1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

Relazione per il Progetto del Corso di Programmazione e Modellazione ad Oggetti

Umberto Trozzi

Matricola 291330

30/10/2024

Indice

1 Analisi

1.1 Analisi dei Requisiti

Pokemon Battle è un gioco a singolo utente, con possibilità di scegliere fra varie modalità di gioco. Lo scopo principale del gioco è quello di costruire un team formato da 6 Pokémon e sfidare un allenatore, pilotato dal computer, anch'esso avente una squadra formata da 6 Pokemon.
In particolare il giocatore potrà scegliere fra due modalità di gioco:

- Partita rapida: il giocatore, previa formazione della squadra, potrà avviare una partita contro un team formato da 6 Pokemon casuali.
- Lega Pokemon: rappresenta la modalità principale del gioco dove il giocatore sarà chiamato ad affrontare la Lega e diventare campione. In particolare l'utente potrà decidere se creare la propria lega, formata da allenatori personalmente selezionati, o affrontare una lega già presente.

1.2 Lista dei Requisiti

Requisiti Funzionali:

- Il programma deve permettere all'utente di scegliere un nome per l'allenatore, una squadra formata da 6 Pokemon e una modalità di gioco.
- A fine partita il giocatore deve poter tornare all'area principale e decidere se continuare a giocare o terminare la sessione.
- I Pokemon dell'avversario, nella modalità di gioco rapida, devono essere scelti casualmente.
- Implementazione delle battaglie:
    - Mosse: ogni Pokemon ne ha a disposizione 4 con relativi PP che, una volta azzerati, rendono indisponibile la suddetta mossa.
    - HP: ogni Pokemon avrà un quantitativo di punti vita che, una volta azzerati, determinerà il cambio Pokemon e l'indisponibilità di quest'ultimo.
    - Tipi: tipi di Pokemon diversi subiscono danni diversi in base al tipo del Pokemon attaccante.
- Implementazione di un Pokedex:
    - Lista: consultabile per conoscere il roster di Pokemon disponibile, insieme alle mosse disponibili
- Implementazione della Lega:
    - Lega: lega precostruita, pronta da sfidare con allenatori già costruiti.
    - Lega Personale: lega con allenatori aventi team costruiti dall'utente.

Requisiti non Funzionali:

- Gestione e salvataggio dei dati di gioco su File.
- Gestione dei suoni.
- Gestione della "banca dati".

1.3 Modello del dominio

I principali attori coinvolti in una partita sono:

- I Pokemon, rappresentati dall'entità **IPokemon**, la quale contiene le caratteristiche principali dei pokemon, come nome, specie e mosse.
- Le Mosse, rappresentate dall'entità **IMoves**, la quale contiene i tipi delle mosse da poter utilizzare, i danni di output, i PP (cosiddetti "power point").
- Le Statistiche proprie di ogni Pokemon, rappresentate dall'entità **IStats**, la quale contiene informazioni come il livello del pokemon e i punti vita.
- Le Specie, rappresentate dall'entità **ISpecie**, la quale contiene informazioni come altezza e peso del pokemon, oltre che l'immagine di questo.
- Il Tipo, rappresentato dall'entità **IType**, si occupa di generare tutti i tipi presenti e contenere la logica dietro il concetto di debolezza e resistenza fra tipi.
- Il Pokedex, rappresentato dall'entità **IPokedex**, che rappresenta la fonte dati principale, contenente tutti i pokemon contenuti nel file csv.

