UNIVERSITÀ DEGLI STUDI FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Elaborato di Ingegneria del Software

Autore:
Borghini Cosimo

N° Matricola:
7010926

Corso principale:
Ingegneria del Software

Docente corso:
Enrico Vicario

Indice

Introduzione

Il progetto in questione ha avuto come obiettivo la realizzazione delle fondamenta di un Role-Play Game (RPG) ispirato ai classici giochi della serie Pokémon. L'intento principale è stato quello di costruire una struttura di base che, attraverso sviluppi successivi, possa evolvere in un'esperienza videoludica completa. Tale esperienza deve essere in grado di rispettare i canoni tipici di un buon gioco di ruolo: offrire al giocatore la possibilità di immergersi in un mondo di fantasia, popolato da personaggi unici e da sfide stimolanti, all'interno di una narrazione coinvolgente e memorabile. L'obiettivo ultimo è quello di rendere il giocatore parte integrante della storia, trasformando le sue scelte e interazioni in elementi fondamentali dell'avventura.

Obiettivi del Progetto

Per raggiungere questo risultato, si è lavorato con particolare attenzione alla progettazione e implementazione delle seguenti funzionalità, considerate essenziali per un RPG di qualità:

- **Gestione dei dialoghi**: è stata implementata una struttura che utilizza file in formato `.xml` per consentire una gestione modulare e flessibile dei dialoghi tra i personaggi. Questo approccio garantisce la possibilità di ampliare facilmente la narrativa e aggiungere nuovi personaggi o eventi.
- **Gestione dei salvataggi**: è stato scelto il formato `.json` per memorizzare i progressi del giocatore. Tale scelta consente una serializzazione dei dati semplice ed efficace, facilitando sia il recupero che la modifica dei salvataggi.
- **Gestione delle mappe**: la configurazione delle mappe è stata affidata a file in formato `.txt`. Questa soluzione minimalista e leggibile offre una base semplice ma estendibile per la creazione di ambientazioni di gioco.
- **Gestione degli oggetti animati**: sono stati implementati strumenti per animare oggetti e personaggi presenti nelle mappe, aumentando il livello di interattività e rendendo il mondo di gioco più vivo e coinvolgente.

Scelte

Pur rifacendosi ai canoni estetici e ludici dei giochi Pokémon, si è scelto di adottare un approccio differente rispetto alla meccanica principale del combattimento. Nel nostro progetto, la lotta è stata abbandonata per lasciare spazio a una dinamica di gioco più incentrata sulla cattura dei mostriciattoli, ispirandosi alla cosiddetta "modalità safari" presente nei giochi originali. Questa scelta ha portato a una ridefinizione delle priorità del gameplay, rendendo centrale una meccanica che in Pokémon è secondaria.

La dinamica di cattura è stata notevolmente ampliata e arricchita:

- **Parco oggetti ampliato**: il giocatore ha accesso a un'ampia varietà di strumenti, ciascuno con un ruolo specifico, che permette di interagire con i mostriciattoli in modi differenti.
- **Comportamenti diversificati**: ogni specie di Pokémon è stata dotata di un comportamento unico, che varia in base agli oggetti utilizzati dal giocatore. Questo introduce un livello di strategia e varietà nell'approccio alla cattura, rendendo ogni incontro unico.

Design del Gioco ed Esplorazione

Nel prototipo attuale, l'avventura del giocatore si apre senza alcuna spiegazione esplicita riguardo al mondo di gioco o alle sue meccaniche. Questa scelta è stata deliberata, in quanto si è voluto dare grande importanza al senso di scoperta e di esplorazione. L'obiettivo è quello di stimolare la curiosità del giocatore, che dovrà orientarsi nel mondo di gioco, esplorare la mappa e interagire con i personaggi e l'ambiente per comprendere le regole che lo governano.

La mappa test realizzata per il prototipo rappresenta un'area limitata, ma funzionale per introdurre il giocatore alle dinamiche principali. Durante l'esplorazione, il giocatore incontrerà un personaggio guida, che gli offrirà una prima introduzione alla ricerca dei mostriciattoli, fornendogli alcuni strumenti di base per la cattura e una lista iniziale di quattro esemplari da individuare.

Questi Pokémon, accessibili tramite l'inventario, rappresentano un obiettivo chiaro per il giocatore, introducendo in maniera graduale le principali meccaniche di gioco.

Librerie Utilizzate

Per lo sviluppo del progetto sono state utilizzate le seguenti librerie:

- **GDX**: È stata utilizzata la libreria libGDX, una framework open-source per lo sviluppo di giochi 2D e 3D. libGDX offre una serie di strumenti per gestire grafica, input, audio e logica di gioco, rendendo facile la gestione delle risorse e l'implementazione delle funzionalità di gioco.
- **JSON**: Il formato JSON è stato scelto per la gestione dei dati di salvataggio, in quanto fornisce un metodo semplice ed efficace per serializzare oggetti complessi. È stato utilizzato per memorizzare lo stato del gioco, come la posizione del giocatore, l'inventario, e altre informazioni cruciali, permettendo un caricamento e salvataggio rapido e flessibile.

