararsi al *duello*, che avviene ogni tre turni e può far retrocedere il giocatore sconfitto di diverse caselle. I duelli tra i giocatori si svolgono tramite un sistema di combattimento semplice ma efficace, che permette di attaccare, difendersi o ricaricare la propria stamina. Il numero degli attacchi disponibili dipende dalla rarità dell'arma in possesso, lo stesso vale per il numero di volte che il giocatore può difendersi con la differenza che è possibile recuperare slot attacchi spendendo il proprio turno ricaricando la stamina, mentre il numero di difese è limitato e dipende, quindi, solo dalla rarità del proprio scudo.

Un'altra caratteristica chiave è la meccanica del *furto*, attivata atterrando su specifiche caselle, che consente a un giocatore di rubare un oggetto casuale a un avversario. Questo oggetto sostituisce automaticamente uno di quelli già posseduti, aggiungendo un livello di incertezza e incentivando scelte tattiche in ogni momento.

La vittoria, quindi, non dipende solo dai lanci del dado del giocatore, bensì dalla sua preparazione cruciale e decisionale nelle varie fasi di gioco.

In conclusione, il nostro gioco rivisita un classico dell'infanzia, rendendolo più strategico e coinvolgente, per offrire un'esperienza unica e competitiva.

Elenco delle caratteristiche previste dall'applicativo:

- 2 o 4 giocatori (con possibili combinazioni di utenti reali o gestiti da cpu)
- una pedina per ogni giocatore
- una tavola da gioco
- 108 caselle totali che possono essere di sei tipi diversi:
  - Vuota (casella base senza alcun effetto)
  - Bonus movimento
  - Malus movimento
  - Bonus monete
  - Malus monete
  - Furto
- un negozio
- meccanica di duello

1.1 Requisiti

Requisiti funzionali

- L'applicazione deve permettere la scelta della modalità di gioco (2 giocatori oppure 4 giocatori).
- L'applicazione deve permettere la scelta dei giocatori reali e giocatori AI.
- L'applicazione deve essere in grado di creare il tabellone di gioco, disponendo in modo casuale le caselle dei vari tipi.
- L'applicazione deve permettere a ciascun giocatore (nel proprio turno) di scegliere l'opzione Negozio.
- L'applicazione deve eseguire un lancio del dado (6 facce) per ogni giocatore (nel proprio turno).

- L’applicazione deve eseguire il movimento delle pedine (associate ai giocatori) sul tabellone a seconda dell’esito del lancio del dado.
- L’applicazione deve essere in grado di gestire gli eventi causati dalle caselle in base alle regole del gioco.
- L’applicazione deve poter apportare modifiche al tabellone di gioco e alle sue caselle.
- L’applicazione deve essere in grado di generare il Negozio e di gestirne il refresh, generando casualmente gli oggetti acquistabili al suo interno.
- L’applicazione deve permettere ai giocatori di acquistare ed equipaggiare (una volta acquistati) gli oggetti presenti nel Negozio.
- L’applicazione deve gestire l’evento periodico Scontro in cui vengono accoppiati i giocatori per sfidarsi in un minigioco.
- L’applicazione deve permettere il corretto svolgimento del minigioco.
- L’applicazione deve gestire ricompense e penalità risultanti dall’esito dello Scontro.
- L’applicazione deve essere in grado di decretare il vincitore del gioco.

Requisiti non Funzionali

- L’applicazione dovrà mostrare animazioni di movimento delle pedine nel tabellone.
- L’applicazione dovrà mostrare le informazioni di ogni giocatore nel corrispettivo turno.
- L'applicazione potrà mostrare una classifica finale per visualizzare l’esito della partita di ogni giocatore

1.2 Modello del dominio

Entità: Ruolo e relazioni

La struttura di WALK of WAR è basata su diverse entità fondamentali che collaborano tra loro.

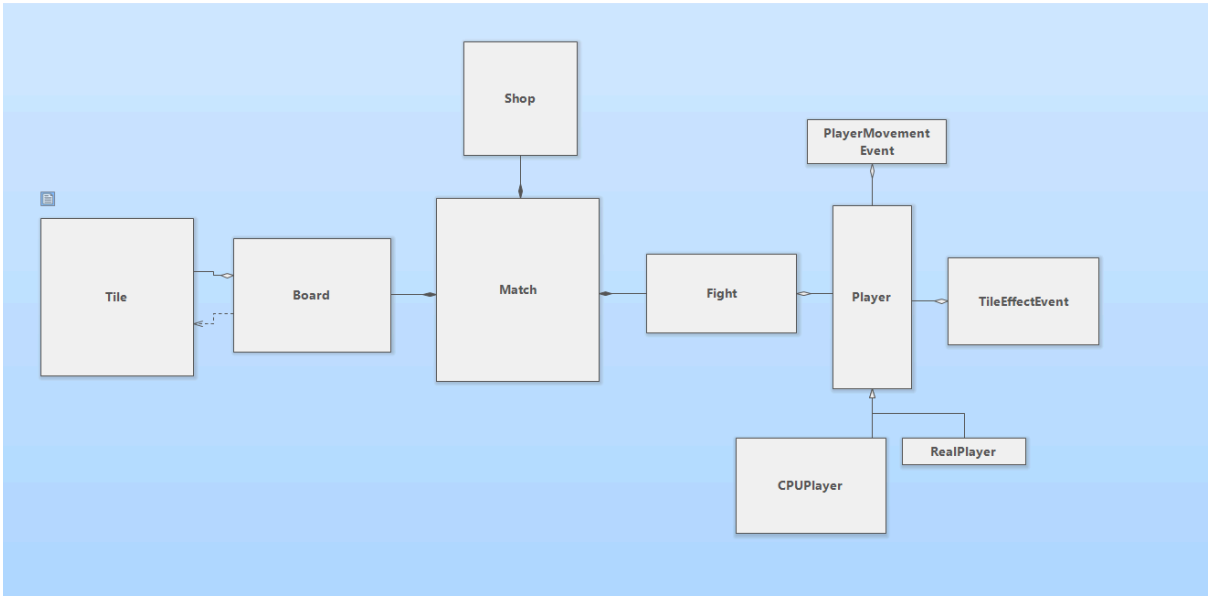
Il cuore della partita è rappresentato dall'entità **Match**, che gestisce e coordina ogni fase del gioco, dal turno dei giocatori alle condizioni di vittoria. Il **Board**, o tabellone, rappresenta il percorso che i giocatori devono completare; è costituito da una serie di **Tile** (caselle) che possono avere effetti speciali, attivandosi quando i giocatori vi si fermano. Ogni casella è progettata per