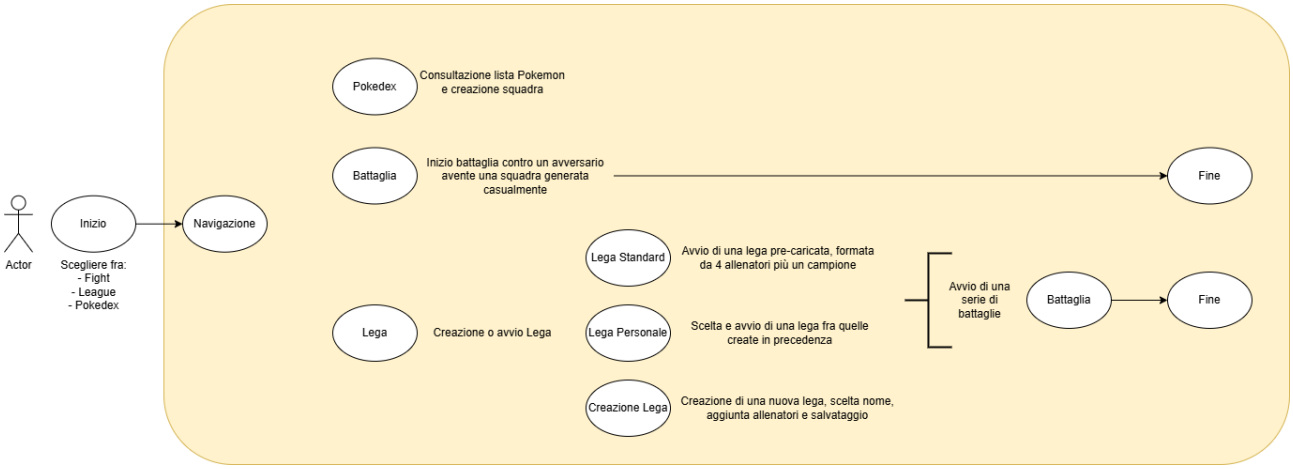


Figura 1: Diagramma d'uso

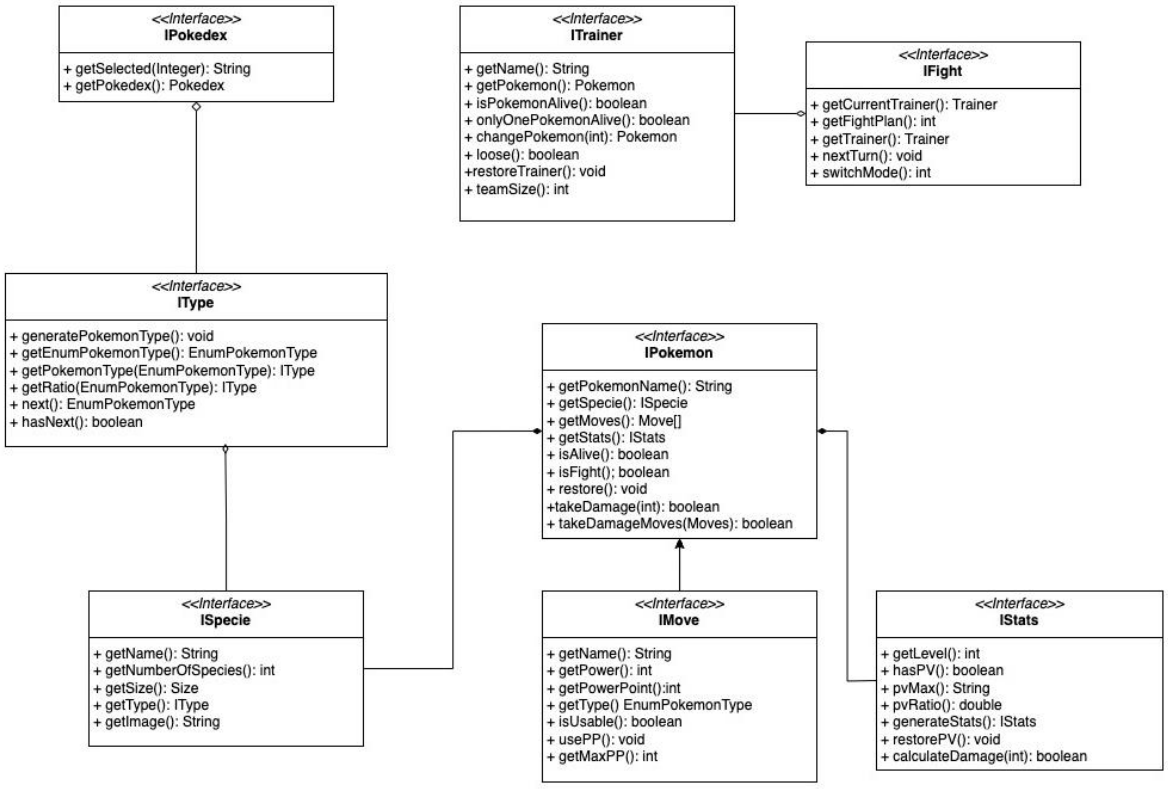


Figura 2: Modello del dominio

La *figura 1* mostra i casi d'uso del software. Ogni partita inizia con la scelta di quale modalità di gioco selezionare. In base alla modalità scelta il giocatore potrà creare la propria squadra e cambiare il suo nickname, effettuare una battaglia rapida contro un allenatore comandato dal computer avente una squadra generata casualmente, sfidare la lega pokemon potendo scegliere fra l'affrontare una lega fornita dall'autore, una creata in precedenza da lui o il crearne una nuova. Il gioco termina al fine della battaglia o dopo aver affrontato tutte le battaglie, quindi una volta completata la lega pokemon.