1 Schermate iniziali

In questo primo capitolo viene esaminato il funzionamento delle schermate iniziali. In particolare, vengono considerate due schermate: *Press X Screen* e *Menu Screen*. La schermata *Press X* ha lo scopo di avviare il gioco, mentre il *Menu Screen* consente di navigare tra le opzioni di gioco.

1.1 Schermata di Pressione del Tasto X

La *Press X Screen* è una schermata che serve a far capire all'utente quale tasto premere per continuare. In questa schermata, viene visualizzato un testo che invita a premere il tasto "X". Il tasto `X` è quindi il principale punto di interazione dell'utente con il gioco. La schermata utilizza un'animazione di fade per rendere il testo più dinamico, attirando l'attenzione dell'utente e creando una transizione visiva.

1.1.1 Funzionamento

Nel ciclo di rendering della schermata, viene eseguita una pulizia dello schermo, seguita dal disegno dello sfondo e del testo. Il testo viene disegnato con un effetto di opacità variabile, che cambia in base a un'animazione di fade. Il testo `"Press X"` viene posizionato nella parte inferiore dello schermo e il suo colore cambia dinamicamente per creare un effetto visivo interessante.

1.1.2 Interazione con l'utente

Quando viene premuto il tasto `X`, il gioco passa alla schermata successiva. Se il tasto viene premuto, il gioco esegue una transizione e cambia schermata. Questo passaggio viene realizzato tramite la classe `TransitionScreen`, che gestisce la transizione visiva tra la schermata corrente e la successiva.

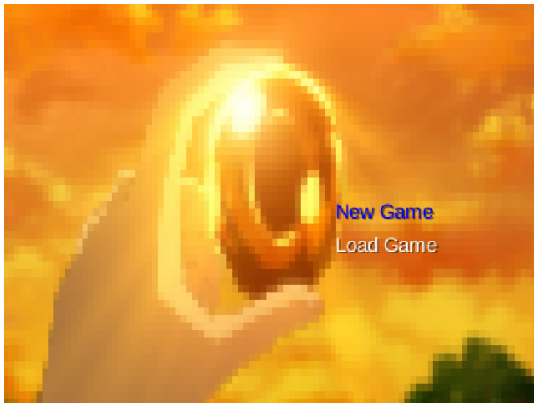

Figura 1: Menu di Gioco

1.2 Menu di Gioco

Il *Menu Screen* è il punto centrale dove l'utente può selezionare l'opzione desiderata per continuare. Il menu offre due opzioni principali: *New Game* e *Load Game*. L'utente può navigare tra queste opzioni utilizzando le frecce direzionali `UP` e `DOWN`, e confermare la selezione con il tasto `X`.

1.2.1 Navigazione nel Menu

Nel menu, l'utente può selezionare un'opzione usando le frecce direzionali `UP` e `DOWN`. Quando viene premuta la freccia `UP`, l'indice della selezione viene decrementato, mentre premendo la freccia `DOWN`, l'indice viene incrementato. Dopo aver selezionato un'opzione, l'utente può confermare la scelta premendo il tasto `X`.

Il sistema di selezione è ciclico, il che significa che se l'utente raggiunge l'ultima opzione e preme la freccia giù, verrà selezionata la prima opzione, e viceversa.

1.2.2 Comandi di Gioco

L'interazione con il menu è facilitata dall'uso delle frecce direzionali e del tasto `X`. L'utente può quindi navigare facilmente tra le opzioni e avviare un nuovo gioco o caricare una partita salvata. La selezione dell'opzione avviene tramite la pressione del tasto `X`, che avvia il processo di transizione verso la schermata successiva, come la schermata del gioco o il caricamento di un salvataggio.

1.3 Caricamento dei Dati da File di Salvataggio

Quando viene selezionata l'opzione *Load Game*, il gioco carica i dati da un file di salvataggio. Questo file contiene informazioni cruciali sullo stato del gioco, tra cui la posizione del giocatore, l'inventario, gli oggetti raccolti, e altri dettagli relativi alla mappa e agli NPC.

1.3.1 Funzionamento del Caricamento

Il processo di caricamento viene gestito dalla classe `SaveManager`, che si occupa della lettura e del ripristino dei dati salvati. Quando l'utente sceglie di caricare un gioco e preme il tasto `X`, il gioco tenta di aprire il file di salvataggio. Se il file esiste e contiene dati validi, il gioco ripristina lo stato precedente, caricando la mappa, il giocatore, e tutti gli oggetti e NPC relativi alla sessione salvata. In caso contrario, se il salvataggio non è presente o è corrotto, viene caricato un salvataggio di stato iniziale, con i valori di default impostati per la mappa e il giocatore. Per caricare la mappa di gioco si sfrutta una classe maploader che, preso il nome delle mappa, traduce il file di testo relativo in un mondo di gioco, associando a ogni elemento del testo un oggetto specifico.

