Università degli studi di Napoli “Parthenope”

Progetto di Programmazione 3

A.A. 2023/2024

WOP

Elaborato da:

Ariano Luigi. Mat. 0124002483
Biondi Morgan. Mat. 0124002876
Tridente Antonio. Mat. 0124002692

Indice:

Capitolo 1: Introduzione a WOP

In questo capitolo ci occuperemo di analizzare la struttura generale dei WOP, di come funzionano i nodi e la struttura gerarchica.

Come obbiettivo si vuole implementare in Java un video game di tipo Platform che permette ad un player di compiere molteplici azioni come sconfiggere nemici, superare livelli e raccogliere oggetti.

Il gameplay è molto vasto e accattivante in quanto presenta diverse dinamiche di movimento e di attacco, del player e non. Altrettanto vasti sono la progettazione e lo sviluppo in codice di WOP in quanto possiede una intricata struttura ad albero. Esiste di fatto un nodo principale, la classe Game, che viene istanziata nel "Main" e che dà vita a tutta la gerarchia di sottoclassi/nodi.

Per "Nodo" non si intende propriamente quello che può essere proprietario di un game engine, ma è più una astrazione concettuale che intende: "*Classe che istanzia ed usa un'altra Classe*".

L'intera logica di programmazione inoltre vive su un cardine assoluto, la onnipresenza di due funzioni fondamentali: "update()" e "render() o draw()". Queste due funzioni sono presenti in ogni classe attrice, ovvero che ha bisogno di mostrarsi sullo schermo e rispondono alla seguente implementazione: una classe (o nodo) A possiede N classi figlie, l'update o render della classe A implementa la logica di update o render della classe A ed IN PIU' il richiamo all'update o render delle N classi figlie secondo la sua logica. Osserviamo come si comportano più generalmente.

1) **La funzione UPDATE** modifica e gestisce la logica di variabili e stati peculiari della classe che la implementa (nel caso del player, ad esempio, la funzione in questione implementa la logica di movimento, quella della gestione della barra della vita, dell'energia ecc.) richiamando inoltre l'update di tutti i nodi figli. Ecco un esempio di codice:

```
public void update() {
        ...
        if (paused) {
            pauseOverlay.update();
        } else if (levelCompleted) {
            levelCompletedOverlay.update();
        } else if (gameOver) {
            gameOverOverlay.update();
        } else if (playerDying) {
            player.update();
        ...
}
```

In questo caso ci troviamo nell'update della classe "Playing", vi è come dicevamo una logica peculiare del suo update ed il richiamo all'update di tutte le classi figlie ove necessario.

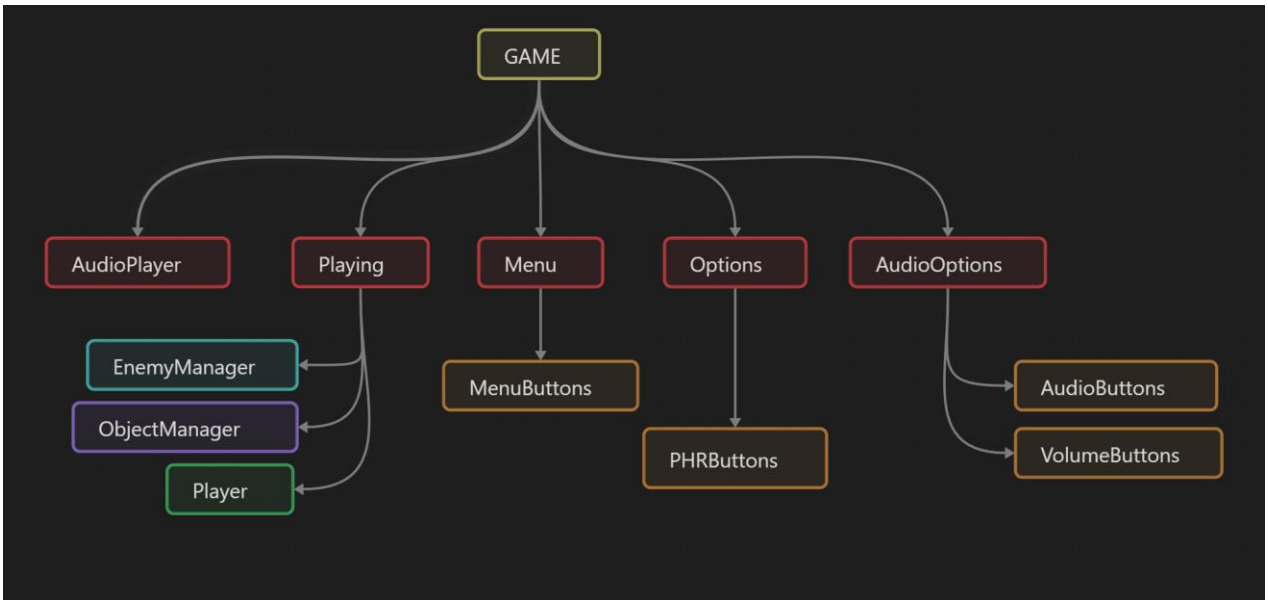
2)**La funzione RENDER** invece non gestisce la logica di stato delle classi (fatta eccezione per alcuni casi estremi) ma si basa su di essa per presentare nella scena, fare il render, disegnare a schermo, lo stato attuale di quel nodo in quel determinato momento. Esattamente come la funzione Update, la funzione Render richiama il render di tutti i nodi figli attori della scena in questione, ancora un esempio:

```
public void draw(Graphics g) {
        ...
        g.drawImage(backgroundImage, 0, 0, Game.GAME_WIDTH, Game.GAME_HEIGHT, null);
        drowLayer1(g);
        drowLayer2(g);
        levelManager.draw(g, xLevelOffset, yLevelOffset);
        objectManager.draw(g, xLevelOffset, yLevelOffset);
        enemyManager.draw(g, xLevelOffset, yLevelOffset);
        player.render(g, xLevelOffset, yLevelOffset);
        ...
}
```

Sempre preso dalla classe “Playing” La funzione render applica una sua logica di rendering della scena più il richiamo al rendering dei suoi nodi figli attori della scena.

In soldoni, la funzione Update modifica delle variabili in base a delle condizioni e la funzione Render disegna su schermo lo stato del nodo in base alle modifiche fatte dalla funzione Update.

Vediamo qui di sotto la struttura gerarchica (non del tutto completa) fino al secondo livello di profondità partendo dal nodo Game.



Andremo ora in osservazioni più approfondite delle singole classi o nodi presenti nel progetto, dividendo lo studio in macrocategorie:

- ***Classi o interfacce indipendenti***: sono quegli elementi java che nel progetto non estendono o implementano altre classi o interfacce, e che sono quindi stand alone, servono a generalizzare la progettazione e la programmazione del software permettendo di sfruttare al massimo la capacità polimorfa delle classi ed interfacce java offrendo così una base solida di partenza.

- ***Classi di primo livello***: sono quegli elementi del progetto che estendono o implementano una o più *classi/interfacce indipendenti* sopra descritte, che eseguono quindi ad un livello di astrazione inferiore e più specifico funzionalità e caratteristiche di classi e nodi che iniziano ad avere una forma più definita verso gli oggetti concreti presenti nelle scene.

- ***Classi di secondo livello***: sono quegli elementi del progetto che estendono o implementano una o più *classi di primo livello ed in aggiunta classi o interfaccie indipendenti.* Arrivati a questo tipo di solito si hanno gli oggetti concreti, i nodi effettivi dell'albero che possiedono i metodi Render ed Update implementati e funzionanti.

Finito lo studio delle categorie di classi passeremo all'illustrazione del complessissimo ***diagramma UML*** ed andremo in esso a ricercare e ad osservare i ***Design pattern*** implementati nel progetto.

Capitolo 2.1: Commento alle classi indipendenti

In questo capitolo andremo ad osservare quelle che nel codice sono le:

- ***Classi o interfacce indipendenti***: sono quegli elementi java che nel progetto non estendono o implementano altre classi o interfacce, e che sono quindi stand alone, servono a generalizzare la progettazione e la programmazione del software permettendo di sfruttare al massimo la capacità polimorfa delle classi ed interfacce java offrendo così una base solida di partenza.

Le classi nel progetto di questa tipologia sono:

1. Game
2. GameState
3. StateMethods
4. Entity
5. Level
6. AbstractObject
7. AbstractProjectile
8. StatusManger
9. AbstractButton
10. MenuOverlayInterface
11. GameWindow
12. EnemyManager
13. LevelManager
14. ObjectManager
15. Costants
16. HelpMethods
17. LoadSave

Game:

Questa classe è il nucleo di un gioco e implementa il design pattern State per gestire gli stati del gioco come Menu, Playing, GameOptions e Quit. Ecco una descrizione dei vari elementi nella classe:

Variabili di ambiente:

- "**gamePanel**": Riferimento a un oggetto di tipo "GamePanel".

- "**gameWindow**": Riferimento a un oggetto di tipo "GameWindow".

- "**gameThread**": Thread che esegue il loop principale del gioco.

- "**SET_FPS**": Costante che rappresenta il numero di fotogrammi al secondo desiderati.

- "**SET_UPS**": Costante che rappresenta il numero di aggiornamenti al secondo desiderati.

- "**frame**": Contatore dei fotogrammi per secondo.

- "**update**": Contatore degli aggiornamenti per secondo.

Variabili per la mappa:

- Costanti e variabili utilizzate per definire le dimensioni della mappa di gioco.

Variabili legate all'audio:

- "**audioOptions**": Oggetto di tipo "AudioOptions" per gestire le impostazioni audio.

- "**audioPlayer**": Oggetto di tipo "AudioPlayer" per gestire gli effetti sonori delle scene.

State design pattern:

- "**stateToUpdate**": Riferimento a un oggetto che implementa l'interfaccia "StateMethods".

- "**currentGameState**": Enumerazione che rappresenta lo stato corrente del gioco (MENU, PLAYING, OPTION, QUIT), sfruttata come catalizzatore per lo state design pattern.

Game States:

- Oggetti che implementano l'interfaccia "**StateMethods**" per rappresentare i vari stati del gioco (**Playing, Menu, GameOptions**).

Costruttore: Inizializza le classi necessarie e avvia il loop principale del gioco.

Metodo "**StartGameLoop()**": Avvia il thread del gioco.

Metodo "i**nitClasses()**": Inizializza le diverse classi necessarie per il gioco, come "AudioOptions", "AudioPlayer", "GameOptions", "Menu", "Playing", "GamePanel", e "GameWindow".

Metodo "**update()**": Chiama il metodo "update()" dello stato corrente per aggiornare lo stato degli elementi del gioco.

Metodo "**render(Graphics g)**": Chiama il metodo "draw()" dello stato corrente per disegnare gli elementi del gioco.

Metodo "**checkGameStateChanged()**": Controlla se lo stato del gioco è cambiato e imposta "stateToUpdate" in base al nuovo stato.

Metodo "**windowFocusLost()**": Gestisce la perdita del focus dalla finestra di gioco, resetta il movimento del giocatore se si è nello stato di gioco (PLAYING).

Metodo "**run()**": Implementa il loop principale del gioco gestendo gli FPS e gli UPS.

