INGEGNERIA E SICUREZZA DEL SOFTWARE

Prof.ssa Simona Ester Rombo

Relazione Progetto, Team 5

Andrea Spinelli
Alessia Bonì
Giovanni Bonura
Raffaele Terracino

Università degli Studi di Palermo

A.A. 2022-2023

// Nel Cap. 1 si parla dei requisiti
// Nel Cap. 2 si parla della progettazione MVC e Scrum Method
// Nel Cap. 3 si parla dei nostri task

Partendo dalle specifiche preliminari, il primo step dell'analisi dei requisiti è stato concepire la storia di base del gioco.

Storia del gioco:

L'idea del gioco è che il protagonista affronti un certo numero di livelli e sfide, a ognuno legato un particolare tema, inoltre al relativo tema sarà legato anche un nuovo personaggio che si aggiungerà al gruppo ad ogni livello.

Durante il viaggio verranno svelati i motivi dietro al male che sta corrompendo il mondo (7 peccati capitali) e si scoprirà la mente dietro a tutto questo.

In base alle scelte compiute durante il gioco il protagonista può scegliere se arrivare a un finale buono o a un finale cattivo. Nel finale buono, il protagonista, insieme agli alleati che raggrupperà, salverà il mondo dalla corruzione, Nel finale cattivo, invece, il protagonista, insieme ai suoi compagni, costituiranno il nuovo male del mondo.

Specifiche dei requisiti

Le specifiche dei requisiti sono suddivise in due categorie: meccaniche del gioco e sistema di livelli.

Meccaniche del gioco:

1. Il giocatore deve poter personalizzare il proprio personaggio;
2. Il giocatore deve poter scegliere almeno tra tre tipologie di classi, che determineranno:
   a) L'equipaggiamento;
   b) Sistema di statistiche, che influiscono sul combattimento;
   c) Un albero delle abilità, che permette al giocatore di sbloccare abilità uniche utilizzabili in combattimento;
3. Il gioco deve presentare un tutorial che introduce alle meccaniche del gioco;
4. Il giocatore deve potere mettere in pausa il gioco;
5. Il gioco deve poter permettere di salvare la partita e riprenderla in un momento successivo;
6. Il gioco deve presentare un sistema di combattimento a turni;
7. Il giocatore alla fine di ogni combattimento deve ricevere:

a) Un certo numero di punti esperienza, che permetteranno di espandere l'albero delle abilità;

b) Un certo numero di valuta di gioco e materiali, che gli permettono di aumentare/potenziare le statistiche ed equipaggiamento.

8. Il gioco presenta diverse situazioni di stallo (almeno una per livello), in cui il giocatore è tenuto a scegliere tra diverse opzioni che influiscono sul proseguimento del gioco;
9. Il gioco deve presentare almeno due finali.
10. Il gioco deve tenere conto delle scelte fatte durante il gioco che stabiliscono il finale del gioco;

Sistema di livelli:

1. Il gioco deve essere suddiviso in livelli, che a loro volta devono essere suddivisi in aree;
2. Il superamento dei livelli si basa sulla sconfitta di un boss di fine livello;
3. Ciascuna area deve presentare un certo numero di nemici che devono essere sconfitti per raggiungere il boss del livello;
4. Il giocatore può trovare degli oggetti durante l'esplorazione dei livelli, che possono facilitare il proseguimento del gioco;
5. Il gioco propone delle sfide al giocatore che offrono **bonus** in caso di vittoria o **malus** in caso di sconfitta.

Requisiti non funzionali

Dalle specifiche dei requisiti, sono stati dedotti i requisiti non funzionali, elencati di seguito.

1. L'introduzione alle meccaniche del gioco deve essere graduale:
   a) Come criterio di verificabilità, il giocatore non deve essere introdotto a tutte le meccaniche del gioco nella prima fase, ma deve risultare suddiviso in più parti, sparse nelle fasi iniziali del gioco.
2. Il salvataggio deve:
   a) Essere rapido, in particolare non deve essere superare (indicativamente) i 30 secondi;
   b) Poter essere effettuato in qualunque momento al di fuori dei combattimenti;
   c) Essere notificato al giocatore una volta correttamente effettuato;
3. L'interfaccia utente deve essere semplice e intuitiva, in particolare devono essere presenti due menù:
   a) Menù di pausa per permettere:
      - Di riprendere la partita;
      - Di uscire dal gioco;
      - Di salvare la partita;
   b) Menù di gioco per permettere:
      - Di visualizzare le proprie statistiche;
      - Di visualizzare e usare gli oggetti raccolti durante i livelli;
      - Di visualizzare e modificare il proprio albero delle abilità;
   c) La schermata di combattimento non deve presentare troppi pulsanti;
4. Requisiti organizzativi:
   a) Il processo di sviluppo dovrà seguire la Metodologia Agile, con particolare riferimento a Scrum;
   b) Il linguaggio di programmazione del software sarà Java;
   c) I software che usati per lo sviluppo includono:
      - L'IDE IntelliJ IDEA;
      - Il CVS Git e la piattaforma Github;
      - Il DBMS MySQL;
      - Strumenti per il disegno del sistema software in UML;

Requisiti di fidatezza

salvataggio

Requisiti funzionali

I requisiti funzionali individuati sono stati definiti nella forma di user story, usando il modello "Who, what, why" e la specifica sintassi "As a <type of user> i want <some goal> so that <some reason>, includendo per ognuna di esse uno o più criteri di accettazione.

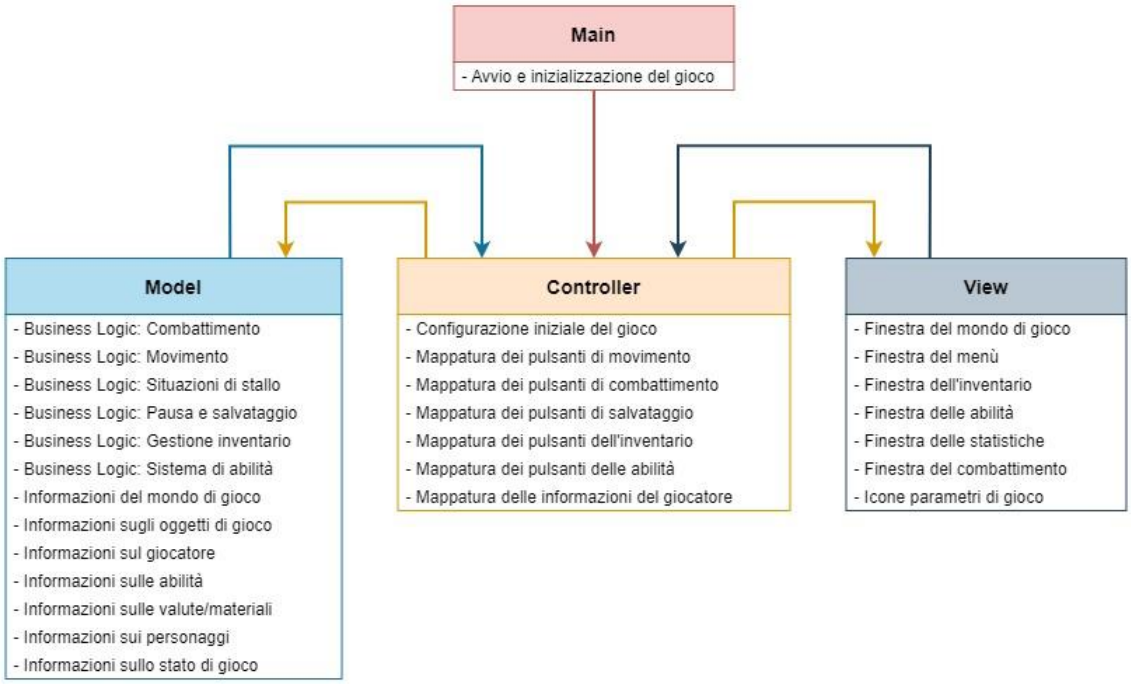
Le user story prodotte hanno contribuito alla formazione del product backlog. A ogni user story è stata assegnata una priorità, un tipo, un criterio di accettazione, un tema, story points e uno status. Per gli story points è stato deciso di utilizzare come unità di misura le taglie delle t-shirt, scelta che si è rivelata vantaggiosa al momento del poker planning. Il product backlog viene elencato di seguito.

ID	Priority	Type	Item	Criteria	Theme	SP	Status
1	**1**	Story	In qualità di giocatore, voglio potermi muovere nei livelli in modo tale da poterli esplorare	Il giocatore deve poter: - Premere uno tra i pulsanti W,A,S,D - Visualizzare il personaggio muoversi verso le rispettive direzioni SU,GIU', SX e DX	Meccaniche	**L**	Completed
2	**1**	Story	In qualità di giocatore, voglio poter avviare un combattimento con un nemico in modo tale da avanzare nella storia	Il giocatore deve poter: - Visualizzare i nemici nella mappa di gioco - Decidere se affrontarli - Avviare un combattimento	Meccaniche	**M**	Completed
3	**1**	Story	In qualità di giocatore, voglio poter attaccare con l'arma in modo tale da infliggere al nemico danni in combattimento	Il giocatore deve poter: - Premere un tasto durante il combattimento - Infliggere un certo danno in base alle proprie statistiche	Meccaniche	**M**	Completed
4	**2**	Story	In qualità di giocatore, voglio poter utilizzare le abilità in modo tale da infliggere danni al nemico in combattimento	Il giocatore deve poter: - Premere un bottone - Scegliere l'abilità - Attaccare il nemico	Meccaniche	**M**	Completed
5	**2**	Story	In qualità di giocatore, voglio poter prendere decisioni in modo tale poter influenzarne il proseguimento	Il giocatore deve poter: - Visualizzare almeno due pulsanti - Sceglierne solo uno - Influenzare (aumentare/diminuire) le caratteristiche del gioco	Meccaniche	**L**	Completed
6	2	Epic	In qualità di sistema, voglio poter suddividere il gioco in livelli in modo tale da strutturare la storia	Ogni livello deve corrispondere a una specifica parte della storia, contenendo oggetti, personaggi, nemici e sfide uniche del livello. Il completamento di un livello fornisce l'accesso al successivo.	Sistema	XL	In progress
7	**2**	Story	In qualità di sistema, voglio poter fornire un primo livello in modo tale da introdurre la storia	Il primo livello deve corrispondere alla parte iniziale della storia di gioco, contenere due mappe di gioco ognuna con almeno una decisione da prendere e vari nemici da affrontare.	Sistema	**L**	Completed
8	**2**	Story	In qualità di sistema, voglio poter fornire un secondo livello in modo tale da sviluppare la storia	Il secondo livello deve corrispondere alla parte centrale della storia di gioco, contenere tre mappe di gioco ognuna con almeno una decisione da prendere, nemici da affrontare e sfide da superare	Sistema	**L**	
9	**2**	Story	In qualità di sistema, voglio poter fornire un terzo livello in modo tale da concludere la storia	Il terzo livello deve corrispondere alla parte finale della storia di gioco, contenere tre mappe di gioco ognuna con almeno una decisione da prendere, nemici da affrontare, sfide da superare boss finale da sconfiggere	Sistema	**L**	

10	**2**	Story	In qualità di giocatore, voglio poter visualizzare la mia salute e stamina, durante l'esplorazione, in modo tale da decidere come proseguire il combattimento	Il giocatore deve poter: - Visualizzare nella schermata delle barre che indicano la salute e la stamina del personaggio	Meccaniche	**S**	
11	**2**	Story	In qualità di giocatore, voglio poter visualizzare il mio attuale obiettivo in modo tale da sapere cosa devo fare	Il giocatore deve poter: - Visualizzare nella schermata una finestra con scritto l'obiettivo attuale	Meccaniche	**M**	
12	**2**	Bug	Più finestre si aprono durante lo switch tra combattimento e esplorazione	Deve essere presente un solo JFrame comune per esplorazione e combattimento, per tutto il gioco	Meccaniche	**M**	Completed
13	**3**	Story	In qualità di giocatore, voglio poter visualizzare il mio inventario in modo tale da utilizzare i miei oggetti, anche in combattimento	Il giocatore deve poter: - Premere un tasto (I) - Scorrere la schermata degli oggetti - Selezionare l'oggetto da scegliere se utilizzare			
14	**3**	Story	In qualità di giocatore, voglio poter visualizzare una descrizione degli oggetti in modo tale da sapere il loro effetto, anche in combattimento	Il giocatore deve poter: - Premere un tasto (I) - Scorrere la schermata degli oggetti - Scegliere un oggetto da vederne l'effetto			
15	**3**	Story	In qualità di giocatore, voglio poter mettere in pausa il gioco in modo tale da riprenderlo più avanti	Il giocatore deve poter: - Premere un tasto (Esc) - Visualizzare un menu - Aspettare il tempo necessario - Premere lo stesso tasto per tornare in partita			
16	**3**	Story	In qualità di giocatore, voglio poter mettere in pausa il gioco In modo tale da salvare i miei progressi	Il giocatore deve poter: - Premere un tasto (Esc) - Visualizzare un menu - Premere un bottone per salvare - Visualizzare se il salvataggio ha successo			
17	**3**	Story	In qualità di giocatore, voglio poter mettere in pausa il gioco in modo tale da uscire da gioco	Il giocatore deve poter: - Premere un tasto (Esc) - Visualizzare un menu - Premere un bottone per uscire			
18	**3**	Story	In qualità di giocatore, voglio poter avviare un combattimento con un nemico in modo tale da guadagnare dei materiali	Il giocatore deve poter: - Visualizzare i nemici - Decidere se affrontarli - Combattere i nemici - Vinto il combattimento ricevere dei materiali			
19	**3**	Epic	In qualità di giocatore, voglio poter affrontare delle sfide durante i livelli in modo tale da ottenere dei vantaggi	Durante i livelli, il giocatore deve poter interagire con un personaggio che gli propone la sfida e decidere se iniziarla o meno			