lab.txt

```
TERRAIN:
F F F F F F F F F
F F F F F F F F F
F F F F F F F F F
F F F F F F F F F
F F F F F F F F F
OBJECTS:
wall 0 0
wall 1 0
wall 2 0
wall 3 0
wall 4 0
wall 5 0
wall 6 0
wall 7 0
wall 8 0
rug 3 0
teleport 4 0 beach 7 10
NPCS:
professor 4 3
```

1.3.2 Struttura del Salvataggio

Il file di salvataggio è strutturato in formato JSON, il che consente una gestione semplice e flessibile dei dati. La classe `SaveData` organizza i dati rilevanti, come la mappa corrente, la posizione del giocatore, l'inventario, e altre informazioni specifiche come lo stato degli oggetti nel mondo di gioco. Quando il gioco viene caricato, i dati vengono letti dal file JSON e mappati in oggetti, ripristinando così lo stato precedente della sessione. Questo processo include anche il recupero di tutti gli oggetti e NPC presenti nella mappa, permettendo al giocatore di riprendere l'esperienza da dove l'aveva interrotta.

1.3.3 Caricamento degli Asset

Oltre ai dati di gioco, quando viene caricato un salvataggio, vengono caricati anche gli asset necessari per visualizzare correttamente il gioco. Questi asset comprendono texture, animazioni, e altre risorse grafiche. La classe `AssetManager` gestisce il caricamento di tutti i file richiesti, come le texture

degli sprite, le animazioni del giocatore e degli NPC, e gli oggetti della mappa. Solo dopo che tutti gli asset sono stati correttamente caricati, il gioco viene avviato, garantendo che tutte le risorse siano disponibili per il rendering corretto dell'ambiente di gioco. Il processo di caricamento degli asset è ottimizzato per ridurre i tempi di attesa, ma è comunque essenziale per garantire che il gioco riprenda con tutte le risorse necessarie già pronte all'uso.

savegame.json

```
{
  "mapFileName": "assets/maps/beach.txt",
  "playerData": {
    "x": 7,
    "y": 8,
    "inventory": {
      "BasicTrap": 2,
      "GreatBall": 3,
      "FloralPerfume": 1,
      "StandardBait": 4,
      "Pokeball": 1
    },
    "team": [],
    "toCatch": {
      "Slakoth": true,
      "Bellossom": false,
      "Parasect": true,
      "Exeggutor": true
    },
    "hasInventory": true
  }
}
```

2 Schermata di Gioco e Inventario

2.1 Game Screen: Implementazione della Schermata Principale

Figura 2: Screenshot di Gioco

2.1.1 Sistema di Caricamento delle Risorse

La gestione delle risorse della schermata principale si realizza attraverso un sistema centralizzato che opera su diverse tipologie di asset. Per quanto riguarda gli sprite degli attori, si effettua il caricamento di atlas specializzati: abbiamo infatti tileset diversificati per protagonista, professore e mondo di gioco.

L'interfaccia utente richiede un insieme dedicato di risorse: uipack.atlas fornisce gli elementi grafici di base, mentre il sistema testuale si appoggia su un font tipico dei giochi pokemon. Di particolare importanza risulta l'implementazione del DialogueDb, che attraverso un loader personalizzato gestisce i dialoghi mediante file XML, garantendo una gestione flessibile dei contenuti narrativi.

Dialogo Post Introduzione

```
<!-- Dialogo post introduzione -->
<dialogue name="professor_post_intro">
    <linear id="0" text="Ah, you're back! Let's
    see what you've brought.">
        <pointer target="1"/>
    </linear>
    <linear id="1" text="I'm interested in
    species you haven't caught before!">
        <pointer target="2"/>
    </linear>
    <linear id="2" text="Keep up the good work!"/>
</dialogue>
```

2.1.2 Architettura dei Controller

Il sistema di controllo si articola attraverso tre componenti principali che cooperano per gestire le diverse sfaccettature del gameplay. Il DialogueController assume la responsabilità primaria nella gestione delle conversazioni, controllando sia la visualizzazione che l'avanzamento dei dialoghi, oltre a gestire il flusso delle scelte narrative quando necessario.

L'InteractionController si occupa di mediare le interazioni tra il giocatore e l'ambiente circostante. Si interfaccia costantemente con il sistema di dialoghi e implementa una logica sofisticata per rispondere appropriatamente agli input contestuali del giocatore. La sua implementazione garantisce una risposta fluida e naturale alle azioni dell'utente.

Il PlayerController, invece, si concentra esclusivamente sulla gestione del personaggio principale. Oltre al movimento base, si occupa di gestire il reset dei controlli durante le transizioni tra le diverse schermate e mantiene lo stato corrente del personaggio.

2.1.3 Sistema di Rendering del Mondo

Il rendering del mondo di gioco si basa su un sistema articolato che coinvolge diversi componenti specializzati. La camera implementa un offset preciso per garantire un centraggio ottimale della visuale sul personaggio, aggiornandosi continuamente in base ai suoi movimenti. Il PlaceRenderer gestisce il rendering specifico per ogni tipologia di elemento presente nel mondo di gioco, occupandosi anche del cambio dinamico dell'ambiente quando il giocatore si sposta tra diverse aree.