Metodo "**getFpsUps()**": Restituisce una stringa formattata contenente gli FPS e gli UPS, posizionaata successivamente nel gamePanel.

Metodi di accesso:

- Vari metodi getters e setters per ottenere riferimenti agli oggetti delle diverse classi utilizzate nel gioco.

Nel costruttore che non ha parametri vengono inizializzati tutti gli oggetti ivi descritti e viene fatto partire il GameLoop attraverso StartGameLoop che al suo interno inizializza il gameThread e lo fa partire.

A questo punto vengono eseguite le istruzioni presenti in run, implementazione dell'interfaccia runnable di java. In questo metodo vengono eseguiti dei calcoli per decidere quando eseguire l'update e quando eseguire il repaint. Ogni aggiornamento dal clock di memoria misura il tempo passato dall'ultimo aggiornamento fatto e se supera una certa quantità, che abbiamo definito precedentemente, viene eseguito l'Update e/o il Render dello stato attuale in cui si trova il gioco.

GameState:

Questa è una enumerazione semplicissima, accessibile in modo statico globalmente con la riga "*GameState.state*", che contiene 4 tipi di enumerazione, ognuna rappresentante uno stato del gioco: PLAYING, MENU, OPTIONS, QUIT. Lo stato iniziale viene impostaato a MENU, così da iniziare il gioco da dove dovrebbe, nel menù 😊.

StateMethods:

Questa interfaccia richiede l'implementazione di tutti i metodi base dei 3 stati principali del gioco, ovvero PLAYING, OPTIONS, MENU, fungerà da reference polimorfa per lo state pattern implementato nella classe Game. Troviamo ovviamente Update e Render, ma se ne presentano alcuni altrettanto importanti e che non abbiamo ancora menzionato.

- **"Update()"**:Permetterà alle implementazioni di aggiornare gli elementi del gioco nel tempo.
- **"Draw()"**: Permetterà alle implementazioni di disegnare quegli stessi elementi nella scena.
- "**mouseCliked(MouseEvent)":** gestisce gli eventi legati al click di rilascio del mouse.
- "**mousePressed(MouseEvent)":** gestisce gli eventi legati alla tenuta pressione del click del mouse.
- "**mouseDragged(MouseEvent)":** gestisce gli eventi legati al click trascinato del mouse.
- "**mouseMoved(MouseEvent)":** gestisce gli eventi legati al movimento generale del mouse.
- **"keyPressed(KeyEvent)":** gestisce gli eventi legati alla tenut pressione di un tasto dellaa tastiera.
- **"keyReleased(KeyEvent)":** gestisce gli eventi legati al rilascio di un tasto della tastiera.

State:

La classe astratta state permette principalmente di gestire il cambiamento degli stati del gioco attraverso il **GameState**, possiede inoltre il metodo "**buttonHovered()"** che restituisce true se il cursore del mouse è posizionato sopra un oggetto MenuButton e false nel caso opposto.

Possiede un costruttore che richiede un oggetto di tipo Game, gli passeremo al momento della creazione delle classi che la estendono il nodo principale "Game". Questa tecnica viene fatta al fine di permettere ad un nodo figlio di accedere agli elementi pubblici del nodo padre, otenendo così il potere di viaggiare all'interno dell'albero risalendo i singoli rami per poterne raggiungere i singoli nodi ai quali saarebbe impossibile accedere in altro modo. Questa scelta di programmazione viene ampiamente utilizzata nel codice per collegare i nodi più disparati nel momento del bisogno, eccone quì un esempio preso dalla classe "**Level**" che non abbaimo ancora visto:

```
...

game.getPlaying().getMementoManager().addPlayerMemento(new PlayerMemento(game.getPlaying().getPlayer()));

...

game.getPlaying().getPlayer().setSpawnPoint(getCurrentLevel().getPlayerSpawnPoint());

...
```

Il cambiamento dello stato del gioco avviene attraverso il metodo "**setGameState(GameState)**" che prende come parametro un oggetto di tipo Gamestate e setta la variaabile statica globale GameState.state a quella in intresso.

Entity:

La classe entity è come si può intuire una classe importantissima. Essa non rappresenta un nodo come già annunciato nell'introoduzione ma l'astrazione assoluta di cosa una entità nel mondo di WOP sia.

E' la classe madre di tutto ciò che ha caratteristiche come la vita, il danno, una velocità di movimento, una hitbox, una attackbox eccetera, verrà quindi estesa dai nemici, dai boss, dal player stesso. Essa possiede:

Caratteristiche principali della classe:

- **"x" e "y**": Coordinate della posizioni di partenza dell'entità.
- "**hitBoxWidth**" e "**hitBoxHeight**": Dimensioni della hitbox.
- "**aniTick**" e "**aniIndex**": Variabili utilizzate per la gestione dell'animazione.
- "**maxHealth**" e "**currentHealth**": Variabili per la vita dell'entità.
- "**walkSpeed**": Velocità di movimento dell'entità.
- "**airSpeed**", "**inAir**": Variabili per il movimento in aria e simulazione gravità.
- "**invulnerability**": Flag che indica se l'entità è invulnerabile.
- "**state**": Variabile per lo stato dell'entità.
- "**hitbox**": Hitbox di collisione di tipo rettangolare attorno all'entità.
- "**attackBox**" e "**circularAttackbox**": Hitbox per gli attacchi dell'entità.
- "**statusManager**": Gestore degli stati per applicare effetti di buff e debuff.

Metodi principali:

- **Costruttore**: Inizializza le coordinate, le dimensioni della hitbox e il gestore degli stati.
- "**initHitbox(float x, float y, float width, float height)**": Inizializza la hitbox con le dimensioni fornite.
- "**drawHitbox(Graphics g, int xLevelOffset, int yLevelOffset)**": Disegna la hitbox rettangolare intorno all'entità.
- "**drowAttackBox(Graphics g, int levelOffsetX, int yLevelOffset)**": Disegna la hitbox per gli attacchi.
- "**drowCircularAttackBox(Graphics g, int levelOffsetX, int yLevelOffset)**": Disegna la hitbox circolare per gli attacchi.
- Metodi Getters e Setter per ottenere informazioni sullo stato della entità.

Una fondamentale caratteristica è che tutte le entità possiedono una "hitbox " ed una "attackbox", aree geometriche posizionate nel mondo di gioco, *cuore cardine della logica di combattimento e di movimento* analizzata in seguito nelle classi più complicate come i "**Manager**" o lo stesso player.

OSSERVAZIONI IMPORTANTI: Prima di proseguire si sente il bisogno di definire alcune dinamiche.

Come per la classe Game, anche le entità hanno gli stati, che rappresentano le righe di una matrice, ma andiamo con calma.

Logica Di disegno di una entità:

La funzione Render che abbiamo già ampiamente descritto, ha una logica di disegno delle entità che si basa su dei cicli for innestati ed eseguiti su una matrice.

Ogni entità possiede un set di animazioni collezionate in dei ".png" del folder "**/res**" per categorie, nelle righe dei png sono presenti set di animazioni coerenti e consecutivi, ad esempio nella riga 1 sono disegnate tutte le immagini necessarie ad una animazione della corsa, i frame o sprite della animazione sono quindi ordinati per colonne.

La funzione Update *delle entità* possiede implementazioni peculiari per ogni nodo come abbiamo sempre detto, ma ha sempre una costante, la funzione "**updateAnimationTick()**", questa ad ogni update, che ricordiamo viene gestito dal nodo padre "Game", viene eseguita ed aumenta un contatore, la variabile "**aniTick**" (*Animation Tick*). Un controllo *if* viene eseguito su di essa ad ogni iterazione, e soltanto quando questa ha raggiunto o superato un limite definito dalla variabile "**aniSpeed**" (*Animation Speed*), viene riportata a 0 e viene incrementato il valore di "**aniIndex**" (*Animation index*) di 1. Il discorso non finisce, vi è una classe non ancora descritta, la classe "**Costants**" *la quale contiene* tutti i valori coostanti del gioco, come nel nostro caso, *il numero di frame proprietari di un determinato stato di una determinata entità*. Possiamo quindi sfruttarla per capire quando "aniIndex" deve smettere di incrementare e reimpostarsi a 0, assegnazione che non fa altro che far ripartire il loop di animazioni della matrice di cui sopra.

La funzione Update inoltre gestisce una altra variabile che vediamo qui dichiarata, "**state**" che non ha nulla a che vedere con in *GameState* descritto precedentemente. *In base a condizioni peculiari e a logiche di gameplay* che vedremo più avanti in questa documentazione, *questa variabile viene settata uguale ad una costante della classe "Costants"* tra un determinato set di costanti che rappresentano tutti gli stati possibili si una determinata entità.

Dati questi 3 presupposti (*Animazioni conservate in una matrice, variabile AnimationIndex, variabile Stat*e) possiamo andare ad osservare uno stralcio di codice esempio per capire immediatamente come la funzioine Render decide quale animazione e quale frame della corrispettiva animazione disegnare:

```
public void render(Graphics g){

...

g.drawImage(animations[state][aniIndex], Posizione_x, Posizione_y, Larghezza, Altezza);

...

}
```

Quello che osserviamo è *che la funzione "**drawImage()**" disegna a schermo in una posizione (x, y) e con una certa (altezza, larghezza) l'immagine presa dalla matrice di riga "**state**" e di colonna "**aniIndex**".* Definiti i modi di cambiare di queste due variabili, risulta adesso semplice ed intuitvo capire come una animazione viene presentata a schermo.

Detto ciò, ci siamo liberati di una bella gatta da pelare e possiamo continuare con la descrizione dei Livelli di gioco.

Level:

La classe “Level” è una classe che conserva tutte le caratteristiche fondamentali di un livello.

Variabili di Ambiente:

- "**enemyFactory**": Istanza della classe **EnemyFactory** utilizzata per la creazione di nemici.
- "**image**": Un'immagine che conserva i dati del livello.
- "**levelData**": Una matrice contenente i dati del livello ottenuti dall'immagine.
- "**enemyList**": Un'ArrayList di nemici presenti nel livello.
- "**potions**": Un'ArrayList di pozioni presenti nel livello.
- "**lootBoxes**": Un'ArrayList di loot box presenti nel livello.
- "**spikes**": Un'ArrayList di spine trappola presenti nel livello.
- "**cannons**": Un'ArrayList di cannoni presenti nel livello.
- "**levelTileWide**" e "**levelTileHeight**": Larghezza e altezza del livello in termini di tile.
- "**maxTileOffset**" e "**maxTileOffsetY**": Limiti oltre i quali la telecamera non deve più spostarsi orizzontalmente e verticalmente.
- "**maxLevelOffsetX**" e "**maxLevelOffsetY**": Limiti convertiti in pixel per il movimento della telecamera.
- "**playerSpawnPoint**": Punto di spawn del giocatore nel livello.