20	**3**	Story	In qualità di giocatore, voglio poter visualizzare i miei punti esperienza in modo tale da sapere se posso potenziare l'albero delle abilità	Il giocatore deve poter: - Premere un tasto (I) - Cambiare finestra - Scorrere la schermata delle abilità			
21	**3**	Story	In qualità di giocatore, voglio poter visualizzare l'albero delle abilità in modo tale da avere un resoconto delle mie abilità, anche in combattimento	Il giocatore deve poter: - Premere un tasto (I) - Cambiare finestra - Scorrere la schermata delle abilità			
22	**3**	Story	In qualità di giocatore, voglio poter visualizzare l'albero delle abilità in modo tale da poter espandere le mie abilità	Il giocatore deve poter: - Premere un tasto (I) - Cambiare finestra - Scorrere la schermata delle abilità scegliere - Selezionare l'abilità da scegliere se espandere			
23	**4**	Story	In qualità di giocatore, voglio poter visualizzare i miei materiali e valuta in modo tale da sapere se posso potenziare le mie statistiche	Il giocatore deve poter: - Premere un tasto (I) - Cambiare finestra - Scorrere la schermata delle statistiche			
24	**4**	Story	In qualità di giocatore, voglio poter visualizzare le mie statistiche in modo tale da avere un resoconto del mio personaggio, anche in combattimento	Il giocatore deve poter: - Premere un tasto (I) - Cambiare finestra - Scorrere la schermata delle statistiche			
25	**4**	Story	In qualità di giocatore, voglio poter visualizzare le mie statistiche in modo tale da potenziarle	Il giocatore deve poter: - Premere un tasto (I) - Cambiare finestra - Scorrere la schermata delle statistiche - Scegliere l'equipaggiamento da potenziare			
26	**5**	Story	In qualità di giocatore, voglio poter visualizzare una guida di gioco in modo tale da sapere come si gioca	Il giocatore deve poter: - Premere un tasto (Esc) - Visualizzare un menu - Premere un bottone per cambiare finestra - Visualizzare la guida del gioco			
27	**5**	Story	In qualità di giocatore, voglio poter controllare una mappa in modo tale da sapere in quale punto mi trovo e decidere dove andare	Il giocatore deve poter: - Premere un tasto (M) - Visualizzare una mappa			
28	**6**	Epic	In qualità di giocatore, voglio poter personalizzare il mio personaggio in modo tale da provare diverse esperienze di gioco	All'inizio del gioco, il giocatore deve poter: - Scegliere la classe del personaggio - Scegliere nome e sesso			

Dopo la stesura del product backlog, il team di sviluppo si è concentrato sulla definizione dell'architettura da utilizzare per il progetto e l'implementazione del gioco. L'architettura che è stata ritenuta più consona per lo sviluppo è stata l'architettura MVC (Model View Controller).



Per il disegno dell'architettura si è scelta una raffigurazione più testuale che pittorica.

L'architettura scelta si compone di 4 strati: il model, il controller, e il main.

La parte di "Model" fa riferimento a tutta la business logic necessaria per il funzionamento del gioco, oltre che a una parte di database per conservare tutte le informazioni necessarie per la gestione dei livelli e del giocatore. Si è deciso di utilizzare un database relazionale e il DBMS MySQL.
La parte di "view" si riferisce a tutte le finestre che compongono ciò con cui l'utente interagisce: tra queste figurano, principalmente, la schermata di esplorazione e la schermata di combattimento. La parte di view è stata interamente realizzata in Java Swing.
La parte denominata come "Controller" si riferisce a tutti i controller di gioco che permettono lo scambio di informazioni tra View e Controller. Tra questi, due controller costituiscono le fondamenta delle meccaniche del gioco: il controller di movimento del personaggio e il controller di mappatura dei pulsanti di movimento.
La parte di "Main" rappresenta il punto di ingresso del software. Questa parte si occupa dell'inizializzazione del gioco, intesa come costruzione degli oggetti controller, model e view, oltre che all'avvio dello stesso. Il criterio di funzionamento cardine dell'intero gioco è basato sull'idea che vi sono due possibili macro stati: lo stato di "esplorazione" e lo stato di "combattimento". Lo stato di esplorazione include il movimento del personaggio per i vari livelli, l'esplorazione degli stessi e l'interazione con i personaggi e oggetti di gioco. Lo stato di "combattimento" include

invece il sistema di combattimento a turni specificato nei requisiti funzionali. Quando un giocatore interagisce con un nemico, passa allo stato di “combattimento”. Una volta finito il combattimento si torna allo stato di “esplorazione”. Il processo si ripete per tutto il gioco.

Dopo la stesura del product backlog, il team di sviluppo si è concentrato sulla definizione dell'architettura da utilizzare per il progetto e l'implementazione del gioco. L'architettura che è stata ritenuta più consona per lo sviluppo è stata l'architettura MVC (Model View Controller).

Definizione degli sprint

Seguendo passo per passo la metodologia agile, con particolare riferimento al framework Scrum, dopo la stesura del product backlog e la definizione dell'architettura il team si è riunito per definire la durata degli sprint e redigere il primo sprint backlog. Per la durata degli sprint si è scelto di usare come unità di misura le ore e si è stabilito che ogni sprint abbia una durata di 20 ore, da suddividere in 10 giorni lavorativi.

Ogni user story è stata suddivisa in task per facilitare il processo di sviluppo e ognuno di essi è stato assegnato a un particolare membro del team, con riferimento alle competenze tecniche di ognuno. Le user story si definivano completate (done) quando i criteri di accettazione risultavano verificati. Approssimativamente, le ore di sviluppo calcolate risultano pari a 100. Alla fine del progetto, risultano effettuati due sprint, i cui sprint goal sono stati entrambi pienamente raggiunti. Di seguito, si elencano i due sprint backlog di riferimento.

Primo sprint backlog

Goal:	*Completare le meccaniche di movimento e combattimento basilari*			IEH:	*100*					
ID	**Item/Acceptance Critera**	**Tasks**	**SP**	**Owner**	**Planned hours**	**Giov**	**Ven**	**Sab**	**Mer**	**Giov**
1	In qualità di giocatore, voglio potermi muovere nel livelli in modo tale da poterli esplorare		**L**			Remaining hours				
	- Premere uno tra i pulsanti W,A,S,D - Visualizzare il personaggio muoversi verso le rispettive direzioni SU,GIU', SX e DX	Creare view mappa di gioco		Andrea						
		Controller movimento personaggio		Raffaele						
		Business logic: movimento personaggio		Raffaele						
		Disegno delle entità di gioco su schermo		Raffaele						
2	In qualità di giocatore, voglio poter avviare un combattimento con un nemico in modo tale da avanzare nella storia		**M**			Remaining hours				
	- Visualizzare i nemici nella mappa di gioco - Decidere se affrontarli - Avviare un combattimento	Creare view combattimento		Andrea						
		Business logic: avvio di un combattimento		Andrea						
		Creare controller combattimento		Alessia						
3	In qualità di giocatore, voglio poter attaccare con l'arma in modo tale da infliggere al nemico danni in combattimento		**M**			Remaining hours				
	- Premere un tasto durante il combattimento - Infliggere un certo danno in base alle proprie statistiche	Business logic: sistema di statistiche		Alessia						
		Business logic: combattimento a turni		Giovanni						
		Progetto database entità di gioco		Giovanni						

Secondo sprint backlog

Goal:	*Completare il primo livello*			IEH:	*100*					
ID	**Item/Acceptance Critera**	**Tasks**	**SP**	**Owner**	**Planned hours**	**Giov**	**Ven**	**Sab**	**Mer**	**Giov**
6	"In qualità di giocatore, voglio poter utilizzare le abilità in modo tale da avere vantaggi in combattimento"		**M**			Remaining hours				
	Il giocatore deve poter: - Premere un bottone - Scegliere l'abilità - Attaccare il nemico	Business logic: skill tree e combattimento		Giovanni						
		Controller combattimento: abilità		Alessia						
		View abilitá		Andrea						
7	"In qualità di giocatore, voglio poter prendere decisioni in modo tale poter influenzare il proseguimento del gioco"		**L**			Remaining hours				
	Il giocatore deve poter: - Visualizzare almeno due pulsanti - Sceglierne solo uno - Influenzare (aumentare/diminuire)le caratteristiche del gioco	Business logic: NPC e dialoghi		Andrea						
		Disegno NPC su schermo		Raffaele						
		Controller: dialoghi		Andrea						
		Business logic: sistema decisionale		Raffaele						
		Controller: sistema decisionale		Raffaele						
		Realizzazione sprite NPC		Andrea						
		View: dialoghi		Andrea						
8	In qualità di sistema, voglio poter fornire un primo livello di gioco in modo tale da introdurre la storia		**L**			Remaining hours				
	Il primo livello deve corrispondere alla parte iniziale della storia di gioco, contenere due mappe di gioco con varie situazioni di stallo	Connessione al DB		Giovanni						
		Business logic: livelli e mappe		Alessia						
		Realizzazione sprite seconda mappa		Andrea						
		Controller livelli e mappe		Raffaele						
25	Bug: più finestre si aprono durante lo switch tra combattimento ed esplorazione		**M**							
	Deve essere presente un solo JFrame comune per esplorazione e combattimento, per tutto il gioco			Andrea						

Per una maggiore chiarezza, la parte di progettazione è suddivisa in tre macroaree, corrispondenti alle meccaniche principali del gioco: esplorazione, combattimento e inizializzazione.

Esplorazione

Entità di gioco: Player e Mob

Sviluppatore: Raffaele Terracino

Le entità di gioco individuate nel primo sprint sono state il Player, rappresentante del personaggio controllato dall'utente, e Mob, rappresentante dei nemici con cui il Player può combattere. Player e Mob incapsulano un oggetto di tipo Statistics, che incapsula le statistiche del combattimento, e rispettivamente un oggetto di tipo PlayerSprite e MobSprite, che incapsulano le informazioni sulle immagini da stampare a schermo. La classe Player contiene anche il metodo move(), che si occupa, in base all'EventType passato, di aggiornare i due interi worldX e worldY dell'oggetto playerSprite, sommando o sottraendo ad essi l'offset movementSpeed, pari a 10. Il metodo move è stato sviluppato seguendo il TDD. Seguono i test progettati per tale metodo.

```
class PlayerTest {
    private Player p;

    @BeforeEach
    void setUp() {
        p = new Player();
    }

    @AfterEach
    void tearDown() {

    }

    @Test
    void testMove() {
        p.move(EventType.MOVED_DOWN);
        assertEquals(130, p.getPlayerSprite().getWorldY());
        p.move(EventType.MOVED_UP);
        assertEquals(120, p.getPlayerSprite().getWorldY());
        p.move(EventType.MOVED_RIGHT);
        assertEquals(110, p.getPlayerSprite().getWorldX());
        p.move(EventType.MOVED_LEFT);
        assertEquals(100, p.getPlayerSprite().getWorldX());
    }
}
```

Il costruttore di default imposta il Player alla posizione di default, utilizzata nei test.

Le classi MobSprite e Mob non contengono altri metodi oltre i getter e setter, per cui non è stato necessario effettuare i test di unità.