aggiungere un elemento di imprevedibilità, influenzando il corso della partita con bonus o malus che costringono i giocatori ad adattare costantemente la propria strategia.
I partecipanti al gioco sono rappresentati dall'entità **Player**, che si suddivide in due categorie: **RealPlayer**, per il giocatore umano, e **CPUPlayer**, che simula un avversario controllato dall'intelligenza artificiale. Ogni giocatore possiede attributi specifici, come la propria posizione sul tabellone, le monete accumulate, e alcune abilità speciali che possono essere attivate durante la partita. Periodicamente, i giocatori possono scontrarsi in un **Fight**, un duello che introduce meccaniche di sfida diretta e permette di influenzare lo stato degli avversari.
Per aggiungere un ulteriore elemento strategico, i giocatori hanno anche accesso ad uno **Shop** (negozio) dove possono spendere le loro monete per acquistare potenziamenti e abilità speciali, dando maggiore profondità alla loro esperienza.

L'applicazione deve poter gestire partite da 2 o 4 giocatori, ottimizzando le componenti del gioco di conseguenza. Inoltre queste due modalità si suddividono in:

- 2 giocatori
  - 2 giocatori reali
  - 1 giocatore reale e 1 giocatore AI
- 4 giocatori
  - 4 giocatori reali
  - 3 giocatori reali e 1 giocatore AI
  - 2 giocatori reali e 2 giocatori AI
  - 1 giocatore reale e 3 giocatori AI

Le modalità di gioco sono definite all'interno di GameMode, il quale comunicherà a Match (all'atto della scelta) come creare e gestire l'istanza di gioco, la mappa, i
giocatori e tutte le altre funzioni ed eventi di gioco.



UML di analisi problema

Capitolo 2
Design

Di seguito riporteremo l'itinerario di scelte e strategie messe in atto per risolvere e rispecchiare adeguatamente i requisiti individuati nella sezione di Analisi precedente.

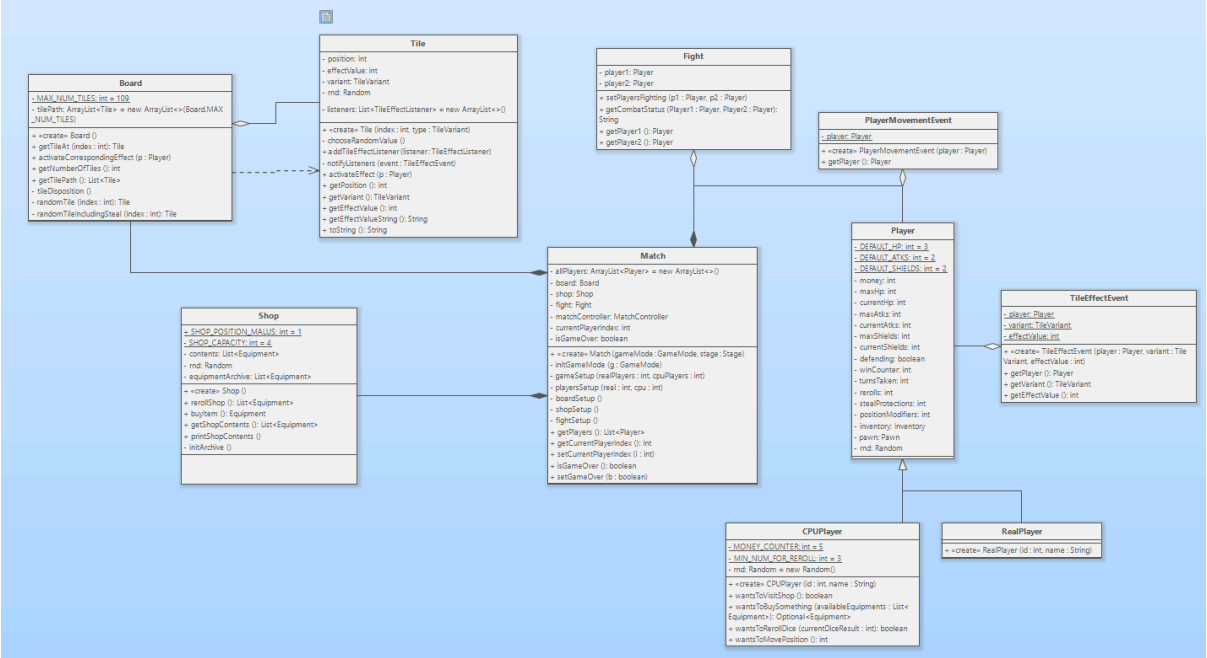
2.1 Architettura

Per quanto riguarda l'architettura del nostro applicativo,abbiamo optato fosse la scelta migliore l'utilizzo del pattern architetturale MVC, che si adatta perfettamente allo sviluppo di un videogioco.