Metodi principali:

- "**calculateCannons()**": Calcola la posizione dei cannoni nel livello.
- "**createSpikes()**": Crea le spine trappola nel livello.
- "**createLootBoxes()**": Crea le loot box nel livello.
- "**createPotions()**": Crea le pozioni nel livello.
- "**calculatePlayerSpawnPoint()**": Calcola il punto di spawn del giocatore nel livello.
- "**createLevelData()**": Ottiene i dati del livello dall'immagine.
- "**createEnemies(BufferedImage img)**": Crea nemici utilizzando la factory in base ai dati dell'immagine.
- "**calculateLevelOffsets()**": Calcola i limiti di spostamento della telecamera in base alle dimensioni del livello.
- "**getSpriteIndex(int x, int y)**": Restituisce il valore RGB riconosciuto durante l'estrazione delle informazioni dalle immagini levelData nella posizione specificata.
- Alcuni metodi getter per ottenere informazioni specifiche sul livello, come i dati del livello, la lista dei nemici, delle pozioni, delle loot box, delle spine trappola, dei cannoni e il punto di spawn del giocatore.

Il costruttore di questa classe richiede una immagine in ingresso, immagine che la classe sfrutta per ottenere il leveldata, ma che cos’è questo level data?

LevelData, ragionare con i colori...

Come fa il player a muoversi sul terreno? Come fa a farlo un nemico? Come capiamo se ci stiamo scontrando con un muro oppure stiamo camminando nel vuoto? Per tutte queste domande esiste una sola risposta, "*esiste il LevelData*".

Il LevelData non è altro che una matrice di interi, calcolata sulla base dei colori di un disegno fatto su un qualunque editor di immagini. La classe Level fa uso di una funzione "**createLevelData()**" che richiama il metodo di una classe (la "**HelpMethods**") che calcola la matrice della *componente RGB rossa* in base ad una immagine data in input, memorizzando questi valori come valori di output, diamo un occhi al codice:

```
public static int[][] getLevelData(BufferedImage img){
    int[][] levelData = new int [img.getHeight()][img.getWidth()];
    for( int j = 0; j<img.getHeight(); j++){
        for (int i = 0; i < img.getWidth(); i++) {
            Color color = new Color(img.getRGB(i, j));
            int value = color.getRed();
            if(value >= 48)
                value = 11;
            levelData[j][i] = value;
        }
    }
    return levelData;
}
```

Si istanzia una nuova matrice di righe e colonne, rispettivamente l'altezza dell'immagine mandata e la larghezza sempre della stessa, si scorre con dei cicli for innestati la struttura così creata e per ogni iterazione, si raccoglie la componente RGB rossa, solo *se il valore è tra 0 e 48 questo viene impostato ad 11, in tutti gli altri casi il valore compreso tra 0 e 48 viene memorizzato direttamente*.

La scelta non è casuale, la mappa di gioco è di fatto costruita sulla base di sprite, blocchetti di terreno che si trovano nel folder "/res/envitoment/terrain.png", questi hanno larghezza ed altezza di 32 pixel (Costante definita nella classe Game) e sono collocati nel .png in 4 righe ognuna avente 12 colonne, il risultato? 4*12=48, esattamente il filtro di componente rossa registrato nella funzione, nella prima riga, l'elemento di colonna 11, ovvero il dodiciesimo sprite, è una parte di immmagine senza colori. Possiamo ora iniziare a collegare i puntini.

Esiste una matrice che conserva per ogni posizione corrispondente ad un pixel di una immagine, la sua componente RGB rossa, e ne imposta il valore ad 11 se questa componente supera la quantità di 48. Possiamo utilizzare durante la Render del livello una logica per la quale, visitata per intero la matrice, per ogni cella, si disegni il blocchetto di terreno presente in un array di blocchetti lungo 48 elementi, quello corrispondente al valore della componente Rossa registrata nella matrice. Ed ecco che magicamente vedremo comparire nel game panel, i blocchi di terreno in base al tipo di immagine che stiamo passando alla funzione, l'aria è l'undicesimo elemento di questo array, pertanto, qualsiasi volta ci sia 11, non verrà disegnato nulla in quanto l'undicesimo elemento dell'array è un png che non ha pixel colorati.

La stessa logica la possiamo usare per definire se un terreno è solido oppure no. Dando in input ad una funzione la posizione di una entità rispetto ai suoi piedi e dividendola per la grandezza di un blocco (Costante della classe Game) otterremo un numero che indicherà la **posizione rispetto alla matrice del LevelData della entità in questione** 😀, confrontando quella numero (che sono un set di coordinate scalate per la matrice del LevelData fondamentalmente) con il valore della matrice **LevelData** rispetto alla direzone del movimento della entità, possiamo decidere se farla muovere oppure no in base al fatto che il LevelData in quella direzione contenga o no il valore 11, ovvero l'aria. Se non lo coontiene, smettiamo di spostare l'entità in quella direzione, altrimenti se vi è necessità, le permetteremo il movimento. Ed ecco sciolto il mistero 😊.

La logica applicata dal level data, viene applicata in modo perfettamente speculare anche durante la creazione di nemici, player ed entità varie come le casse o i cannoni, un nuovo nemico viene generato in uno specifico set di coordinate in base alla componente RGB Blu della immagine in ingresso, dove viene trovato che il blu corrisponde al valore 0, viene posizionato un nemico di tipo "**NightBorne**", dove invece il valore rosso è uguale a 256 viene creato il punto di spawn del Player e così via.

Quasi tutte le funzioni della classe "Level" ragionano in questo modo per ottenere gli array dei nemici, delle lootbox, informaizoni sul player e della conformazione del terreno.

Abstract Object:

La classe madre di tutti gli oggetti interattivi del gioco, come casse pozioni e spuntoni.

Osserviamone le principali caratteristiche:

- "x", "y": Coordinate dell'oggetto nel gioco.
- "**objType**": Tipo di oggetto, identificato da costanti come "BARREL", "BOX", "CANNON_LEFT", ecc.
- "**hitbox**": Hitbox dell'oggetto, rappresentata come un rettangolo.
- "**doAnimation**": Flag che indica se l'oggetto deve eseguire un'animazione.
- "**active**": Flag che indica se l'oggetto è attivo.
- "**aniTick**", "**aniIndex**": Variabili utilizzate per la gestione dell'animazione.
- "**xDrawOffset**", "**yDrawOffset**": Offset utilizzati per il disegno dell'oggetto.

Metodi principali:

- Costruttore "A**bstractObject(int x, int y, int objType)**": che richiede una posizione di partenza ed un intero che descrive il tipo di oggetto.
- "**initHitbox(float width, float height)**": Inizializza la hitbox con le dimensioni fornite.
- "**drawHitbox(Graphics g, int xLevelOffset, int yLevelOffset)**": Disegna la hitbox dell'oggetto.
- "**updateAnimationTick()**": Aggiorna l'indice di animazione e gestisce l'animazione dell'oggetto.
- "**reset()**": Reimposta lo stato dell'oggetto, l'indice di animazione e i tick di animazione.
- Alcuni metodi getter e setter per ottenere o impostare informazioni sull'oggetto, come il tipo, la hitbox, l'attivazione, gli offset, ecc.

Questa classe ci permette di gestire in modo uniforme tutte le caratteristiche basi di un oggetto interattivo del gioco condividendo risorse comuni, le più importanti delle quali come "anindex" e compagne abbiamo già analizzato della sezione dedicata alle entità, funzionano allo stesso modo.

Le uniche cose importanti da tenere a mente sono la variabile"**objType**" (*Object Type*) e la variabile booleana "**active**".

La prima definisce quale oggetto stiamo andando ad istanziare al momento del richiamo al costruttore e tra i vari metodi getters spicca nell'utilizzo quello riguardante questa variabile, nella logica dei "Manager", infatti, sotto determinate condizioni per applicare eventi di gioco, vi troviamo il richiamo a questo valore, se ad esempio è uguale a quello di una pozione rossa, viene aggiornata la vita del player, se blu, la sua energia, se il valore è quello di una cassa questa rimane statica nel gioco fin quando il player non la colpisce.

La seconda invece definisce quando un oggetto è attivo, essere attivo per un oggetto, significa essere capace di collidere con le hitbox ed attackBox del player nello specifico, e subirne di conseguenza gli effetti, nel caso delle pozioni per esempio, al momento della rottura di una cassa questa viene spawnata nel mondo di gioco, viene pertanto inizializzata con lo stato di "attiva" e soltanto quando il player collide con la propria hitbox con quella della pozione, verrà applicato l'effetto desiderato (come il ripristino della vita) e verrà impostato lo stato a "non attivo". Le personali funzioni di Update, smetteranno di aggiornare quell'oggetto in base a questa condizione causando la scomparsa dal mondo di gioco dell'oggetto

Presente anche qui la funzione "**updateAnimationTick()**" che segue una sua logica per aggiornare le animazioni, se l'object type è quella di un barile o di una cassa, l'animazione non verrà fatta partire fin quando il player non colpisce l'oggetto in questione, nel caso delle pozioni verrà aggiornato sempre per dare l'effetto di movimento fin da subito.

AbstractButton:

Attributi:

- "x", "y": Le coordinate del pulsante nell'area di gioco.
- "**width**", "**height**": Larghezza e altezza del pulsante.
- "**mouseOver**", "**mousePressed**": Flag che indica se il mouse è sopra il pulsante o se il pulsante è stato-premuto.
- "**rowIndex**", "**columnIndex**": Indici utilizzati per gestire l'aspetto visivo del pulsante.
- "**hitbox**": Un oggetto rettangolare che rappresenta l'area sensibile del pulsante.

Metodi:

- "**update**()": Aggiorna gli indici in base agli stati del mouse (sopra, premuto).
- "**resetBools**()": Resetta i flag "mouseOver" e "mousePressed".
- "**createHitbox**()": Inizializza l'oggetto rettangolare hitbox con le dimensioni del pulsante.

Getters e Setters:

- Diversi metodi per accedere e modificare gli attributi della classe, come "getX()", "setY()", ecc.

Il costruttore prende in input la posizione X ed Y del pulsante e le dimensioni, le assegna poi alle sue variabili interne eseguendo il metodo createHitbox() che posiziona l'area di riconoscimento del mouse.

Le variabili booleane mouseOver e **mousePressed servono alla logica di update per capire se un bottone è stato cliccato o se semplicemente il mouse ci è passato sopra**. Viene settata la "aniIndex" in base a queste variabili booleane e la logica di Render che abbiamo già affrontato in paragrafi precedenti. In base alle variabili booleane mouseOver e mousePressed viene aggiornata la posizione dell'indice colonna a 0 se il mouse non si trova sul bottone, a 1 se si trova sopra al bottone e a 2 se l ha cliccato.

I bottoni sono presente ampiamente in tutti gli stati di gioco e li vedremo comparire in tutti i menù di UI.

AbstractProjectile:

E’ una classe semplicissima che permette di istanziare tutti i tipi di proietili, accorpa dentro di sè le caratteristiche più generali e comuni dei proiettili. Tra gli attributi troviamo infatti:

- “projType” o “Projectile Type” che definisce la tipologia del proiettile.
- “hitbox” che è un Rettangolo rappresentante il corpo di collisione.
- “direction”, un intero che varia tra 1 e –1 usato per fare l’update della posizione, è un moltiplicatore che modifica il segno della prossima variabile.
- “active”, una variabile booleana che ne indica lo stato, attivo o non attivo.
- “canDoDamage”, una variabile che indica se un proiettile può applicare danno ad una entità.
- “projectileSpeed” indica quanto veloce si muove il proiettile

Nel **costruttore** vengono richieste le posizioni di partenza (*x* ed *y*), la direzione che può essere solo 1 o –1, ed un tipo di proiettile che serve a definire le interazioni possibile con l’ambiente, una sorta di bitmask di collisione. Vengono anche inizializzate la velocità in base al tipo di proiettile scelto ed una hitbox.