La *figura 2* mostra il modello del dominio rappresentato con le interfacce. Si possono notare fin da subito le relazioni tra le diverse entità: le più importanti sono quelle di aggregazione e composizione che riguardano le entità IPokemon, IMove, IStats e ISpecie.

2 Design

2.1 Architettura

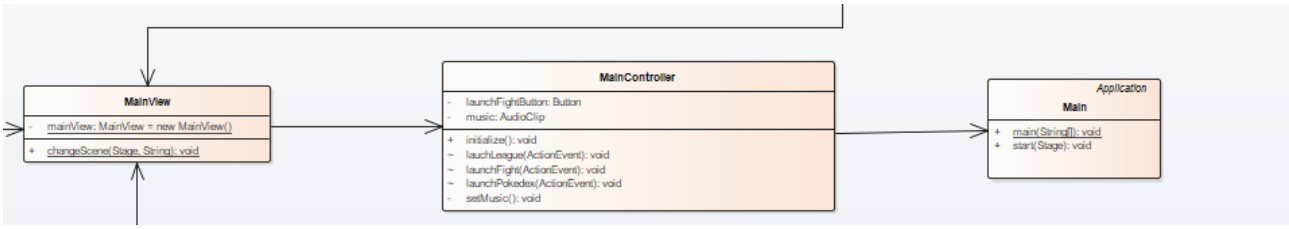
L’architettura del software si basa sul pattern **MVC** (Model- View- Controller). Il Controller funge da disacoppiatore, in quanto elimina ogni interazione diretta tra Model e View passando i riferimenti del primo alla View e utilizzando i metodi pubblici delle entità del modello per modificare i dati a fronte di ogni evento generato dal viewer.

2.2 Design dettagliato

In questa sezione vengono riportato i diagrammi UML con le relative descrizioni delle entità che rappresentano le macro componenti del progetto. Oltre all'impostazione architetturale del software secondo il pattern MVC previo descritto, sono stati utilizzati anche altri pattern come:

- Command Pattern, in FightController (appartenente al controller).
- Observer Pattern, in PokedexController (appartenente al controller).
- Factory Method, in PokemonType (appartenente al model).
- Iterator Pattern, in EnumSetPokemonType (appartenente al model).
- Singleton Pattern, nel Main (appartenente alla view).