Model

Il nostro Model sarà quell'insieme di entità che si occupa di contenere e preservare i dati che costituiscono le componenti di gioco e il suo stato corrente. Inoltre sarà compito del Model anche quello di gestire adeguatamente l'accesso e la modifica dei dati menzionati, in modo che gli eventi di gioco e interazioni degli utenti siano in grado di modificare lo stato del gioco, garantendo comunque protezione dall'esterno. A tal proposito torna decisamente utile la separazione architetturale proposta dal pattern MVC: il

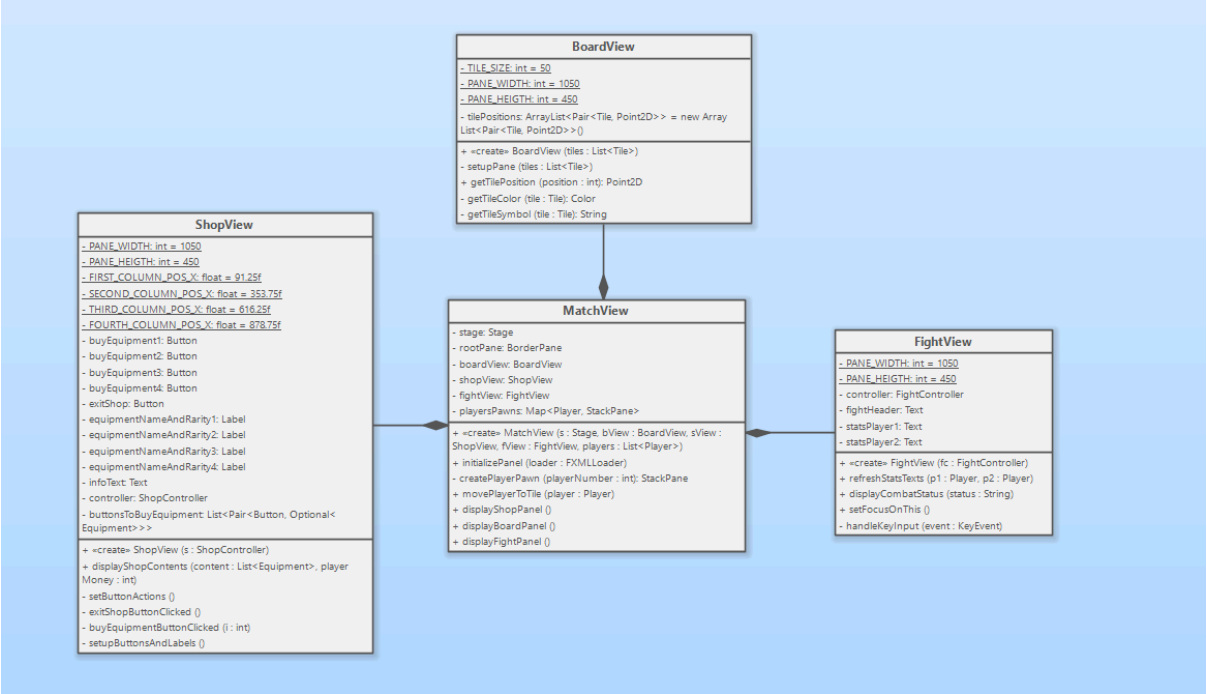
Model fornirà al Controller accesso ai dati interessati dell'applicazione, il quale servirà da intermezzo tra Model e View.
Data la natura dell'entità Match descritta nel capitolo precedente, risulta naturale considerarla parte integrante dell'implementazione del Model, in quanto contiene la maggior parte di informazioni circa il gioco e le sue componenti.
Di seguito viene riportato lo schema generale del Model.



UML delle classi Model

View

La View è la componente dell'applicazione che si occupa della gestione della GUI e dell'interazione con l'utente. Per questa parte del progetto abbiamo deciso di utilizzare JavaFX, uno dei più famosi framework Java orientato alla creazione di interfacce grafiche.
La View conterrà dunque le diverse finestre e interfacce che compongono la GUI che, come specificato precedentemente, saranno associate ciascuna ad un personale Controller, il quale si occuperà di gestire solamente le interazioni che riguardano la propria View.

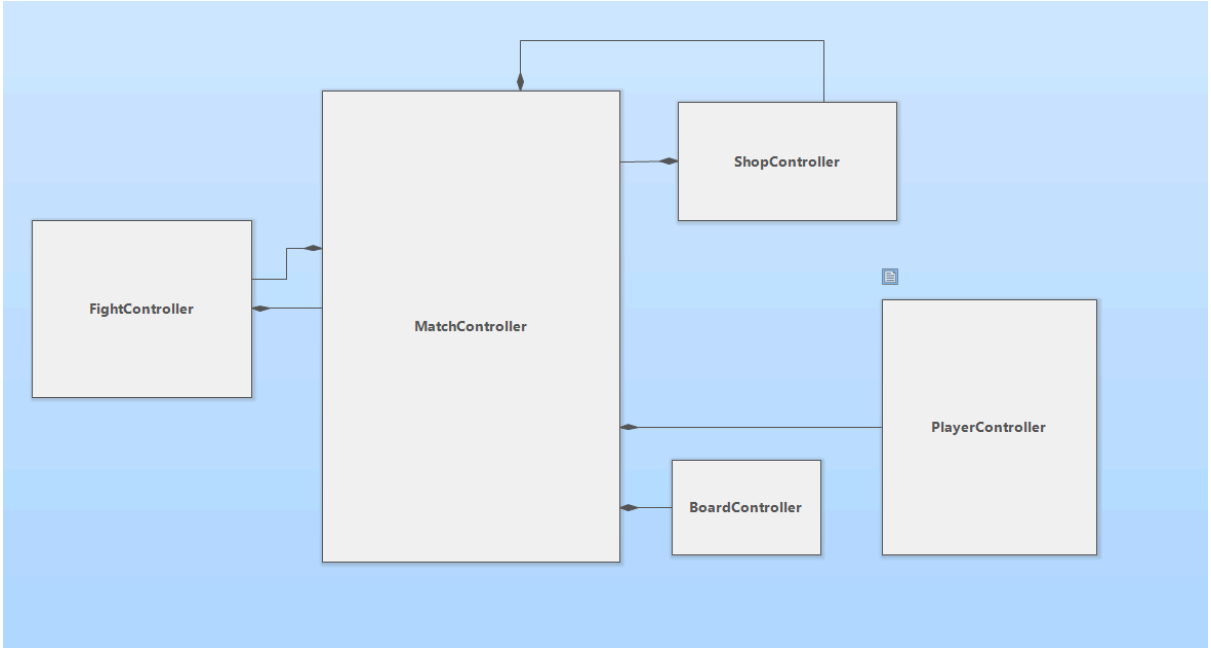


UML delle classi View

Controller

Il Controller è la componente architetturale che si occupa di gestire la comunicazione tra Model e View, in modo che i cambiamenti di uno influenzino quelli dell'altro.