Fatta eccezione per getter e setters troviamo un solo metodo che è importante da descrivere, il metodo “**updatePosition()**”, super semplice, super diretto, modifica la posizione in x del proiettile aggiungendo alla sua attuale posizione la “*projectileSpeed*” di cui prima moltiplicata per la direzione.

StatusManager:

Questa classe è tanto semplice quanto peculiare, non possiede un costruttore ma soltanto metodi pubblici legati strettamente alla classe “Entità” discussa prima.

Diverse funzioni sono presenti al suo interno che implementano una logica di base uguale per tutte.

Si vuole creare una funzione che passati come parametro una entità ed una durata, applichi degli effetti di buff e debuff alla entità per il tempo indicato. Prendiamo ad esempio la funzione di “slow”:

```
public void applySlow(Entity entity, int duration, float slowValue){
      float startingWalkSpeed = entity.getWalkSpeed();
     Thread slowThread = new Thread(() -> {
       etity.setWalkSpeed(slowValue);
       try {
                Thread.sleep(duration * 1000);
                } catch (Exception e) {
                System.out.println("Qualcosa è andato storto nella funzione di slow");
                      e.printStackTrace();
                 }
           entity.setWalkSpeed(startingWalkSpeed);
       });

       slowThread.start();

   }
```

Della entità in questione viene presa tramite le funzioni get, un valore specifico come in questo caso la velocità di movimento, e viene conservata in una variabile temporanea, dopodiché vinee creato un nuovo Thread separato da quello padre, che imposta tramite il metodo set, la velocità di movimento desiderata presa in input per poi iniziare uno “sleep” personale, un timer praticamente, prima di proseguire e resettare la componente dell’entità a quella di partenza. Ciò causerà alle funzioni Update delle entità di effettuare aggiornamenti differenti su quella specifica variabile, ed in questo caso facendo muovere più lentamente l’entità scelta fin quando il Thread in questione non riprenda la sua esecuzione dopo la funzione “sleep”. Vi sono diverse funzioni presenti in questa classe e tutte applicano questa stessa logica di base.

MenuOverlayInterface:

Interfaccia estensiva per tutti i tipi di menù di gioco, possiede i soliti due metodi Update e Draw più altri metodi comuni al riconoscimento dei movimenti del mouse, il “mousePressed” “mouseMoved “mouseReleased” “mouseHovering” “mouseDragged” “mouseReleased”. Va implementata ed offre una buona approssimazione di tutte le funzionalità di base dei menù di gioco.

GameWindow:

La game window è un’altra classe semplicissima che utilizza la classe nativa “**JFrame**” per il contenimento del pannello di gioco, il suo costruttore di fatto richiede un oggetto di tipo “**GamePanel**” che verrà aggiunto poi col metodo “**add**” al frame inizializzato. Vi sono inoltre diverse funzioni standard per settare le funzionalità di base come la scalabilità, la richiesta di focus, la chiusura del programma al chiudersi della finestra e così via.

Game Managers:

Tra tutte le classi, spiccano di prepotenza le 3 classi così definite come “Managers”.

I Managers sono fondamentali al gioco in quanto gestiscono la logica degli eventi non legati all’utente, come quella degli oggetti, dei nemici e dei livelli, compiti dai quali prendono i nomi, esse sono di fatto:

- L’ “**EnemyManager**”: delegato alla gestione di tutti i nemici del livello corrente, delle loro collisioni, dei loro update, dei loro render, delle loro morti, il caricamento delle texture e così via.
- Il “**LevelManager**”: classe delegata alla gestione dei “Level” (Classe analizzata precedentemente), alla creazione di questi sia a livello di Rendering che di caricamento in memoria degli sprite del terreno. Una delle funzionalità più importanti è sicuramente quella della creazione dei “**savingPoints**” i punti di salvataggio, gestiti dal “**Memento design Pattern**” per riportare un livello al suo stato di partenza se mai lo si volesse resettare, per gioia di giocarlo ... o per rabbia ;D.
- L’ “**ObjectManager**”: delegato alla gestione di Animazioni, creazione ed esistenza degli oggetti di gioco, Pozioni e casse per intenderci.

Tutti e tre i manager vivono dei capisaldi di render e di update finora descritti e ribaditi, con le loro funzioni di Render ed Update. Li analizzeremo uno ad uno per poi addentrarci nel prossimo capitolo.

Game Managers: EnemyManager

L'**Enemy Manager** come abbiamo detto gestisce la logica ed il rendering di tutti i nemici di un determinato livello, il suo costruttore richiede l'ingresso del nodo padre "Playing", per Avere un collegamento ad esso e quindi un collegamento anche agli altri suoi nodi figli (Come gli altri manager).

Come variabili di ambiente possiede:

- L'oggetto rappresentante il nodo "Playing"
- Delle Matrici di tipo "**BufferedImage**" che conterranno tutti gli sprite di tutte le animazioni di ogni nemico possibile nel gioco
- Un "**ArrayList**" classe nativa di Java per implementare le liste di tipo "**AbstractEnemy**" (classe che non abbiamo ancora studiato) che contiene tutti i nemici di un determinato livello.
- Istanze della interfaccia "**RenderingInterface**" che analizzeremo nel Capitolo dedicato ai design pattern, esse corrispondono ai Manager per il rendering dei singoli nemici. Una istanza renderizza un tipo di nemico,
- Un array di "**RenderingRequest**" che analizzeremo sempre nel capitolo legato ai Design Pattern e che impacchetta le risorse necessarie al rendering di un entità.

Una volta istanziato l'oggetto EnemyManager fa 3 cose:

1. Col metodo "**LoadEnemyImages()**" carica nelle matrici corrispondenti gli sprite di tutti i nemici possibili nel gioco.
2. "**InitrenderRequests()**" impacchetta le risorse di rendering nell'array di RenderingRequests da mandare poi ai renderers.
3. "**initRenderers()"** crea il Design Pattern della "**Chain of Responsability**" collegando tra loro le RenderingInterface di cui prima, per impostare e rendere funzionante la "Rendering chain" dei nemici.

In aggiunta l'**Enemy Manager** possiede altri metodi utili a gestire la logica dei nemici.

- "**addEnemyes(Level)**" fa un richiamo alla classe "Level" in ingresso richiamandone il metodo "**getEnemyes()**" associandone il valore all'array di nemici ivi descritto. Viene così inizializzato l'array contenenti i nemici di un determinato livello.
- "**Update()**" viene richiamata dal nodo padre seguendo la logica di programmazione dell'intero progetto, esegue un ciclo for sull'array di nemici appena creato, e se questi sono vivi (viene fatto un controllo sulla variabile "active") ne viene eseguito l'update. Ogni volta che un nemico viene aggiornato, una variabile booleana istanziata prima dle ciclo for, viene settata a true. Basta quindi che almeno un nemico sia vivo, per rendere questa flag vera. Quando tutti i nemici sono stati uccisi, la flag rimane impostata su falso e viene di conseguenza segnalato al nodo padre "Playing" che il livello è stato completato, in modo da gestire l'evento e proseguire con il prossimo livello.
- "**Draw()**" è la nostra funzione Render, che semplicemente delega alla "Renderingchain" la respoondabilità di renderizzare i nemici a schermo.
- "**checkPlayerHitEnemy(AttackBox, areaAttack)**" è una funzione che quando richiamata, presa in input la "attackBox" del Player, ed una flag che indica se il player sta facendo un attacco ad area oppure no, esegue un ciclo for sull'array di nemici, ***controllando che vi sia intersezione tra la attackbox mandata in ingresso e la hitbox del nemico scelto in quel momento,*** **se ciò è vero, se il nemico è attivo, se il nemico ha ancora punti salute e se questo non è al momento invulnerabile**, viene applicato il danno al nemico in questione e tramite le funzioni, appartenenti sempre a questo nodo, "playSFX()" "playHurtEffect()" "playDeathEffect()" viene eseguito un suono in base alle condizioni di vita e del tipo del nemico rappresentative di ciò chegli sta accadendo, se è stato colpito ma è ancora in vita viene eseguito un suono di "Hit" altrimenti viene eseguito il "DeathSound" del nemico in questione.
- "**resetAllEnemyes()**" è la funzione quando invocata analizza tutti i nemici e tramite il punto di salvataggio creato nella classe "Level" implementato tramite il "**Memento design patter**" è in grado di ripristinare lo stato iniziale del nemico su quello di partenza del livello.

Game Managers: LevelManager

Il **LevelManager** abbiamo detto che è il delegato alla gestione dei livelli, osserviamone la struttura.

Come variabil di ambiente troviamo:

- Il riferimento al nodo padre "**Game**".
- Un Array di **BufferedImage** che dovrà contenere le immagini di tutti i livelli
- Un **ArrayList** atta al contenimento di tutti i livelli.
- Un intero a rappresentare il livello corrente, il "**LevelIndex**"
- Ed una variabile booleana per controllare se il gioco sia appena iniziato.

Il nodo di riferimento è come descritto nelle variabili la stessa classe "Game" che deve gestire in base ai segnali inviati dalla classe "Playing" i livelli, ha quindi necessità di accedere al "LevelManager". **Nel costruttore** vi troviamo pertanto il bisogno di un oggetto "Game" in ingresso, successivamente il richiamo a due funzioni:

- "importSprites()" che carica all'interno di un array di 48 elementi, i 48 sprite differenti che compongo il terreno di gioco, questo array verrà sfruttato dalla Render dello stesso nodo seguendo *la Logica del LevelData* di cui abbiamo già discusso precedentemente, per disegnare il livello a schermo.
- "buildAllLevels()" invece, come di facile intuizione dal nome, costruisce tutti i livelli del gioco. Utilizza una funzione della classe "LoadSave" di cui ancora non abbiamo discusso per ottenere un array di immagini rappresentanti i livelli (Queste immagini vengono importate nel programma dalla cartella "/res/lvls"). Una volta ottenuto l'array, tramite un ciclo for vengono aggiunti in una "ArrayLyst " i livelli dall' 1 all'ennesimo livello presenti nella cartella:

```
private void buildAllLevels() {
        levels = new ArrayList<>();
        BufferedImage[] alllevels = LoadSave.getAllLevels();
        for (BufferedImage img : alllevels) {
            levels.add(new Level(img));
        }
}
```

La classe del LevelManager possiede inoltre altri 3 metodi molto importanti:

- Il Metodo "**Update()**" classico, che ha una importante differenza rispetto a tutti gli altri, non viene eseguito ad ogni iterazione dell'update del nodo "Game", ma soltanto una volta quando un livello inizia, questo perchè ha bisogno di utilizzare una sola volta il prossimo metodo.
- "**createSavingPoint()**": come da nome, questo blocco di codice serve a creare un punto di salvataggio o di ripristino per la partita corrente, salva di fatto tramire il "Memento Design Pattern" implementato, lo stato di tutte le entità presenti nel livello al momento in cui il giocatore ci arriva. Quì vi è un peculiare utilizzo della logica ad albero utilizzata, per risalire e scendere nell'albero al fine di raggiungere nodi molto distanti da quello corrente.
- "**loadNextLevel()**", questo metodo quando invocato carica il prossimo livello dalla ArrayList nominata in precedenza in una variabile di appoggio e risalendo l'albero di programmaizione, porta tanti altri nodi, compresi gli altri manager, a caricare le nuove informazioni presenti nel nuovo livello. Conclusa l'operazione il metodo "**createSavingPoint()**" viene invocato andando a salvare lo stato corrente del nuovo livello
- Getters e setters per muoversi tra i nodi ed ottenere informazioni.