Il sistema di viewport implementa uno scaling dell'interfaccia con un fattore fisso di 2, bilanciando leggibilità e definizione degli elementi visuali. Particolare attenzione si pone al mantenimento del corretto aspect ratio nelle diverse risoluzioni supportate, garantendo un'esperienza visiva consistente su differenti dispositivi.

2.1.4 Interfaccia Utente di Gioco

L'interfaccia della schermata principale si struttura secondo una gerarchia precisa che parte da una Root Table come contenitore principale. Al suo interno, la DialogueBox, inizialmente nascosta, si posiziona nella parte inferiore dello schermo, pronta ad apparire quando necessario per mostrare conversazioni e informazioni narrative. Complementare ad essa, l'OptionBox si allinea sulla destra dello schermo, attivandosi per presentare le scelte disponibili durante i dialoghi interattivi.

Figura 3: Esempio di dialogo

2.2 Inventory Screen: Implementazione della Schermata Inventario

2.2.1 Struttura Base e Inizializzazione

La schermata dell'inventario si costruisce su un sistema di rendering che combina SpriteBatch per gli elementi generali e ShapeRenderer per la grafica custom. Il sistema implementa una gestione sofisticata dei font, utilizzando due scale distinte: una per il testo standard e una per i titoli, garantendo una chiara gerarchia visiva delle informazioni.

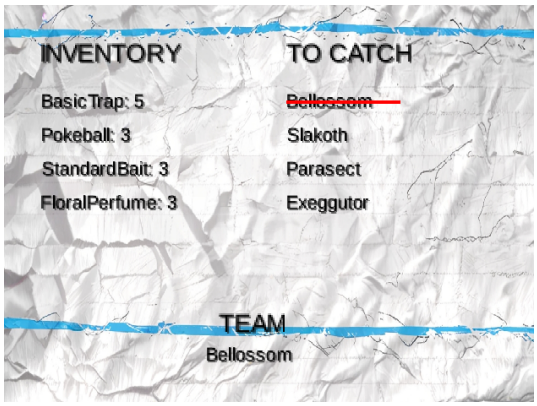

Figura 4: Inventario

Lo sfondo utilizza una texture specifica che si adatta dinamicamente alle dimensioni dello schermo, contribuendo a creare un'atmosfera coerente con il tema del gioco. La gestione del colore dei font si adatta per garantire la massima leggibilità su questo sfondo texture.

2.2.2 Layout delle Sezioni

Il layout della schermata si articola in tre aree distinte, ciascuna con un ruolo specifico nell'organizzazione delle informazioni. La sezione dell'inventario occupa la parte sinistra dello schermo mentre sulla destra si colloca la lista dei Pokémon da catturare. Questa sezione implementa un sistema visivo che utilizza linee rosse per evidenziare i Pokémon catturati, fornendo un feedback immediato sullo stato di completamento della collezione.

La squadra Pokémon trova il suo spazio nella parte inferiore dello schermo, centrata dinamicamente in base al numero di membri presenti.

2.2.3 Gestione degli Input e Transizioni

Il sistema di input dell'inventario si concentra sulla semplicità d'uso, implementando una transizione fluida verso la GameScreen attraverso il tasto Z. Durante questa transizione, si garantisce il corretto salvataggio dello stato corrente per mantenere la coerenza del gioco.

3 Schermata di Cattura

3.1 Architettura della Schermata

La schermata di cattura rappresenta uno dei componenti fondamentali del gioco, implementando l'interfaccia Screen di LibGDX e il pattern Observer attraverso l'interfaccia CaptureObserver. Si caratterizza per una struttura modulare che separa la logica di cattura, il rendering e il controllo degli input in componenti distinti.

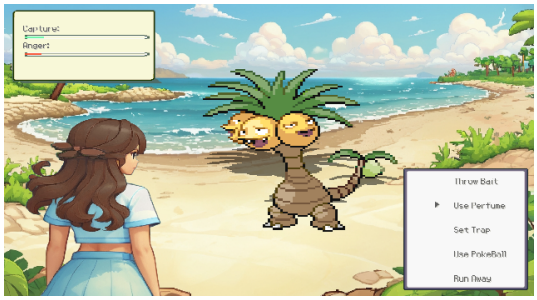

Figura 5: Capture

3.1.1 Gestione degli Stati

Il sistema mantiene diversi stati interni attraverso variabili booleane che controllano il flusso di esecuzione. Lo stato di uscita dalla schermata viene gestito attraverso il flag notExit, mentre l'attesa dell'input dell'utente viene controllata mediante waitingForKeyPress. Un timer dedicato (exitTimer) coordina la transizione tra gli stati, introducendo un ritardo configurabile prima del cambio di schermata.