In particolare abbiamo optato per un'associazione 1:1 tra Controller e View in modo da mantenere il codice ben strutturato e comprensibile. Dato che ogni View sarà controllata e gestita dal proprio e unico Controller risulta facile individuare e risolvere eventuali problemi di interazione, ma anche integrare nuove funzionalità separatamente. Ora semplicemente indicando un'interfaccia apposita per i Controller possiamo incapsulare il comportamento che ci si aspetta in generale per poi specializzare le diverse implementazioni mirate.



UML delle classi Controller

2.2 Design dettagliato

Baffioni William

Sistema a turni

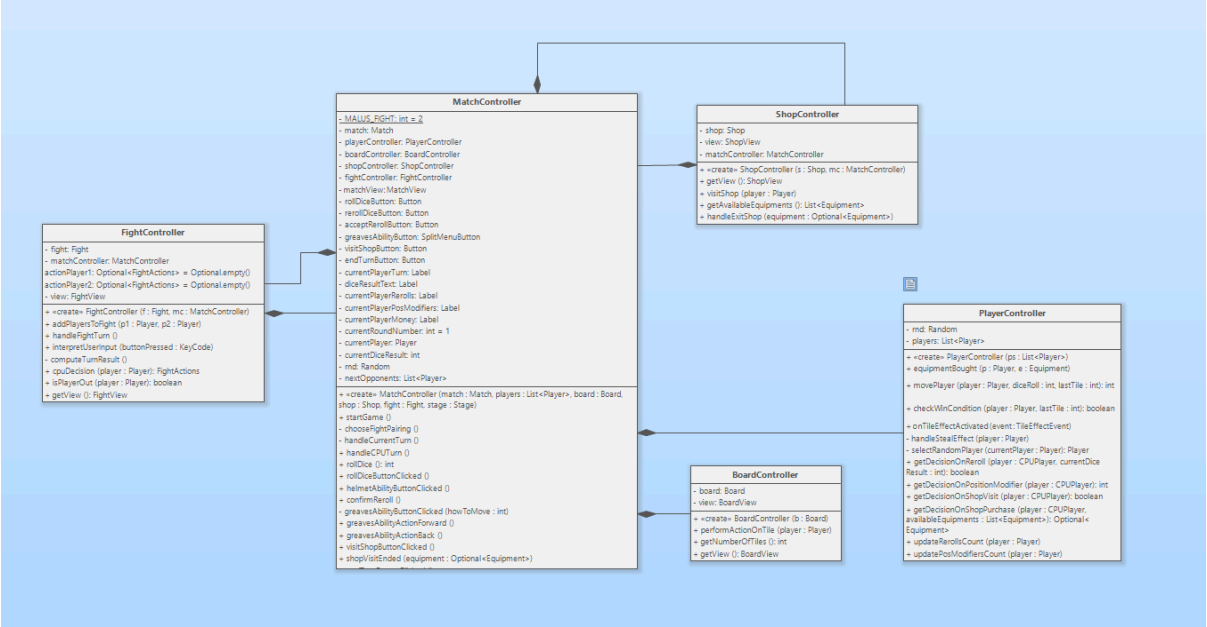
L'obiettivo principale era quello di creare un gestore centrale per la logica di gioco in un ambiente interattivo. La sfida consisteva nel coordinare le varie componenti del gioco, garantendo che l'interazione tra i giocatori, il tabellone e le meccaniche di gioco fosse fluida e intuitiva.
L'implementazione di MatchController ha affrontato questo problema creando una struttura che consente di gestire i turni dei giocatori, sia umani che controllati dalla CPU, e di orchestrare le azioni che ciascun giocatore può compiere. Grazie a questa classe, siamo in grado di mantenere il controllo del flusso di gioco, garantendo che ogni fase del turno sia gestita correttamente.

Funzionalità principali del sistema a turni

MatchController offre una serie di funzionalità chiave:

- Gestione dei Turni: la classe si occupa di determinare il giocatore attivo e gestire le azioni di ciascun turno, inclusi il lancio dei dadi e il movimento sulla tavola;

- Interazione con l'Interfaccia Utente: fornisce un'interfaccia reattiva, abilitando e disabilitando i bottoni in base alle azioni disponibili, e aggiornando le etichette con informazioni contestuali per i giocatori;
- Supporto per CPU e Giocatori Umani: gestisce le decisioni automatizzate per i giocatori controllati dalla CPU, mantenendo l'interesse e il dinamismo del gioco anche in assenza di input umano;
- Controllo delle Condizioni di Vittoria: tiene traccia delle condizioni di vittoria e gestisce la transizione tra i turni, garantendo che il gioco possa proseguire senza intoppi fino alla sua conclusione.