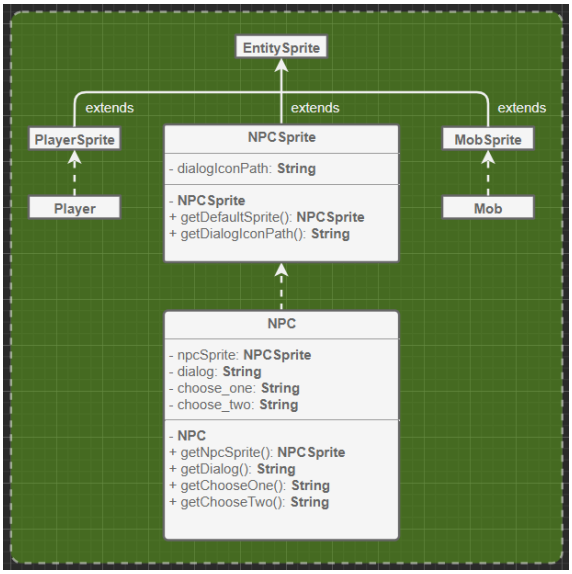
Entità di gioco: NPC

Sviluppatore: Andrea Spinelli

- Introduzione a cos'è un npc --

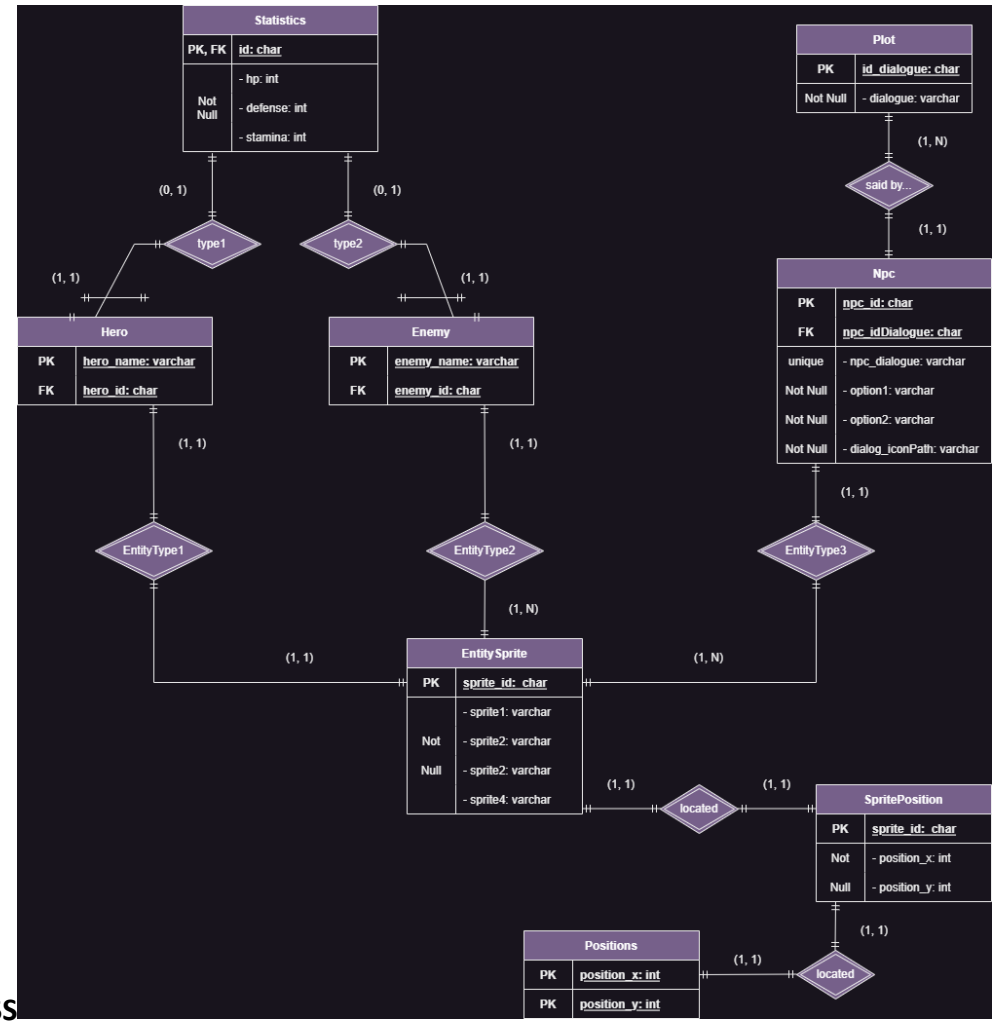
Si è notato come la classe `NPC` possa detenere i dati dei Dialoghi; pertanto, quest'ultimi sono stati inseriti al suo interno, si può osservare che il funzionamento è molto simile ai dati contenuti nei Mob per il combattimento.

`NPC` inoltre implementa la classe `NPCSprite`, la quale questa detiene i dati riguardo gli Sprite e i relativi metodi `get` per la loro restituzione.



Progettazione database relazionale per le entità di gioco

Sviluppatore: Giovanni Bonura



SS

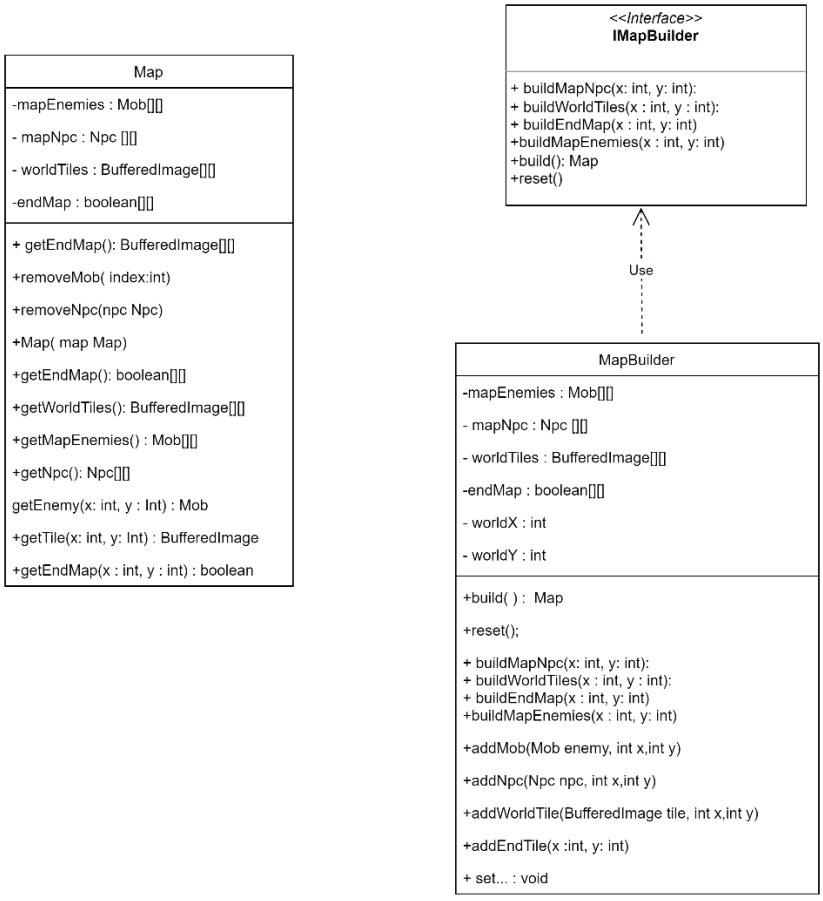
Nel primo sprint si è prevista la presenza di un database relazionale per le entità di gioco, come anche suggerito dall'architettura scelta MVC. Di seguito verrà allegato lo schema concettuale del DB.
Come si può vedere dallo schema concettuale, per ogni entità sono stati definiti gli attributi, specificandone 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.

Sistema di mappe e livelli

Sviluppatore: Alessia Bonì



Classe Map

La classe Map contiene quattro matrici:

- mapEnemies: contiene le posizioni e i riferimenti dei nemici
- mapNpc : contiene le posizioni e i riferimenti agli npc
- worldTiles: contiene tutti i tile del mondo relative all'esplorazione
- endMap: matrice booleana che contiene la fine della mappa: ovvero i tile che indicano al controller di cambiare mappa

Metodi get restituiscono sia le singole matrici ma, permettono anche di ottenere il singolo nemico ,tile o Npc.

Classe MapBuilder

Classe Map, si avvantaggia di un builder per la costruzione dei suoi oggetti, in quanto essendo quattro matrici, il costruttore prenderebbe a parametro 8 coordinate, causando il problema dei costruttori telescopici, inoltre il pattern builder apporta un vantaggio fondamentale: costruzione Step-by-Step degli oggetti, che permette un codice più leggibile e una costruzione semplificata e organica dei singoli oggetti

Il sistema è così composto:

- Un'interfaccia Builder che propone i metodi principali per costruire i vari oggetti
- Un classe MapBuilder che implementa builder, che supporta Map
- La classe Map che contiene la logica fondamentale delle mappe

In *MapBuilder* sono presenti anche dei metodi set per facilitare il riuso di tutte le matrici , e i metodi build… relativo alle matrici da costruire che prende a parametro le coordinate della grandezza della specifica mappa, con infine il metodo build() che restituisce un oggetto completo di tipo Map

Oltre ai vari metodi build…() si sono forniti anche dei metodi specifici per costruire i singoli Nemici, Npc , worldTile e endTile per semplificare la costruzione della mappa, rispettivamente addMob,addNpc,addWorldTile,addEndTile che prendono a parametro le cordinate per aggiungere l'elemento.

Il metodo reset( ) setta tutti i parametri a null, in caso d'errore di inserimento

Classe Level

Level
- counter: int - maps: ArrayList<Map>
+ switchMap( ) + Level(ArrayList<Map> maps) + getCurrentMap(): Map

La classe Level si occupa dell'organizzazione di un livello, contiene un ArrayList di mappe: maps che contiene due mappe per livello, e un counter che permette di scegliere le mappe

Il metodo ***switchMap***: si occupa di cambiare la mappa, ovvero passare da Mappa 1 a Mappa 2 rispettivamente maps(0) e maps(1) incrementando o decrementando il counter.

Il costruttore si occupa di aggiungere le mappe passate a parametro , i metodi get restituiscono il counter e la mappa corrente.

View dell'esplorazione

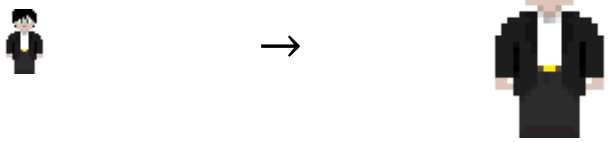
Sviluppatore: Andrea Spinelli

In base alle conoscenze personali del membro, si è deciso, che per la realizzazione degli Sprite, di dividere il lavoro su due supporti: uno web "piskelapp.com", per la realizzazione dei personaggi, e un'applicazione "Paint.net" per la realizzazione della mappa.

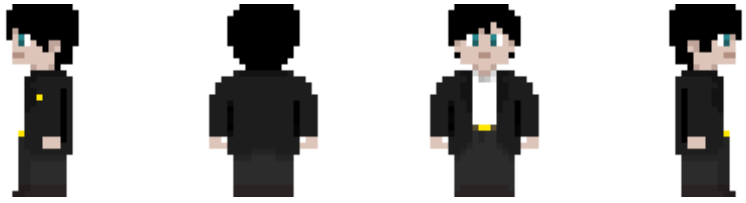
Per Sprite s'intende quell'elemento grafico di gioco, di una determinata dimensione, che descrive visivamente: un personaggio, una zona della mappa….

Nel nostro si è deciso di creare Sprite di dimensione 32×32 pixel, per avere una visibilità minima dei dettagli, dopodiché questi verranno ridimensionati in modo consono agli schermi attuali.

32×32 pixel **Ridimensionamento** **96×96 pixel**



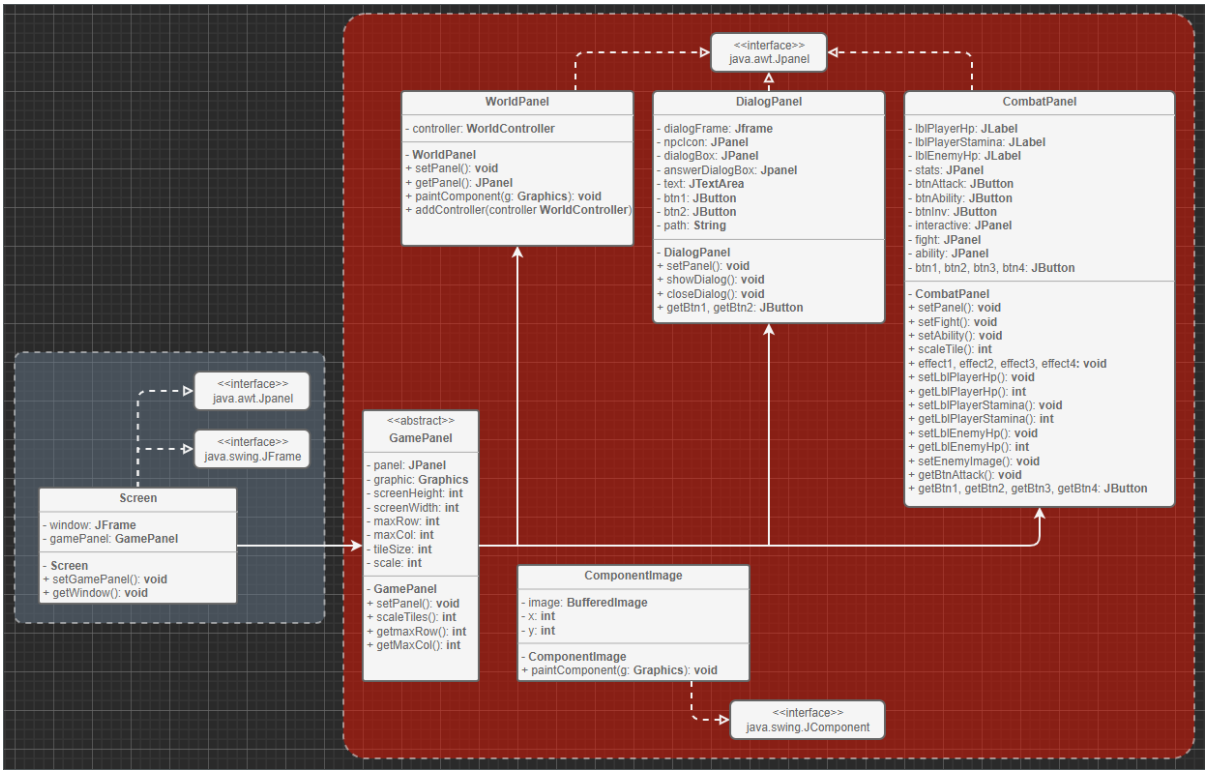
Mediante il supporto web si è innanzitutto il personaggio giocabile in varie posizioni, in modo tale da dare una bozza di movimento grafico nel gioco



Successivamente si è passato alla realizzazione di alcuni Mob applicando la stessa metodologia.