Game Managers: ObjectManager

La classe ObjectManager abbiamo detto che si occupa della logica di Render e di Update riguardante gli oggetti interattivi del gioco in un determinato livello.

Parliamo delle variabili di Ambiente:

- Riferimento al nodo padre "Playing"
- Diverse BufferedImage (Forma matriciale e array) che conservano, le immagini di: Pozioni, Casse, Cannoni
- BufferedImage singole contenenti gli sprite statici di palle di cannone e spine.
- ArrayList per contenere gli oggetti istanziati effettifi. Queste conservano: le Pozioni, le Casse, i Cannoni, le Spine, le Palle di Cannone.
- L'istanza di una "**CloningFactory**" implementazione del prorotype design pattern.

Il **costruttore** semplicissimo crea un riferimento al nodo padre "**Playing**" ed invoca il metodo "**loadImages()**" che carica le immagini nelle matrici ed array di tipo BufferedImage prima menzionati.

I Metodi presenti sono:

- "**loadObjects(Level)**": quando richiamata associa ad ogni ArrayList prima descritto l'output una specifica funzione "getElement" dal livello ingresso, esempio:

  ```
  potions = new ArrayList<>(level.getPotions());
  ```

- "**Update()**" richiamata dal nodo padre, esegue dei cicli for sugli array di oggetti richiamando oggetto per oggetto il suo metodo update secondo lo schema ad albero.
- "**Draw()**" richiamata dal nodo padre, esegue il rendering degli oggetti usufruendo di 5 metodi proprietari della classe per applicare logiche di draw ad ogni tipo di oggetto, questi metodi sono "**drawPotions()**", "**drawCannons()**", "**drawTraps()**", "**drawCannonBall()**", "**drawBoxes()**". Tutte quante le draw usufruiscono della variabile "isActive" ereditata dalle **classi Madri "AbstractObject ed AbstractProjectile"**, per disegnare o no in base a se questa variabile è settata su *true* oppure *false* + altre condizioni peculiari e che descriveremo in seguito, dei singoli oggetti.
- "**ceckSpikeTouched(Player)**": quando invocata controlla che il player stia toccando le spine, se così fosse (le spine sono posizionate in fossati o punti strategici molto angusti) uccidono istantaneamente il player.
- "**checkPlayerTouchedPotions(hitbox)**": quando invocata viene controllato che il player stia intersecando la hitbox di qualceh pozione, se coosì fosse viene invocato il prossimo metodo
- "**applyEffectToPlayer(Potion)**": fatto un controllo sul tipo di pozione viene applicato al player un effetto diverso (La pozione rossa ripristina vita, la blu energia).
- "**checkObjectHit(attackBox)**": quando invocata controlla che la attackbox del player stia intersecando una cassa eseguendo un ciclo for, in tal caso modifica lo stato di animazione della cassa a true, per farle iniziare l'agiornamento della animazione (vedi AbstractObject), facendo poi spawnare una Pozione al suo posto.
- "**updateCannons(Player)**": gestisce l'update dei cannoni in base a determinati fattori. Un cannone deve restare in attesa di diverse cose prima di poter essere updatato, ha bisogno che: la posizione in y del player sia la stessa del cannone, che il player sia nella direzione di sparo del cannone (se il cannone punta a destra il player deve stare a destra del cannone e così il contrario), che non vi siano muri tra il cannone ed il player e che quest'ultimo sia in range di attacco.
- "**shootCannon(Cannon)**": che viene invocata quando viene eseguito l'update dei cannoni e questi si trovano in un determinato frame di animazione, tramite il "**Prototype Design Pattern**" vengono generati nuove palle di cannone sulla base del cannone che le sta sparando.
- "**updateProjectiles()**": esegue dei cicli for su tutti i proiettili della scena e se questi sono attivi, vengono agiornati richiamando gli update secondo lo schema ad albero, e secondo la logica descritta nella classe "**AbstractProjectile**", se il proiettile colpisce il Player gli viene aplicato il danno per poi venire desattivato, se il proiettile colpisce un muro viene disattivato.
- "**resetAllObject()**": che resetta lo stato inziale degli oggeti della scena.

Constants:

La classe "**Constants**" è una classe molto lunga e strabordante di valori costanti, ma di una semplicità unica. Definiamone la struttura

Dentro di essa vi sono classi statiche rappresentanti macrocategorie di oggetti. Esse sono: **ObjectCostants**, **EnemyCostants**, **Projectiles**, **Enviroment**, **UI**, **Direction** e **PlayerCostants**.

- ## ObjectConstants:

Contiene costanti riguardanti gli oggetti di gioco, Pozioni, Cannoni, Spine e Lootboxes. Queste costanti descrivono, le grandezze degli sprite in ambo le dimensioni, soprattutto definisce delle costanti intere alle quali vengono assegnate dei numeri, queste costanti molto espressive, eccone un esempio:

```
public static final int RED_POTION = 0;
public static final int BLUE_POTION = 1;
public static final int BARREL = 2;
```

Possono essere usate in tutte le parti del codice quando si trova bisogno di fare riverimento ad un determinato tipo di oggetto. Quando istanziamo un abstractObject ad esempio, possiamo inviare come argomento in ingresso al costruttore la variabile *BARREL* per capire perfettamente cosa vogliamo istanziare.

Possiede inoltre un metodo "**getSpriteAmount(objectType)**" che preso in input un tipo di oggetto restituisce il numero di frame della sua animazione. **Questa funzione è di fondamentale iportanza nella funzione di "updateAnimationTick()" discussa in precedenza perchè definisce il limite entro il quale la variabile "aniIndex" deve essere resettata a 0 per ricominciare il ciclo di animaizoni.**

- ## EnemyConstants:

Contiene a sua volta, sottoclassi che definiscono costanti per ogni tipo di namico "Nightborne", "HellBound" e "Ghost" con lo stesso schema illustrato sopra, vi sono quindi costanti che misurano le misure degli sprite, in più costanti che descrivono lo stato del nemico:

```
public static final int NIGHT_BORNE = 0;

public static final int NIGHT_BORNE_IDLE = 0;
public static final int NIGHT_BORNE_RUN = 1;
public static final int NIGHT_BORNE_ATTACK = 2;
public static final int NIGHT_BORNE_HITTED = 3;
public static final int NIGHT_BORNE_DIE = 4;
```

Ogni nemico possiede i suoi stati peculiari ed un intero che lo descrive, intero utilizzato con la l***ogica del LevelData*** per decidere tramite la componente RBG Blu quale nemico posizionare in quel punto al momento della creazione di questi.

Associati a questa sottoclasse nascono 5 metodi:

- "**getSpriteAmount(enemyType, enemystate)**": restituisce in base al tipo di nemico, il numero di sprite corrispondenti allo stato in ingresso di quel nemico. Utilizzata anch'essa ampiamente nell' "**updateAnimazionTick()**" dei nemici per far ricominciare il loop di animaizone.
- "getMaxHealth(enemytype): restituisce in base al tipo di nemico la sua vita massima.
- "getEnemyDamage(enemytype): restituisce in base al tipo di nemico il suo danno
- "getAttackDistance(enemytype): restituisce in base al tipo di nemico la sua distanza di attacco, utilizzata nella logica dei nemici per definire il momento in cui questo determinato nemico può entrare nell'animazione di attacco.
- "getVisiondistance(enemytype): restituisce in base al tipo di nemico il suo raggio di visione o di azione entro il quale riesce a vedere il Player, se il player entra in questo raggio di azioneil nemico viene triggerato e portandolo a fare determinate azioni.

- **Projectiles:**

Sottoclasse contenente le costanti di misure e tipo dei vari proiettili del gioco. I suoi 3 metodi "**getProjectileWidth()**", "**getProjectileHeight()**" e "**getProjectileSpeed()**" restituiscono come da nome, rispettivamente, la larghezza del proiettile, la altezza, e la velocità.

- **UI:**

Contenente costanti per i bottoni dei menù di gioco, staright forward, altezze e larghezze, basta.

- **PlayerConstants**

Anch'essa molto simile a quella dei nemici, contiene tutte le costanti del player, le misure degli sprite delle animazioni, quelle delle abilità speciali, tutte le costanti che rappresentano i suoi molteplici stati (Idle, Jumping, Running, ecc.) .

Possiede il metodo "**getSpriteAmoun(PlayerAction)**" che data una azione, restituisce il numero di frame di quella animazione, utilizzata nella "**updateAnimationTick()**" per far ripartere il loop di animazioni.

HelpMethods:

La classe "HelpMetods" fornisce un insieme di metodi di utilità per diverse operazioni legate alla gestione del gioco. Ecco una descrizione dettagliata:

- **Metodo "getLevelData(BufferedImage img)"**Questo metodo riceve un'immagine rappresentante i dati di un livello del gioco e restituisce una matrice bidimensionale contenente le informazioni sulla disposizione dei blocchi nel livello.
- **Metodo "GetPlayerSpawnPoint(BufferedImage img):** Restituisce il punto di spawn del giocatore all'interno del livello basato su un'immagine.
- **Metodi "getPotions(BufferedImage img)" e "getLootBoxes(BufferedImage img):** Entrambi i metodi ricevono un'immagine e restituiscono una lista di oggetti "Potion" o "LootBox", rispettivamente, posizionati all'interno del livello in base ai colori dell'immagine.
- **Metodi "getSpikes(BufferedImage img)" e "getCannons(BufferedImage img):** Questi metodi generano una lista di oggetti "Spike" o "Cannon" basati sui colori presenti nell'immagine.
- **Metodo "canMoveHere(float x, float y, float width, float height, int[][] lvlData):** Descrizione: Verifica se l'entità può muoversi in una posizione specifica senza collisioni con i muri.
- **Metodo "isPathClear(int[][] levelData, Rectangle2D.Float enemyHitbox, Rectangle2D.Float playerHitbox, int enemyY):** Descrizione: Determina se il percorso tra un nemico e il giocatore è libero da ostacoli.
- **Metodo "projectileHittingWall(AbstractProjectile projectile, int[][] levelData):** Descrizione: Verifica se un proiettile sta colpendo un muro all'interno del livello.
- **Altri metodi di verifica e gestione del terreno e delle collisioni:** I metodi come "canCannonSeePlayer", "isPlayereInFrontOfCannon", "isPlayereInRange", "areAllTilesClear", "isSolid", "isTileSolid", "getEntityXPosNextWall", "getEntityYPosFloorRoofRelative", "isEntityOnFloor", "isFloor" , gestiscono diversi aspetti delle collisioni, dei movimenti e delle verifiche di visibilità tra oggetti all'interno del gioco.