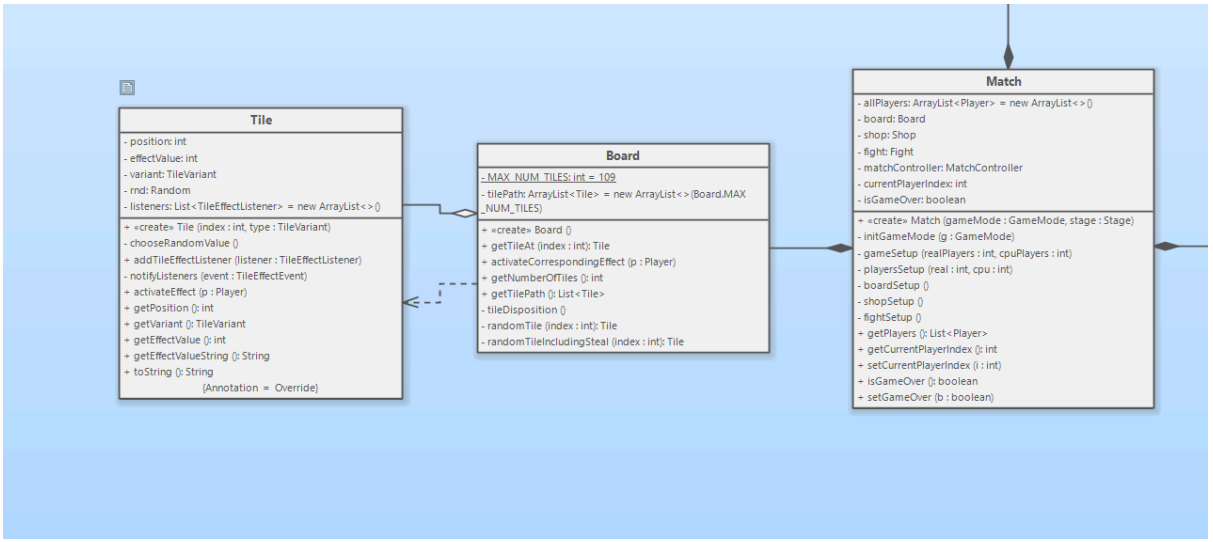
Gestione interfaccia grafica

Purtroppo ho riscontrato difficoltà con l'implementazione corretta del MVC per quanto riguarda la dipendenza tra MatchController e MatchView: l'inesperienza con l'utilizzo di file .fxml ed il programma SceneBuilder mi ha portato a gestire in modo scorretto la separazione dei concern e ho dovuto inserire riferimenti a componenti della UI in MatchController per poter gestire i loro eventi. Come risultato non è stato possibile separare il flusso di gioco da quello della UI e perciò la View si blocca nei turni dei giocatori CPU. Il problema è stato in seguito corretto per la gestione della View di Shop e Fight, scegliendo di abbandonare i file .fxml e SceneBuilder, impostando manualmente disposizione e comportamento dei vari elementi di UI e questo

ci ha permesso di implementare correttamente la suddivisione dei compiti richiesta da MVC. Per motivi di tempo non siamo riusciti ad aggiornare MatchController e MatchView seguendo lo stesso metodo.

Creazione e gestione Board di gioco

La gestione della board è una delle componenti più importanti del nostro sistema di gioco: Questa parte è cruciale perché permette di gestire le interazioni tra i giocatori e le caselle in modo efficace, garantendo che gli effetti associati vengano applicati correttamente. Il nostro approccio (come nei precedenti e vedremo anche nei successivi) si articola su tre elementi principali: il modello della board, il suo controller e la visualizzazione. La classe Board funge da fondamento per il nostro sistema, occupandosi di organizzare le caselle e di disporle in un percorso casuale. Questa classe non solo definisce il numero massimo di caselle, ma si occupa anche di attivare gli effetti quando un giocatore atterra su una di esse. Ciò significa che ogni volta che un giocatore si posiziona su una casella, vengono gestiti automaticamente effetti come bonus o malus. A stretto contatto con la classe Board troviamo il BoardController, il quale svolge un ruolo essenziale di coordinamento. Questo controller funge da intermediario tra il modello e la vista, gestendo le azioni dei giocatori sulla board. Quando un giocatore si muove, il BoardController determina quale casella è stata raggiunta e si occupa di attivare l'effetto corrispondente. In questo modo, il sistema diventa fluido e reattivo, permettendo un'esperienza di gioco senza interruzioni. Infine, la BoardView si occupa della rappresentazione visiva della board. Essa crea un'interfaccia chiara e accattivante, mostrando ogni casella con le relative informazioni sui tipi e sui valori degli effetti. La disposizione delle caselle avviene in modo ordinato, rendendo la navigazione attraverso la board intuitiva per i giocatori.



UML Board (Model)

Flusso di gioco

La nostra prima sfida è stata progettare un sistema che permettesse al giocatore di scegliere facilmente la modalità di gioco e di iniziare la partita senza interruzioni. Come soluzione abbiamo strutturato il *flusso* in modo che tutto parta dalla view, l'interfaccia grafica, dove il giocatore può selezionare la modalità preferita. Questa scelta viene immediatamente trasmessa al *MenuController*, il controller del menu principale, che non solo gestisce tutte le opzioni ma crea anche un'istanza di *Match*, il modello principale del gioco. Il passo successivo è stata la gestione del complesso insieme di componenti e regole di gioco. Per farlo, *Match* diventa il nucleo di tutti i modelli principali, e crea un *MatchController*, che prende in carico i vari modelli e organizza la partita. Qui, ogni elemento del gioco ha il suo specifico controller per assicurare una gestione modulare e coerente. Il *MatchController*, a sua volta, inizializza controller specifici per ogni modello, tra cui il *PlayerController*.