In seguito, mediante l'applicazione, grazie all'uso dei livelli grafici, si sono potuti realizzare gli Sprite della mappa, avendo sempre il riferimento dello Sprite precedente; si è fatto uso, inoltre, di una griglia che ha permesso di comporre visivamente la prima mappa:

Innanzitutto, si è deciso che per la realizzazione delle view si è deciso di far uso delle interfacce AWT e Swing. La finestra del gioco deve essere una sola, pertanto, in vista delle duplici view, World e Combattimento, è stata stabilita un'astrazione in modo tale che le possa racchiudere tutte:

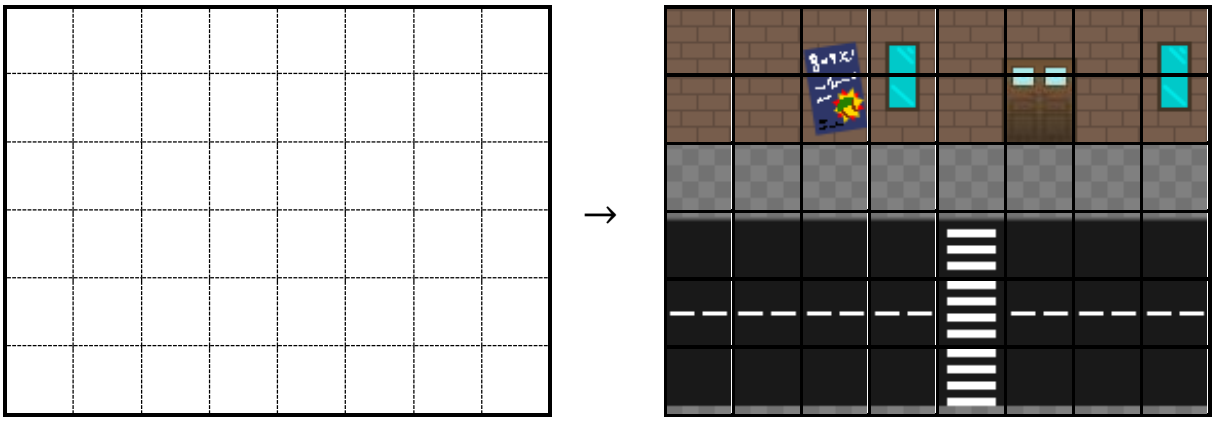


In primi termini generali si ha il seguente funzionamento:

La classe `Screen` include entrambe le librerie AWT e Swing, così facendo, fa da classe principe per la finestra di gioco. La classe introduce due metodi:
`setGamePanel()`, in modo tale da impostare il pannello corretto in base agli eventi del gioco,
`getWindow()`, che restituisce l'intera finestra che verrà fatta visualizzare all'avvia del gioco.

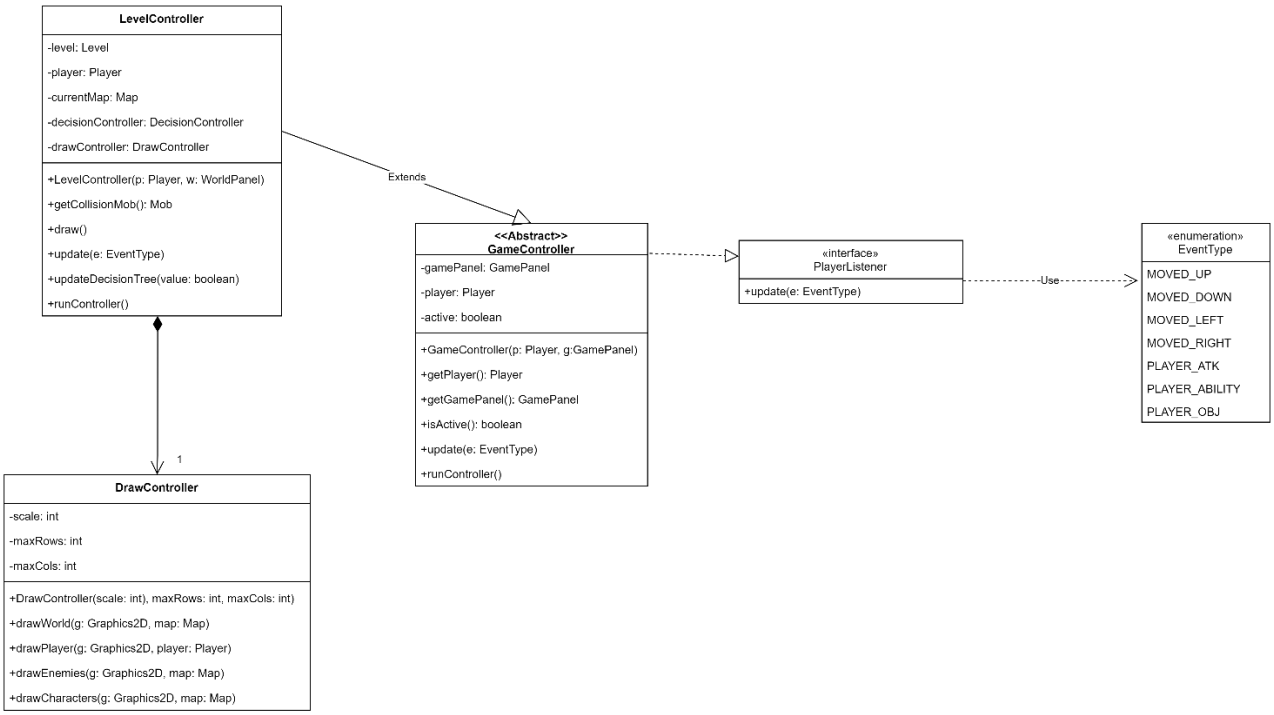
La classe `GamePanel` funziona come astrazione per i vari pannelli delle situazioni di gioco:
`setPanel()`, imposta i componenti grafici di gioco in base al tipo di pannello,
`scaleTale()`, imposta le dimensioni grafiche, in pixel, degli Sprite,
`getMaxRow()`, imposta il numero di righe nella mappa di gioco,
`getMaxCol()`, imposta il numero di colonne nella mappa di gioco.

Per la View: World lo schema si basa semplicemente su un semplice pannello, il quale richiama i metodi `getMaxRow()` e `getMaxCol()` impostando una matrice di gioco dove verranno caricati gli Sprite.



Disegno delle entità di gioco su schermo

Sviluppatore: Raffaele Terracino



La classe LevelController è una sottoclasse di GameController, che rappresenta un generico controller di meccaniche di gioco. La classe GameController incapsula un riferimento a un GamePanel, classe astratta rappresentante di una generica interfaccia di gioco, un riferimento a un Player e un booleano active, che stabilisce lo stato del controller. Oltre il costruttore e il metodo runController, che si occupa di svolgere le funzioni cardine del controller, la classe implementa il metodo update() dell'interfaccia PlayerListener.

Connessione con level e map

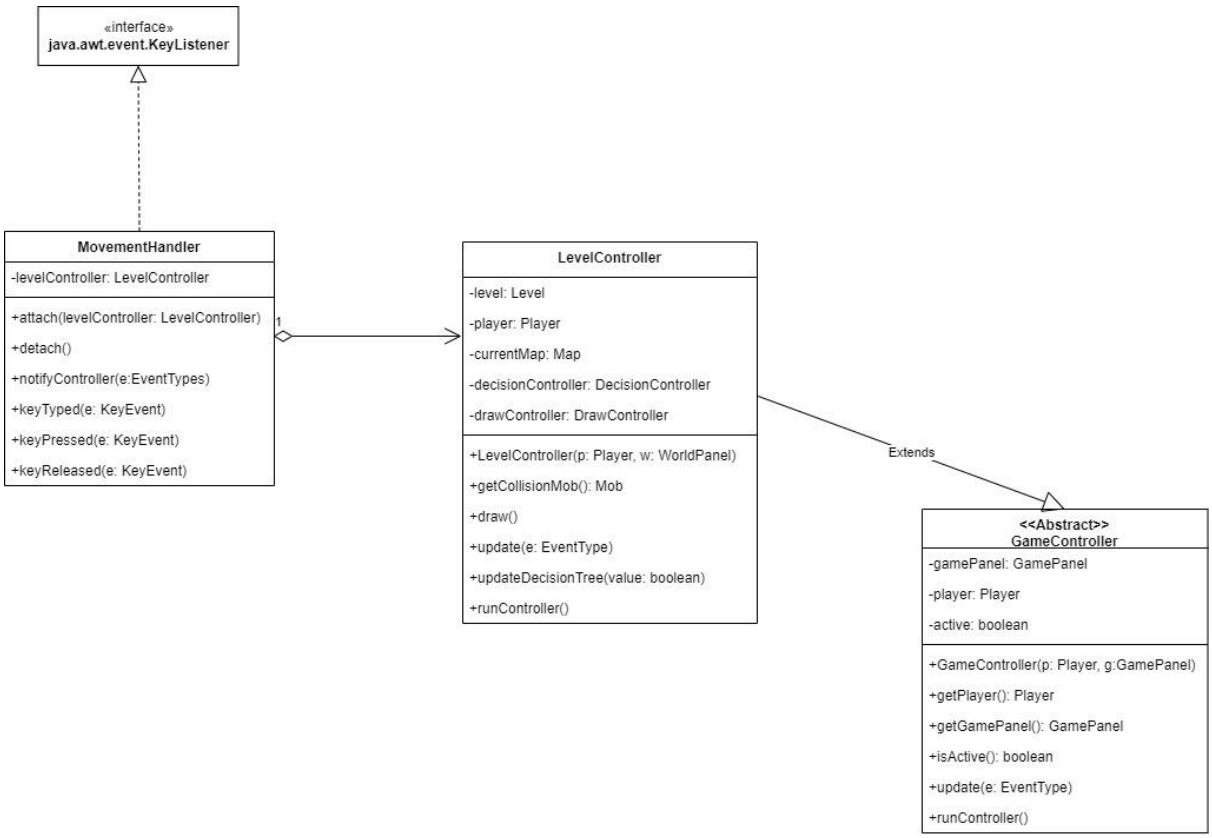
Il disegno delle entità di gioco è gestito dalle classi dalle classi LevelController e DrawController. Il LevelController, all'interno del metodo runController, 60 volte per secondo, chiama il metodo repaint della classe WorldPanel. Il metodo repaint(), proprio della classe JPanel di cui WorldPanel è

una sottoclasse, a sua volta chiama il metodo paintComponent di worldPanel, che non fa altro che chiamare il metodo draw del LevelController passandogli un oggetto di classe Graphics2D, classe di JavaSwing per il disegno di immagini su schermo. Tale metodo chiama, nel seguente ordine, i metodi drawWorld(), drawEnemies(), drawCharacter e drawPlayer(), che si occupano di richiedere a Swing il disegno su schermo del livello e dei personaggi. La logica di disegno si basa sul seguente criterio: poiché la schermata di esplorazione è suddivisa in righe e colonne, all'interno di un doppio ciclo for si esplorano le possibili combinazioni di indici (i, j), prendendo dall'oggetto map immagini di gioco, nemici e personaggi, invocando per ognuno di essi il metodo drawImage() di Graphics2D. Per il giocatore, invece, è sufficiente chiamare una sola volta tale metodo. Questo processo viene ripetuto 60 volte per secondo finché, a causa di un evento, la variabile booleana active viene settata a false, facendo terminare l'esecuzione del metodo runController().

Poiché il disegno delle entità di gioco si basa su meccanismi interamente gestiti da Java Swing e AWT, non è stato possibile sviluppare test di unità per questa parte, in quanto non vi è modo di verificare, se non visivamente, che ogni entità di gioco sia stata correttamente disegnata.

Movimento del personaggio

Sviluppatore: Raffaele Terracino



La classe MovementHandler realizza la logica di business del movimento del personaggio facendo uso del design pattern **Observer**. La classe mantiene un riferimento a un oggetto di classe LevelController e implementa l'interfaccia KeyListener di Java, di cui interessa soltanto il metodo KeyPressed. Quando si vuole che il sistema di movimento sia attivo, basta invocare su un riferimento a MovementHandler il metodo attach, passando un riferimento a un LevelController, che viene settato nella classe. Se invece lo si vuole disattivare basta invocare detach(), che setta a null il riferimento. L'attivazione e disattivazione dinamica del sistema di movimento è utile nel

momento in cui si deve passare dalla schermata di esplorazione a quella di combattimento. Infatti, disattivato il sistema di movimento, se l'utente preme uno dei tasti W, A, S, D durante il combattimento, il personaggio del giocatore rimarrà esattamente alla stessa posizione in cui era prima del cambio di schermata. Quando viene premuto un tasto della tastiera viene invocato il metodo KeyPressed che opera nel seguente modo. All'inizio della sua esecuzione, con un if statement, controlla che il riferimento levelController sia nullo, nel qual caso termina l'esecuzione senza fare altro, altrimenti, in base al tasto premuto, invoca il metodo notifyController passandogli un valore enumerativo EventType secondo il seguente schema:

- il tasto "W" corrisponde a EventType.MOVED_UP;
- Il tasto "A" corrisponde a EventType.MOVED_DOWN;
- Il tasto "S" corrisponde a EventType.MOVED_UP;
- Il tasto "D" corrisponde a EventType.MOVED_UP.

Per altri tasti, il metodo non viene invocato. L'attach del LevelController viene fatto all'inizio del metodo runController() della classe stessa, invece il detach viene fatto prima che il metodo stesso termini l'esecuzione.

Il metodo notifyController invoca il metodo update di LevelController, dichiarato nell'interfaccia PlayerListener, che si occupa di due compiti:

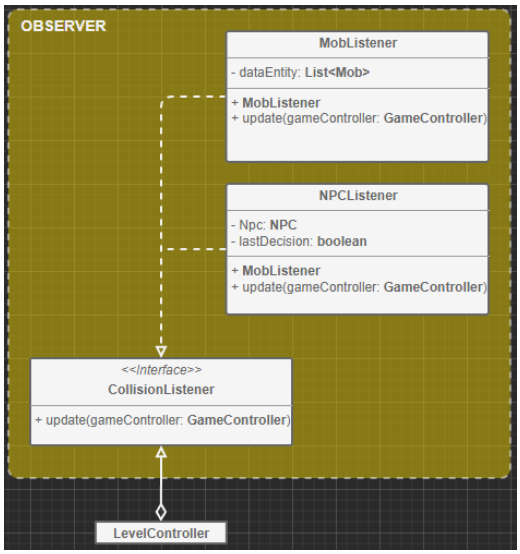
1. di evocare il metodo move sull'oggetto player, che di fatto realizza il movimento
2. di controllare che sia presente una collisione, argomento trattata nella relativa sezione.