3.2 Meccaniche di Cattura

Il processo di cattura si articola attraverso diverse meccaniche di gioco:

3.2.1 Sistema delle Esche

Il sistema implementa quattro tipologie di esche: standard, piccante, dolce e maleodorante. Ciascuna esca influenza il comportamento del Pokémon selvatico in maniera differente, modificando la probabilità di cattura e il livello di rabbia della creatura. La gestione delle esche viene coordinata attraverso il metodo handleBait, che elabora l'evento specifico e aggiorna lo stato del gioco di conseguenza.

3.2.2 Sistema dei Profumi

Si distinguono quattro varietà di profumi: floreale, fruttato, erbaceo e mistico. Il sistema dei profumi opera in sinergia con quello delle esche, offrendo ulteriori possibilità di manipolazione delle meccaniche di cattura. L'implementazione prevede una gestione dedicata attraverso il metodo handlePerfume.

3.2.3 Sistema delle Trappole

Le trappole si suddividono in base, avanzata, astuta e rapida. Il sistema delle trappole rappresenta un ulteriore livello di strategia nel processo di cattura, con ciascuna tipologia che offre vantaggi specifici in determinate situazioni. La gestione viene effettuata attraverso il metodo handleTrap.

3.3 Sistema di Eventi

L'architettura implementa un robusto sistema di gestione degli eventi che coordina le diverse fasi della cattura. Gli eventi vengono processati attraverso il metodo update, che gestisce una vasta gamma di situazioni:

3.3.1 Eventi di Cattura

Si distinguono eventi di successo (CAPTURE_SUCCESS), fallimento (CAPTURE_FAIL), fuga del Pokémon (POKEMON_FLED) e stati di rabbia (POKEMON_ANGER). Ogni evento genera una risposta appropriata nel sistema, aggiornando sia lo stato interno che l'interfaccia utente.

3.3.2 Eventi di Utilizzo Strumenti

Il sistema processa gli eventi relativi all'utilizzo di strumenti quali Poké Ball, esche, profumi e trappole. Ciascun evento viene gestito attraverso handler dedicati che aggiornano la logica di gioco e forniscono feedback visivo all'utente.

attemptCapture Method

```
public void attemptCapture(int pokeballIndex) {
    double adjustedCaptureProbability;

    if (pokeballIndex != 3) {
        captureProbability = Math.min(100,
        captureProbability +
currentPokemon.getPokeballEffect(pokeballIndex));
        double angerImpact = angerLevel / 100.0;
        adjustedCaptureProbability =
        captureProbability * (1 - angerImpact);
    } else {
        adjustedCaptureProbability = 100;
    }

    // Verifico se la cattura ha successo o fallisce
    if (Math.random() * 100 < adjustedCaptureProbability) {
        eventNotifier.notifyObservers("CAPTURE_SUCCESS");
    } else {
        eventNotifier.notifyObservers("CAPTURE_FAIL");
        increaseAngerLevel(10);  // Aumento livello di rabbia
        handlePokemonReaction(); // Gestisco la reazione
    }
}
```

3.4 Interfaccia Utente

L'interfaccia utente viene gestita attraverso il CaptureRenderer, che si occupa di visualizzare le informazioni relative alla probabilità di cattura, il livello di rabbia del Pokémon e i messaggi di stato. Il sistema mantiene l'utente costantemente informato sullo stato della cattura attraverso messaggi contestuali e feedback visivo.

4 Implementazione dei Pattern

4.1 State Pattern

L'implementazione del pattern State è stata realizzata utilizzando la libreria LibGDX per gestire il cambiamento dinamico dello stato del gioco e separare i vari comportamenti del gioco in base agli stati. Quando necessario, sono stati creati diversi oggetti che rappresentano gli "stati" del gioco, come lo stato di gioco attivo o lo stato di cattura. Ogni stato è stato implementato come una classe che implementa l'interfaccia `Screen` di LibGDX, permettendo così di alternare le schermate del gioco in modo modulare.

Nel contesto della schermata di gioco (`GameScreen`), è stato implementato un sistema che consente di cambiare tra i diversi stati tramite il controller del giocatore, il controller delle interazioni e il controller del dialogo, gestendo l'input e l'aggiornamento delle variabili di gioco. Quando il gioco passa a uno stato diverso, come durante una battaglia o una cattura, vengono creati nuovi oggetti di stato che interagiscono con il sistema principale.

Nella schermata di cattura, ad esempio, si utilizza una logica simile: un controller di cattura gestisce le azioni del giocatore mentre cattura un Pokémon, cambiando lo stato in base agli eventi come il lancio della Poké Ball o il fallimento della cattura. Ogni evento modifica lo stato e determina quale azione intraprendere, aggiornando l'interfaccia di gioco in tempo reale.

Il pattern State consente così di isolare ciascun stato del gioco (come il gioco in corso, la cattura di un Pokémon o il dialogo con un NPC) in una struttura ben definita, separando la logica di gioco e migliorando la manutenzione e l'estensibilità del codice.

