Relazione del progetto "PokèRogue"

Alex Casadio
Pietro Maretti
Tommaso Cosimo Miraglia
Egor Tverdohleeb

10 giugno 2025

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il progetto di PokeRogue mira a realizzare una ricostruzione semplificata dell'omonimo gioco[1]. Nello specifico, si intende riprodurre la maggior parte degli aspetti funzionali, tralasciando quelli grafici e sonori.
L'obiettivo finale è ottenere un gioco in stile roguelike (da cui il nome **Poke + Rogue**) ossia un gioco in cui si completa un salvataggio tramite molteplici run. Per run intendiamo una serie di scontri vinti conseguitivamente fino alla vittoria finale o al game over. Durante una singola run, il giocatore sceglie tre pokémon tra quelli disponibili nel proprio box e forma una squadra. Per "box" si intende il contenitore dei pokémon selezionabili all'inizio di ogni partita; esso può essere espanso catturando nuovi pokémon nel corso della run. Il giocatore utilizza la propria squadra in una serie di scontri continui contro nemici via via più forti.
Gli scontri possono avvenire contro un allenatore con la sua squadra di pokémon oppure contro pokémon selvatici. Il gioco termina dopo aver vinto 100 scontri.
Ogni scontro è suddiviso in turni; a ogni turno, sia il nemico che il giocatore devono compiere una scelta. Durante il combattimento, il giocatore può scegliere tra diverse azioni: utilizzare una mossa con il pokémon attualmente in campo, provare a catturare il pokèmon avversario se selvatico usando una delle pokéball disponibili, cambiare il pokémon attivo oppure saltare il turno usando l'opzione "run".
Se un pokémon avversario viene mandato KO, i pokémon del giocatore ricevono esperienza, che permette loro di salire di livello, diventare più forti e imparare nuove mosse.

[1] `https://pokerogue.net/`

Alla fine di ogni scontro, viene presentato un negozio in cui il giocatore può scegliere gratuitamente uno tra tre oggetti oppure acquistare uno dei tre oggetti a pagamento offerti.
Terminata la run, è possibile salvare il proprio box con i pokémon catturati, per poterli riutilizzare in una futura run.

Requisiti funzionali

- Il giocatore deve poter selezionare la propria squadra iniziale prima di ogni run.
- Il gioco deve permettere la gestione di molteplici salvataggi mantenuti in memoria.
- La difficoltà del gioco deve risultare bilanciata.
- Dopo che il giocatore e il nemico hanno effettuato le proprie scelte, deve essere gestita correttamente la priorità tra le due azioni.
- Durante ogni azione, il sistema deve tenere conto di tutti i modificatori attivi nel gioco.

Requisiti non funzionali

- Il gioco deve adattarsi correttamente a schermi di qualsiasi dimensione, gestendo lo stretch della risoluzione.
- Il sistema deve essere in grado di ricevere ed elaborare correttamente l'input da tastiera del giocatore.
- L'interfaccia di gioco deve essere semplice e intuitiva, in modo da rendere fluidi e rapidi sia gli scontri che l'intera esperienza di gioco.

1.2 Modello del Dominio

Le battaglie si svolgono tra due `Trainer` ossia entità in possesso di una squadra di Pokémon e in grado di prendere `Decision`. Anche i Pokémon selvatici, sebbene dotati di una sola unità in squadra, vengono rappresentati secondo la stessa struttura concettuale. Ciascun `Trainer` dispone di una squadra di `Pokemon` composta da un massimo di sei elementi. Ogni `Pokemon` è associato a una singola `Ability` e a un insieme di `Move`. Entrambe avranno effetti diversi sul pokemon del `Trainer` o dell'avversario in base a quale si

stia utilizzando. si noti dunque che Abilità, Mosse e Item hanno un `effect` modelizzalibile in modo unico

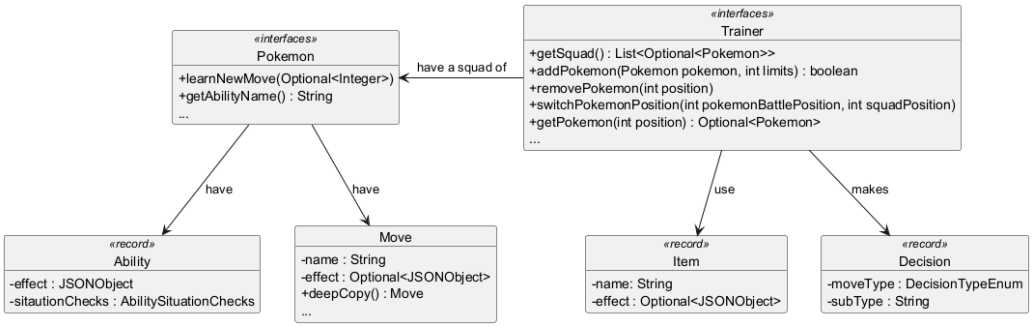


Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

L'architettura del gioco PokéRogue adotta il patternMVC (Model-View-Controller), che consente una chiara separazione delle responsabilità e una maggiore flessibilità nello sviluppo. Grazie a questa struttura, è possibile sostituire intere componenti della View senza influire sul Controller o sul Model. Questa suddivisione favorisce inoltre lo sviluppo modulare, in cui ogni parte può essere implementata e testata isolatamente. La comunicazione tra i componenti MVC avviene tramite metodi definiti nelle interfacce, garantendo una struttura coerente e facilmente estendibile.
L'architettura del progetto si basa su cinque elementi principali:

- **Scene**
- **GameEngine**
- **GraphicEngine**
- **KeyListener**

Il **GameEngine** svolge un ruolo centrale, coordinando la comunicazione tra le Scene, il KeyListener e il GraphicEngine. Il gioco si sviluppa attorno a un ciclo semplice ma efficace: il KeyListener intercetta gli input da tastiera e li inoltra al GameEngin, che contiene un riferimento alla Scene attiva, alla quale delega l'elaborazione dell'input.
La Scene gestisce l'input aggiornando lo stato interno e modificando di conseguenza gli elementi visivi attraverso la SceneView. Inoltre, ogni Scene interagisce con i componenti del Model per eseguire operazioni come:

- Caricare i Pokémon da un salvataggio

- Salvare una partita
- Caricare le caratteristiche degli elementi grafici
- Eseguire altre funzionalità di gioco

Infine, il **GameEngine** agisce come intermediario tra la **Scene** attiva e il **GraphicEngine**: per ogni ciclo di esecuzione, recupera dalla **Scene** l'elenco aggiornato degli elementi grafici da visualizzare e li trasmette al **GraphicEngine**, che ne gestisce il posizionamento e la renderizzazione a schermo. Questo meccanismo viene ripetuto a ogni input dell'utente, assicurando che ogni tasto premuto sulla tastiera produca un aggiornamento immediato e visibile dell'interfaccia di gioco.

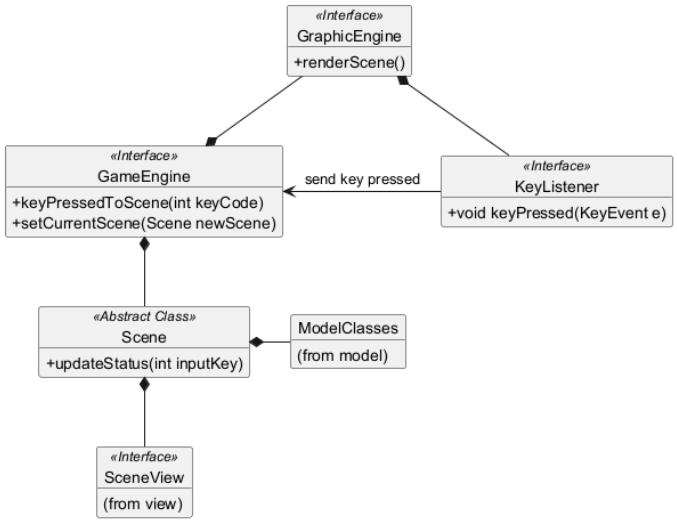


Figura 2.1: Diagramma UML che rappresenta l'architettura del progetto, evidenziando l'adozione del pattern architetturale MVC.