Per quanto riguarda i test, il funzionamento del metodo KeyPressed non è testabile poiché esso è gestito interamente dalla JVM. Nemmeno per notifyController sono stati necessari test, poiché l'unica cosa che fa è invocare il metodo update della classe LevelController, per il quale sono stati sviluppati i test di unità.

Sistema di collisioni

Sviluppatore: Andrea Spinelli

Riguardo l'avvio del combattimento si è osservata la necessità di introdurre un Sistema di Collisione, la quale è stato analizzato e identificato nel pattern Observer (accanto si ha lo schema).



La classe `CollisionListener` si mette in ascolto della classe `LevelController`, il listener dopodiché, non appena avviene una collisione con lo Sprite di un Mob, avvia il metodo `update()`, la quale (controllando il tipo di entità con cui ci si scontra) avvia la (relativa) schermata di combattimento, con i dati del Mob a cui ci si è andato a collidere.

`MobListener` presenta:

- Una lista, `dataEntity`, di Mob, in quanto è possibile che in un combattimento si affrontino più nemici;
- `update()`, che trasferisce i dati del Mob al `GameController` e avvia il pannello di combattimento.

Il Controller dei Dialoghi segue la stessa logica del combattimento, questa volta però si ha una collisione con un NPC, quindi notificherà la classe `NPCListener` e chiamerà il metodo update().

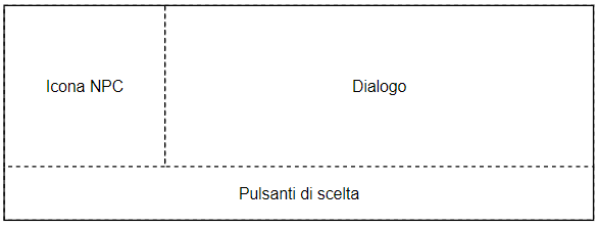
`NPCListener` presenta:

- Una classe, `NPC`, che detiene tutte le informazioni relative al dialogo;
- `update()`, che trasferisce i dati dell'NPC al `GameController` che avvierà una nuova finestra, mantenendo la finestra del mondo intatto.

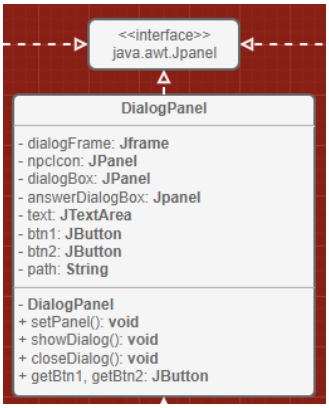
View: dialoghi

Sviluppatore: Andrea Spinelli

Per la View: Dialoghi si è tenuto è osservato che non è necessario l'intero cambiamento del pannello della finestra di gioco, si è pensato quindi l'apertura di un'ulteriore finestra temporanea dove avviene il dialogo stesso e poi chiusa (vedi figura per lo schema).



L'implementazione di tale schema avviene nella classe `DialogPanel`, si nota che in questa, per l'appunto, è stata introdotta una variabile JFrame dove verrà costruito tutto il dialogo al suo interno e mostrata quando richiamata; per una maggiore visione della classe si descrivono in particolare i seguenti metodi:



- `showDialog()`, mostra il dialogo costruito non appena avviene una collisione con l'NPC;
- `closeDialog()`, chiude il dialogo dopo aver dato una risposta;
- `getBtn1`, `getBtn2()`, rispettivamente i metodi per restituire il due pulsanti di scelta del dialogo.

Bisogna precisare che gli ultimi due metodi get servono principalmente per sapere il tipo di scelta che viene effettuata, in modo tale da proseguire nel sistema decisionale, il quale porterà vantaggi o svantaggi a secondo della scelta fatta.

Combattimento

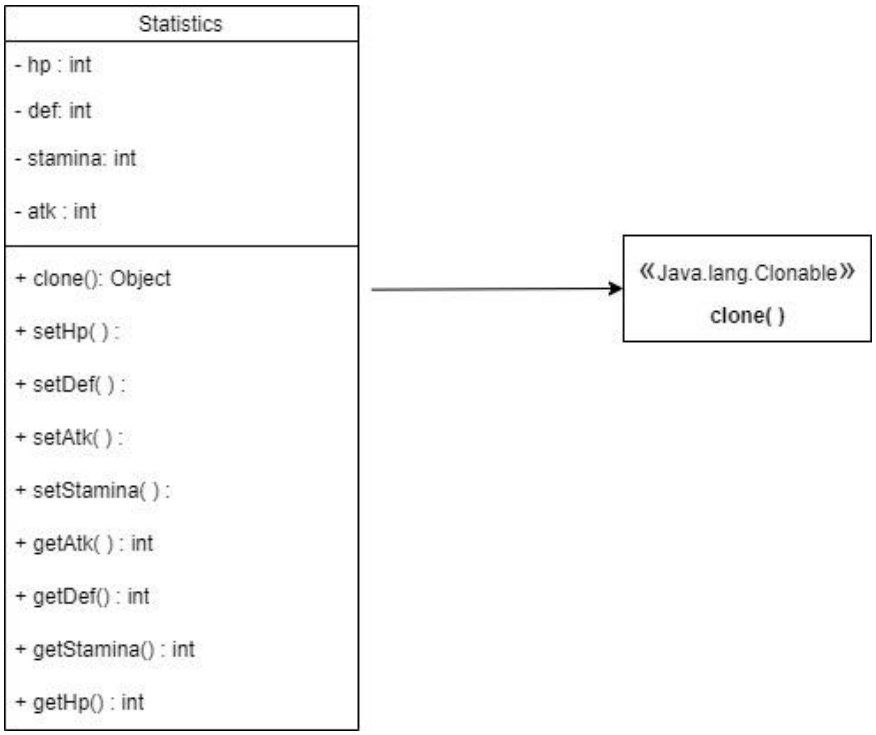
Sistema di statistiche

Sviluppatore: Alessia Bonì

La classe Statistics definisce le statistiche usate dai personaggi/nemici, fondamentali per eseguire le funzioni del sistema di combattimento, quali attacco e abilità speciali

Oltre ai metodi get e set delle statistiche si è pensato di usare il prototype pattern per avere un rapido accesso alle statistiche in quanto molti nemici , specialmente in una mappa riusano le stesse statistiche.

Il secondo utilizzo è per il sistema di combattimento, in quanto usa il clone delle statistiche



Prototype Pattern:

L'implementazione di clone ( ) crea un oggetto dalla classe corrente e trasporta tutte le sue caratteristiche (attributi, riferimenti ad oggetti, ecc.…) su un nuovo oggetto della stessa classe, questo è possibile in quanto clone( ), viene implementato nella stessa classe di cui vogliamo la copia dell' oggetto

In questo caso clone( ) è un metodo implementato nella classe Statistics che implementa l'interfaccia Java.lang.Object.Clonable

Uno dei vantaggi di usare clone( ) è che non si sono ripetute tante linee di codice dovuta all' inizializzazione , utile nelle fasi del sistema di combattimento in quanto non devo creare nuove statistiche ripetutamente per il personaggio e il nemico e una volta finito il combattimento, le statistiche iniziali del giocatore vengono ripristinate

Test:

La classe è stata sviluppata seguendo il TDD:

```java
class StatisticsTest {
    private Statistics stats;
    private Statistics v;

    @BeforeEach
    void setUp() {
        this.stats = new Statistics(100, 20, 50, 20);
    }


    @Test
    void testClone() {
        try {
            this.v = (Statistics)this.stats.clone();
        } catch (CloneNotSupportedException var2) {
            var2.printStackTrace();
        }

        Assertions.assertTrue(this.v instanceof Statistics);
        Assertions.assertEquals(this.stats.getHp(), this.v.getHp());
        Assertions.assertEquals(this.stats.getStamina(), this.v.getStamina());
        Assertions.assertEquals(this.stats.getDef(), this.v.getDef());
        Assertions.assertEquals(this.stats.getAtk(), this.v.getAtk());
    }
}
```

Sono riportati i test :

Sono state creati due oggetti di tipo statistics dov'è *v* sarà semplicemente il clone di *stats.*

In setUp( ) si è inizializzato stats, utile nel metodo testClone( ).

In testClone( ) si testa il metodo clone( ), dato che viene implementato da java.lang.Object.Clonable() , si deve anche implementare un try-catch che in caso di fallimento ,

permetterà un utile devugging tramite printStackTrace( );

Per formalità si testa con Assertion.assertTrue (verifica se una condizione o un boolean è vera)se effettivamente v è un istanza di Statistics.

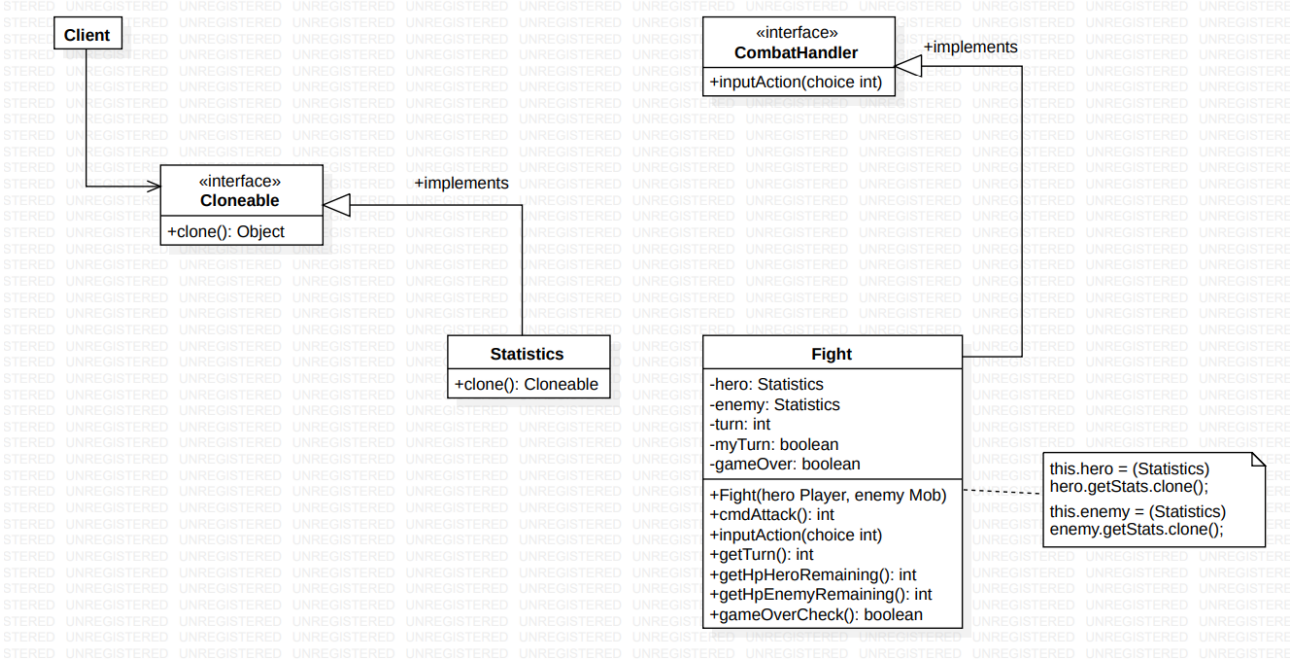
Infine con Assertion.assertEquals si testano l'eguaglianza degli attributi Hp,Stamina,Def,Atk

Sistema di combattimento a turni

Sviluppatore: Giovanni Bonura

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.