2.2.1 Gestione delle varie schermate



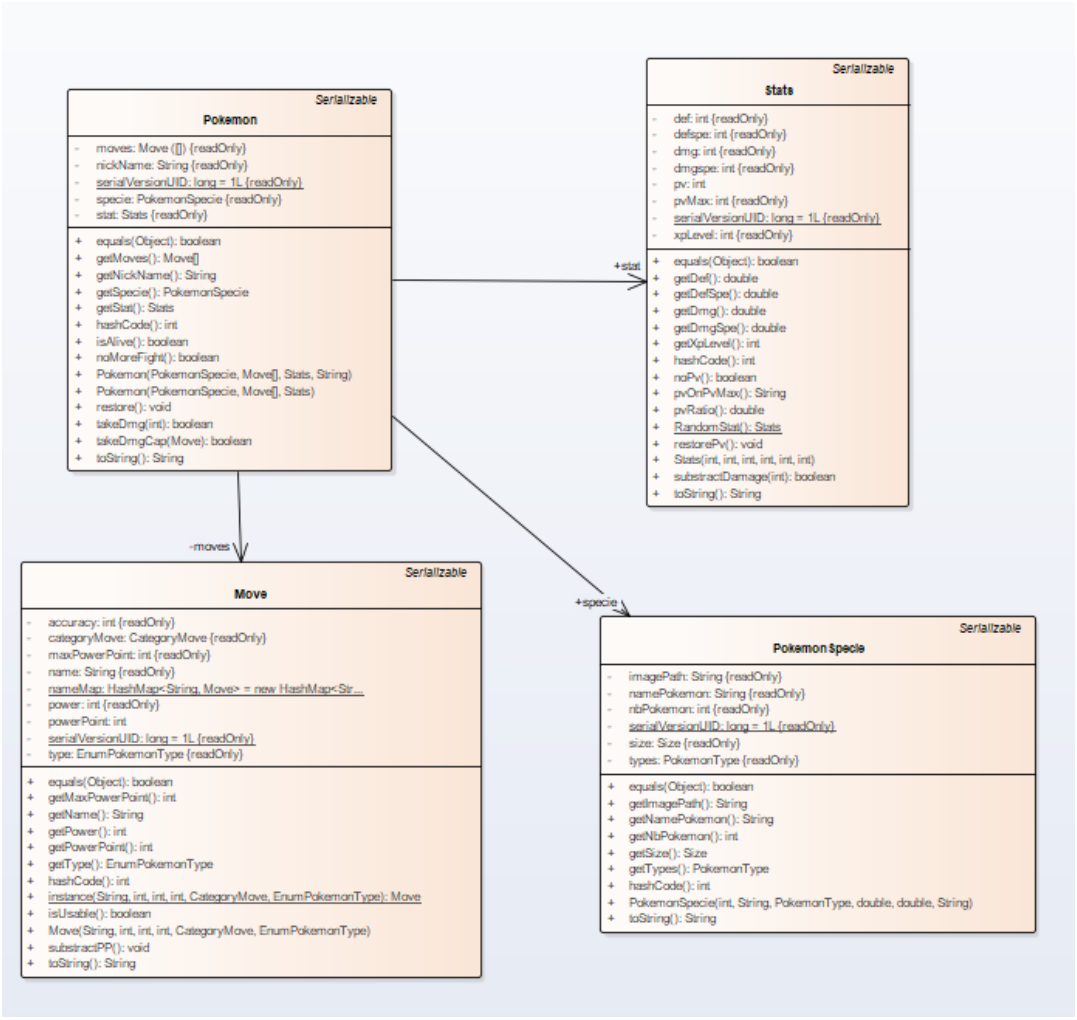
Problema

Una delle parti fondamentali del gioco è la navigazione fra le varie schermate. L'implementazione deve tener conto delle schermate raggiungibili e dell'instradamento da poter metter mettere a disposizione dell'utente, così da permettere un orientamento coerente alla struttura presentata.

Soluzione

Si è reso necessario implementare una classe **MainView** che tenesse conto delle schermate raggiungibili dall'utente. Al suo interno infatti si sviluppano i metodi richiamabili per raggiungere le schermate **Fight.fxml**, **Pokedex.fxml** e **League.fxml**. Queste interfacce grafiche sono state implementate attraverso l'uso del linguaggio **FXML che** permette di separare la presentazione dalla logica applicativa di un software scritto attraverso **JavaFX**. Avendo avuto già a che fare con linguaggio xml, si è preferito adottare un insieme di software familiari.