2.2 Design dettagliato

2.2.1 Egor Tverdohleb

Generazione dei Pokemon

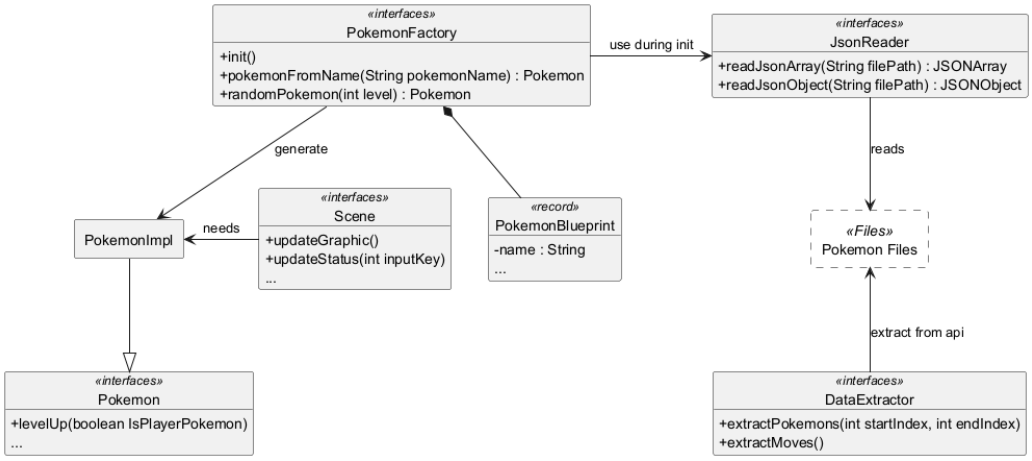


Figura 2.2: Rappresentazione UML della pokemon Factory

Problema È necessario implementare un sistema di generazione dei Pokémon tenendo conto che condividono un insieme di valori fissi determinati dalla loro specie, ma presentano anche alcune caratteristiche che devono essere generate casualmente al momento della creazione.

Soluzione Per gestire in modo efficiente la creazione dei Pokémon, è stato adottato il *pattern Factory*. I dati statici relativi a ciascuna specie di Pokémon sono stati estratti e salvati in appositi file JSON. In particolare si è costruito un DataExtractor per utilizzare la REST API pokeApi [1]. Per evitare accessi ripetuti al file system e ridurre i tempi di caricamento percepiti dall'utente, tali dati vengono caricati una sola volta durante l'inizializzazione del gioco e mantenuti in memoria. La `PokemonFactory` si occupa della lettura dei file JSON e della memorizzazione dei relativi oggetti `PokemonBlueprint` in una struttura indicizzata. I metodi esposti dalla factory consentono di creare nuovi Pokémon clonando le informazioni fisse dal blueprint corrispondente e completandole con valori generati casualmente (es. statistiche variabili, abilità, ecc.). In questo modo, il sistema garantisce

[1] `https://pokeapi.co//`

efficienza nella generazione dei Pokémon e coerenza tra gli esemplari della stessa specie.

Gestione degli effetti

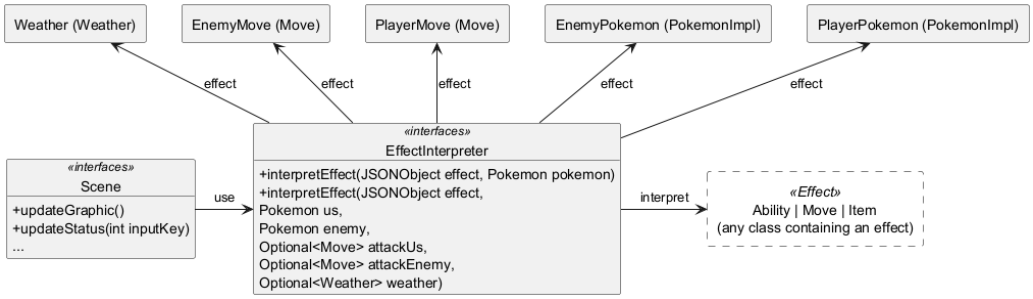


Figura 2.3: Rappresentazione UML della gestione degli effetti

Problema Si desidera gestire Abilità, Mosse e Oggetti considerando che ognuno ha un effetto specifico. Ogni effetto deve essere applicato in modo contestuale alla battaglia in corso.

Soluzione Come descritto nella fase di modellazione, il concetto di effetto è stato unificato e strutturato secondo la seguente rappresentazione all'interno dei rispetti file JSON:

- **checks**: lista di liste di stringhe.

  Ogni elemento della lista rappresenta un singolo *check*, composto da tre stringhe nel formato:

  `Valore1  Segno  Valore2`

  - `Valore1` e `Valore2` rappresentano rispettivamente un parametro (o campo) e un valore atteso, e possono fare riferimento a costanti (es. `enum`) o a proprietà dell'entità in gioco.
  - `Segno` indica un operatore logico tra i seguenti: `==`, $\geq$, $>$, $\leq$, $<$, `!=`.

  Un insieme di `check` è considerato valido se tutte le condizioni in esso presenti risultano vere contemporaneamente.

- **activations**: lista di liste di stringhe.

  Ogni elemento rappresenta un'*activation*, ovvero un'azione che modifica uno stato del sistema. È composta da due stringhe:

  - `Valore1`: il nome del parametro o campo da modificare.
  - `Valore2`: il nuovo valore da assegnare a `Valore1`.

In questo modo è stato possibile definire un EffectInterpreter seguendo il *pattern Interpreter* lasciando grande elasticità al sistema per l'aggiunta di nuove abilita, mosse o ogetti

Generazione delle Abilita e Mosse

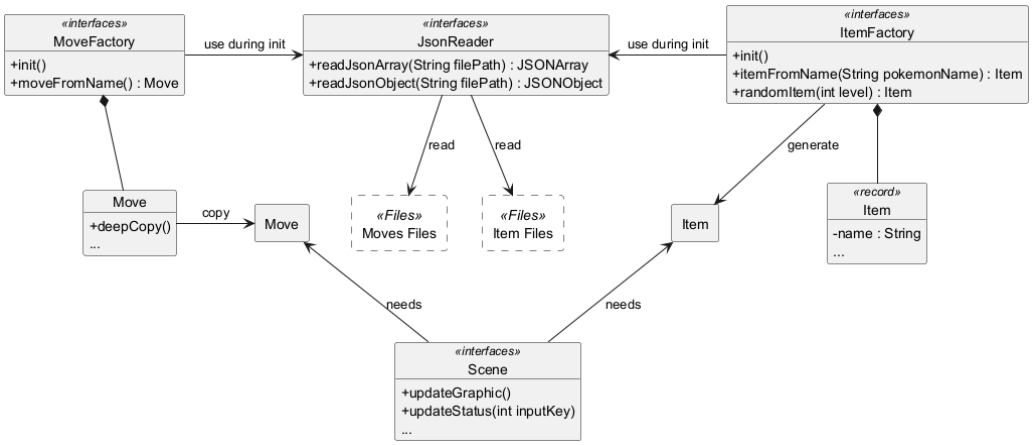


Figura 2.4: Rappresentazione UML delle Move e Ability Factory

Problema Si desidera generare Abilità e Mosse, tenendo considerazione dell'overhead causato dalla lettura in memoria

Soluzione Il problema risulta sostanzialmente analogo a quello della generazione dei Pokémon, e viene affrontato con una soluzione simile, con alcuni accorgimenti specifici. Le Abilità, non necessitando di modificare i propri campi, sono state modellate come oggetti immutabili. Seguendo il *pattern Factory*, è stata realizzata una classe factory incaricata di effettuare un'unica lettura dei dati da memoria all'avvio, per poi restituire le istanze appropriate su richiesta. Le Mosse, al contrario, richiedono uno stato interno modificabile e sono quindi state modellate come oggetti mutabili. Per mantenere l'efficienza del singolo accesso in memoria, alla factory è stato affiancato l'utilizzo

del *pattern Prototype*, permettendo di restituire copie delle mosse originali ogni volta che vengono richieste.

2.2.2 Maretti Pietro

Creazione di un'intelligenza artificiale per i nemici

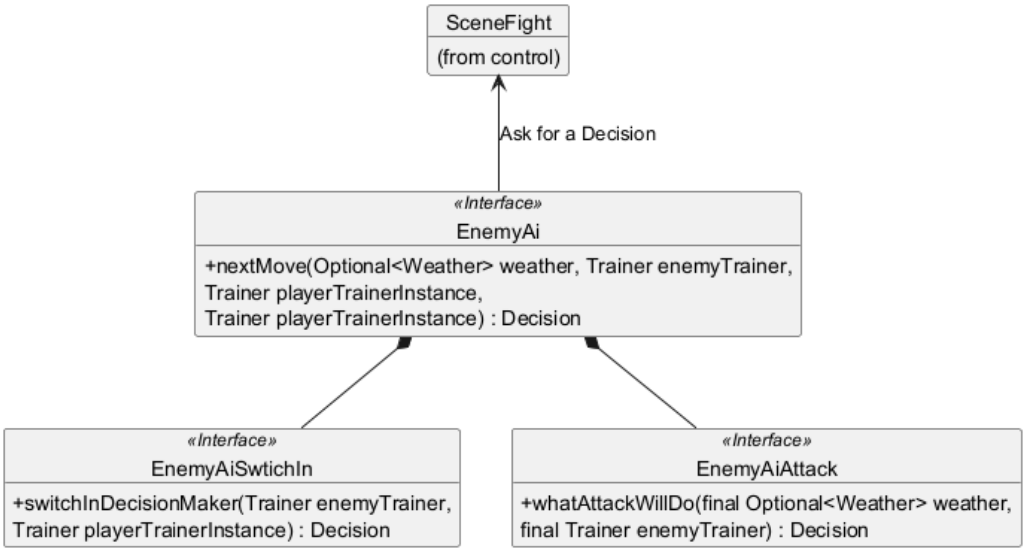


Figura 2.5: Schema UML che raffigura come la SceneFight utilizza l'Ai e di come essa divide il suo compito

Problema Nel contesto del gioco, si desidera implementare un sistema di intelligenza artificiale in grado di prendere decisioni per un nemico durante i combattimenti. Tale sistema deve adattarsi a diversi livelli di difficoltà, influenzando la complessità e l'efficacia delle scelte effettuate dall'IA.