In generale, la classe fornisce un insieme di metodi di utilità necessari per varie operazioni fondamentali nel contesto del gioco implementato.

LoadSave:

La classe LoadSave menzionata precedentemente in qualche occasione, è la responsabile dell'estrazione immagini dalla cartella "**/res**". Questa un pò come la classe "**Costants**" discussa pocanzi, contiene decine di Stringhe costanti indicanti il Path delle cartelle corretto per raggioungere una determinata risorsa, alcuni esempi:

```
public static final String LEVEL_ATLAS = "enviroment/Terrain.png";
public static final String FOREST_LAYER_1 = "enviroment/background_layer_1.png";
public static final String PLAYER_ATLAS = "entity/MageAnimations.png";
```

Due metoodi gestiscono questi percorsi:

- "**getSpriteAtlas(PathName)**": questa funzione richiede in ingresso una costante di percorso file e tramite un algoritmo, converte l'imput stream dell'immagine in una buffered image, che infine restituisce.

  ```
  ...
  BufferedImage img = null;
  InputStream is = LoadSave.class.getResourceAsStream("/Progetto_prog_3/res/" +
  fileName);
  ...

  img = ImageIO.read(is);
  return img;
  ```

- "**getAllLevels()**": è una funzione un attimino più complessa della precedente ma che ha come obbiettivo di base quasi la stessa cosa, restituire un array di immagini, quelle dei livelli da mandare poi in pasto al "**LevelManager**" per la costruzione degli stessi e la creazione dei "**LevelData**". La funzioone definisce una variabile contenente il path per raggiunrere la cartella contenente i livelli, dopodichè crea un nuovo oggetto "File". Questo viene inizializzato come un oggetto descrittore della cartella raggiunta tramite path dichiarato prima. Vengono poi ordinati i file contenuti in base al nomi, memorizzati in un array di BufferedImage e ritornati coome ooutput della funzione.

Conclusioni capitolo 2.1:

E’ stata una lunga carrellata ma siamo arrivati alla conclusione del *Capitolo 2.1: “Classi e Interfaccie Indipendenti”.*

Abbiamo descritto tutte le classi e le interfaccie che sono colonne portanti di tutta la struttura del gioco, da qui in poi, le logiche di base saranno già state definite e bisognerà definire quelle più specifiche delle singole classi o nodi che estendono o implementano queste appena illustrate. *Fiondiamoci quindi ancora di più nella tana del Bianconiglio...*

Capitolo 2.2: Commento alle Classi di primo livello

In questo capitolo andremo ad osservare quelle che nel codice sono le:

- ***Classi di primo livello***: sono quegli elementi del progetto che estendono o implementano una o più *classi/interfacce indipendenti* descritte nel capitolo precedente, che eseguono quindi ad un livello di astrazione inferiore e più specifico funzionalità e caratteristiche di classi e nodi che iniziano ad avere una forma più definita verso gli oggetti concreti presenti nelle scene.

Le classi nel progetto di questa tipologia sono:

1. Playing
2. Player
3. AbstractEnemy
4. Cannon
5. CannonBall
6. LootBox
7. Potion
8. Spike
9. MenuButton
10. PHRButtons
11. SoundButton
12. VolumeButton
13. AudioOptions
14. Menu
15. GameOptions
16. GameOverOverlay
17. LevelComplitedOverlay
18. PauseOverlay

Playing (Estensione di State ed implementazione di StateMethods):

Questa è la classe a cui facciamo riferimento quando parliamo dello stato di Playing. Essa gestisce tutta la dinamica di quando l'utente gioca ed interagisce col mondo di gioco, è inoltre uno dei nodi figli della classe Game. **Possiede** anche le **la caratteristica di essere un nodo padre per le classi** di tipo: "**EnemyManager**", "**ObjectManager**", "**LevelManager**", "**Player**", "**GameOverOverlay**", "**LevelComplitedOverlay**" e "**PauseOverlay**" e richiama i loro metodi Render ed Update seguendo la logica base di programmazione di questo software.

Variabili di ambiente:

- Istanze delle classi figlie appena descritte.
- "**Paused**": variabile booleana per indicare se il Playing si trova nello stato di pausa.
- "**PlayerDying**": una flag che indica se il Player si trova nello stato di morte.
- "**GameOver**":variabile boleana per indicare se il Player è morto e bisogna quindi settare lo stato a GameOver.
- "**LevelComplited**": flag che indica se il livello corrente è stato completato
- "**x ed y LevelOffset**": servono a gestire il movimento della telecamera nel mondo di gioco.
- "**Left, Right, Bottom ed Upper Border**": delimitano delle linee immaginarie che se toccate dal player, causano il movimento di telecamere in base alla direzione del player (L'effetto ottenuto è che la telecamera segue il Player).
- "**x ed y MaxLevelOffset**": limiti di mappa per le quali raggiunti i bordi del livello la telecamera smette di spostarsi anche se il player si trova oltre i bordi designati pocanzi.
- "**BufferedImage per i Backgrounds**": immagini che conservano i layer dello sfondo.
- "**MementoManager**": il manager che gestisce i punti di salvataggio implementante il *Memento Design Pattern.*

Il **costruttore** prende in ingresso il nodo padre "Game" e lo manda al costruttore sella superclasse "State" della quale abbiamo già parlato nel capitolo precedente. Esegue poi 3 richiami a delle sue funzioni.

- "**initClasses()**": semplice e diretta inizializza tutte le classi ed i nodi figli istanziati e dei queli abbiamo parlato durante la parte delle variabili di ambiente.
- "**calculateLevelOffset()**": assegna alle variabili x ed y maxLevelOffset i valori di ouput dal richiamo del livello corrente tramite il LevelManager, il quale possiede un metodo per calcolare questi valori. Ecco il codice per essere più chiari possibili.

```
maxLevelOffsetX = levelManager.getCurrentLevel().getLevelOffset();
maxLevelOffsetY = levelManager.getCurrentLevel().getLevelOffsetY();
```

- "**loadStartGame()**": molto semplice anch'essa, richiama i metodi dell'**EnemyManager** e dell'**ObjectManager** per caricare i corrispettivi oggetti in base al livello corrente, passato coome riferimento dal **LevelManager.**

```
enemyManager.addEnemies(levelManager.getCurrentLevel());
objectManager.loadObjects(levelManager.getCurrentLevel());
```

Ora siamo pronti per passare alla logica di **Update**. La classe Game farà i suoi conti per calcolare il tempo tra un update all'altro, e richiamerà l'Update dello stato corrente, nel caso lo stato corrente sia quello di Playing, questo nodo passerà per una serie di if i quali oserveranno lo stato dei valori booleani prima descritti, ed in base a quello attivo, eseguirà l'update di uno o più dei sui nodi figli seguendo l'attuale stato della partita descritto per l'appunto, da questi valori booleani. Esempio:

```
. . .
if (paused) {
            pauseOverlay.update();
} else if (levelCompleted) {
            levelCompletedOverlay.update();
. . .
```

Se il gioco non è in stato di GameOver, viene richiamato l'update dei nodi figli "Player", "EnemyManager", "ObjectManager" e "Levelmanager" (Il quale se è la prima volta ad essere updatato creerà un punto di salvataggio).

La funzione di Rendering "draw" segue la stessa identica logica, esegue dei controlli sulle variabili booleane rappresentative dello stato corrente, ed esegue il rendering dello stato corrente.

Durante la funzione Update viene fatto un controllo con la funzione "**checkCloseToBorders()**" per gestire i movimenti della telecamera.

Questo nodo inoltre implementa le funzioni della interfaccia "StateMethods" di cui abbiamo discusso. Seguendo un po' la stessa logica di update e render in base alle variabili booleane vengono mandati segnali e richiamati metodi di nodi figli, spesso quello che viene fatto è che il segnale del Mouse o della Tastiera catturato, viene inviato come riferimento a dei nodi figli per permettere a loro di elaborarlo in base alle loro necessità, vediamo un esempio:

```
public void keyPressed(KeyEvent e) {

        if (gameOver) {
            gameOverOverlay.keyPressed(e);
        } else {

            switch (e.getKeyCode()) {
                case KeyEvent.VK_W:
                    player.setUp(true);
                    break;
                case KeyEvent.VK_A:
                    player.setLeft(true);
                    break;
      . . .
```

Qui osserviamo come vengano settate a true alcune variabili legate al player quando un determinato tasto viene premuto, oppure nel caso di GameOver come il segnale venga passato al nodo figlio "**gameOverOverlay**".

La classe Playing inoltre possiede oltre ai soliti metodi di get e set, anche alcuni metodi peculiari che richiamano funzioni dei nodi figli, questi metodi sono:

- "**loadNextLevel()**": che resetta tute le variabili booleane di stato e richiama il metodo "loadNextLevel" del nodo figlio "LevelManager".
- "**checkPotionTouched, checkPlayerHitEnemy, checkObjectHit, checkSpikesTouched**": richiamano i rispettivi manager per controllare se c'è stata una collisione tra corpi.

Troviamo poi metodi standard utlizzati da altri nodi come:

- "**resetAll()**": che viene richiamato quando c'è bisogno di resettare il livello, vengono richiamati a cascata i reset di tutti i manager + vengono implementate logiche di reset personali come i reset delle variabili booleane.
- "**resetBools()**": richiamata dalla precedente resetta le flag booleane allo stato di partenza.
-

Una funzionalità importantissima dello stato di playing sono le sue innumerevoli implementazioni della "StateMethods" che offre diverse funzioni di **cattura dei segnali di input**, da mouse e da tastiera, **segnali che il nodo invia a nodi figli quando necessario** e facendone scaterare determinati eventi. Prendiamo ad esempio il Player di cui discuteremo a breve, possiede diverse "flag" con i propri getters e setters (Flag come quele che gestiscono il movimento direzionale) che vengono attivate e disattivate in base alla tenuta pressione o al rilascio dei tasti WASD della tastiera.

Player (Estensione di Entity):

Oooh, finalmente la classe “Player”, il nodo che tutti stavamo aspettando.

Il Player è il modo con cui l’utente si interfaccia con il mondo gi gioco, è quella Entità che noi possiamo controllare, alla quale impartiamo i comandi di movimento, fuga, attacco leggero, attacco pesante, schivata ed ultimate. Ecco come si presenta all’interno del gioco in alcune dei suoi frame.

Lo vediamo nello stato di Idle, corsa, attacco ed ultimate.

Iniziamo col definire che il player è figlio della classe Playing sopra illustrata e che pertanto effettua Render ed Update solo quando Playing fa richiamo ai suoi.

