RELAZIONE PROGETTO INGEGNERIA DEL SOFTWARE A.A.2022-23

MEMBRO DEL TEAM: Bonura Giovanni

Assegnazione task degli sprint

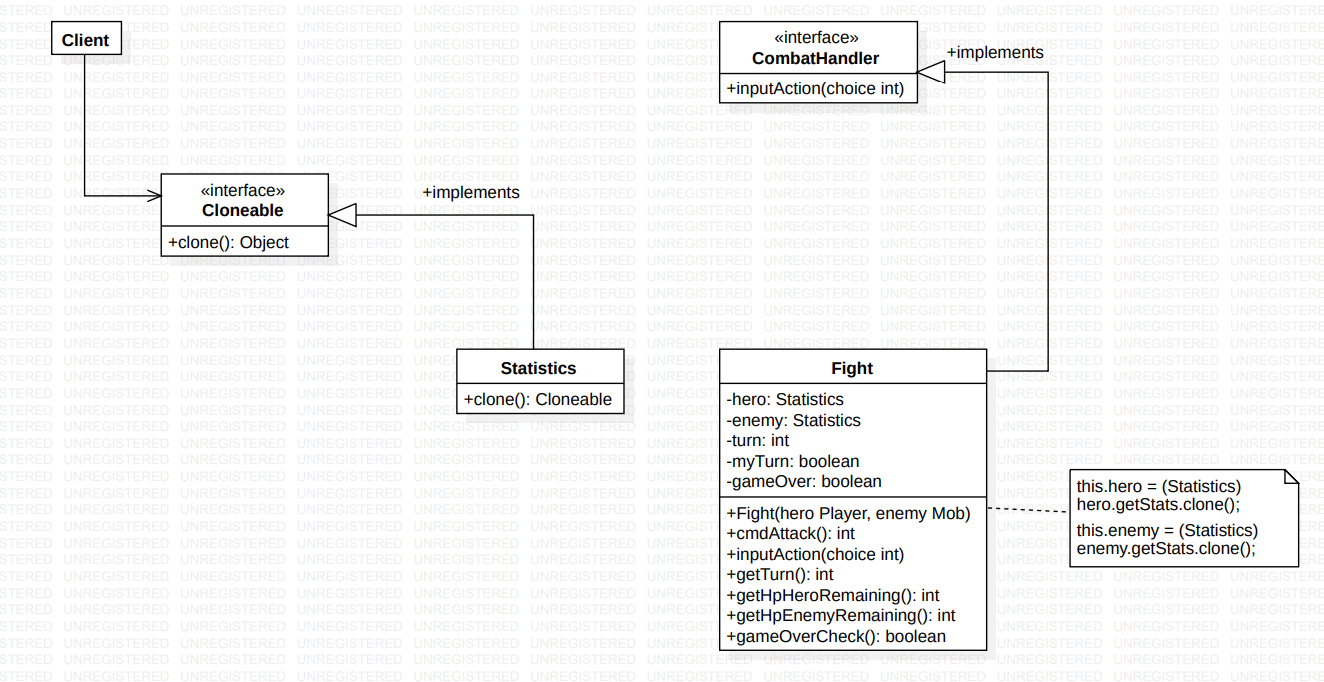
In base alla definizione dei due sprint backlog, al membro del team in questione sono stati assegnati i seguenti task:

* business logic del sistema di combattimento;
* progettazione database relazionale per le entità di gioco;
* business logic del sistema di abilità in combattimento;
* realizzazione del DB e testare la sua connessione con le classi.

Business logic del sistema di combattimento

La business logic del sistema di combattimento rientra nel campo Model dell’architettura scelta dal team (MVC).

Qui di seguito viene fornita un’illustrazione di quello che è il diagramma delle classi relativo al task.



Per la realizzazione del sistema di combattimento è stato pensato un sistema a turni, tipico dei giochi RPG.

Come possiamo vedere dal diagramma delle classi, si inizia dalla dichiarazione di un’interfaccia, CombatHandler, che fornisce le funzionalità utilizzabili durante il combattimento.

La classe Fight che si occupa di gestire tale evento contiene una serie di attributi e metodi che permettono di attaccare l’avversario, stabilire e assegnare il turno di gioco, imporre le condizioni di gameOver con conseguente assegnazione della vittoria a seguito dello scontro.

Si è fatto riferimento alla classe delle statistiche, poiché costituiscono la base per poter implementare un sistema di combattimento. L’utilizzo del pattern Prototype mediante l’implementazione dell’interfaccia Cloneable e del conseguente override del metodo clone, ha reso sì che attraverso la generazione di un clone (dei personaggi) risultasse più efficiente gestire localmente l’aggiornamento delle statistiche a seguito di un attacco.