Soluzione Per gestire queste scelte è stata definita l'interfaccia `EnemyAI`, che espone il metodo `nextMove`. Questo metodo restituisce un oggetto di tipo `Decision` e accetta i seguenti parametri:

- un `Optional<Weather>`, che rappresenta le condizioni atmosferiche attuali
- il `Trainer` del giocatore

- il `Trainer` nemico (controllato dall'IA)

La classe `EnemyAiImpl`, che implementa l'interfaccia, riceve in fase di costruzione il livello di difficoltà. Tale valore viene utilizzato per impostare delle flag interne che influenzano il processo decisionale, rendendolo più o meno sofisticato.
Quando viene richiesta la prossima mossa, la classe `EnemyAimpl` utilizza due classi di supporto create segundo il *pattern Separation of concerns*: `EnemyAiSwitchIn` e `EnemyAiAttack`.
Viene invocato il metodo `switchInDecisionMaker()` della classe `EnemyAiSwitchIn`, che valuta le condizioni della battaglia e le flag impostate per decidere se effettuare uno switch e, in tal caso, quale Pokémon mandare in campo. Se questo metodo non restituisce una decisione valida, si procede con la chiamata al metodo `whatAttackWillDo()` della classe `EnemyAiAttack`, che valuta se attaccare e quale mossa utilizzare.
Il primo tra questi due metodi che restituisce una decisione valida fornisce il risultato finale, che viene restituito da `EnemyAiImpl` come mossa scelta dall'intelligenza artificiale.
Nel caso in cui non venga scelta né un'azione di switch né un attacco ad esempio per via dell'impossibilità di agire o dell'inutilità di qualsiasi scelta disponibile l'oggetto `Decision` risulterà vuoto. In tal caso, il nemico non eseguirà alcuna azione durante il turno corrente.

Creazione di un sistema grafico flessibili e che si adatti allo schermo

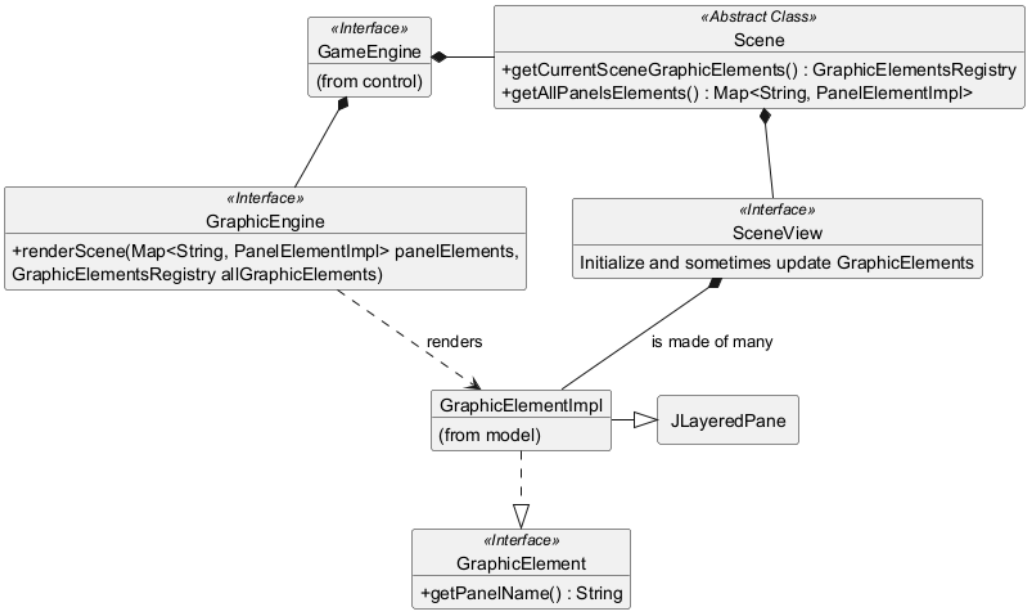


Figura 2.6: Schema UML che mostra il legame fra i GraphicElements le Scene, il GraphicEngine e il GameEngine

Problema Il gioco deve avere una struttura grafica altamente flessibile, in grado di gestire componenti eterogenei come box, sprite, sfondi e testi. L'obiettivo è evitare vincoli rigidi imposti dai layout tradizionali, pur garantendo un comportamento adattivo: l'interfaccia deve essere in grado di ridimensionarsi automaticamente al variare delle dimensioni della schermata.

Soluzione La grafica del gioco si basa su due componenti principali: i `GraphicElement` e il `GraphicEngine`. I primi rappresentano gli elementi visivi che compongono una scena, mentre il secondo è responsabile della loro visualizzazione a schermo.
La classe `GraphicElementImpl` implementa l'interfaccia `GraphicElement` ed estende `JLayeredPane`. Ogni `GraphicElement` è quindi un pannello personalizzato, all'interno del quale vengono disegnati i contenuti grafici tramite il metodo `paintComponent()`, che consente di renderizzare box, testi, immagini e altri elementi visivi.
Sono stati definiti sei tipi principali di `GraphicElement`, ciascuno dei quali

estende `GraphicElementImpl` e ridefinisce il metodo `paintComponent()` in base alle proprie esigenze. Gli elementi fondamentali sono:

- `BoxElement`: un contenitore grafico generico;
- `SpriteElement`: utilizzato per visualizzare immagini;
- `TextElement`: per la gestione e visualizzazione del testo;
- `ButtonElement`: un wrapper di `BoxElement` che aggiunge la funzione `setSelected()`, utile per evidenziare il bordo del pulsante quando selezionato;
- `BackgroundElement`: un wrapper di `SpriteElement`, utilizzato per visualizzare immagini di sfondo a schermo intero.

Ogni `GraphicElement` può essere istanziato specificando manualmente parametri come posizione, dimensioni e stile, oppure tramite un costruttore che accetta un oggetto `JSONObject`. Quest'ultima modalità consente di caricare la configurazione degli elementi da file JSON, rendendo flessibile la definizione delle scene grafiche.
Le coordinate e le dimensioni sono espresse come percentuali rispetto alla risoluzione dello schermo, garantendo un ridimensionamento automatico e proporzionale degli elementi al variare della finestra di gioco.
Per gestire la profondità e la disposizione degli elementi, è stata introdotta l'interfaccia `PanelElement`, che rappresenta un contenitore dotato di `Layout`, configurabile tramite il costruttore. Questo consente di suddividere ogni scena in più layer e di controllare accuratamente la sovrapposizione degli elementi grafici.
Tutti i `GraphicElement` vengono assegnati a un `PanelElement`. Durante la fase di rendering, il `GraphicEngine` utilizza il metodo `getPanelName()` di ciascun elemento per determinarne il pannello di appartenenza e visualizzarlo correttamente nella scena.
L'intera architettura grafica si basa sul *Composite Pattern*. In questo contesto, ogni scena (`View`) è modellata come una gerarchia di elementi: i `GraphicElement` fungono da componenti foglia (*leaf*), ovvero elementi grafici atomici e non suddivisibili, mentre i `PanelElement` rappresentano i compositi (*composite*), in quanto possono contenere altri `GraphicElement`, compresi altri `PanelElement`. Questo approccio consente di strutturare l'interfaccia in maniera modulare, flessibile e scalabile.