2.2.2 Creazione dei Pokemon



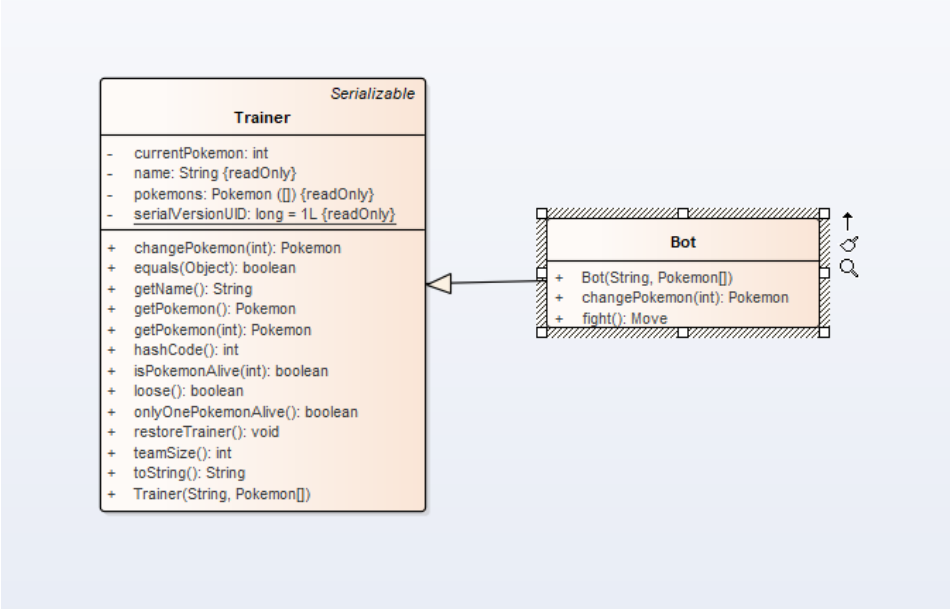
Problema

I Pokemon rappresentano la componente principale del gioco poiché, attraverso mosse, statistiche e specie differenti, vanno a caratterizzare ogni battaglia rendendola così unica. Va quindi proposto un roster di Pokemon sufficientemente ampio così da permettere la rigiocabilità del titolo, così come un roster di mosse utilizzabile ampio per permettere all'utente di mettere in campo varie strategie per sconfiggere gli allenatori. Vanno inoltre garantite tutte le specie ad oggi presenti.

Soluzione

Ogni Pokemon ha statistiche, mosse e specie diverse. Poiché queste sono informazioni non banali si è resa necessaria la creazione di tre classi di supporto, rispettivamente **Stats**, **Move** e **Specie**, per andare a caratterizzare ogni Pokemon. In particolare, nella classe **Move** vengono gestite le mosse che un Pokemon può avere, attraverso i metodi "*getPower*()", "*getType*()" e la gestione dei PP vengono così simulate le mosse in battaglia. La classe **PokemonSpecie** invece va a coprire la struttura del Pokemon, come le dimensioni, lo sprite (immagine) e il tipo, fondamentale nelle battaglie per applicare le logiche di debolezza e resistenza fra tipi. Infine si è resa necessaria l'implementazione di una classe **Stats** per poter così caratterizzare ogni Pokemon con punti vita, attacco, difesa, livello e danno. Tutte queste informazioni vanno a comporre un Pokemon, avente mosse, specie e statistiche differenti.
Si noti come nella classe Pokemon sia stato utilizzato il **Composite Pattern,** poiché questa utilizza un array di **Move**, il che sottolinea una struttura composita in cui un Pokémon può avere più mosse, consente di trattare le mosse come una collezione di oggetti omogenei all'interno di un singolo Pokémon.

2.2.3 Creazione degli Allenatori



Problema

Le battaglie sono caratterizzate da incontri 1vs1, ovvero fra due allenatori.
Bisogna quindi avere a disposizione un allenatore che impersoni l'utente ed uno che sia controllato dal programma stesso.