Possiede tantissime variabili di ambiente, escludendo quelle ereditate da “Entity”, andiamole a vedere:

- **Variabili per la gestione dei frame:**
  - **"aniSpeed "** - Definisce la velocità di animazione.
- **Variabile per definire l'azione del player:**
  - **"left, right, up, down, jump"** - Variabili booleane che indicano la direzione di movimento e l'azione di salto.
- **Variabili per le animazioni e il livello:**
  - **"animations"** - Matrice di immagini per le animazioni.
  - **"levelData;"** - Matrice di dati di livello.
- **Variabili per le hitbox:**
  - **"XOffset"** - Offset orizzontale della hitbox.
  - **"YOffset"** - Offset verticale della hitbox.
- **Variabili per il salto:**
  - **"jumpSpeed"** - Velocità di salto iniziale.
  - **"fallSpeedAfterCollision"** - Velocità di caduta dopo una collisione.
- **Variabili per StatusBarUI e Ultimate:**
  - **"statusBarImg"** - Immagine per la barra dello stato.
  - **"ultimate"** - Immagini per l'abilità speciale.
  - **"Variabili per le dimensioni e la posizione della StatusBarUI e delle barre di salute ed Ultimate."**
- **Variabili per la posizione verticale:**
  - **"tyleY "** - Posizione verticale in termini di blocchi nel livello.
- **Variabili per l'abilità speciale e gli attacchi normali:**
  - **"powerAttackActive, ultimateActive"** - Flag per indicare l'attivazione dell'abilità speciale e dell'attacco finale.
  - **"powerAttackTick, powerGrowSpeed, powerGrowTick"** - Contatori e velocità di crescita per l'abilità speciale.
  - **"attackChecked, canPlayAttackSound, hurted"** - Flag per controlli di attacco e suoni, e stato di ferimento.
- **Altre variabili:**
  - **"ultimateAttackBox"** - Hitbox per l'attacco finale.
  - **"damage"** - Danno inflitto agli avversari.
  - **"playing"** - Oggetto Playing per la gestione del gioco.

Il **costruttore** richiede un riferimento al nodo padre “Playing”, una posizione di partenza in coordinate X ed Y, una altezza ed una larghezza. Vengono inviate le varibaili necessarie alla costruzione della superclasse a questa, tramite la funzione “**initStates()**” vengono inizializzate alcune variabili importanti come la “walkSpeed” oppure la “currentHealth”, la funzione “**loadAnimation()**” carica gli sprite delle animazioni utilizzando anche la classe “**LoadSave**”, “**initHitbox()**” inizializza la hitbox ereditata da “**Entity**” (è infatti un metodo proprietario della classe madre) ed “**initattackBox()**” inizializza le due attackBox, quella dell’attacco standard e quella della suprema.

Tenendo conto di tutto ciò che può fare il Player, è facile immaginare quanto sia difficile parlare della funzione Update. Questa effettua numerosi controlli molto stratificati e richiama funzioni peculiari della classe che aggiornano ed osservano tutte le condizioni possibili, di ogni caratteristica del player in relazione all'ambiente di gioco ed agli input del giocatore. Osserviamola step by step.

Funzione Update:

Nella funzione Update la prima cosa che incontriamo sono due funzioni, la "**updateHealthBar()**" e la "**updatePoewrBar()**". Nella GUI di gioco queste due barre sono presenti e responsive, e sono di fatto gestite dal nodo del Player. **La prima modifica la lunghezza della "healthBar"**, o barra della vita, in base alla vita corrente del player. E' un rettangolo rosso posizionato sopra l'immagine della "statusBar", **la seconda fa lo stesso per la "powerBar"**, un rettangolo giallo posizionato sulla status bar, agiornandone la lunghezza anche in relazione al tempo (Una meccanica di gioco è che la powerBar si ricarica nel tempo).

Troviamo poi un controllo if molto esteso il quale aggiorna stati del nodo padre Playing, nel caso il player stia morendo (si ricorda infatti che nel nodo Playing vi è una variabile booleana "playirDying" che osserva la morte del payer").

Per dare un effetto pathos alla morte del player, prima di agiornare lo stato di Playing su "GameOver", viene fatto una sleep del thread aumentando solo l'animation tick del player, in modo che esso si aggiorni mentre tutto il resto rimanga fermo. **Se il player ha ancora degli HP, questo blocco di codice viene saltato e viene fatto u nomrale "updateAnimationTick()" prima effettuare un return della funzione**.

Proseguiamo il viaggio verso "**updateAttackBox()**" che aggiorna la posizione della attackBox in relazione alla direzione del movimento del Player e verso una funzione estremamente profonda ovvero "***updatePosition()***" che analizzeremo successivamente poichè un capitolo a parte.

Un controllo sulla flag di movimento "moving" fà sì che dei controlli riguardanti la collisione con degli oggetti vengano eseguiti richiamando le funzioni del nodo padre Playing di cui abbiamo già parlato, queste sono infatti "**checkPotionTouched()**" e "**checkSpikesTouched()**". Alcuni controlli successivi aggiornano i tick di una abilità speciale del player, il "***Dash()***" del quale parleremo successivamente.

Un ultimo controllo if gestisce l'attivazione della funzione "**checkAttack()**". L'evento viene triggerato nel caso una tra due flag tra "*attacking*" e "*ultimateActive*" sia attiva e che lo stato attuale del player sia diverso da "*HURT*".

Passato anche questo ultimo controllo, viene eseguita una "**updateAnimationTick()**" e la funzione "setAnimation()" che se pur lunga possiamo descrivere nel seguente modo: ***il player ha degli stati, utilizzati per scegliere quale riga della matrice di animazioni andare a renderizzare, ogni stato ha velocità di animazione (aniSpeed) differenti e viene attivato sotto determinate condizioni.*** Degli if assegnagno questi nuovi stati e l'aniSpeed in base a quale flag di stato sia attiva in quel momento, un esempio renderà tutto estremamente più chiaro:

```
. . .

if (moving) {
            aniSpeed = 15;
            state = RUNNING;
}

. . .

if (ultimateActive) {
            state = USING_ULTIMATE;
            aniSpeed = 23;
            return;
}
. . .
```

Semplice e pulito 😀. Una osservazione che possiamo fare è che su abilità speciali, viene efettuato un return della funzione per evitare che stati incompatibili si accavallino tra loro.

Detto ciò, analizziamo più a fondo delle funzioni nominate nella funzione Update che hanno bisogno di avere maggiore attenzione.

Funzione UpdatePosition():

Lo spostamento di una qualsiasi entità avviene sempre allo stesso modo, **la velocità di movimento viene aggiunta ad ogni iterazione di un Update alla coordinata spaziale delle X ed Y**, in negativo se l'entitò si sta muovendo verso la sinistra per le X od in alto per la Y, ed in positivo se si sta muovendo verso la destra per le X ed in basso per la Y. Nonostante il player abbia un "Dash" neaanche lui scappa da questa regola. Quello che è importante osservare è come e quando questo approccio viene applicato, anche qui andiamo con ordine.

In modo sequenziale vengono eseguiti questi seguenti controlli if:

- Se la flag "**jump**" e lo stato è diverso da "**HURT**" viene eseguita la funzione "**jump()**", che semplicemente setta la flag "inAir" a true e la velocità di movimento in aria "airSpeed" uguale alla costante "jumpSpeed" (Entrambe atte alla simulazione della gravità).
- **Se la ultimate è attiva oppure se, il player non è in aria, non sta eseguendo il dash, e sta effettuando movimento destra/sinistra concorrenti oppure non li sta effettuando affatto**, viene eseguito un return dalla funzione "updatePosition()", tutte qeste sono condizioni che escludono il movimento e che quindi causano questo evento di return.
- **Se il player non sta attaccando e non sta andando verso sinistra ma sta andando verso *destra*, viene aggiornata la posizione per il movimento verso destra**, vengono inoltre settate delle variabili additive e moltiplicative per la rendering.
- **Se il player non sta attaccando e non sta andando verso destra ma sta andando *verso sinistra*, viene aggiornata la posizione per il movimento verso sinistra**, vengono inoltre settate delle variabili additive e moltiplicative per la rendering.
- **Se il dash è attivo, viene aumentata per la durata di questo, la velocità di movimento orizzontale**, così da permetter al Player di muoversi velocemente per una breve durata di tempo
- **Se non si trova in aria** viene fatto un controllo per vedere l'opposto, ovvero se si trova in aria, questo serve a settare la flag a vera nel caso il player abbia iniziato l'evento di salto.
- **L'ultimo if serve a gestire la simulazione della gravità**, se di fatto si trova in aria e non sta usando la ultimate, la componente y della sua posizione viene aggiornata così come la x, funzioni per controllare se il player si trova in aria, sul terreno e se può muoversi in una certa direzione orizzontale sono ampiamente utilizzate. Queste vengono importate dalla classe "HelpMethods" di cui abbiamo già discusso.

Funzione checkAttack():

Questa viene chiamata come una delle ultime funzioni della Update principale. La sua attivazione è in corrispondenza della flag "attacking" oppure della flag "ultimateActive". Utilizza una flag delle variabili di ambiente chiamata "attackChecked" per gestire l'ingresso a questo blocco di codice, di fatto quando questa è attiva, la prima istruzione "**if(!attackChecked) return;**" ci fa subito uscire dalla funzione per evitare di applicare il danno innumerevoli volte nei tanti tick di memoria tra un update e l'altro.

Viene analizzata la flag di ultimate per la quale se questa non è attiva viene impostato il danno su 5.

Due *if* effettueranno dei richiami al nodo padre per controllare se il player stia attaccando una qualche altra entità sullo schermo. Il primo riguarda gli attacchi semplici ed il secondo la ultimate, richiamano i metodi "**checkPlayerHitEnemy**" e "**checkObjectHit**" del nodo padre **Playing** (*Che a sua volta richiama quello dei manager*) per controllare se il player in quella determinata animazione di attacco stia colpendo qualcosa. In entrambi i casi viene scelto di fare il controllo solo quando il Player si trova in un determinato "aniIndex" ed una volta effeettuati i conti, **la flag "attackChecked" viene impostata a false così da bloccare l'ingresso a future iterazioni** (*La variabile viene resettata nella "**updateAnimationTick()**" quando l'animazione di attacco o di ultimate finisce così da permettere di applicare il danno al prossimo attacco*).

Se pur semplice questa funzione meritava un suo paragrafo poichè gestisce un ruolo fondamentale nell'interazione del player con il mondo di gioco.

Altre funzioni:

La classe Player possiede ovviamente decine di altre funzioni, fatta eccezione per i getters e setters delle sue variabili di ambiente, ampiamente utilizzate da perte del nodo Playing per inviare segnali a questa entità, e delle funzioni già nominate vi troviamo:

- La solita funzione **Render** che seguendo perfettamente la "logica di disegno di una entità" renderzza il player nei suoi stati e nelle sua animazioni.
- "**drawUI()**": che disegna la status bar la barra della vita e dell'energia in alto a sinistra del pannello di gioco.
- "**ultimateAbility()**" e "**dash()**" che attivano le flag per le due abilità da cui prendono il nome, dano inoltre danni differenti per un breve periodo come nel caso della prima ed in entrambi i casi diminuendo i livelli di energia correnti se possibile, altrimenti quando invocate, queste non verranno arrivate (Il player deve avere abbastanza energia per dashare e usare la Ultimate).
- "**changeHealth()**" e "**changePower()**" che quando invocate modificano i valori della vita e della barra delle abilità
- "**resetAll()**" che utilizzando il Memento Design Pattern, ripristina lo stato iniziale del Player quando invocata.
- "**loadAnimations()**" che carica gli sprite di animazione nella matrice di animazioni.
- "**loadLevelData()**" carica il "current LevelData" al momento dell'inizializzazione di un nuovo livello, per far sapere al Player dove può e non può muoversi.
- "**resetMovement()**" porta a 0 tutte le variabili booleane legate al movimento direzionale.
- "**setSpawnPoint()**" posiziona le coordinate di partenza del Player all'inizio di un nuovo livello.
- "**jump()**" attiva le flag per la meccanica del salto.
- "**die()**" attiva le flag per lo stato di DIE posizionando la vita a 0 ed impedendo il movimento aereo.
- "**saveState()**" sempre legata al Memento Design Patter, ritorna un nuovo "PlayerMemento" analizzato nel capitolo legato ai Design Pattern da poter usare quando vengono creati i saving point nel LevelManager.