Aggiungere e far selezionare i pokèmon al giocatore prima di iniziare la run

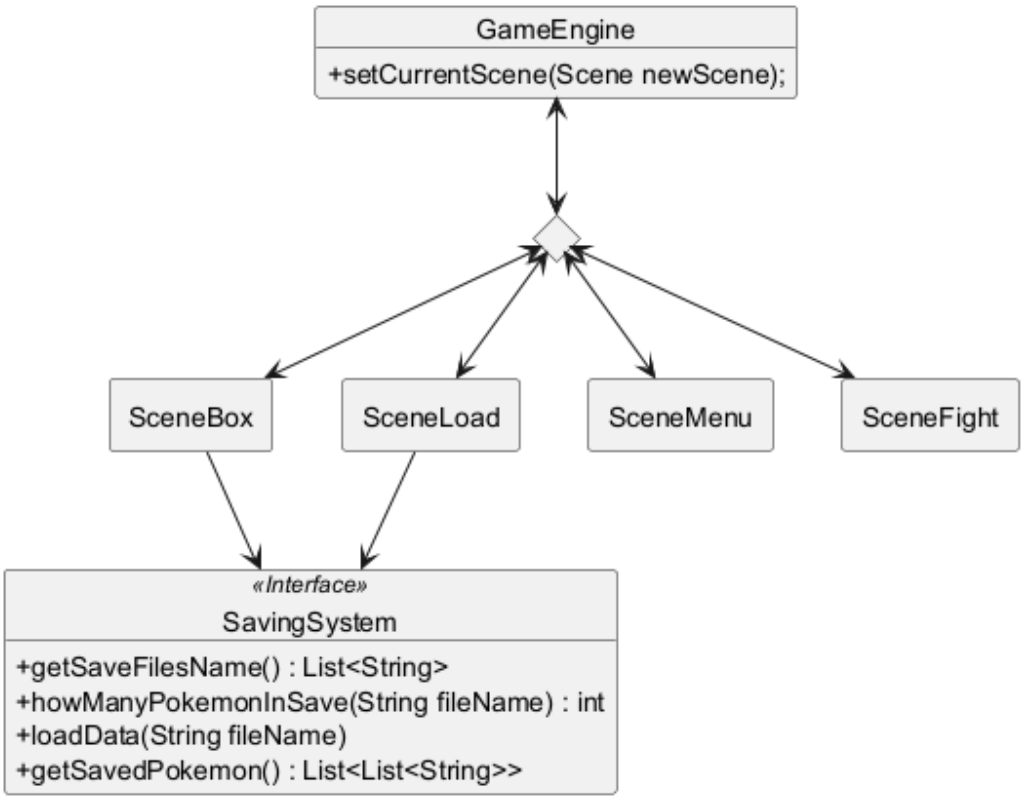


Figura 2.7: Schema UML che raffigura le connessioni fra le Scene: Menu, Load, Box e Fight rispetto al GameEngine.

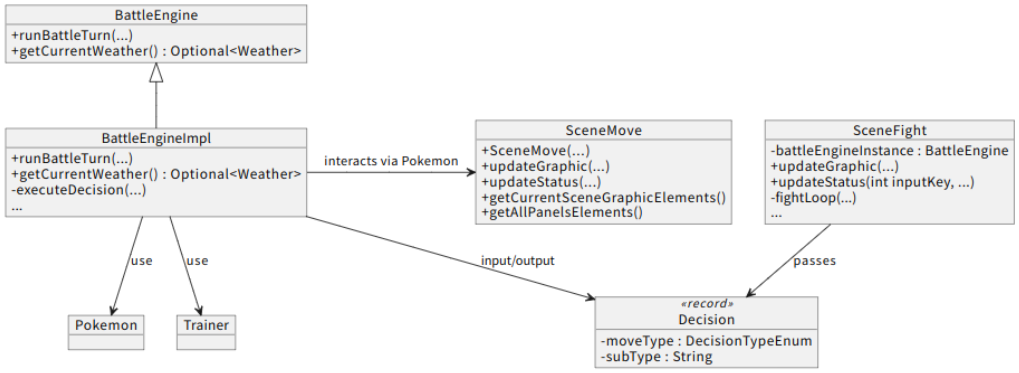
Problema All'avvio di una run, il giocatore deve poter selezionare i Pokémon con cui iniziare. Questa selezione deve avvenire caricando un set di Pokémon da un salvataggio precedente, oppure iniziando da una nuova box contenente i tre Pokémon iniziali predefiniti.

Soluzione Per far selezionare al giocatore la squadra iniziale in una delle due modalità all'avvio del gioco, viene mostrato all'avvio del gioco o alla fine di ogni run la `SceneMenu`, che permette di iniziare una nuova run scegliendo tra due opzioni: caricare un salvataggio esistente oppure partire con la box iniziale contenente i tre Pokémon starter.
Nel caso in cui il giocatore scelga di caricare un salvataggio, la `SceneMenu` richiama il `GameEngine`, che provvede a passare il controllo alla `SceneLoad`.

Questa scena si occupa di caricare tutti i salvataggi disponibili dalla cartella dedicata, utilizzando il `SavingSystem`. Una volta visualizzati, la `SceneLoad` consente di scorrere e selezionare uno dei salvataggi. Il file selezionato viene quindi passato come parametro alla `SceneBox` tramite il `GameEngine`.
Nel caso si opti invece per una nuova partita, viene comunque utilizzata la stessa `SceneBox`, ma inizializzata con una stringa vuota nel costruttore. Questo indica alla `SceneBox` che non deve essere effettuato alcun caricamento da file, ma devono essere invece mostrati i Pokémon starter di default.
Indipendentemente dal percorso scelto, la `SceneBox` ha il compito di mostrare al giocatore la lista dei Pokémon disponibili (da salvataggio o predefiniti) e permettere la selezione di un massimo di tre Pokémon con cui iniziare la run. Ogni volta che un Pokémon viene selezionato, esso viene aggiunto alla squadra del `Trainer` del giocatore; in caso di deselezione, il Pokémon viene rimosso dalla squadra.
Una volta selezionato almeno un Pokémon, il giocatore può avviare ufficialmente la run premendo il pulsante di `Start`, il quale richiama l'inizializzazione della scena di combattimento, la `SceneFight`.

2.2.3 Miraglia Tommaso Cosimo

Gestione del turno di battaglia

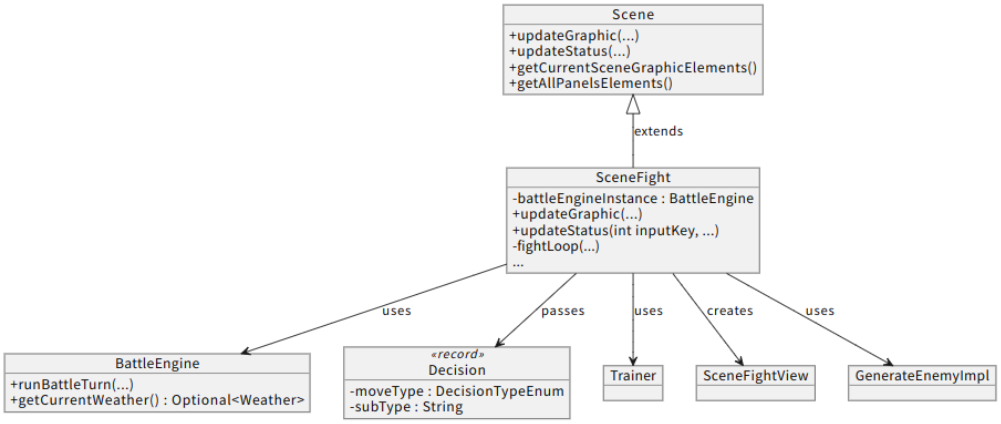


Problema Nel contesto del gioco, è necessario un sistema in grado di orchestrare le dinamiche di combattimento tra il giocatore e un nemico, tenendo conto di numerosi fattori come le mosse selezionate, gli effetti delle abilità, le condizioni meteorologiche, i cambi di Pokémon, e l'interazione con un'eventuale intelligenza artificiale. Questo sistema deve anche integrare meccanismi

per la cattura dei Pokémon, l'assegnazione delle ricompense e la gestione della progressione del giocatore.

Soluzione Per affrontare questo problema è stata sviluppata la classe `BattleEngineImpl`, un'implementazione dell'interfaccia `BattleEngine`. Questa classe si occupa della gestione completa del turno di battaglia e coordina le azioni del giocatore e del nemico, comprese quelle decise dall'intelligenza artificiale.
Il metodo principale, `runBattleTurn`, riceve le decisioni del giocatore e del nemico (sotto forma di oggetti `Decision`) e, a partire dalla priorità delle mosse e dalla velocità dei Pokémon, stabilisce l'ordine di esecuzione. Durante l'esecuzione, vengono valutati e applicati eventuali effetti di abilità (`handleAbilityEffects`), condizioni di stato (`applyStatusForAllPokemon`) e danni derivanti da attacchi (`handleAttack`). Se una delle due decisioni prevede l'uso di una Pokéball, il metodo `handlePokeball` viene invocato per determinare l'esito della cattura, aggiornare il salvataggio e, se necessario, modificare lo stato della partita. In caso di sconfitta del Pokémon nemico o di cambio forzato, viene effettuato un controllo tramite il metodo `newEnemyCheck`, che gestisce l'eventuale ingresso di un nuovo avversario oppure il passaggio a una fase successiva della partita, come lo shop o il salvataggio finale.
Il design della classe adotta un approccio modulare, delegando le responsabilità specifiche (calcolo del danno, applicazione effetti, cambio Pokémon) a metodi separati, seguendo il principio della *Separation of Concerns*. L'integrazione con altre componenti del gioco (es. `GameEngine`, `SavingSystem`) consente di mantenere aggiornata la progressione del giocatore e l'evoluzione della battaglia.