4.2 Singleton Pattern

Nel mio gioco Pokémon, l'implementazione del pattern Singleton è stata realizzata per garantire che alcune classi cruciali abbiano una sola istanza globale, accessibile da qualsiasi parte del gioco. In particolare, la classe `GameState` è stata progettata come Singleton per gestire lo stato generale del gioco, come lo stato del giocatore, la mappa del mondo e la gestione delle risorse, degli eventi e delle schermate.

La classe `GameState` contiene al suo interno vari componenti necessari per il gioco, come il `PlayerState`, il `MapState`, il `ResourceManager`, e il `EventManager`. La gestione di questi oggetti centralizzati permette di avere un punto di riferimento unico e modulare per l'intero gioco. Il costruttore di `GameState` è privato, assicurando che l'istanza venga creata solo una volta, attraverso il metodo statico `getInstance()`.

4.2.1 PlayerState

La classe `PlayerState` è utilizzata per tenere traccia dello stato del giocatore nel gioco, inclusi i dettagli come la posizione e il personaggio controllato. Implementata come parte del Singleton `GameState`, questa classe assicura che lo stato del giocatore sia centralizzato e possa essere facilmente modificato e recuperato in qualsiasi punto del gioco.

Il `PlayerState` gestisce la variabile `player`, che rappresenta il giocatore, e due variabili di posizione `safeX` e `safeY` che memorizzano le coordinate di salvataggio della posizione del giocatore. Questa separazione permette al gioco di gestire il movimento del giocatore e di ripristinare facilmente la sua posizione in caso di necessità, migliorando la gestione della persistenza del gioco. L'adozione del pattern Singleton per `PlayerState` consente di evitare la creazione di più istanze di questa classe, centralizzando il controllo del personaggio in un unico punto.

4.2.2 MapState

La classe `MapState` è responsabile della gestione delle informazioni relative alla mappa di gioco. Mantiene una lista di luoghi (`places`) in cui il gioco può trovarsi e consente di passare da un luogo all'altro. Come per `PlayerState`, `MapState` è un Singleton, il che significa che solo una singola istanza della mappa esiste nel gioco, centralizzando la logica di navigazione e migliorando l'efficienza nella gestione delle transizioni tra le diverse aree.

La classe `MapState` contiene un oggetto `currentPlace`, che rappresenta la posizione attuale del giocatore nella mappa, e un `HashMap` che mappa i nomi dei luoghi agli oggetti `World`. Grazie a questa struttura, è possibile caricare e passare tra diverse aree di gioco in modo modulare, evitando di dover ricreare istanze della mappa ogni volta che il giocatore cambia zona. L'uso del Singleton assicura che il gioco mantenga una visione coerente e centralizzata dello stato della mappa durante l'intera sessione di gioco.

4.2.3 ScreenManager

Un'altra classe importante che utilizza il pattern Singleton è `ScreenManager`, che gestisce la schermata attuale del gioco. Utilizzando questo approccio, è possibile passare da una schermata all'altra senza creare più istanze della classe `ScreenManager`, garantendo una gestione coerente delle transizioni tra le schermate del gioco. Questo approccio semplifica la gestione dell'interfaccia utente e facilita il passaggio tra diverse modalità di gioco, come la battaglia, la cattura dei Pokémon o l'esplorazione.

4.2.4 ResourceManager ed EventManager

L'uso del pattern Singleton consente anche di centralizzare la gestione di risorse come il `ResourceManager`, che si occupa del caricamento e dello smaltimento delle risorse, evitando duplicazioni e migliorando l'efficienza nel gioco. Allo stesso modo, `EventManager` gestisce gli eventi di gioco, come le notifiche di cattura dei Pokémon, mantenendo una sola istanza per coordinare questi eventi globali.

In sintesi, l'adozione del pattern Singleton in queste classi critiche ha migliorato la manutenzione e l'affidabilità del gioco, consentendo una gestione centralizzata e facilmente accessibile di risorse e stati. Questo approccio ha anche semplificato l'espansione del gioco in futuro, rendendo il codice più modulare e facilmente gestibile.

4.3 Factory Pattern

Successivamente, il pattern Factory è stato implementato per centralizzare la creazione di oggetti complessi come Pokémon, oggetti di mappa e articoli utilizzati nel gioco. Le classi `PokemonFactory`, `MapObjectFactory` e `CaptureItemFactory` sono esempi dell'applicazione di questo pattern, che consente di creare istanze di vari oggetti senza dover gestire direttamente la logica di costruzione nel resto del gioco.

4.3.1 PokemonFactory

La `PokemonFactory` centralizza la creazione dei Pokémon. Il metodo `createPokemon` restituisce l'oggetto corrispondente al nome dato (es. `Exeggutor` o `Slakoth`). Se il nome non è valido, viene sollevata un'eccezione. Questo approccio semplifica l'aggiunta di nuovi Pokémon senza modificare il codice principale.

Figura 6: Factory dei Pokemon

4.3.2 MapObjectFactory

La `MapObjectFactory` crea oggetti di mappa come edifici, piante, zone di teletrasporto e aree di incontro. Ogni tipo di oggetto viene creato tramite metodi specifici per oggetti statici, animati o complessi (teletrasporto, encounter). Questo approccio centralizza la logica di creazione e facilita l'espansione del gioco con nuovi oggetti.