Per dare un’idea di cosa si occupa la classe verranno spiegati in poche righe il funzionamento dei metodi principali:

* cmdAttack: il seguente metodo restituisce un intero relativo al valore di danno di ogni singolo attacco (mediante un’espressione aritmetica che coinvolge le statistiche dei personaggi impegnati nello scontro), sia da parte del personaggio che da parte dell’avversario;
* inputAction: il seguente metodo, sovrascritto poiché la classe Fight implementa l’interfaccia CombatHandler, prende a parametro un intero che rappresenta la scelta effettuata dall’utente per selezionare una determinata funzionalità in combattimento. Il sistema di combattimento a turni si basa sul seguente criterio: nella classe è stato dichiarato un attributo di tipo boolean myTurn che all’interno del ciclo switch del metodo inputAction cambia più volte il suo stato (true/false) finché non si arriva ad un gameOver. La ripetuta variazione di stato permette l’alternanza del turno di gioco tra il personaggio e l’avversario;
* gameOverCheck: il seguente metodo restituisce il valore di un boolean (gameOver) dichiarato all’interno della classe. A seguito di ogni turno, viene effettuato un check dei punti salute (Hp) dei personaggi mediante i metodi getHpHeroRemaining e getHpEnemyRemaining. Nel caso in cui gli Hp di uno dei personaggi risulta essere minore o uguale a zero, l’attributo gameOver verrà settato a true e sarà restituito dal metodo di cui stiamo parlando, andando a decretare la fine del combattimento.

Ovviamente la verificabilità di tutto il flusso di esecuzione in combattimento è stata accertata mediante la realizzazione dei test, seguendo il Test Driven Development (TDD), i cui test sono allegati di seguito.

Immagine che contiene testo

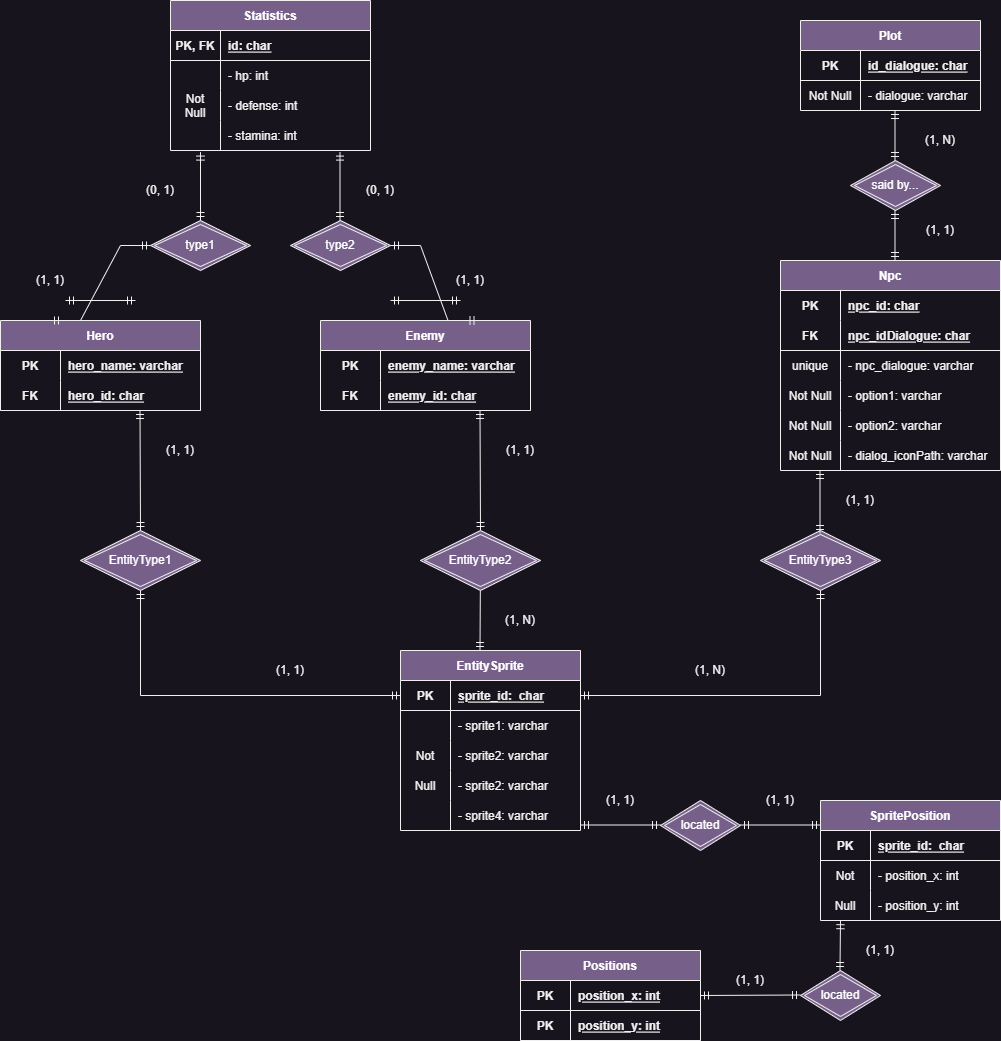
Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamente

Progettazione database relazionale per le entità di gioco

Nel primo sprint si è prevista la presenza di un database relazionale per le entità di gioco, come anche suggerito dall’architettura scelta MVC. Di seguito verrà allegato lo schema concettuale del DB.



Come si può vedere dallo schema concettuale, per ogni entità sono stati definiti gli attributi, specificandone il tipo, le chiavi primarie ed eventuali vincoli di chiave esterna. Occorre notare che le entità Hero ed Enemy inizialmente erano viste come una generalizzazione dell’entità Stats; il processo di normalizzazione dello schema entity-relationship, ha portato a definirle come delle entità figlie mediante delle associazioni.

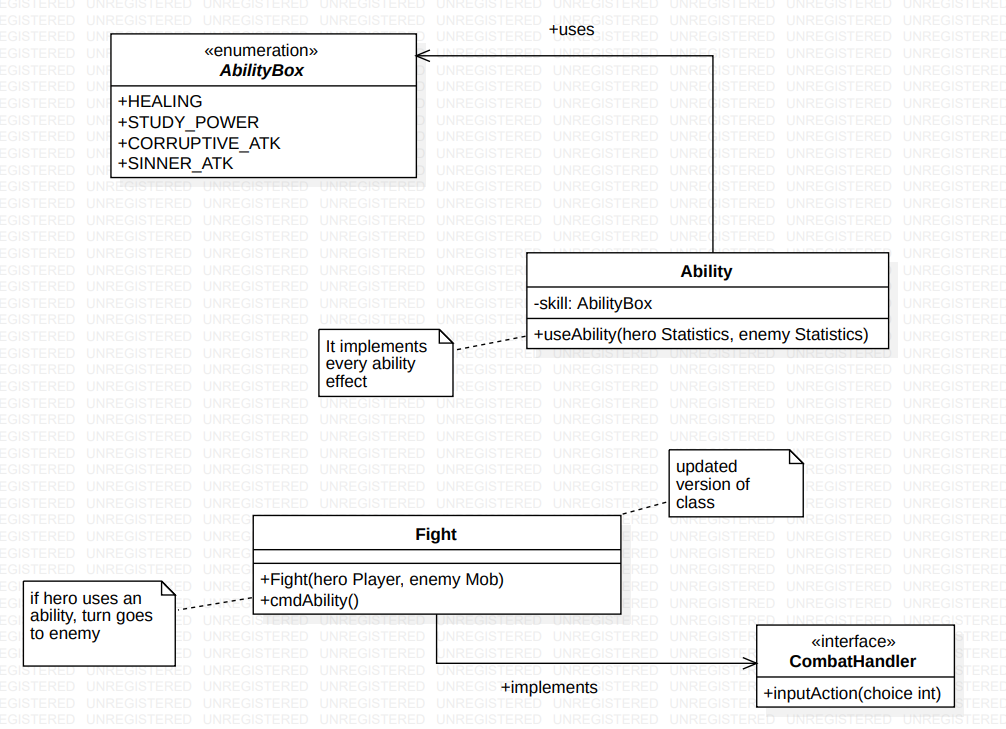
Le entità Plot ed Npc si occupano di registrare i dialoghi di ogni singolo personaggio, le scelte che possono essere intraprese e l’icona della finestra di dialogo. Mentre le entità EntitySprite, SpritePosition, Positions rispettivamente si occupano di memorizzare i seguenti dati:

* EntitySprite: memorizza gli sprite (modelli poligonali realizzati in pixel art) di ogni singolo personaggio. Come si nota dallo schema concettuale, gli attributi sono stati dichiarati di tipo varchar, dal momento che rappresentano stringhe di lunghezza variabile;
* SpritePosition: contiene le informazioni circa le posizioni (coordinate x,y) di ogni singolo personaggio sulla mappa;
* Positions: questa entità memorizza soltanto le posizioni, che mediante un vincolo di chiave esterna, si risale a quale personaggio appartengono quelle determinate coordinate.