Implementazione della scena di combattimento

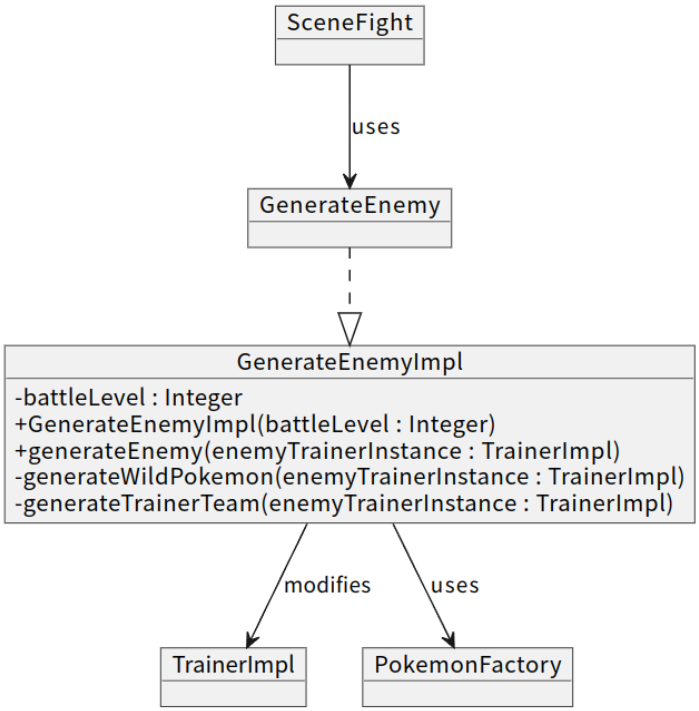


Problema Gestire la scena di combattimento nel gioco, coordinando input utente, aggiornamenti grafici e logica di battaglia, inclusa la generazione del nemico.

Soluzione La classe `SceneFight` estende `Scene` e rappresenta il controller principale della scena di combattimento, orchestrando l'interazione tra l'input utente, la logica di battaglia e la visualizzazione grafica. Al suo interno, la classe gestisce l'inizializzazione e il mantenimento dello stato del trainer nemico tramite `TrainerImpl`, e crea un'istanza di intelligenza artificiale avversaria mediante `EnemyAiImpl`, configurata in base al livello di difficoltà della battaglia.
La gestione degli input da tastiera avviene tramite il metodo `updateStatus`, il quale interpreta i comandi dell'utente aggiornando la selezione corrente o attivando l'esecuzione delle azioni corrispondenti, come attacchi, uso di oggetti o cambio di Pokémon. In particolare, l'input è mappato su codici interni che rappresentano le varie azioni disponibili, consentendo una navigazione fluida e coerente tra i diversi elementi dell'interfaccia.
Quando l'utente conferma un'azione, la classe richiama `fightLoop`, che invoca il metodo `nextMove` dell'istanza di `EnemyAi` per ottenere la decisione strategica del nemico. Successivamente, tramite `BattleEngine.runBattleTurn`, vengono elaborate simultaneamente entrambe le decisioni (giocatore e IA), aggiornando lo stato della battaglia.
L'aggiornamento degli elementi grafici è affidato a `SceneFightView`, che riflette in tempo reale lo stato della scena, assicurando una sincronizzazione visiva con le scelte e gli eventi di gioco. La struttura complessiva garantisce così una chiara separazione tra gestione degli input, logica di combattimen-

to e rendering grafico, permettendo un'architettura modulare, scalabile e facilmente manutenibile.

Generazione Dinamica degli Avversari



Problema Nel contesto del sistema di combattimento, è necessario generare nemici che siano coerenti con il livello di difficoltà e il contesto della battaglia, differenziando tra Pokémon selvatici e allenatori avversari, e gestendo le relative implicazioni di gameplay come la possibilità di cattura.

Soluzione La classe `GenerateEnemyImpl` si occupa di creare nemici adeguati al livello di battaglia. Il sistema prevede due tipologie di avversari: Pokémon selvatici e allenatori nemici con una squadra completa di Pokémon. Nel dettaglio, quando il livello della battaglia è un multiplo di cinque, viene generato un allenatore nemico dotato di una squadra composta da più Pokémon, il cui numero cresce in base al livello, fino a un massimo stabilito. In caso contrario, si genera un singolo Pokémon selvatico, contraddistinto da uno stato che ne permette la cattura durante il combattimento. I livelli dei Pokémon, sia selvatici che della squadra degli allenatori, sono calcolati

in base al livello della battaglia, con una componente casuale che introduce variazioni per aumentare la varietà degli incontri. La generazione degli specifici Pokémon è delegata a una factory dedicata che crea istanze coerenti con il livello stabilito. Questa distinzione tra nemici offre una differenziazione importante nel gameplay: i Pokémon selvatici rappresentano incontri occasionali che il giocatore può catturare e aggiungere alla propria squadra, mentre i Pokémon degli allenatori sono avversari fissi che non possono essere catturati, ma devono essere sconfitti per progredire. In questo modo, la generazione dinamica degli avversari contribuisce a creare una progressione di difficoltà bilanciata e un'esperienza di gioco varia e coinvolgente.

2.2.4 Casadio Alex

Gestione della scena di salvataggio

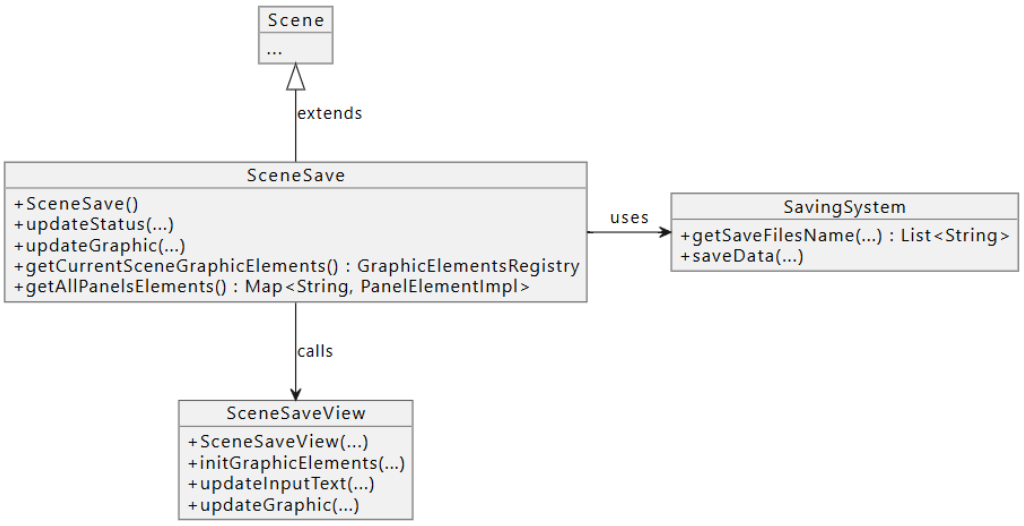


Figura 2.8: Schema UML che raffigura la scena di salvataggio.

Problema Era necessario implementare una schermata di salvataggio che consentisse all'utente di inserire un nome, verificare eventuali conflitti con salvataggi esistenti e scegliere se continuare o uscire salvando.

Soluzione La classe `SceneSave` estende `Scene` ed è responsabile della gestione logica e grafica della schermata di salvataggio. Vengono inizializzati gli elementi grafici tramite file JSON, seguendo lo stesso approccio delle altre scene. I pulsanti sono mappati attraverso una struttura dati che associa i nomi degli elementi a interi, rendendo più semplice gestirne lo stato.

La scena gestisce due pulsanti principali: "EXIT_AND_SAVE" e "EXIT_WITHOUT_SAVING". L'input dell'utente viene elaborato tramite il metodo `updateStatus()`, che si occupa della navigazione tra i pulsanti e dell'attivazione delle azioni. In particolare:

- Premendo `Enter` su "EXIT_WITHOUT_SAVING", il gioco ritorna alla scena principale.
- Premendo `Enter` su "EXIT_AND_SAVE", si attiva una procedura di salvataggio.

Il salvataggio viene effettuato solo se il nome inserito non è già presente tra i file nella directory `resources/saves`. In caso contrario, l'utente riceve un messaggio che lo informa del conflitto.
L'inserimento del nome avviene tramite il metodo `handleTextInput()`, che gestisce lettere, numeri, spazi e il tasto `Backspace`. Il nome inserito è limitato a una lunghezza massima, per evitare problemi di visualizzazione o salvataggio.
La classe interagisce con il sistema di salvataggio e con il motore grafico, aggiornando dinamicamente i contenuti a schermo tramite la classe `SceneSaveView`. Tutti gli elementi grafici vengono inseriti in un `GraphicElementsRegistry` temporaneo e poi utilizzati per il rendering.
Questa scena rappresenta un buon esempio di coordinazione tra logica di gioco, gestione dell'input utente e flessibilità grafica. È stata progettata per essere facilmente estendibile e integrabile con il resto dell'architettura.