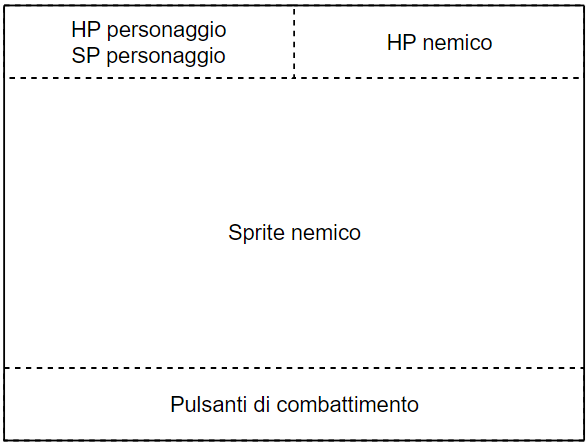
```
class FightTest {
    Statistics stats = new Statistics();
    Player hero = new Player();
    Mob enemy = new Mob(stats);
    private Fight fight;

    @BeforeEach
    void setUp() throws CloneNotSupportedException {
        fight = new Fight(hero, enemy);
        assertNotNull(fight);
    }

    @AfterEach
    void tearDown() {
        fight = null;
        assertNull(fight);
    }

    @Test
    void cmdAttack() {
        assertEquals(75, fight.getHpHeroRemaining() - fight.cmdAttack(),
                "Considering boolean default value of myTurn variable");
    }

    @Test
    void inputAction() {
        do{
            fight.inputAction(1);
        }while(!fight.gameOverCheck());

        assertTrue(fight.gameOverCheck());
```

```
    }


    @Test
    public void gameOverCheck(){
        assertFalse(fight.gameOverCheck(), "We are considering default
value of a boolean");
    }
}
```

View: combattimento

Sviluppatore: Andrea Spinelli

Per la View: Combattimento si è voluto creare una struttura semplice ed intuibile, per questo si è pensato di seguire il seguente schema:



L'implementazione di tale schema, nella classe `CombatPanel`, è stata facilitata grazie ai metodi di Layout per i pannelli che la libreria AWT fornisce; per una maggiore visione della classe si descrivono in particolare i seguenti metodi:

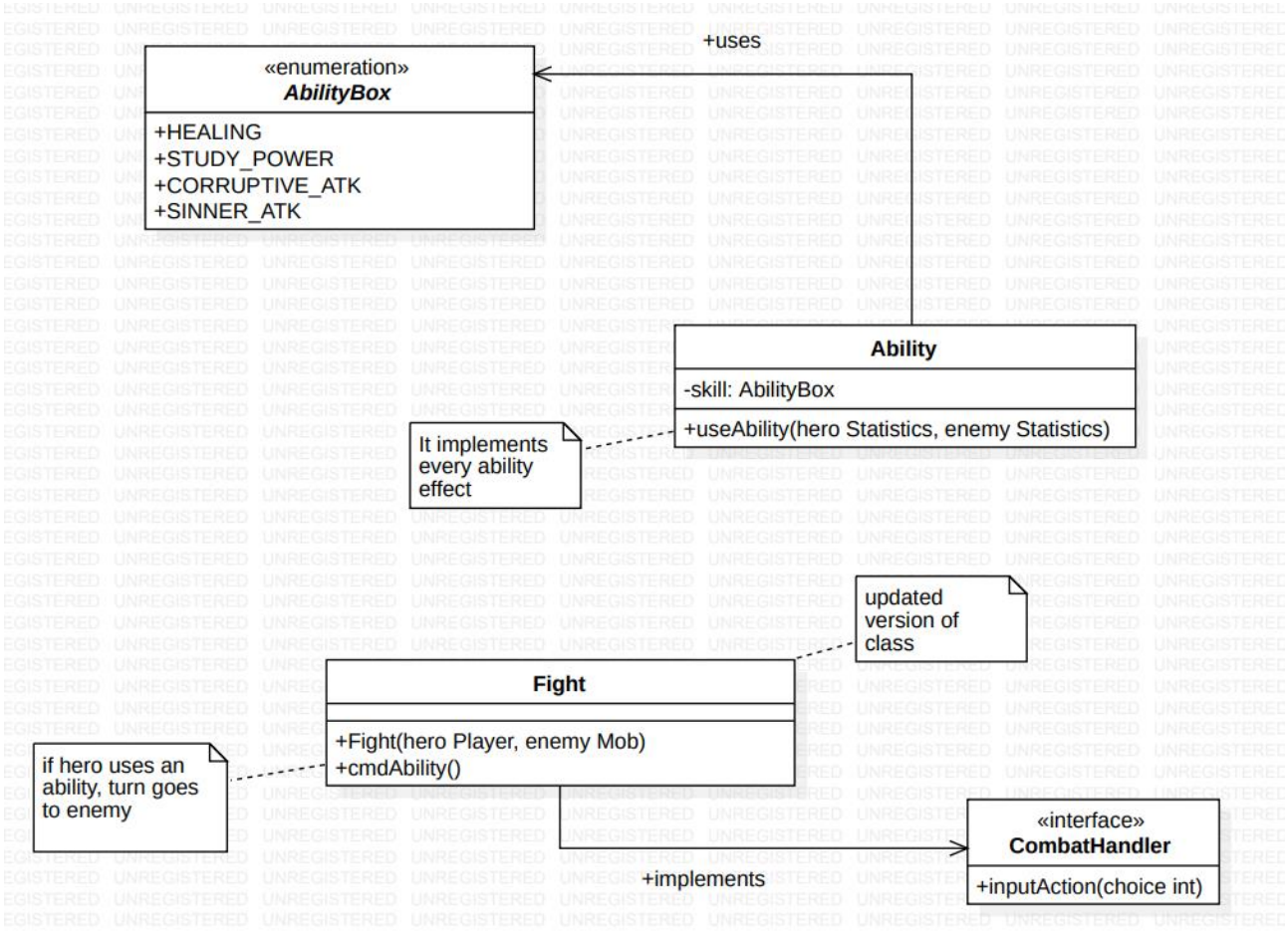
- `setFight()`, imposta/scambia in "Pulsanti di combattimento" un pannello con i pulsanti sul come affrontare il nemico, i quali: *Attack*, *Ability*, *Inventory*;
- `setAbility()`, imposta/scambia in "Pulsanti di combattimento" un pannello specifico per le abilità con i relativi pulsanti, i quali: *Healing*, *Study Power*, *Cursed Attack*, *Sinner Attack*;
- `scaleTile()`, ridimensiona il tile del nemico che va impostato al centro;
- `effect1`, `effect2`, `effect3`, `effect4()`, rispettivamente imposta gli effetti delle relative abilità.

Sistema di abilità

Sviluppatore: Giovanni Bonura

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.

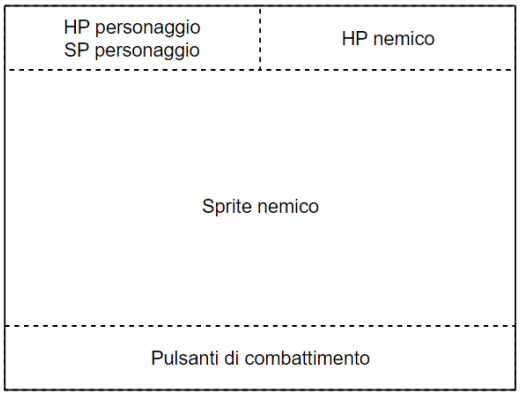
```
class AbilityTest {
    Statistics stats;
    Player hero;
    Mob enemy;

    @BeforeEach
    void setUp()  {
        stats = new Statistics(); assertNotNull(stats);
        hero = new Player(); assertNotNull(hero);
        enemy = new Mob(stats); assertNotNull(enemy);
    }

    @AfterEach
    void tearDown() {
        stats = null; assertNull(stats);
        hero = null; assertNull(hero);
        enemy = null; assertNull(enemy);
    }

    @Test
    void useAbility() {
        Ability skillType_1 = new Ability(AbilityBox.HEALING);
        Ability skillType_2 = new Ability(AbilityBox.STUDY_POWER);
        Ability skillType_3 = new Ability(AbilityBox.CORRUPTIVE_ATK);
        Ability skillType_4 = new Ability(AbilityBox.SINNER_ATK);

        skillType_1.useAbility(hero.getStats(), enemy.getStats());
        assertEquals(130, hero.getStats().getHp());

        skillType_2.useAbility(hero.getStats(), enemy.getStats());
        assertEquals(83, enemy.getStats().getHp());

        skillType_3.useAbility(hero.getStats(), enemy.getStats());
        assertEquals(65, enemy.getStats().getHp());

        skillType_4.useAbility(hero.getStats(), enemy.getStats());
        assertEquals(57, enemy.getStats().getHp());
    }
}
```

View: abilità

Sviluppatore: Andrea Spinelli

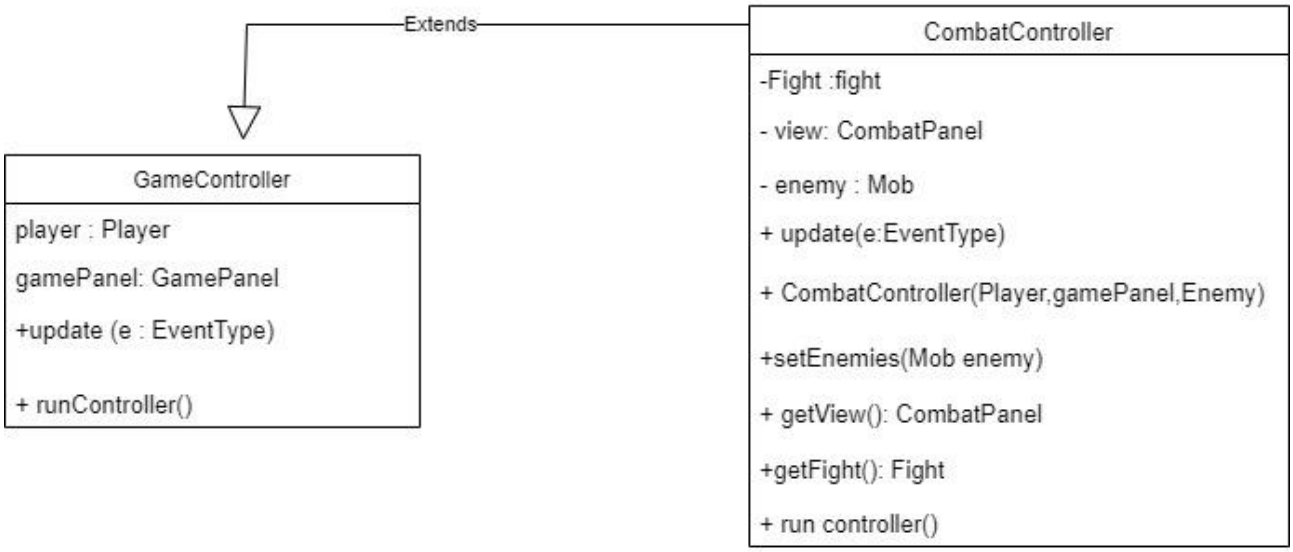
Per la View: Abilità si è preso la struttura precedente del combattimento, impostato già in modo tale da anticipare l'onero dell'implementazione.

La classe CombatPanel non fa altro che richiamare il metodo `setAbility()` che, come descritto in precedenza, imposta/scambia in "Pulsanti di combattimento" un pannello specifico per le abilità con i relativi pulsanti, i quali: *Healing*, *Study Power*, *Cursed Attack*, *Sinner Attack*.



Controller: combattimento

Sviluppatore: Andrea Spinelli



La classe CombatController si occupa di inizializzare e aggiornare le componenti relative al tutto il sistema di combattimento, durante tutte le fasi di esso.

Viene richiamata dalla classe Game nel caso di una collisione con un nemico.

Contiene gli oggetti di tipo Fight e CombatPanel rispettivamente model e view relativi al sistema di combattimento

CombatController: Costruttore

Inizializzazione del model: Fight:

Alla classe fight passo come parametri nel costruttore Player e enemy, fondamentali per il model per eseguire tutte le operazioni quali abilità e attacco

Inizializzazione della view: CombatPanel

inizializzazione delle label relative a Hp player e nemico, stamina del player

Si usano i metodi setLblPlayerHp ,setLblPlayerStamina,setEnemyHp di CombatPanel per settare le statistiche iniziali del nemico e del eroe ,usando i metodi getHpHeroRemaing ,getHeroStamina, getHpEnemyRemaing della classe Fight , che restituiscono le statistiche corrispettive.

Inizializzazione dei Bottoni Fight e Ability: Healing, Corruptive_Atk ,Sinner_Atk, Study_Power

Il bottone Fight si inizializza tramite getBtnFight, a cui si aggiunge un listener che quando il pulsante viene premuto ,esegue queste operazioni:

- richiamerà il metodo cmdAttack di classe Fight,
- passera a inputAction il valore 1,

- invoca il Metodo Update di CombatController passando come parametro il tipo enumerativo: PLAYER_ATK relativo all' enum EventType;

Per i bottoni Healing, Corruptive_Atk, Sinner_Atk, Study_Power, usando i metodi getBtn1,getBtn2, ecc… assegna ad ognuno un distinto listener il quale si occupa di:

- Invocare il metodo setSkill, che prende come parametro una abilità di tipo enumerativo Ability box, la quale riporta gli stessi nomi delle abilità : HEALING , CORRUPTIVE_ATK ,SINNER_ATK ,STUDY_POWER;
- Passare a parametro a inputAction il valore 2
- Invocare il metodo update con parametro enumerativo PLAYER_ABILITY di EventType;

CombactController: Update

Il metodo update si occupa di aggiornare la classe CombatPanel , secondo la business logic della classe Fight.

Riceve a parametro un enum di Tipo EventType, che indica semplicemente se si è scelto in vista un attacco o una abilità:

1. **PLAYER_ATK:** La vista verrà aggiornata con i valori di HP del Player e nemico secondo la business logic del model : in questo caso getHpHeroRemaing e getHpEnemyRemaing. L'aggiornamento avviene attraverso i metodi : setLblPlayerHp e setLblEnemyHp, che prendono a parametro i valori corretti rispettivamente dell'eroe e del nemico.
2. **PLAYER_ABILITY:** Uguale al caso precedente , semplicemente si aggiorna la label relativa alla stamina del player secondo business logic della classe Fight. L'aggiornamento avviene tramite setLblPlayerStamina in cui si passa a parametro la stamina corrente ottenuta dal metodo getHeroStamina contenuto in fight

CombatController : runController

Si occupa di far rimanere il controller attivo attraverso un ciclo while: finche isActive è vero il controller è attivo,quando è falso si distruggono i vari listener si passa il controllo al GameController , che si occupa dell'esplorazione.