Business logic del sistema di abilità in combattimento

La business logic del sistema di abilità rientra anch’essa nel campo Model dell’architettura MVC.

Ecco mostrata di seguito un’illustrazione del diagramma delle classi relativa al seguente task.



Si inizia dalla dichiarazione di un enum contenente l’elenco di tutte le abilità che sono presenti nel gioco, i cui nomi fanno riferimento alla storia e alla descrizione del personaggio.

La classe Ability fornisce l’implementazione di tutte le abilità, in particolare il metodo useAbility (che prende a parametro due oggetti di tipo Statistics), il cui corpo contiene un ciclo switch corrispondente all’elenco di abilità messe a disposizione del giocatore in combattimento. Ognuna delle seguenti abilità provoca degli effetti bonus/malus nei confronti sia del nostro personaggio che dell’avversario.

Ad esempio, l’abilità HEALING permette di ripristinare una piccola parte dei punti salute, mentre le altre abilità come STUDY\_POWER, CORRUPTIVE\_ATK, SINNER\_ATK rappresentano degli attacchi speciali che infliggono maggior danno ma possono causare degli effetti negativi sul personaggio, per rendere il sistema di combattimento più bilanciato.

La classe Fight è stata aggiornata, aggiungendo nel metodo inputAction il comando relativo alle abilità, ovvero cmdAbility. La funzione del metodo cmdAbility è quella di istanziare un oggetto di tipo Ability ed invocare il metodo useAbility, passando a parametro le statistiche relative ai personaggi impegnati nello scontro. Bisogna precisare che nel momento in cui il personaggio decide di usare una delle sue abilità, tale azione corrisponde ad un turno giocato e di conseguenza il turno passa all’avversario.

Di seguito vengono mostrati i test relativi alle abilità.

Immagine che contiene testo

Descrizione generata automaticamente

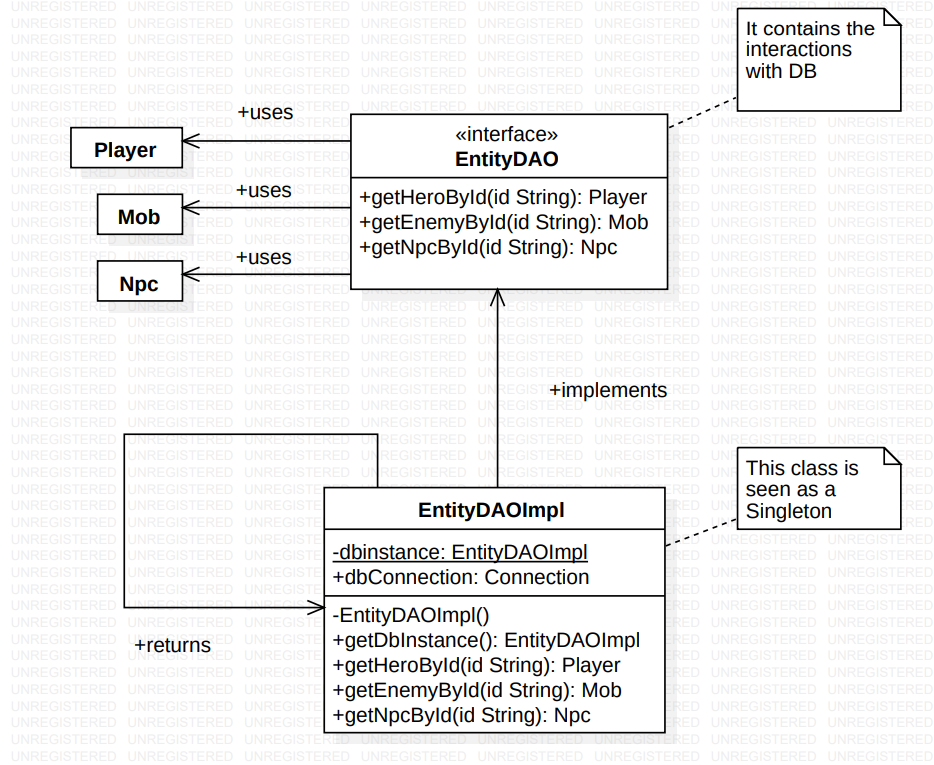
Immagine che contiene testo

Descrizione generata automaticamente

Realizzazione del DB e testare la sua connessione con le classi

L’obiettivo di questo task è stato quello di implementare il database relazionale, basandoci sullo schema concettuale concepito nel precedente sprint, e soprattutto testare la sua connessione con le classi.