Gestione dell'acquisto ed utilizzo di items

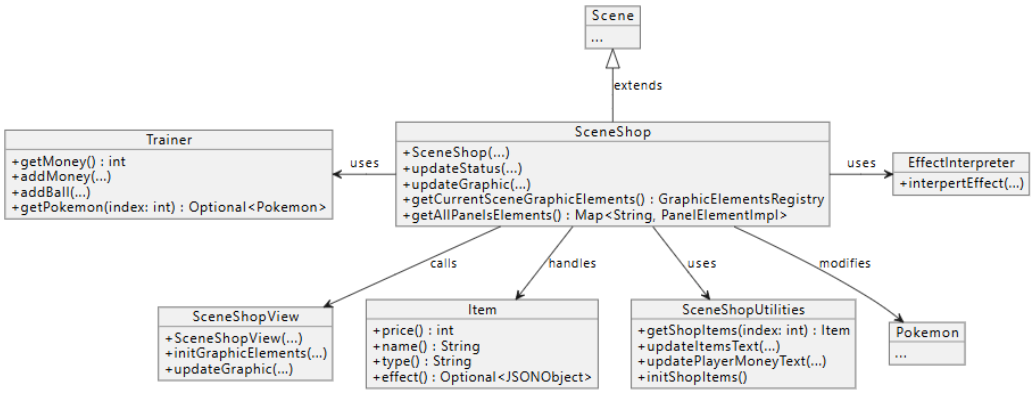


Figura 2.9: Schema UML che raffigura la scena dello shop.

Problema Era necessario creare un'interfaccia interattiva che permettesse al giocatore di acquistare, usare oggetti e gestire il proprio team Pokémon durante la fase di "Shop", garantendo navigazione fluida e feedback coerente.

Soluzione Per affrontare la complessità derivante dalla gestione di molteplici elementi interattivi con comportamenti diversi, la progettazione di `SceneShop` si è basata su alcuni principi fondamentali: separazione della logica, astrazione delle interazioni utente e modularità della visualizzazione.
Il primo passo è stato quello di rappresentare ogni elemento grafico (pulsanti per oggetti, comandi, squadra) tramite un nome simbolico associato a un intero. Questa associazione è memorizzata all'interno di una mappa (`graphicElementNameToInt`), che permette una navigazione efficiente e flessibile, evitando la necessità di hardcodare indici numerici. In questo modo, il codice resta facilmente adattabile a eventuali modifiche nella disposizione o nel numero di pulsanti.
Un'altra scelta fondamentale è stata quella di distinguere chiaramente tra lo stato corrente del cursore (`currentSelectedButton`) e quello verso cui si sta navigando (`newSelectedButton`). Questa distinzione ha permesso di implementare meccaniche di evidenziazione visiva e di aggiornamento progressivo dello stato, consentendo ad esempio animazioni o transizioni fluide nel rendering.
La logica di navigazione (gestita in `updateStatus()`) è strutturata tramite controlli condizionali che verificano in quale "zona" del negozio si trovi l'utente: oggetti gratuiti, oggetti acquistabili, pulsanti speciali, o selezione dei Pokémon. Per ciascuna zona vengono applicate regole di spostamento coerenti: ad esempio, nel caso dei Pokémon, la navigazione avviene in senso orizzontale, mentre per i pulsanti oggetto è verticale.
L'acquisto e l'uso degli oggetti sono stati separati in due momenti logici. Alla pressione di `Enter` su un oggetto acquistabile, viene verificata la disponibilità di denaro e, in caso positivo, l'oggetto viene segnato come "in attesa di utilizzo" (`selectedItemForUse = true`) e si apre la schermata di selezione del Pokémon. Una volta che il giocatore seleziona un Pokémon, viene invocato `applyItemToPokemon()`, che interpreta l'effetto dell'oggetto tramite l'`EffectInterpreter` e lo applica al soggetto scelto.
Nel caso in cui il giocatore decida di tornare indietro senza applicare l'oggetto, viene attivata una procedura di `compensazione` (metodo `compensation()`), che rimborsa il denaro speso. Questo garantisce che nessuna risorsa venga persa a causa di input accidentali o ripensamenti, migliorando l'esperienza utente e la robustezza del sistema.
Il `reroll` degli oggetti è stato trattato come operazione distinta, con un costo

fisso. La sua logica è incapsulata in un metodo dedicato (`rerollShopItems()`), che aggiorna le descrizioni degli oggetti a schermo e scala la quantità di denaro del giocatore. Questo approccio favorisce la riusabilità del codice in altri contesti futuri (ad esempio, mercati diversi o oggetti temporanei).
Per la parte grafica, la classe delega il disegno degli elementi a `SceneShopView`, seguendo il principio di separazione tra logica e presentazione. Ogni aggiornamento visivo è guidato da un registro di elementi (`GraphicElementsRegistry`) che viene aggiornato dinamicamente in risposta alle azioni dell'utente.
Infine, l'intero design della scena è stato pensato per essere facilmente estendibile: l'aggiunta di nuovi oggetti, comandi, o tipologie di interazioni può avvenire con interventi minimi sulla logica principale, mantenendo la compatibilità con il motore grafico e con le altre componenti del gioco.

Creazione degli items

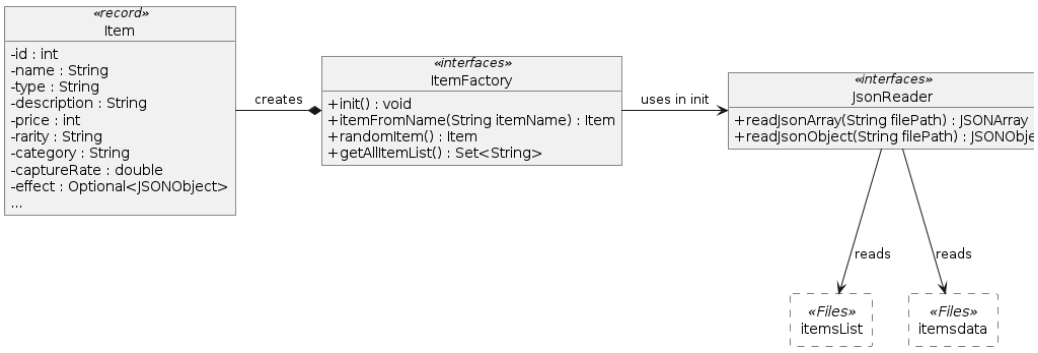


Figura 2.10: Schema UML che raffigura l'itemFactory.

Problema Nel contesto del gioco, è necessario gestire una grande quantità di oggetti con caratteristiche diverse, definite esternamente in file JSON. È importante garantire un accesso rapido e sicuro a queste informazioni, evitando letture ridondanti da disco e potenziali incoerenze. Inoltre, si desidera poter accedere agli oggetti sia per nome che in maniera casuale, ad esempio durante la generazione del negozio o il drop di premi.

Soluzione La soluzione adottata consiste nell'utilizzare un approccio centralizzato tramite la classe `ItemFactory`, che funge da punto unico di accesso e gestione per tutti gli oggetti. La fase di inizializzazione (`init()`) legge in anticipo tutti i file JSON e popola una mappa contenente oggetti immutabili. Questa scelta consente:

- di evitare letture ripetute da disco durante il gioco, migliorando l'efficienza;
- di garantire che ogni oggetto esista in una sola versione coerente (immutabilità del record `Item`);
- di semplificare l'accesso e la gestione degli oggetti tramite nomi simbolici.

L'utilizzo di `Optional<JSONObject>` per rappresentare l'effetto consente di gestire in modo elegante la presenza o assenza di un comportamento speciale associato all'oggetto. Inoltre, la funzione `randomItem()` sfrutta un `Set` e uno `Stream` per selezionare un oggetto casuale, mantenendo l'implementazione semplice ma efficace.
L'intero design è facilmente estendibile: per aggiungere un nuovo oggetto, è sufficiente aggiornare il file `itemsList.json` e fornire il relativo file `.json` con le specifiche. Questo approccio favorisce la modularità, la scalabilità e la manutenzione del progetto.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

In questo progetto abbiamo deciso di testare le principali classi del Model utilizzando la suite *JUnit*.

Riportiamo brevemente i componenti che abbiamo sottoposto a test automatizzato:

- `MoveFactory`: viene verificata la corretta creazione delle mosse, la loro potenza, precisione, tipo e PP.
- `AbilityFactory`: viene testata la corretta istanziazione delle abilità, verificando che le condizioni di attivazione siano correttamente interpretate.
- `Move copy`: viene verificato che oggetti `Move` uguali siano distinti in memoria e che modifiche su uno non influenzino l'altro.
- `EffectInterpreter su mosse`: viene testata l'interpretazione degli effetti delle mosse a partire da un file JSON.
- `EffectInterpreter su abilità`: viene testata l'applicazione corretta degli effetti delle abilità da file JSON.
- `PokeEffectivenessCalc`: viene verificata la correttezza del calcolo dell'efficacia degli attacchi tra diversi tipi di Pokémon.
- `EffectInterpreter su oggetti`: viene verificata l'applicazione corretta degli effetti degli oggetti su un Pokémon e sul trainer.
- `PPItemsTest`: viene testato il corretto funzionamento dell'oggetto `Elixir`, assicurandosi che ripristini tutti i PP delle mosse.