CombatController : Test

I test si focalizzano sul simulare un combat controller dichiarando tutte le componenti necessarie per testarlo:

- Un controller di CombatController:
- stats di tipo statistics
- Un Mob enemy
- Un combatPanel view
- Un Player player

Nel metodo setUp i componenti vengono inizializzati, per cui si scrive alcuni numeri per Statistics, passo a Mob stats, e a controller prende a parametro il player, la view e enemy

In updateTets si eseguono i test: si controlla che i parametri passati alla view siano corretti, e che non ci siano differenze fra i valori della view e del model

Attenzione: I valori in view sono visti come una label in questa forma:

Hero Hp: Value, Stamina Hero: Value, Enemy Hp Value

Quindi tramite funzione substring si estrae il valore numerico corretto in questa forma:

this.controller.getWiew().getLblPlayerHp().getText().substring(11,
this.controller.getView().getPlayerHp().getText().lenght()));

Inoltre dato che i metodi getHpHeroReaming ,getHpEnemyRemaing e getHeroStamina restituiscono valori interi ogni volta che vengono richiamati vengono convertiti in stringhe con Integer.toString per facilità di confronto con le substring ottenute dalla view.

I test si dividono in :

- Test del bottone Fight
- Test delle Abilità

Fight Test

Si testa il bottone fight simulando la scelta 1 in ImputAction e invocando update con parametro PLAYER_ATK

Tramite assertEquals confrontiamo due stringhe : quella ottenuta da substring su getLblPlayerHp e Integer.toString su getHpHeroRemaing rispettivamente di view e model.

Con lo stesso procedimento di prima confronto anche getLblEnemyHp e getHpEnemyRemaing per confrontare se anche i valori di hp del nemico combaciano

Ability Test

Per qualsiasi abilità da testare si usa la seguente formula:

this.controller.getFight().setSkill(AbilityBox.ABILITY);

this.controller.getFight().inputAction(2);

this.controller.update(EventType.PLAYER_ABILITY)

Dove AbilityBox.ABILITY : può essere una delle quattro abilità:
HEALING,STUDYPOWER,SINNER_ATK, CORRUPTIVE_ATK

<u>Healing Test:</u>

Per Healing testo con assertTrue se gli hp e la stamina dell' eroe sono visualizzati correttamente in view , quindi si procede analogamente per classe fight:

Un AssertEquals che confrontera gli hp del model con quelli della view relativo al player

E un Secondo AssertTrue che confronta la stamina del model con la stamina della view relativa al player

Ovviamene tutti i valori di classe Fight verrano convertite in stringhe per garantire consistenza con i metodi getLbl di classe CombatPanel

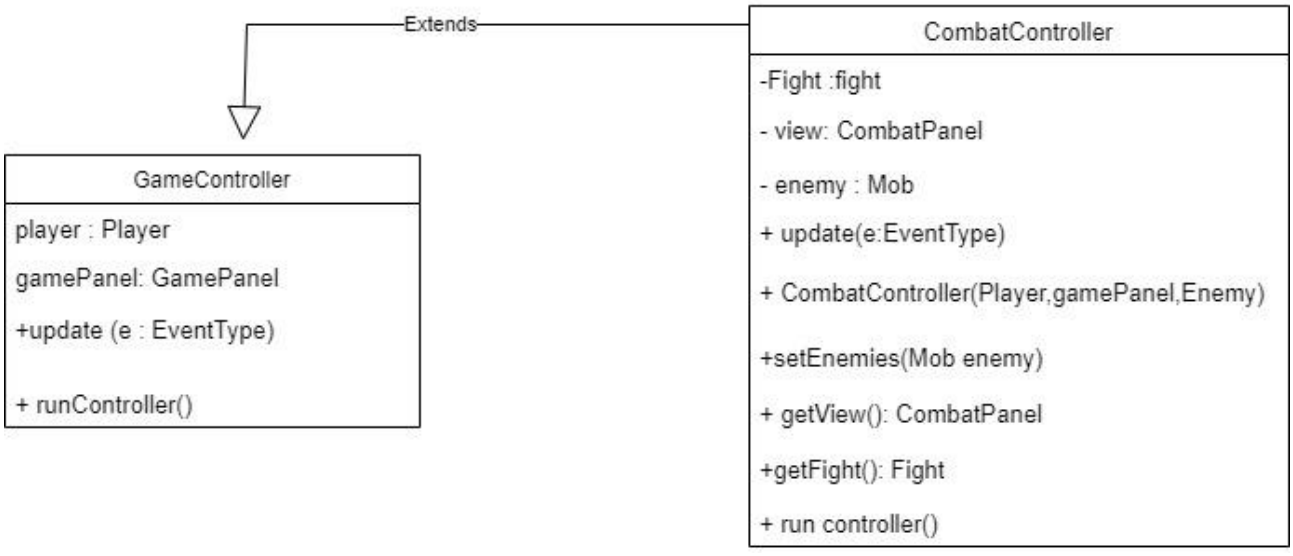
<u>Study_Power ,Sinner_Atk, Corruptive_Atk Test :</u>

Analogamente per Healing e Fight Test si eseguono tre test AssertEquals:

- Un AssertEquals che confronta gli hp relativi al player del model e della view
- Un AssertEquals che confronta gli hp relativi del nemico del model e della view
- Un AssertEquals che confronta la stamina relativa del player in model e in view

Ovviamente tutti i valori fight sono convertiti in stringhe per garantire consistenza con i metodi di CombatPanel

Logic controller per il sistema di combattimento: Classe CombatController



La classe CombatController si occupa di inizializzare e aggiornare le componenti relative al tutto il sistema di combattimento, durante tutte le fasi di esso.

Viene richiamata dal GameController nel caso di una collisione con un nemico.

Contiene gli oggetti di tipo Fight e CombatPanel rispettivamente model e view relativi al sistema di combattimento

CombatController: Metodi

CombatController: Costruttore

Inizializzazione del model: Fight:

Alla classe fight passo come parametri nel costruttore Player e enemy, fondamentali per il model per eseguire tutte le operazioni quali abilità e attacco

Inizializzazione della view: CombatPanel

inizializzazione delle label relative a Hp player e nemico, stamina del player

Si usano i metodi setLblPlayerHp ,setLblPlayerStamina,setEnemyHp di CombatPanel per settare le statistiche iniziali del nemico e del eroe ,usando i metodi getHpHeroRemaing ,getHeroStamina, getHpEnemyRemaing della classe Fight , che restituiscono le statistiche corrispettive.

Inizializzazione dei Bottoni Fight e Ability: Healing, Corruptive_Atk ,Sinner_Atk, Study_Power

Il bottone Fight si inizializza tramite getBtnFight, a cui si aggiunge un listener che quando il pulsante viene premuto ,esegue queste operazioni:

- richiamerà il metodo cmdAttack di classe Fight,
- passera a inputAction il valore 1,
- invoca il Metodo Update di CombatController passando come parametro il tipo enumerativo: PLAYER_ATK relativo all' enum EventType;

Per i bottoni Healing, Corruptive_Atk, Sinner_Atk, Study_Power, usando i metodi getBtn1,getBtn2, ecc… assegna ad ognuno un distinto listener il quale si occupa di:

- Invocare il metodo setSkill, che prende come parametro una abilità di tipo enumerativo Ability box, la quale riporta gli stessi nomi delle abilità : HEALING , CORRUPTIVE_ATK ,SINNER_ATK ,STUDY_POWER;
- Passare a parametro a inputAction il valore 2
- Invocare il metodo update con parametro enumerativo PLAYER_ABILITY di EventType;

CombactController: Update

Il metodo update si occupa di aggiornare la classe CombatPanel , secondo la business logic della classe Fight.

Riceve a parametro un enum di Tipo EventType, che indica semplicemente se si è scelto in vista un attacco o una abilità:

3. **PLAYER_ATK:** La vista verrà aggiornata con i valori di HP del Player e nemico secondo la business logic del model : in questo caso getHpHeroRemaing e getHpEnemyRemaing. L'aggiornamento avviene attraverso i metodi : setLblPlayerHp e setLblEnemyHp, che prendono a parametro i valori corretti rispettivamente dell'eroe e del nemico.
4. **PLAYER_ABILITY:** Uguale al caso precedente , semplicemente si aggiorna la label relativa alla stamina del player secondo business logic della classe Fight. L'aggiornamento avviene tramite setLblPlayerStamina in cui si passa a parametro la stamina corrente ottenuta dal metodo getHeroStamina contenuto in fight

CombatController : runController

Si occupa di far rimanere il controller attivo attraverso un ciclo while: finche isActive è vero il controller è attivo,quando è falso si distruggono i vari listener si passa il controllo al GameController , che si occupa dell'esplorazione.

CombatController : Test

I test si focalizzano sul simulare un combat controller dichiarando tutte le componenti necessarie per testarlo:

- Un controller di CombatController:
- stats di tipo statistics
- Un Mob enemy
- Un combatPanel view
- Un Player player

Nel metodo setUp i componenti vengono inizializzati, per cui si scrive alcuni numeri per Statistics, passo a Mob stats, e a controller prende a parametro il player, la view e enemy

In updateTets si eseguono i test: si controlla che i parametri passati alla view siano corretti, e che non ci siano differenze fra i valori della view e del model

Attenzione: I valori in view sono visti come una label in questa forma:

Hero Hp: Value, Stamina Hero: Value, Enemy Hp Value

Quindi tramite funzione substring si estrae il valore numerico corretto in questa forma:

this.controller.getWiew().getLblPlayerHp().getText().substring(11,
this.controller.getView().getPlayerHp().getText().lenght()));

Inoltre dato che i metodi getHpHeroReaming ,getHpEnemyRemaing e getHeroStamina restituiscono valori interi ogni volta che vengono richiamati vengono convertiti in stringhe con Integer.toString per facilità di confronto con le substring ottenute dalla view.

I test si dividono in :

- Test del bottone Fight
- Test delle Abilità

Fight Test

Si testa il bottone fight simulando la scelta 1 in ImputAction e invocando update con parametro PLAYER_ATK

Tramite assertEquals confrontiamo due stringhe : quella ottenuta da substring su getLblPlayerHp e Integer.toString su getHpHeroRemaing rispettivamente di view e model.

Con lo stesso procedimento di prima confronto anche getLblEnemyHp e getHpEnemyRemaing per confrontare se anche i valori di hp del nemico combaciano

Ability Test

Per qualsiasi abilità da testare si usa la seguente formula:

this.controller.getFight().setSkill(AbilityBox.ABILITY);

this.controller.getFight().inputAction(2);

this.controller.update(EventType.PLAYER_ABILITY)

Dove AbilityBox.ABILITY : può essere una delle quattro abilità:
HEALING,STUDYPOWER,SINNER_ATK, CORRUPTIVE_ATK

Healing Test:

Per Healing testo con assertTrue se gli hp e la stamina dell' eroe sono visualizzati correttamente in view , quindi si procede analogamente per classe fight:

Un AssertEquals che confrontera gli hp del model con quelli della view relativo al player

E un Secondo AssertTrue che confronta la stamina del model con la stamina della view relativa al player

Ovviamene tutti i valori di classe Fight verrano convertite in stringhe per garantire consistenza con i metodi getLbl di classe CombatPanel

Study_Power ,Sinner_Atk, Corruptive_Atk Test :

Analogamente per Healing e Fight Test si eseguono tre test AssertEquals:

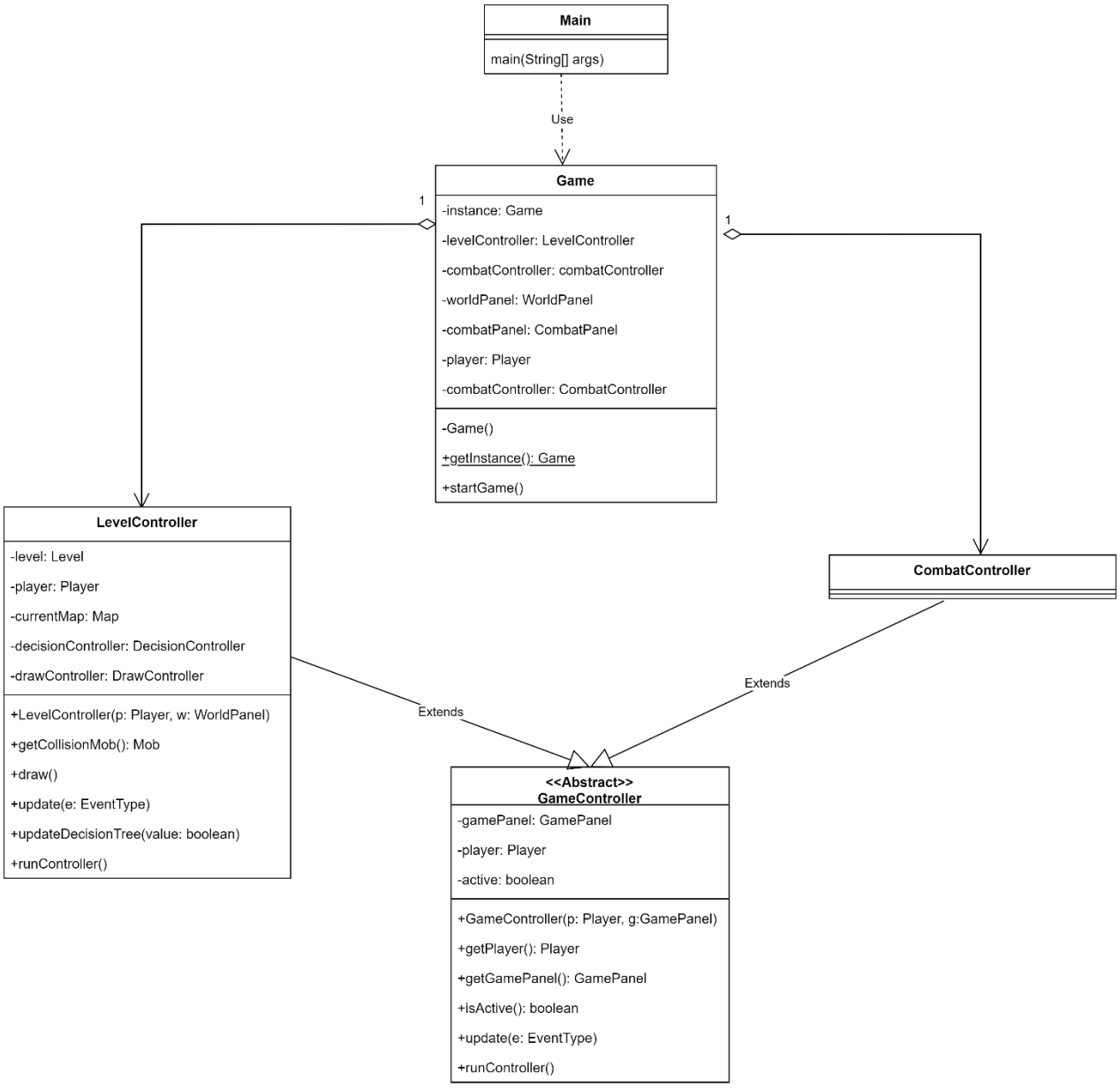
- Un AssertEquals che confronta gli hp relativi al player del model e della view
- Un AssertEquals che confronta gli hp relativi del nemico del model e della view
- Un AssertEquals che confronta la stamina relativa del player in model e in view