Il diagramma delle classi relativo al task è il seguente.



Come si può dedurre dal diagramma delle classi, si è ricorso all’implementazione del pattern DAO (Data Access Object)

Il DAO è un pattern architetturale che ha come scopo quello di separare le logiche di business da quelle di accesso ai dati. L’idea alla base di questo pattern è quello di descrivere le informazioni necessarie per la persistenza del modello in un’interfaccia e di implementare la logica specifica di accesso ai dati in apposite classi.

Facendo riferimento al diagramma UML, la struttura del pattern è costituita dai seguenti componenti:

* Model, di cui ne fanno parte le classi Player, Mob, ed Npc e contengono la logica di business per la persistenza del modello;
* InterfacciaDAO: è stata chiamata EntityDAO e fornisce tutte le possibili interazioni col database. In questo caso la necessità del team è stata quella di ottenere le informazioni riguardo ai personaggi, dalle statistiche fino alla loro posizione sulla mappa;
* DAOClass: chiamata EntityDAOImpl ed è rivolta a gestire la logica di accesso ai dati mediante la tecnologia JDBC (Java DataBase Connectivity).

NOTA: occorre precisare che il database è stato implementato utilizzando il DBMS MySQL, andando ad aggiungere nella libreria delle classi il cosiddetto Connector-J, così da poter permetterci di sfruttare la tecnologia JDBC. Il gioco non necessita di una connessione ad Internet per funzionare, per cui sui dispositivi su cui si deve installare il gioco, si deve a sua volta installare e configurare MySQL.

Adesso entriamo un po' più nel dettaglio per comprendere di cosa si occupa la classe EntityDAOImpl.

Per l’implementazione di tale classe si è ricorso all’utilizzo del pattern Singleton, in modo tale da avere una singola istanza di questa classe e che rappresenti un punto d’accesso comune per chi la utilizza.

L’implementazione del Singleton prevede la dichiarazione di un attributo static di tipo della classe stessa cui appartiene (dbInstance, inizializzato a null), un costruttore privato senza parametri ed un metodo static che restituisce la singola istanza della classe (getDbInstance).

Per quanto riguarda la connessione al database, nel costruttore privato sono stati aggiunti rispettivamente il path relativo al DBMS MySQL e l’URL del database.

I metodi getHeroById, getEnemyById, getNpcById rappresentano le query da effettuare al DB per ottenere le informazioni che possono essere utilizzate dagli altri membri del team.

Prendiamo in esempio il metodo getHeroById: prende a parametro una stringa corrispondente al codice identificativo di ogni singolo personaggio. Per ogni personaggio le informazioni che ci interessano sono: statistiche, sprite del personaggio e posizione sulla mappa. In merito a queste informazioni, nel corpo del metodo viene inizializzato a null un oggetto di tipo Player e vengono dichiarati un oggetto di tipo Statistics, uno di tipo PlayerSprite e viene inizializzato un ArrayList di stringhe, che dovrà contenere tutti i path del personaggio. Il passo successivo è stato la formulazione della query seguendo la sintassi SQL per restituire i campi interessati. Attraverso l’utilizzo di alcuni metodi della libreria java.sql, tra cui il metodo executeQuery per citarne uno, come il nome stesso suggerisce viene eseguita la query. Ecco una piccola dimostrazione per rendere più chiaro ciò che è stato fatto. Immagine che contiene testo

Descrizione generata automaticamente

Si nota che eseguendo la query, successivamente all’interno di un ciclo while, vengono selezionate tutte le colonne che dovranno essere restituite dalla query, specificandone il nome della colonna e il tipo di indice (int o stringa in questo caso). Il passo finale prevede di istanziare l’oggetto di tipo Player (inizialmente impostato a null) passando al costruttore i parametri dei campi dichiarati all’interno del ciclo e restituire l’oggetto stesso.

La stessa procedura si ripete per i metodi getEnemyById, getNpcById.

Ciò che segue adesso sono i test realizzati per testare ognuno dei tre metodi sopra citati. Per ognuno di essi si è effettuata una verifica attraverso il caricamento di un personaggio dal database e la creazione di un oggetto (Player, Mob, Npc). Effettuando il confronto, per mezzo delle asserzioni abbiamo testato che le informazioni passate a parametro sono equivalenti.

Immagine che contiene testo

Descrizione generata automaticamente Immagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamente Immagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamente Immagine che contiene testo

Descrizione generata automaticamente