- `AI`: viene testata la logica decisionale dell'intelligenza artificiale, verificando che venga scelta l'azione più coerente in funzione del contesto.
- `BattleRewards`: viene verificato che, al termine della battaglia, l'esperienza venga correttamente assegnata al Pokémon del vincitore.
- `GenerateEnemyInstance`: viene verificata la corretta generazione di un avversario con un numero coerente di Pokémon nella squadra.
- `BattleEngine con decisione NOTHING`: viene testato il comportamento del motore di battaglia quando il giocatore non compie alcuna azione.
- `BattleEngine con Pokéball`: viene testata la cattura di un Pokémon avversario utilizzando una Masterball, controllando che entri nella squadra del giocatore.
- `BattleEngine con cambio Pokémon`: viene verificata la corretta gestione del cambio Pokémon durante una battaglia.
- `GameEngine - Menu principale`: vengono testate le interazioni del giocatore con il menu principale tramite input da tastiera.
- `GameEngine - Shop`: viene verificata la corretta navigazione grafica all'interno del negozio.
- `GameEngine - Salvataggio`: viene testato il comportamento dell'interfaccia di salvataggio e caricamento del gioco.
- `GameEngine - Schermata di battaglia`: vengono testate le scelte possibili durante una schermata di combattimento.
- `GameEngine - Box Pokémon`: viene verificata la navigazione nel box dei Pokémon catturati.
- `GameEngine - Schermata info Pokémon`: viene testata la corretta selezione iniziale e navigazione nella schermata informativa del Pokémon.

3.2 Note di sviluppo

3.2.1 Tverdohleb Egor

Creazione di una classe Generica `Range<T>`

Utilizza per tutti i valori da mantenere dentro un range, in particolare in Pokemon Permalink: `https://github.com/PietroMaro/OOP24-Poke-Rogue/`

blob/4db76a3d9340507e382e7b99884116cdded2243f/src/main/java/it/unibo/pokerogue/model/impl/RangeImpl.java

Utilizzo di Jexl3

Utilizzato dall'effect interpreter come effettivo interprete Permalink: https://github.com/PietroMaro/OOP24-Poke-Rogue/blob/4db76a3d9340507e382e7b99884116cdde src/main/java/it/unibo/pokerogue/controller/impl/EffectInterpreter.java

Utilizzo di JSONArray e JSONObject

Utilizzato ampiemamente tra gestione degli effect e soprattuto dal JSON reader Permalink: https://github.com/PietroMaro/OOP24-Poke-Rogue/blob/4db76a3d9340507e382e7b99884116cdded2243f/src/main/java/it/unibo/pokerogue/model/impl/SavingSystemImpl.java

Utilizzo di Optional

Utilizzato per la gestione dei Pokemon, delle mosse e del Weather. Permalink: https://github.com/PietroMaro/OOP24-Poke-Rogue/blob/4db76a3d9340507e382e7b998841 src/main/java/it/unibo/pokerogue/controller/impl/EffectInterpreter.java

Utilizzo di lambda expressions

Utilizzato nel SavingSystem per filtrare. Permalink: https://github.com/PietroMaro/OOP24-Poke-Rogue/blob/4db76a3d9340507e382e7b99884116cdded2243f/src/main/java/it/unibo/pokerogue/model/impl/SavingSystemImpl.java

3.2.2 Maretti Pietro

Utilizzo di JSONArray e JSONObject

Utilizzato in vaire classi di utility e nella Scene. Permalink: https://github.com/PietroMaro/OOP24-Poke-Rogue/blob/4db76a3d9340507e382e7b99884116cdded2 src/main/java/it/unibo/pokerogue/utilities/PokeEffectivenessCalc.java

Utilizzo di `Optional`

Utilizzato per tutti i valori opzionali in Pokemon, Abilità, Mosse, Weather. Permalink: `https://github.com/PietroMaro/OOP24-Poke-Rogue/blob/4db76a3d9340507e382e7b99884116cdded2243f/src/main/java/it/unibo/pokerogue/controller/impl/ai/EnemyAiImpl.java`

Utilizzo delle libreria esterna `Lombok`

Nel progetto è stata utilizzata la libreria esterna `Lombok` in vari punti per ottenere Getter e Setter Permalink: `https://github.com/PietroMaro/OOP24-Poke-Rogue/blob/4db76a3d9340507e382e7b99884116cdded2243f/src/main/java/it/unibo/pokerogue/model/impl/graphic/BoxElementImpl.java`

3.2.3 Miraglia Tommaso Cosimo

Utilizzo di `Optional`

Utilizzato per tutti i valori opzionali nel `BattleEngine` come ad esempio le Mosse permalink: `https://github.com/PietroMaro/OOP24-Poke-Rogue/blob/4db76a3d9340507e382e7b99884116cdded2243f/src/main/java/it/unibo/pokerogue/controller/impl/scene/fight/BattleEngineImpl.java`

Utilizzo di `lambda expressions`

Utilizzato in alcune classi per semplficare controlli permalink: `https://github.com/PietroMaro/OOP24-Poke-Rogue/blob/4db76a3d9340507e382e7b99884116cdded2src/main/java/it/unibo/pokerogue/utilities/BattleUtilities.java`

Utilizzo delle libreria esterna `Lombok`

Un esempio significativo è il campo `currentWeather`, utilizzato per ottenere il meteo durante la partita, annotato con `@Getter`. permalink: `https://github.com/PietroMaro/OOP24-Poke-Rogue/blob/4db76a3d9340507e382e7b99884116cdded2src/main/java/it/unibo/pokerogue/controller/impl/scene/fight/BattleEngineImpl.java`

3.2.4 Casadio Alex

Utilizzo di `JSONArray` e `JSONObject`

utilizzato in itemFactory
permalink: `https://github.com/PietroMaro/OOP24-Poke-Rogue/blob/4db76a3d9340507e382`
`src/main/java/it/unibo/pokerogue/model/impl/item/ItemFactory.java`

Utilizzo di `Optional`

Utilizzato per tutti i valori opzionali in Pokemon nelle utilities per la scene shop
permalink: `https://github.com/PietroMaro/OOP24-Poke-Rogue/blob/4db76a3d9340507e382`
`src/main/java/it/unibo/pokerogue/controller/impl/scene/SceneShop.java`

Utilizzo delle libreria esterna `Lombok`

Nel progetto è stata utilizzata la libreria esterna `Lombok` in vari punti per ottenere Getter e Setter
permalink: `https://github.com/PietroMaro/OOP24-Poke-Rogue/blob/4db76a3d9340507e382`
`src/main/java/it/unibo/pokerogue/controller/impl/scene/SceneShop.java`

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Miraglia Cosimo Tommaso

Nel complesso, ho trovato questa esperienza stimolante e formativa. Essendo il mio primo progetto di gruppo di questa portata, ho potuto confrontarmi con sfide concrete legate sia alla parte tecnica che alle dinamiche collaborative. Mi ha fatto piacere contribuire attivamente alla realizzazione del lavoro e ho imparato molto, sia in termini pratici che sul piano dell'organizzazione. Detto questo, riflettendo a posteriori sul progetto, non posso non sollevare alcune considerazioni critiche riguardo ad alcune scelte iniziali. In particolare, la decisione di sviluppare un motore grafico interno, sebbene interessante in teoria, mi è sembrata poco funzionale rispetto agli obiettivi reali del lavoro. Anche il forte orientamento verso una riproduzione quanto più fedele possibile del gioco originale ha finito per rallentare il lavoro, spostando l'attenzione su dettagli minori e sacrificando l'efficacia e la solidità complessiva del progetto. Mi dispiace molto non essere riuscito a soddisfare le aspettative di un altro membro del gruppo, soprattutto considerando il tempo e l'impegno che ho dedicato. Ho cercato di contribuire in modo costante e responsabile, pur partendo da un livello di esperienza iniziale limitato. È stato sinceramente frustrante vedere che questo sforzo non sia stato riconosciuto, in un contesto che, idealmente, avrebbe dovuto valorizzare la collaborazione e la crescita condivisa. Nonostante tutto, considero questo progetto una tappa importante per la mia formazione. Ho acquisito consapevolezza non solo su aspetti tecnici, ma anche sull'importanza della pianificazione, delle scelte strategiche e del confronto costruttivo all'interno di un gruppo di lavoro. Per quanto riguarda eventuali sviluppi futuri del progetto, personalmente non sono particolarmente motivato a proseguirne l'evoluzione. Ritengo che

le basi da cui siamo partiti non siano sufficientemente solide per giustificare un investimento ulteriore di tempo, e preferirei dedicarmi a nuove idee dove possa applicare quanto ho imparato.