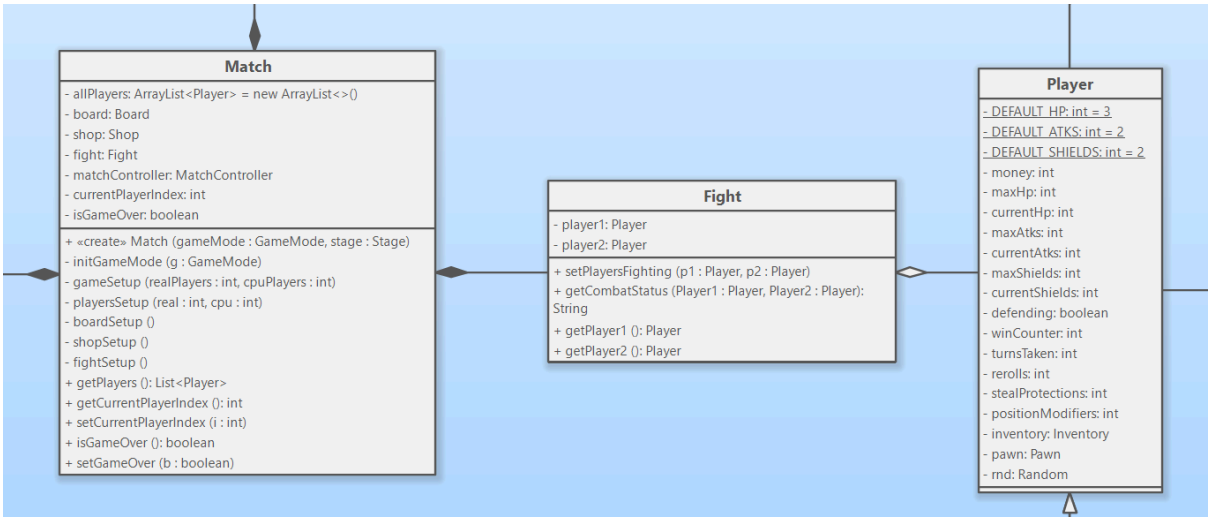
Infine, per il controllo dei giocatori, abbiamo creato il *PlayerController*, che amministra tutti i player, sia umani che CPU, e garantisce che ognuno possa agire in partita secondo le proprie logiche e regole.

Galli Thomas

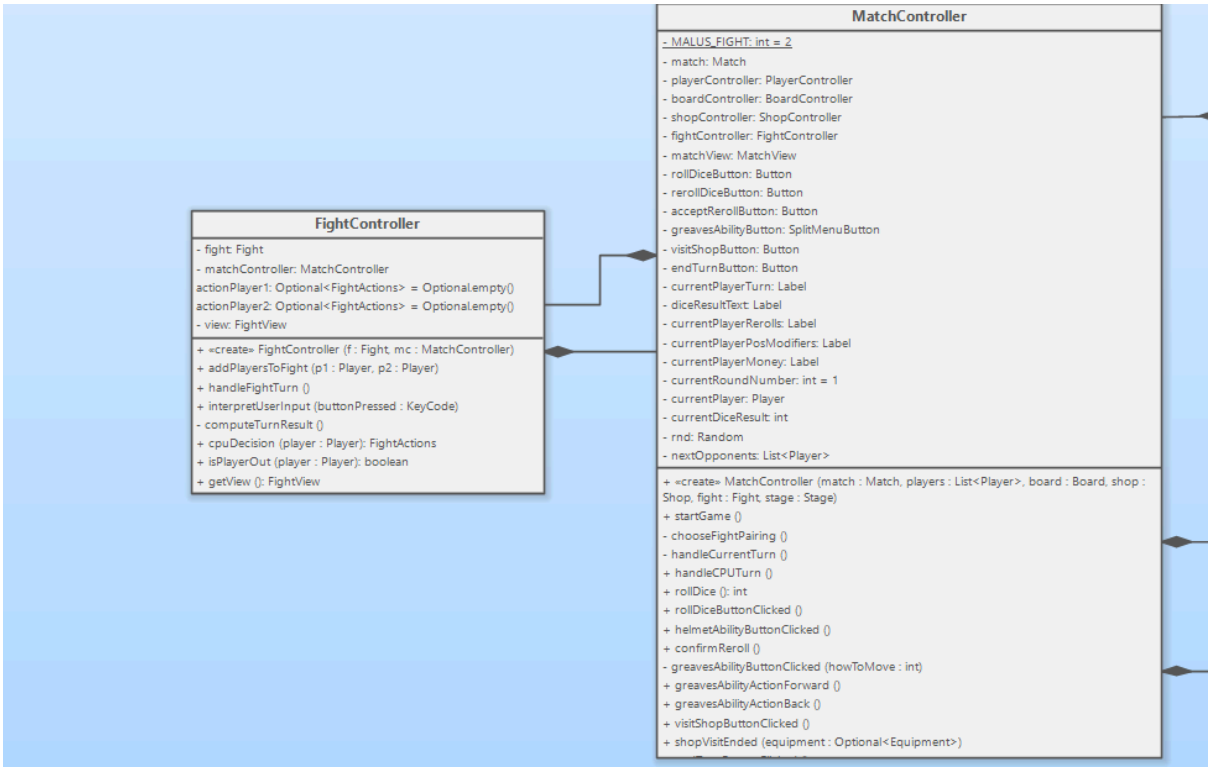
Creazione e gestione delle meccaniche di Scontro