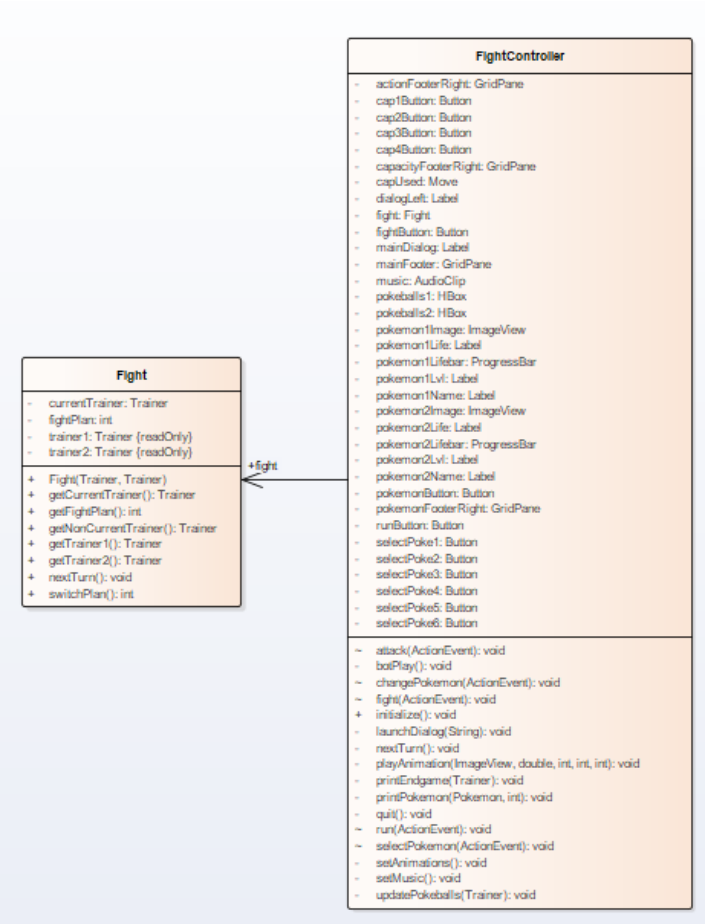
Soluzione

Sono state implementate due classi, **Trainer** e **Bot** che vanno ad intercettare i due casi forniti dal problema. La prima costituisce la classe principale che va a modellare un allenatore tipo, avente un proprio nome e una squadra di Pokemon. La seconda costituisce un'estensione della classe **Trainer**, condividendone molte della caratteristiche, che introduce elementi propri di un "computer" come avversario attraverso i metodi "*fight()*" e "*changePokemon()*" rivisti per simulare una battaglia contro una IA. A tal proposito nella classe Trainer sono stati inseriti metodi, fondamentali durante le battaglie, per interrogare lo stato di un Pokemon, della squadra e l'eventuale sconfitta dell'allenatore attraverso, rispettivamente, i metodi "*isPokemonAlive()*", "*onlyOnePokemonAlive()*" e "*loose()*".

2.2.4 Creazione delle Battaglie

Problema

Le battaglie costituiscono il core del gioco, rappresentano le interazioni fra i Pokemon e per questo motivo devono tener conto di una discreta mole di informazioni da presentare ed elaborare in base ad ogni interazione con l'utente.

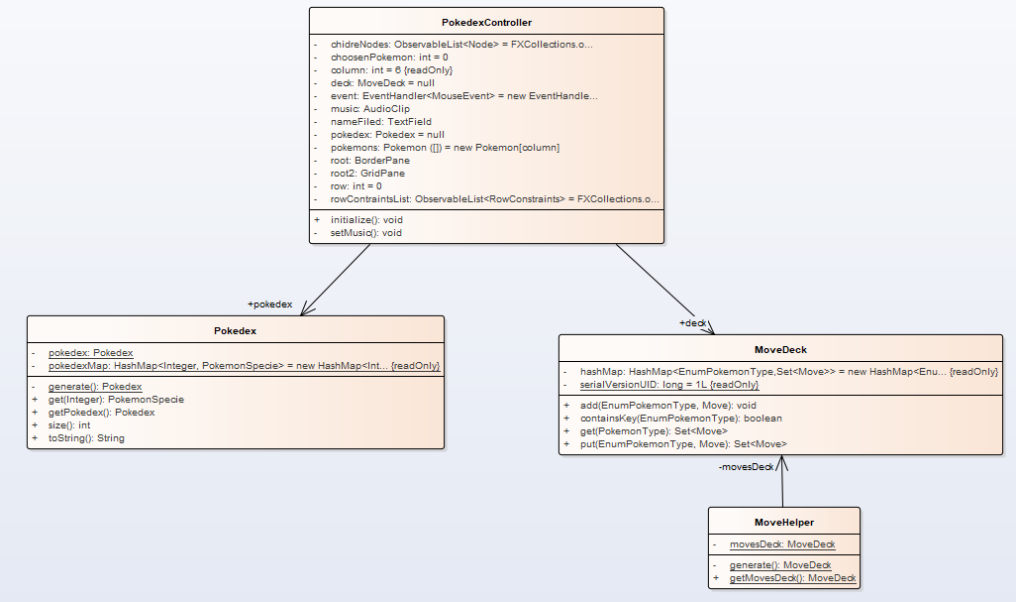


Soluzione

E' stata implementata una classe **Fight** che, insieme all'implementazione del corrispettivo controller **FightController**, vanno a definire le logiche di interazione fra Pokemon. In particolare la classe **Fight** mette a disposizione i due allenatori coinvolti nello scontro (Trainer1 e Trainer2) oltre che al metodo "*nextTurn()*" mediante il quale è possibile avanzare e cambiare il focus su quale dei due allenatori sta agendo. La classe **FightController** si occupa di fare da intermediario fra il modello e la vista, in particolare ha il compito di catturare tutti gli

input forniti dall'utente e trasformali in azioni in-game. Per fare questo sono stati implementati metodi come "*attack()*"che permette ad un Pokemon di attaccare ed innesca tutta una serie di meccanismi e controlli della situazione, e "*changePokemon()*" per permettere all'utente di cambiare Pokemon. A tal proposito, per l'implementazione dei metodi *fight*, *attack*, *run*, *changePokemon* e *selectPokemon* si è presa ispirazione dal **Command Pattern** potendo essere considerati comandi che, una volta invocati, eseguono specifiche operazioni sull'interfaccia utente e sull'oggetto **Fight**, seguendo un ordine preciso per gestire il combattimento.