4.1.2 Egor Tverdohleb

Ho trovato quest'esperienza un'ottima palestra per la progettazione e l'implementazione di un sistema tutto sommato complesso, e ho particolarmente apprezzato la sua stretta vicinanza con dinamiche che riscontro anche nella mia esperienza lavorativa. Nel complesso mi ritengo sufficientemente soddisfatto del risultato da me prodotto, e considero molto preziosi gli insegnamenti tratti dagli errori di progettazione e implementazione, inevitabili data l'inesperienza. Sono invece piuttosto rammaricato per il risultato finale complessivo: il grande impegno di parte del gruppo non è stato minimamente eguagliato dall'altra metà, che, essendo anche meno esperta, ha purtroppo influito negativamente sulla qualità generale del progetto. Questo mi porta, con dispiacere, a non essere interessato a un'evoluzione futura del lavoro, né a considerarlo una rappresentazione fedele delle mie reali capacità

4.1.3 Maretti Pietro

Ritengo che questa esperienza abbia rappresentato un'importante occasione di crescita per le mie competenze nello sviluppo di videogiochi. Non avevo mai affrontato un progetto di queste dimensioni né collaborato per un periodo così prolungato con un team. Sono generalmente soddisfatto del mio contributo, nonostante alcune difficoltà, soprattutto nella gestione della grafica e nelle fasi iniziali di progettazione. Nonostante ciò, credo di aver dato un apporto significativo, intervenendo in parti del Control, del Model e, più del previsto, anche nella View. Le scene che ho realizzato non sono le più complesse, ma mi hanno permesso di costruire un piccolo ecosistema di interazioni tra Menu, Load e Box, che considero riuscito. L'aspetto meno soddisfacente è stato dover apportare modifiche non trascurabili al codice in corso d'opera, a causa di scelte progettuali iniziali non sempre efficaci; tuttavia, anche queste difficoltà mi hanno permesso di imparare a gestire situazioni simili in futuro. Il principale punto critico è stata l'organizzazione, condizionata da una gestione del tempo non ottimale, che ha prolungato lo sviluppo oltre quanto previsto. Nel complesso, però, questa esperienza mi ha reso più consapevole e preparato: oggi affronterei la creazione di un videogioco con maggiore sicurezza e metodo rispetto a prima di questo progetto. In futuro penso che il gioco necessiterebbe di una modifica all'UI per rendere il tutto più comprensibile e godibile. Mentre dal punto di vista delle

meccaniche si potrebbe lavorare verso una maggiore fedeltà rispetto ai giochi pokèmon, dato che alcune di esse sono state tagliate per non appesantire troppo il carico di lavoro.

4.1.4 Casadio Alex

Questa esperienza ha rappresentato per me un'importante opportunità di crescita, essendo il mio primo progetto di queste dimensioni. Ho affrontato molte sfide nuove, sia tecniche che collaborative, che mi hanno permesso di imparare molto e di migliorare sensibilmente il mio approccio alla programmazione. Col senno di poi, mi sono pentito di aver assecondato la scelta di utilizzare un motore grafico. Allo stesso modo, ho trovato poco produttiva l'insistenza nel voler replicare in maniera eccessivamente fedele il gioco originale, quando soluzioni più semplici e creative avrebbero probabilmente portato a un risultato migliore. Dal punto di vista umano, non posso nascondere il dispiacere per alcuni momenti difficili all'interno del gruppo: ho infatti ricevuto critiche nei confronti del mio codice ,da parte di un membro del gruppo, espresse in modo offensivo, senza il giusto riconoscimento del mio percorso e della mia minore esperienza rispetto ad altri membri. Questo ha reso il lavoro di squadra a tratti poco sereno. Nonostante tutto, considero l'esperienza estremamente formativa: ho imparato dai miei errori, ho acquisito consapevolezza sui miei limiti e punti di forza, e porto con me molti insegnamenti utili per affrontare progetti futuri con maggiore maturità.

Appendice A

Guida utente

A.1 Inizio del gioco

In tutte le schermate, l'utente può utilizzare i tasti freccia **su** e **giù** per spostarsi tra le opzioni disponibili. Una volta selezionata l'opzione desiderata, è sufficiente premere il tasto **Invio** per confermare. L'uso del mouse non è supportato.

Figura A.1: Schermata principale del gioco al lancio dell'applicazione

Dopo aver aperto l'applicativo, ci si troverà nella schermata principale (Figura A.1). Questa schermata presenta tre opzioni principali:

- **Continue**: consente di riprendere una partita precedentemente salvata;
- **New Game**: avvia una nuova partita da zero;
- **Info**: mostra informazioni aggiuntive sul gioco.

Per iniziare una nuova partita, selezionare l'opzione `New Game` e premere Invio. Se si desidera esplorare i salvataggi disponibili, ad esempio per sperimentare il funzionamento di elementi specifici come il box oppure per riprendere una sessione precedente, è possibile accedere all'elenco dei salvataggi attraverso l'opzione `Load Game`, disponibile una volta selezionata `Continue`.

A.1 Panoramica del Pokémon Selezionato

Prima di iniziare il round, viene sempre visualizzata una schermata riepilogativa del Pokémon selezionato dall'utente (Figura A.2), contenente i dettagli principali delle sue caratteristiche.

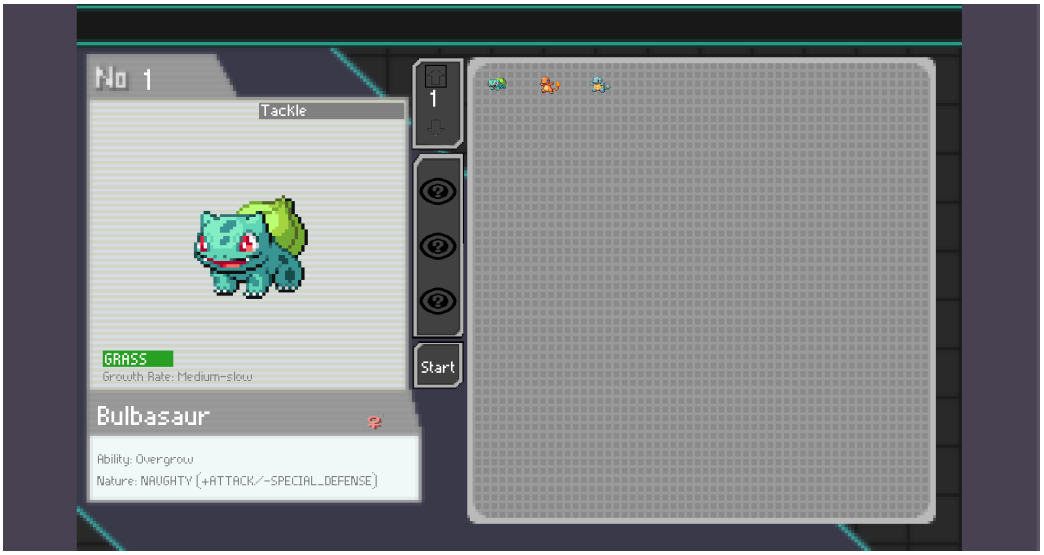


Figura A.2: Panoramica del Pokémon selezionato prima dell'inizio del round

Nel **pannello a sinistra** vengono mostrate le informazioni relative al Pokémon selezionato, come il nome, il tipo, l'abilità e la mossa attualmente assegnata.
Nel **pannello a destra**, in alto, sono visibili i primi tre Pokémon disponibili. Nel caso si carichi un salvataggio in cui sono stati catturati altri Pokémon

durante partite precedenti, anche questi verranno mostrati e potranno essere selezionati.
Una volta selezionato almeno un Pokémon, sarà possibile premere il pulsante `Start` per avviare il gioco.

A.2 Fase di Combattimento

Una volta avviato il round, l'utente accede alla fase di combattimento, durante la quale i Pokémon si affrontano.

Figura A.3: Visualizzazione della fase di combattimento

A destra sono disponibili quattro opzioni, rappresentate da pulsanti rettangolari:

- **FIGHT**: per attaccare utilizzando una delle mosse del Pokémon. Inizialmente ogni Pokémon avrà a disposizione una mossa, ma con il crescere del livello se ne sbloccheranno di nuove;
- **BALL**: per tentare la cattura del Pokémon avversario, da tenere conto che se si starà affrontando un Trainer e non un Pokémon selvatico non sarà possibile catturarlo, infatti si potrà notare che le PokéBall non diminuiranno;
- **POKEMON**: per cambiare il Pokémon attivo con un altro del proprio team;

- **RUN**: per saltare il turno.

A.3 Shop

Dopo la fase di combattimento, il giocatore ha accesso a un negozio in cui può acquistare oggetti utili per i round successivi.

Figura A.4: Visualizzazione della schermata dello shop.

Nella **parte superiore destra** è visualizzato il totale di denaro a disposizione del giocatore, mentre al centro sono disposti i vari oggetti acquistabili, ciascuno rappresentato da il nome dell'oggetto e il relativo prezzo. Gli oggetti sono divisi in gratuiti e a pagamento (presentano il prezzo), gli oggetti a pagamento sono acquistabili solo se si hanno abbstanza soldi.
Nella **parte inferiore sinistra** è presente un pulsante "`REROLL`" che permette di aggiornare casualmente gli oggetti in vendita al costo di 50 monete.
In basso a destra, un pulsante "`TEAM`" consente di accedere alla schermata del proprio team.
Infine, nella **parte inferiore centrale** si trova un riquadro descrittivo che mostra informazioni dettagliate sull'oggetto attualmente selezionato, aiutando il giocatore nella scelta d'acquisto.