Ovviamente tutti i valori fight sono convertiti in stringhe per garantire consistenza con i metodi di CombatPanel

```
class CombatControllerTest {
    private CombatController controller;
    private Statistics stats;
    private Mob enemy;
    private CombatPanel view;
    private Player player;

    @BeforeEach
    void setUp() {
        this.stats = new Statistics(100, 20, 50, 20);
        this.enemy = new Mob(this.stats);
        this.view = new CombatPanel();
        this.player = new Player();
        this.controller = new CombatController(this.player, this.view,
this.enemy);
    }

    @Test
    void updateTest() {
        this.controller.getFight().inputAction(1);
        this.controller.update(EventType.PLAYER_ATK);

Assertions.assertEquals(Integer.toString(this.controller.getFight().getHpHeroRem
aining()), this.controller.getView().getLblPlayerHp().getText().substring(11,
this.controller.getView().getLblPlayerHp().getText().length()));

Assertions.assertEquals(Integer.toString(this.controller.getFight().getHpEnemyRe
maining()), this.controller.getView().getLblEnemyHp().getText().substring(10,
this.controller.getView().getLblEnemyHp().getText().length()));
        this.controller.getFight().setSkill(AbilityBox.CORRUPTIVE_ATK);
        this.controller.getFight().inputAction(2);
        this.controller.update(EventType.PLAYER_ABILITY);

Assertions.assertEquals(Integer.toString(this.controller.getFight().getHpHeroRem
aining()), this.controller.getView().getLblPlayerHp().getText().substring(11,
this.controller.getView().getLblPlayerHp().getText().length()));

Assertions.assertEquals(Integer.toString(this.controller.getFight().getHpEnemyRe
maining()), this.controller.getView().getLblEnemyHp().getText().substring(10,
this.controller.getView().getLblEnemyHp().getText().length()));

Assertions.assertEquals(Integer.toString(this.controller.getFight().getStaminaHe
roRemaining()),
this.controller.getView().getLblPlayerStamina().getText().substring(15,
this.controller.getView().getLblPlayerStamina().getText().length()));
        this.controller.getFight().setSkill(AbilityBox.SINNER_ATK);
        this.controller.getFight().inputAction(2);
        this.controller.update(EventType.PLAYER_ABILITY);

Assertions.assertEquals(Integer.toString(this.controller.getFight().getHpHeroRem
aining()), this.controller.getView().getLblPlayerHp().getText().substring(11,
this.controller.getView().getLblPlayerHp().getText().length()));

Assertions.assertEquals(Integer.toString(this.controller.getFight().getHpEnemyRe
maining()), this.controller.getView().getLblEnemyHp().getText().substring(10,
this.controller.getView().getLblEnemyHp().getText().length()));

Assertions.assertEquals(Integer.toString(this.controller.getFight().getStaminaHe
roRemaining()),
this.controller.getView().getLblPlayerStamina().getText().substring(15,
this.controller.getView().getLblPlayerStamina().getText().length()));
        this.controller.getFight().setSkill(AbilityBox.STUDY_POWER);
```

Avvio e inizializzazione del gioco

Avvio e inizializzazione del gioco

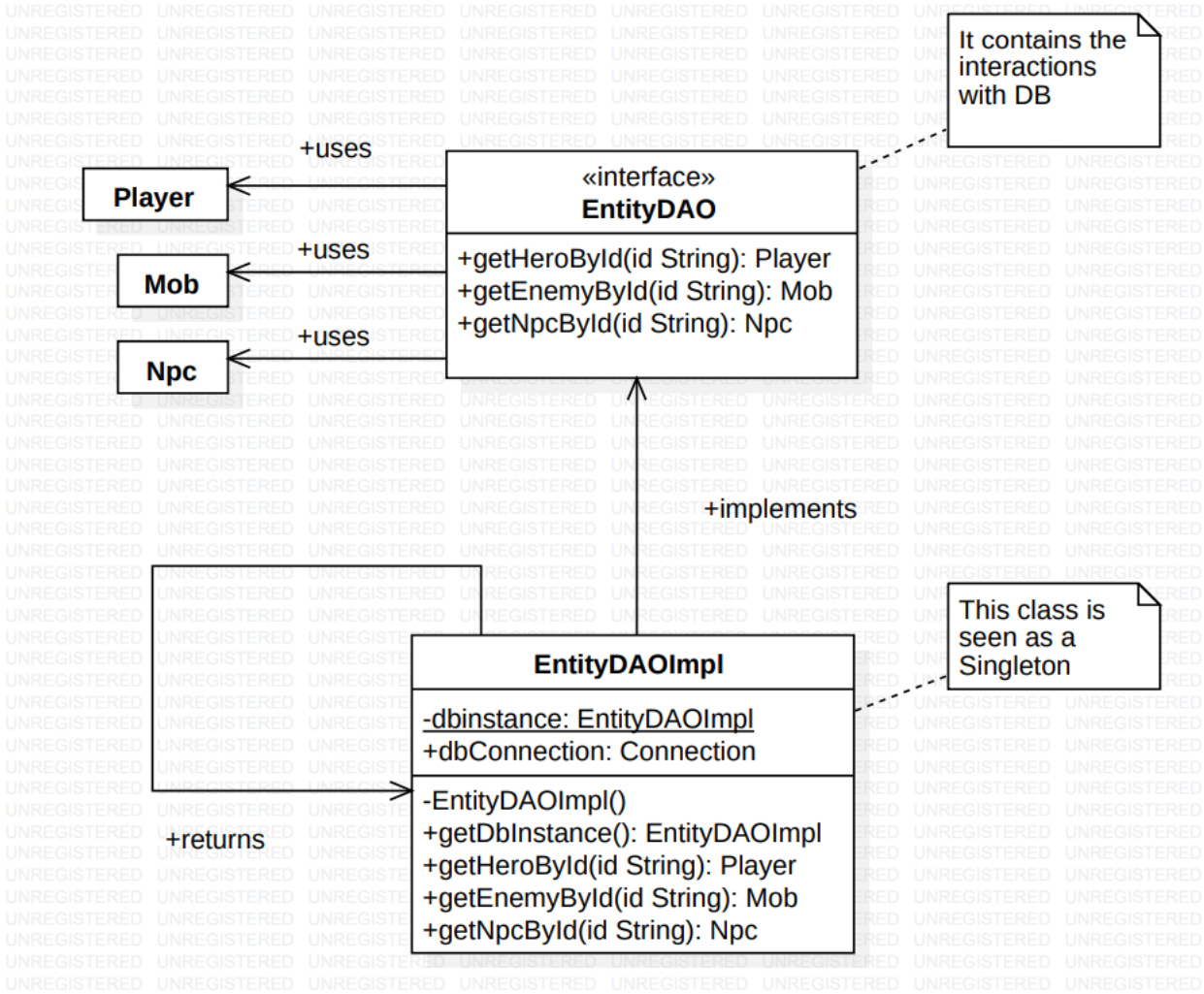


Nel primo sprint ci si è dovuti concentrare sulle meccaniche basilari di gioco: tra queste, era necessario implementare un meccanismo che consentisse il passaggio tra la view dell'esplorazione dei livelli e quella di combattimento. La classe pensata per avere la responsabilità di inizializzare il gioco è la classe Game, astrazione del concetto di "gioco in esecuzione" e pertanto realizzata come Singleton. Il membro del team ha inoltre deciso di applicare il pattern Facade, nascondendo nella classe Main tutta la logica di gioco: il compito di tale classe è richiedere un'istanza di classe Game, classe realizzata come singleton, per poi invocare su di essa il metodo startGame(). All'interno di un ciclo while tale metodo utilizza il metodo runController() dei controller LevelController e CombatController per effettuare il passaggio, quando necessario, tra i due. Il ciclo while termina con la chiusura della finestra di gioco da parte dell'utente.

Connessione al database progettato

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.

```
private EntityDAOImpl() {
    try {
        Class.forName("com.mysql.cj.jdbc.Driver");
        this.dbConnection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/entitystats",
                "agar", "agarproject");
    }

    catch (SQLException | ClassNotFoundException e){
        e.printStackTrace();
    }
}
@Override
public Player getHeroById(String id) {
    Player player = null;
    Statistics stats;
    PlayerSprite ps;
    ArrayList<String> paths = new ArrayList<>();

    try {

        String query = "select hero_hp, hero_defense, hero_atk, hero_stamina,
sprite1, sprite2, sprite3, sprite4, position_x, position_y\n" +
                "from hero join entitysprite on hero.hero_id =
entitysprite.sprite_id join spriteposition on entitysprite.sprite_id =
spriteposition.sprite_id\n" +
                "where hero.hero_id = " + id;

        Statement stmt = this.dbConnection.createStatement();
        ResultSet rs = stmt.executeQuery(query);

        while (rs.next()){
            int hero_hp = rs.getInt("hero_hp");
            int hero_defense = rs.getInt("hero_defense");
```

```
            int hero_atk = rs.getInt("hero_atk");
            int hero_stamina = rs.getInt("hero_stamina");
            String path1 = rs.getString("sprite1");
            paths.add(path1);
            String path2 = rs.getString("sprite2");
            paths.add(path2);
            String path3 = rs.getString("sprite3");
            paths.add(path3);
            String path4 = rs.getString("sprite4");
            paths.add(path4);
            int position_x = rs.getInt("position_x");
            int position_y = rs.getInt("position_y");


            stats = new Statistics(hero_hp, hero_defense, hero_stamina,
hero_atk);
            ps = new PlayerSprite(position_x, position_y);
            ps.setSpritesPath(paths);
            player = new Player(stats, ps);
        }
    }

    catch (Exception e){
        e.printStackTrace();
    }
    return player;
}
```

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.

```
class EntityDAOImplTest {
    Player player = null;
    Mob mob = null;
    Npc npc = null;
    Statistics statsHero, statsEnemy;
    PlayerSprite ps;
    MobSprite ms;
    NPCSprite ns;

    @BeforeEach
    void setUp() {
        statsHero = new Statistics(100, 60,100, 40);
        ps = new PlayerSprite(100, 120);
        ArrayList<String> pathsHero = new ArrayList<>();
        pathsHero.add("src/res/character/move/down/char_down_00.png");
        pathsHero.add("src/res/character/move/up/char_up_00.png");
        pathsHero.add("src/res/character/move/left/char_left_00.png");
        pathsHero.add("src/res/character/move/right/char_right_00.png");
        ps.setSpritesPath(pathsHero);
        player = new Player(statsHero, ps);
        assertNotNull(player);
```

```
        statsEnemy = new Statistics(80, 50, 100, 50);
        ms = new MobSprite(3, 0);
        ArrayList<String> pathsEnemy = new ArrayList<>();
        pathsEnemy.add("src/res/world/level_start/enemies/0.png");
        ms.setSpritesPath(pathsEnemy);
        mob = new Mob(statsEnemy, ms);
        assertNotNull(mob);

        ArrayList<String> pathsNpc = new ArrayList<>();
        pathsNpc.add("src/res/npc/bob_down.png");
        ns = new NPCSprite(2, 2, "src/res/npc/bob.png");
        ns.setSpritesPath(pathsNpc);
        String dialog = "Ehi tu, per favore aiutami, non capisco dove siamo, chi
sono quelli?\n" +
                "E soprattutto cosa sono!?\n" +
                "Se mi proteggi posso darti le chiavi della struttura qui alle
nostre spalle.";
        String option1 = "Va bene";
        String option2 = "Devo già badare a me";
        npc = new Npc(ns, dialog, option1, option2);
        assertNotNull(npc);
    }

    @AfterEach
    void tearDown() {
        player = null; assertNull(player);
        mob = null; assertNull(mob);
        npc = null; assertNull(npc);
    }

    @Test
    void getHeroById() {
        EntityDAOImpl test = EntityDAOImpl.getDbInstance();

        assertEquals(player, test.getHeroById("001"));
    }

    @Test
    void getEnemyById() {
        EntityDAOImpl test = EntityDAOImpl.getDbInstance();

        assertEquals(mob, test.getEnemyById("001"));
    }

    @Test
    void getNpcById() {
        EntityDAOImpl test = EntityDAOImpl.getDbInstance();

        assertEquals(npc, test.getNpcById("001"));
    }
}
```

CAPITOLO 5:
CONCLUSIONE