4.3.3 CaptureItemFactory

La `CaptureItemFactory` centralizza la creazione di oggetti per la cattura dei Pokémon, come Pokéball, esche e trappole. Ogni tipo di oggetto (es. `Pokeball`, `SpicyBait`, `QuickTrap`) viene creato tramite un caso specifico nel `switch`. Questo permette di aggiungere facilmente nuovi oggetti senza modificare altre parti del codice.

4.3.4 Vantaggi del Pattern Factory

L'adozione del pattern Factory ha i seguenti vantaggi:

- **Modularità**: Ogni tipo di oggetto è creato in modo separato e centralizzato.
- **Scalabilità**: Aggiungere nuovi oggetti o Pokémon è semplice.
- **Manutenibilità**: Modifiche nella logica di creazione degli oggetti sono limitate alla Factory.

- **Evitare duplicazioni**: La logica di creazione è centralizzata e non duplicata.

Il pattern Factory semplifica la gestione della creazione degli oggetti nel gioco, centralizzando la logica in classi dedicate. Questo approccio migliora la qualità del codice e rende più facile l'espansione e la manutenzione del gioco.

4.4 Observer Pattern

Il pattern Observer è stato utilizzato per notificare gli osservatori di eventi importanti come l'uso di esche, trappole, catture e i movimenti degli attori. Ogni classe che implementa l'interfaccia `Observer` riceve notifiche su eventi specifici e aggiorna di conseguenza l'interfaccia o esegue azioni nel gioco. Esploriamo due implementazioni principali di questo pattern nel codice:

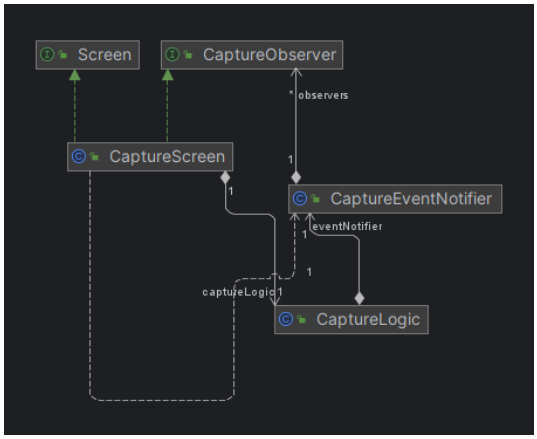

Figura 7: Observer di cattura

4.4.1 CaptureEventNotifier e CaptureObserver

La classe `CaptureEventNotifier` gestisce la registrazione e la notifica degli eventi a tutti gli osservatori registrati, come la `CaptureScreen`, che si occupa di visualizzare l'evento al giocatore. Gli eventi notificati includono azioni come l'uso di esche, trappole e Pokéball, e la successiva cattura di un Pokémon.

CaptureScreen La classe `CaptureScreen` implementa l'interfaccia `CaptureObserver` e si occupa di visualizzare e gestire gli eventi di cattura. Ad esempio, quando un Pokémon fugge o una cattura fallisce, `CaptureScreen` aggiorna il messaggio e attende l'input dell'utente per continuare l'azione.

4.4.2 ActorObserver

Il pattern Observer è usato anche per monitorare i movimenti degli attori nel gioco. La classe `ActorObserver` definisce tre metodi principali che devono essere implementati dagli osservatori:

- `actorMoved`: notificato quando un attore si muove in una nuova posizione.
- `attemptedMove`: notificato quando un attore tenta di muoversi senza successo.
- `actorBeforeMoved`: notificato prima che un attore si muova.

World e ActorBehavior La classe `World` implementa l'interfaccia `ActorObserver` per monitorare e reagire ai movimenti degli attori nel mondo. Essa tiene traccia del giocatore, degli NPC e degli oggetti nel mondo, aggiornando la posizione e lo stato degli attori. Sebbene i metodi dell'interfaccia `ActorObserver` siano attualmente non implementati, la classe `World` è pronta per gestire le notifiche relative ai movimenti degli attori.

4.4.3 Vantaggi del Pattern Observer

L'uso del pattern Observer in questo contesto offre numerosi benefici:

- **Separation of Concerns**: La logica del gioco è separata dalla gestione degli eventi e degli attori.
- **Flessibilità**: È facile aggiungere nuovi osservatori o eventi senza modificare il resto del sistema.
- **Scalabilità**: Il numero di osservatori può crescere senza influire sulle prestazioni o sulla gestione degli eventi.
- **Gestione Centralizzata**: Gli eventi vengono gestiti in un unico punto, centralizzando la logica di aggiornamento degli osservatori.

5 Unit Testing

Per quanto riguarda lo l'unit testing ci si è concentrati prevelentemente sul testing dei pattern prima descritti.

5.1 Test del Singleton `GameState`

Il pattern Singleton è utilizzato per garantire che esista una sola istanza della classe `GameState`. Sono stati implementati vari test per verificare la corretta implementazione di questo pattern.