2.2.5 Creazione della struttura Dati



Problema

Il mondo Pokemon è composto da svariati Pokemon di svariati tipi e mosse differenti, che nel complesso vanno a determinare una mole di informazioni importante, non di facile gestione.

Soluzione

Si è scelto di organizzare Pokemon, tipi e mosse in 3 file .csv distinti da andare ad interrogare un'unica volta. Questo perché i file .csv hanno una struttura semplice e sono anche facilmente consultabili nonché modificabili, oltre al riuscire a fornire molte informazioni in file di ridotte dimensioni, perfetti per il caso d'uso in esame.

In particolare i Pokemon vengono recuperati dalla classe **Pokedex** che mediante il metodo "*generate()*" popola l'oggetto **Pokedex** che, grazie al pattern **Singleton** che garantisce che ci sia solo un'istanza della classe (questo è evidente nel metodo *getPokedex()*, dove viene controllato se pokedex è nullo prima di crearne una nuova istanza), mettendo a disposizione un insieme completo di Pokemon.

Per quanto riguarda le mosse è stato applicato la stessa logica, implementando una classe **MoveHelper** che si incarica di leggere un file .csv contenente le varie mosse e di creare, sempre attraverso un metodo "*generate()*", un oggetto **MoveDeck** contenente tutte le varie mosse disponibili.

3 Sviluppo

3.1 Testing Automatizzato

Per il sistema di entità sono state testate le caratteristiche principali di ciascuna delle macro-implementazioni. Poiché ogni package è formato da almeno una classe, si è scelto una classe per ogni package così da evidenziarne il corretto funzionamento:

- Per il package fight è stata testata la classe Fight nella sua interezza.
- Per il package move è stata testata la classe Move nella sua interezza.
- Per il package pokemon è stata testata la classe Pokemon nella sua interezza.
- Per il package specie è stata testata la classe PokemonSpecie nella sua interezza.
- Per il package stats è stata testata la classe Stats nella sua interezza.
- Per il package trainer è stata testata la classe Trainer nella sua interezza.
- Per il package type è stata testata la classe PokemonType nella sua interezza.

3.1.1 Fight

La classe è stata testata per tutti i suoi metodi ovvero:

- Costruttore, viene testata la corretta costruzione della classe con tutti i suoi attributi iniziali, ovvero che i due allenatori non siano nulli e che siano rispettivamente Trainer1 e Trainer2 .
- NextTurn, per testare che dopo l'allenatore Trainer1 il turno venga passato al secondo allenatore Trainer2 e viceversa.
- SwitchPlan, per testare che il piano di battaglia venga correttamente cambiato al cambiare del turno.

3.1.2 Move

La classe è stata testata per tutti i suoi metodi ovvero:

- Costruttore, viene testata la corretta costruzione della classe con tutti i suoi attributi iniziali, ovvero nome della mossa, potenza, power point e tipo. Inoltre

sono stati inseriti test per simulare eventuali eccezioni in caso di attributi errati.
- IsUsable, viene testato che una mossa senza PP non sia utilizzabile.
- GetPowerPoint, viene testato che spendendo un PP questo venga effettivamente decrementato di un'unità.
- GetName, viene testato che possa esistere un'unica mossa con un dato nome.
- GetMaxPowerPoint, viene testato che i PP vengano restituiti in modo corretto.

3.1.3 Pokemon

La classe è stata testata per tutti i suoi metodi ovvero:

- Costruttore, viene testata la corretta costruzione della classe con tutti i suoi attributi iniziali, ovvero che venga creata correttamente un'istanza dell'oggetto Pokemon avente nome, specie, mosse e statistiche impostate in precedenza.
- GetNickName, viene testato che l'istanza dell'oggetto abbia il nome corretto.
- IsAlive, per testare che il pokemon non sia ancora vivo dopo un danno che lo mette ko.
- TakeDamage, per testare che i pv del pokemon vengano decrementati del quantitativo passato.
- Restore, per testare che i pv del pokemon selezionato vengano ristabiliti.
- NoMoreFight, per testare che un pokemon non sia più in grado di combattere al termine di tutti i PP di tutte le mosse da lui conosciute.