Inizialmente per l'evento periodico di "Scontro" avevamo pensato di introdurre una meccanica di confronto basata sui livelli dei giocatori, che sarebbero aumentati in base alla rarità degli oggetti equipaggiati, dove il giocatore che aveva accumulato il livello più alto, sommando i vari livelli del proprio equipaggiamento, ne usciva vincitore. Tuttavia, ci siamo resi conto che questa soluzione si limitava a un semplice confronto numerico tra i giocatori, senza creare una vera dinamica di combattimento. Volevamo eliminare la componente aleatoria tipica del gioco dell'oca e introdurre una modalità di interazione più coinvolgente e strategica, quindi questa non ci è sembrata la soluzione più adatta. Perciò, abbiamo sviluppato una nuova meccanica di combattimento che include azioni di attacco, difesa e ricarica.
Abbiamo, quindi, implementato un sistema di combattimento completo gestito dalla classe FightController, che media tra la logica di gioco e l'interfaccia utente per garantire un'esperienza di combattimento dinamica e coinvolgente. La classe FightController gestisce tutti gli aspetti chiave del combattimento, partendo dalla creazione di un'istanza di Fight che rappresenta l'intero contesto dello scontro. In questo modo, possiamo aggiungere due giocatori (che possono essere umani, CPU o una combinazione di entrambi) all'inizio della battaglia.
Una volta impostati i giocatori, il controller avvia un ciclo di combattimento che prosegue fino all'esaurimento dei punti vita (HP) di uno dei due contendenti. Durante ogni turno, lo stato attuale del combattimento viene aggiornato e visualizzato, mostrando i punti vita rimanenti e le possibili azioni: attacco, difesa, e ricarica dei punti stamina. Per i giocatori umani, il controller acquisisce l'input tramite la classe FightView, mentre per le CPU simula una scelta il più possibile vicina alla logica di un giocatore reale, garantendo così una sfida bilanciata. Questa struttura di azioni offre una vasta gamma di strategie, rendendo ogni turno più avvincente e competitivo. Ad ogni fase del ciclo, le azioni scelte vengono risolte, e lo stato del combattimento viene aggiornato. Il ciclo si conclude quando uno dei due giocatori esaurisce i propri punti vita, determinando così il vincitore. Se il combattimento termina con un vincitore o in caso di pareggio, il sistema fornisce un messaggio chiaro con il risultato, risolvendo così in maniera efficace la nostra esigenza di un duello strategico. Questa struttura permette un approccio più tattico, in cui ogni

giocatore deve prendere decisioni strategiche, dando vita a un vero e proprio duello competitivo.



UML classe Fight (Model)



UML classe Fight Controller

Gestione sistema Shop

La gestione efficace degli oggetti da acquistare è fondamentale per migliorare l'esperienza del giocatore. Per affrontare questa sfida, ho sviluppato un sistema che integra tre componenti principali: il modello dello shop, il controller e la vista.

Il problema centrale era la necessità di consentire ai giocatori di visualizzare e acquistare oggetti in modo intuitivo e funzionale, garantendo al contempo un aggiornamento corretto delle risorse del giocatore, come il denaro e l'inventario. In seguito descrivo brevemente il funzionamento delle varie funzionalità MVC:

Modello dello Shop: Questa parte del sistema si occupa della gestione degli oggetti disponibili per l'acquisto. Ho creato una classe Shop che mantiene un archivio di equipaggiamenti. Ogni volta che il giocatore interagisce con lo shop, il modello genera casualmente un elenco di oggetti disponibili, limitato a una capacità predefinita.

View: La vista, implementata nella classe ShopView, è responsabile della presentazione degli oggetti disponibili. Mostra il nome e il prezzo degli oggetti, nonché i bottoni per l'acquisto. La vista si aggiorna dinamicamente per riflettere le modifiche apportate dal controller, disabilitando i bottoni degli oggetti che il giocatore non può permettersi.

Controller: Il controller, rappresentato dalla classe ShopController, funge da intermediario tra il modello e la vista. Gestisce le interazioni dell'utente e la logica necessaria per l'acquisto. Quando un giocatore seleziona un oggetto da acquistare, il controller verifica se il giocatore ha sufficienti fondi. Se l'acquisto è valido, l'oggetto viene rimosso dallo shop e il denaro del giocatore viene aggiornato.

Quando il gioco viene avviato, la vista si connette al controller per ricevere l'elenco degli oggetti disponibili. Successivamente, visualizza gli oggetti, consentendo al giocatore di selezionare un acquisto. Al clic sul pulsante di acquisto, il controller gestisce l'azione, controllando le risorse del giocatore e aggiornando di conseguenza il modello e la vista.

Modalità di gioco

Fin dall'inizio, il nostro obiettivo è stato offrire un'esperienza di gioco che permettesse sia sfide tra giocatori reali che contro CPU, in modo che il giocatore potesse scegliere liberamente la modalità di gioco più adatta alle sue preferenze. Per gestire in modo efficace questa flessibilità, abbiamo implementato diverse classi, tra cui il *MenuController* e il *CPUPlayer*. Il *MenuController* è la classe responsabile della gestione delle azioni degli utenti e delle transizioni tra le varie schermate del gioco. Utilizzando JavaFX per comunicare con l'interfaccia utente, questa classe rende semplice e intuitiva la navigazione tra le opzioni di gioco, come l'accesso alle classifiche, l'uscita e l'inizio delle partite nelle varie modalità. Ogni azione nel menu è gestita da un metodo specifico, garantendo un controllo completo sulle funzionalità del gioco e un'esperienza fluida per l'utente. Per la modalità contro CPU, abbiamo progettato la classe *CPUPlayer*. Il nostro intento è stato creare una logica semplice ma efficace, che imitasse il ragionamento di un giocatore reale. La CPU, infatti, valuta il contesto del combattimento per scegliere tra azioni di *attacco*, *difesa* e *ricarica stamina* in modo simile a come farebbe un player umano, rendendo l'interazione più naturale e mantenendo alto il livello di sfida. Grazie alla classe *CPUPlayer*, il giocatore può quindi affrontare una CPU che si comporta in maniera razionale e coerente, rendendo lo scontro stimolante e competitivo anche per chi gioca in modalità singola. Questa struttura, supportata dal *MenuController* per una navigazione intuitiva e dal *CPUPlayer* per una logica CPU credibile, realizza il nostro obiettivo di offrire un gioco accessibile e coinvolgente in tutte le sue modalità.

Capitolo 3

Sviluppo

3.1 Testing automatizzato (Galli Thomas)

Il testing automatico è una pratica fondamentale nello sviluppo software che mira a garantire che il codice funzioni come previsto senza dover verificare manualmente ogni parte del programma. È una forma di controllo di qualità che permette di rilevare errori, bug o regressioni nelle funzionalità in modo rapido e preciso. Per fare ciò abbiamo utilizzato JUnit, una libreria Java ampiamente usata per scrivere test automatici. Ogni metodo di test è progettato per verificare un comportamento specifico di una classe o di una

funzione, e facciamo uso dell'annotazione “@Test” per indicare a JUnit che si tratta di un test. JUnit esegue automaticamente questi test, uno dopo l'altro, e restituisce il risultato come "pass" o "fail" per ciascun caso, permettendoci di avere un feedback immediato sulla correttezza del nostro codice.