5.1.1 Test delle Istanze Singleton

Il test `testSingletonInstance()` verifica che due chiamate consecutive a `GameState.getInstance()` restituiscano la stessa istanza, garantendo così il comportamento da Singleton. Il test utilizza `assertSame()` per confrontare le istanze.

testSingletonInstance Method

```
@Test
void testSingletonInstance() {
    System.out.println(YELLOW + "Eseguendo
    testSingletonInstance()..." + RESET);

    // Ottieni due istanze del Singleton
    GameState instance1 = GameState.getInstance();
    GameState instance2 = GameState.getInstance();

    // Verifica che le due istanze siano la stessa
    assertSame(instance1, instance2, RED + "GameState non è un
    Singleton, le istanze differiscono!" + RESET);

    System.out.println(GREEN + "Test completato
    con successo: testSingletonInstance()" + RESET);
}
```

5.1.2 Test di Inizializzazione

Il test `testInitialization()` simula l'inizializzazione del gioco e verifica che vari componenti, come `PlayerState`, `MapState`, e `ScreenManager`, siano correttamente inizializzati. Vengono utilizzati diversi `assertNotNull()` per controllare che nessuno di questi oggetti sia nullo dopo l'inizializzazione.

5.2 Test della Creazione degli Oggetti: `CaptureItemFactory`

`CaptureItemFactory` è responsabile della creazione di diversi tipi di oggetti nel gioco, come Pokéball, esche, trappole, e profumi. I test esaminano la corretta creazione di questi oggetti.

5.2.1 Test di Creazione di Oggetti Specifici

I test come `testCreatePokeball()`, `testCreateBait()`, `testCreatePerfume()`, e `testCreateTrap()` verificano che ogni tipo di oggetto venga creato correttamente. Ogni test crea un oggetto utilizzando `CaptureItemFactory.createItem()` e confronta il risultato con il tipo e il nome attesi.

testCreatePokeball Method

```
@Test
void testCreatePokeball() {
    System.out.println(YELLOW +
    "Eseguendo testCreatePokeball()..." + RESET);

    // Test per la creazione di una Pokeball
    CaptureItem item =
    CaptureItemFactory.createItem("POKEBALL");
    assertNotNull(item, RED + "Pokeball non è stato creato
    correttamente!" + RESET);
    assertEquals("Pokeball", item.getName(), RED + "Il nome
    della Pokeball non è corretto!" + RESET);

    System.out.println(GREEN + "Test completato con successo:
    testCreatePokeball()" + RESET);
}
```

5.2.2 Test per Tipo di Oggetto Non Valido

Il test `testInvalidItemType()` verifica che, quando viene passato un tipo di oggetto non valido, venga sollevata un'eccezione `IllegalArgumentException` con il messaggio appropriato.

5.3 Test della Notifica degli Eventi: `CaptureEventNotifier`

`CaptureEventNotifier` è utilizzato per notificare gli osservatori riguardo gli eventi di cattura. I test qui descritti verificano che gli osservatori vengano notificati correttamente quando si verifica un evento.

5.3.1 Test di Registrazione e Notifica degli Osservatori

Il test `testRegisterAndNotifyObserver()` registra un osservatore con il `CaptureEventNotifier`, simula un evento e verifica che l'osservatore venga notificato correttamente. Viene utilizzata una variabile `AtomicBoolean` per verificare se l'osservatore è stato notificato.

5.3.2 Test di Deregistrazione degli Osservatori

Il test `testDeregisterObserver()` verifica che un osservatore possa essere rimosso e non venga notificato per eventi successivi. Viene registrato un osservatore, quindi deregistrato, e si verifica che non venga notificato per un evento.

Test della Notifica a Più Osservatori

Il test `testNotifyMultipleObservers()` verifica che più osservatori registrati ricevano la notifica di un evento. Il contatore di notifiche deve essere incrementato correttamente per ogni osservatore.

5.3.3 Test di Pulizia degli Osservatori

Il test `testClearObservers()` verifica che tutti gli osservatori vengano rimossi correttamente quando viene chiamato `clearObservers()`. Si verifica che nessun osservatore venga notificato dopo questa operazione.

testRegisterAndNotifyObserver Method

```
@Test
void testRegisterAndNotifyObserver() {
    System.out.println(YELLOW + "Eseguendo
    testRegisterAndNotifyObserver()..." + RESET);

    CaptureEventNotifier notifier = new CaptureEventNotifier();
    AtomicBoolean notified = new AtomicBoolean(false);

    CaptureObserver mockObserver = eventType -> {
        if ("TEST_EVENT".equals(eventType)) {
            notified.set(true);
        }
    };

    notifier.registerObserver(mockObserver);
    notifier.notifyObservers("TEST_EVENT");

    assertTrue(notified.get(), RED + "L'osservatore non è stato
    notificato correttamente!" + RESET);
    System.out.println(GREEN + "Test completato con successo:
    testRegisterAndNotifyObserver()" + RESET);
}
```