3.1.4 Specie

La classe è stata testata per tutti i suoi metodi ovvero:

- Costruttore, viene testata la corretta costruzione della classe con tutti i suoi attributi iniziali, ovvero che venga creata correttamente un'istanza dell'oggetto Specie avente numero specie, nome pokemon, tipo, altezza, peso e sprite. Inoltre sono stati inseriti test per simulare eventuali eccezioni in caso di attributi errati.
- GetNbPokemon, per testare che venga recuperato il numero di specie corretto.

- GetNamePokemon, per testare che venga recuperato il nome corretto.
- GetTypes, per testare che venga recuperato il tipo corretto.
- GetSize, per testare che vengano recuperati altezza e peso corretti
- GetImagePath, per testare che venga recuperato lo sprite corretto.

3.1.5 Stats

La classe è stata testata per tutti i suoi metodi ovvero:

- Costruttore, viene testata la corretta costruzione della classe con tutti i suoi attributi iniziali, ovvero che venga creata correttamente un'istanza dell'oggetto Stats avente attacco, attacco speciale, difesa, difesa speciale, livello exp e punti vita.
- NoPV, per testare che, a seguito di un danno che vada ad azzerare gli hp del pokemon, questo non abbia più punti vita a disposizione.
- PvOnPvMax, per testare che, a seguito di un danno che vada a dimezzare gli hp del pokemon, questo abbia il giusto quantitativo di pv sui pvmax di partenza.
- SubstractDamage, per testare danni di vari entità e che il pokemon sia o meno esausto.
- RestorePv, per testare che i punti vita vengano correttamente ripristinati.
- RandomStat, per testare che venga generato un oggetto di tipo Stats aventi attributi generati randomicamente.

3.1.6 Trainer

La classe è stata testata per tutti i suoi metodi ovvero:

- Costruttore, viene testata la corretta costruzione della classe con tutti i suoi attributi iniziali, ovvero nome e squadra pokemon.
- GetPokemon, per testare che l'allenatore abbia i pokemon precedentemente creati in squadra.
- TeamSize, per testare che il numero dei componenti della squadra dell'allenatore sia composto dal numero corretto di pokemon.
- ChangePokemon, per testare che il primo pokemon a disposizione venga correttamente cambiato con l'altro disponibile.
- IsPokemonAlive, per testare che entrambi i pokemon siano vivi.

- RestoreTrainer, per testare che tutti i pokemon dell'allenatore recuperino i loro punti salute.
- Loose, per testare che una volta messi ko tutti i pokemon dell'allenatore, questo risulti sconfitto.
- `OnlyOnePokemonAlive`
- OnlyOnePokemonAlive, per testare che, supponendo un team composto da due pokemon, una volta messo ko un pokemon, l'allenatore rimanga con un solo pokemon a disposizione.

3.1.7 PokemonType

La classe è stata testata per tutti i suoi metodi ovvero:

- Costruttore, viene testata la corretta costruzione della classe con tutti i suoi attributi iniziali, ovvero che venga creata correttamente un'istanza dell'oggetto PokemonType contenente il set di enumerati per il tipo corretti.
- GetPokemonType, per testare che passato un tipo non presente nel set di partenza, questo non ritorni nulla.
- ValidFile, per testare la corretta apertura e lettura di un file .csv (viene effettuato qui poiché anche in altre sezione è stata adottata la stessa logica).
- GeneratePokemonType, per testare che la generazione dei tipi avvenga correttamente.
- ValidIndex, per testare che passato un indice i venga restituito il tipo di indice corretto.

3.2 Metodologia di Lavoro

Strumenti Utilizzati:

- Git: utilizzato per il versionamento e la condivisione del codice.
- Eclipse: utilizzato come IDE di riferimento.
- GitHub: utilizzato come repository principale.
- draw.io: utilizzato per la creazione del modello del dominio di analisi.
- Enterprise Architect: utilizzato per la creazione e la modellazione dell'UML.
- Microsoft Word: utilizzato per la redazione della relazione.

3.3 Note di Sviluppo

In questo progetto ho avuto modo di apprendere e rafforzare concetti come gli stream e le lambda expressions, strumenti preziosi per rendere il codice più moderno e leggibile. Oltre agli aspetti puramente tecnici, ho compreso quanto sia importante standardizzare elementi come l'indentazione, l'uso di un IDE specifico per il linguaggio, lo stile di programmazione e l'utilizzo di un sistema per la gestione delle versioni, per garantire coerenza e facilità di estensione.