CPUPlayer e RealPlayer Test

Come descritto precedentemente io mi sono occupato tra le varie cose della gestione del player quindi mi sono anche occupato di realizzare il suo testing automatizzato, in particolar modo sfruttando le classi CPUPlayerTest e RealPlayerTest. Ho utilizzato JUnit per verificare che ogni classe e i relativi metodi rispettassero i comportamenti attesi, e ho impiegato Mockito per simulare i risultati casuali in modo da ottenere test consistenti e riproducibili.

Per la classe CPUPlayer, ho concentrato il testing sulla logica decisionale, come la probabilità di entrare nel negozio, acquistare oggetti e decidere se spostarsi. Ad esempio, ho simulato le scelte casuali del metodo Random per assicurare che le decisioni seguissero le regole di probabilità impostate. Utilizzando Mockito, ho potuto isolare queste decisioni e verificarle in modo deterministico, controllando che il giocatore CPU si comportasse correttamente in base alle risorse e ai parametri di gioco. Per la classe Player, mi sono occupato di verificare i metodi principali legati alla gestione dell’inventario, l’aggiunta di oggetti, l’aggiornamento delle statistiche e l’interazione con le risorse di gioco. Ogni test ha coperto un aspetto specifico, come l’aggiornamento delle statistiche o l’utilizzo delle protezioni contro i furti, assicurandomi che ogni metodo rispondesse correttamente. Questo approccio mi ha permesso di simulare scenari complessi e garantire la qualità e affidabilità delle logiche di Player e CPUPlayer, rendendo il gioco più stabile e prevedibile anche in condizioni di gioco reali.

- *CPUPlayerTest*
- *RealPlayerTest*

Testing di CPUPlayer

Il testing della classe CPUPlayer si concentra sulla logica decisionale (ad esempio, se entrare nel negozio o acquistare oggetti) che spesso dipende dalla generazione casuale. Per questo motivo, usiamo Mockito per simulare i risultati di Random, in modo da garantire i comportamenti.

Testing di RealPlayer

Per RealPlayer, il testing è più focalizzato sulla corretta gestione delle azioni, come l'aggiunta e la rimozione di oggetti dall'inventario o la modifica delle statistiche. A differenza di CPUPlayer, RealPlayer non ha logiche basate su Random, quindi i test sono più diretti.

MatchController Test

Oltre al player mi è sembrato necessario testare una delle principali classi del nostro gioco, devo ammettere che non ho incontrato poche difficoltà a fare ciò essendo di base una classe molto complessa, ho provato a fare del mio meglio testando:

- Inizializzazione (startGameShouldInitializeGame): Verifica che, una volta chiamato startGame, il gioco sia avviato correttamente e non sia terminato.
- Risultato del dado (rollDiceShouldReturnValueBetweenOneAndSix): Verifica che il metodo rollDice restituisca un valore compreso tra 1 e 6.
- Fine turno (endTurnButtonClickedShouldMoveToNextPlayer): Simula la fine del turno e verifica che il giocatore corrente cambi correttamente.
- Turno CPU (handleCPUTurnShouldCallExpectedMethods): Testa il comportamento del metodo handleCPUTurn, verificando che i metodi pertinenti siano chiamati per la logica di decisione del CPUPlayer.
- Visita al negozio (visitShopButtonClickedShouldCallShopVisit): Verifica che il metodo visitShopButtonClicked esegua correttamente la visita al negozio e aggiorni l'interfaccia grafica.
- Penalità dopo combattimento (fightEndedShouldPenalizeLosingPlayer): Verifica che un giocatore perdente venga penalizzato correttamente dopo un combattimento.

3.2 Metodologia di lavoro

Per quanto riguarda la fase di progettazione abbiamo trascorso la maggior parte del tempo insieme (fisicamente o telematicamente) per definire l'approccio da utilizzare per realizzare la nostra architettura di progetto. Per fare ciò ci siamo serviti di diagrammi UML creati utilizzando il programma "Software Ideas Modeler".

Per quanto riguarda la fase di implementazione abbiamo cercato il più possibile di lavorare separandoci i concern in modo tale da non intaccare troppo il lavoro dell'altro. Nonostante ciò non è stato raro confrontarsi insieme per alcune parti in cui il lavoro di ciascuno diventava complementare a quello dell'altro. In ogni caso abbiamo utilizzato Trello per suddividerci al meglio i Task correnti tenendo traccia dello stato di dello sviluppo.

Infine come strumento per il Version Control abbiamo utilizzato Git, più in particolare GitHub Desktop, tramite cui abbiamo separato il workflow in due branch, uno per ciascuno, e facendo convergere sempre il tutto in un branch main.

3.3 Note di sviluppo

William Baffioni

- **Optional** nel sistema di inventario per creare degli "slot" vuoti in cui inserire equipaggiamenti all'occorrenza;
- **Classe parametrizzata Pair** per associare tra loro degli oggetti: in ShopView associo i bottoni per comprare qualcosa agli equipaggiamenti corrispondenti, in modo simile in BoardView associo le caselle alla loro posizione nella UI;
- **Lambda** negli switch e event handler dei bottoni nelle View;
- **JavaFX** per la creazione delle varie View;
- **SceneBuilder** per la creazione di file .fxml.

Algoritmi particolari:

- Algoritmo per creare la Board di gioco, in cui viene generata in modo casuale la disposizione delle caselle normali e bonus/malus;
- Algoritmo per disporre a serpentina il percorso della Board nella UI.

Thomas Galli

- **Optional** in FightController per gestire la presenza o meno di azioni da compiere;
- **Lambda** negli switch, in particolare nell'inventario.