In generale, tante flag gestiscono l'indice di riga e di colonna del rendering e delle animazioni, queste stesse flag quando attive eseguono anche i movimenti direzionali e delle abilità speciali, getters e setters richiamati dal nodo padre o da altre parti del codice come i Manager, inviano segnali tramite queste flag per impostare nuovi stati e rendere il player interattivo con il mondo di gioco. Fatto finito 😀. Speriamo di non aver dimenticato nulla di importante.

Il player gioca ovviamente un ruolo fondamentale all'nterno di WOP e meritava pertanto una spiegazione (se pur sorvolata su alcuni punti) più approfondita e dettagliata. *Possiamo ora proseguire con le altri Classi di primo livello.*

AbstractEnemy (Estensione di Entity):

E qui ricomincia un capitolone.

Anche **AbstractEnemy** come Player vive di un ruolo fondamentale all'interno del mondo di gioco, astrae di fatto tutte le caratteristiche generali dei nemici e le accorpa in una unica classe omogenea capace di gestire ogni meccanica fondamentale di un nemico, offrendo anche spunto di creatività lasciando non implementati diversi metodi astratti sfruttando pure l'utilizzo del **Template Method** studiato nella sezione dedicata dei Design Pattern. Praticamente una personalizzazione del nemico estrema.

Partiamo dalle variabili di ambiente:

- "**enemyType**": descrive con un intero il tipo di nemico (intero che viene definito descrittivo nella classe "Costants".
- "**aniSpeed**": sappiamo cos'è, che viene impostata a 20 di default.
- "**firstUpdate**": segnala che il nemico è stato appena creato e viene gestito l'evento di conseguenza.
- "**walkDir**": la direzione iniziale del movimento direzionale, viene impostata per andare a Sinistra.
- "**enemyTyleY**": variabile molto importante che indica in che altezza rispetto al livello si trovi il neimco, ampiamente usata nel calcoli per interagire con il Player.
- "**attackdistance**" e "**visionDistance**": indicano da quando lontano un nemico può rispettivamente provare ad attaccare il player e vedere il player.
- "**attackChecked**": usata esattamente come nel caso del player per gestire la meccanica di attacco.
- "**active**": forse la flag più importante. Indica se un nemico è attivo o no nel mondo di gioco la morte praticamente.

Il **costruttore** richiede i soliti parametri di una entità in aggiunta di un intero, descrittivo del tipo di nemico. Con questa variabile viene di fatto eseguita l'assegnazione ad "enemyTye", viene poi inizializzata una "walkSpeed" di default (ereditata da Entity) e la Hitbox (non la attackBox che è definita in modo diverso dalle classi figlie). Come passo successivo, il costrutore utilizzando i metodi della classe "**Costants**" "**getVisionDistance(), getAttackDistance() e getMaxHealth()**" inizializza le tre variabili "**visionDistance, attackDistance e maxHealth**" in modo dinamico in base alla variabile "enemyType".

Metodi Astratti:

Alcuni metodi astratti vengono lasciati all'implementazione delle classi figlie per dare loro scelta di come eseguire determinate azioni. Questi sono

- **Update**: classico ma con necessità in questo caso di essere astratto.
- "**makeMovement()**": implementazione del Template Method Design Pattern che nelle classi figlie implementa tutta la logica di movimento e di stati peculiari per ogni nemico (qualcuno semplicemente cammina, altri saltano e possono cadere in dei burroni, altri si teletrasportano ;D).
- "**flipX()**", "**flipY()**", "**flipXP(Player)**" e "**flipYP(Player)**": che servono a modificare dei valori moltiplicativi usati nel render di una entità. Spieghiamoci un attimo meglio, uno sprite viene renderizzato a schermo così come è stato conservato in memoria, ma se moltiplichiamo la sua ampiezza per un valore negativo, possiamo disegnare la stessa immagine ma specchiata rispetto all'asse Y, così da dare l'illusione di una entità che si sposta in direzioni diverse. Queste funzioni tramite dei semplici if, controllano se il movimento o il player sono verso destra, e quindi i moltiplicatori in output saranno positivi, oppure a sinistra e quindi negativi.

Metodi utili e varie implementazioni:

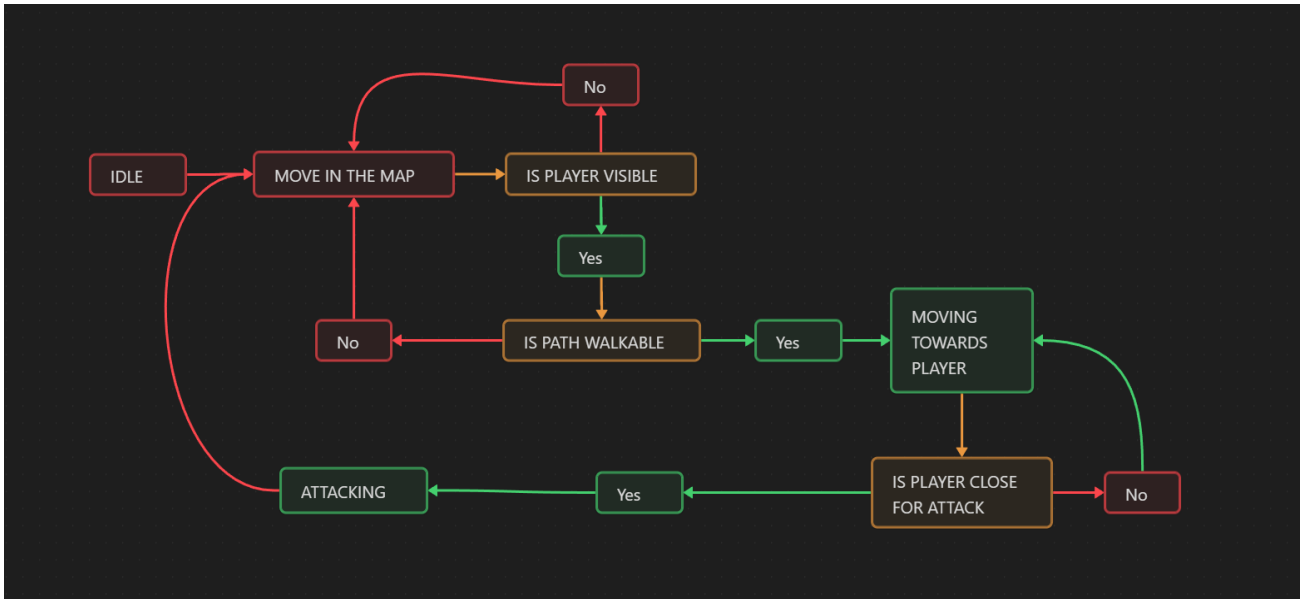
Se nel paragrafo precedente abbiamo esposto i metodi astratti in questo paragrafo andremo a studiare l'implementazione di quei metodi che sono di solito standard per tutti e che in alcuni casi possono ricevere "Override" per gestire le dinamiche in modo differente.

- Il primo che incontriamo è "**act()**": act è l'implementazione del Template Method Pattern cheesegue 3 controlli. Se il nemico si trova nella fare di "**firstUpdate**" viene eseguita la funzione "**firstUpdateCheck()**" che arriva la flag "**inAir**" se il nemico non si trova sul terreno (una cosa comune dato che spawnano in aria). Se la flag "inAir" è attiva il prossimo if gestisce la simulazione della gravità e fa cadere il nemico verso il terreno tramite la funzione "**updateInAir()**". Richiede infine il metodo astratto "**makeMovement()**" che dovrà essere implementato dalle classi figlie.
- "**turnTowardsPlayer(Player)**" cambia la "walkDir" in base alla posiizone del player rispetto al nemico in questione.
- "**canSeePlayer(Player, levelData)**": Questa funzione che ritorna un valore booleano restiuisce true se il player viene visto dal nemico. Utilizza due principali metodi, uno di implementazione propria "**isPlayerInRangeOfVision()**" che ritorna vero se la differenza in valore assoluto tra la posizione X del player e quella del nemico è minore della "**visionDistance** ed "**isPathClear()**" richiamata dalla class "**HelpMethods**" che restituisce vero se non ci sono blocchi di terreno tra il player ed il nemico e se nn ci sono fossati tra il player ed il nemico. **Se entrambe le funzioni restituiscono vero allora la "canSeePlayer()" ritorna vero.**
- "**isPlayerCloseForAttack(Player)**" controlla che il player si trovi abbastanza vicino per essere attaccato, ritorna vero se la differenza in valore assoluto tra la posizione X del player e quella del nemico è minore della "**attackDistance**" altrimenti ritorna falso.
- "**newState(int)**" che preso un intero in input assegna un nuovo behavioral state al nemico in questione.
- "**move(levelData)**": è la funzione usata dai nemici non statici o non volanti per muoversi sul terreno. Il movimento segue la stessa logica del player, ovvero viene aggiunta una costante alla posizione in X del nemico (nel nostro caso "**walkSpeed**") posizionata a negativa se il movimento è verso sinistra ed in positivo se il movimento è verso destra. Le funzioni "**canMoveHere()**" ed "**isFloor()**" prese dalla classe "**HelpMethods**" controllano se il movimento desiderato sia permesso al nemico, **la prima** presa in input la posizione di movimento prevista ed il levelData restituisce veero se il nemico può muoversi in quella direzione poichè esente di muri. **La seconda** applica la stessa logica della prima ma applicata al pavimento. **Se entrambe restituiscono vero il movimento in quella direzione viene permesso e la posizione in x viene aggiornata**. La funzione poi effettua un "return". **Nel caso una delle due funzioni abbia restituito falso** significa che il movimento previsto non può essere eseguito ed return non verrà effettuato, portanto alla prossima funzione "**changeWalkDir()**" che in base alla direzione di movimento corrente, la setta al suo opposto, esempio pratico: un nemico vuole muoversi a destra ma non può farlo perchè c'è un muro oppure un burrone, deve quindi smettere di andare a destra e iniziare a muoversi a sinistra, ecco che la funzione "changeWalkDir()" entra in azione ed effettua questo cambiamento cambiando la variabile "walkDir" al suo opposto. Il nemico alla prossima iterazione inizierà a muoversi nella direzione opposta.
- "**hurt(int)**": preso un valore in input diminuisce la vita corrente del nemico in base al valore dato. Se la vita corrente è minore di 0 viene settato il nuovo stato a quello di morte "DIE", altrimento a quello di "HURT".
- "**checkEnemyHit(attackBox, Player)**" controlla se vi sia intersezione tra la attackbox del nemico in questione e la hitbox del player, se questo è vero richiamando il medoto "**changeHealt()**" di Player, viene applicato il danno del nemico in base al tipo di questo.
- "**updateAnimazionTick()**" di cui abbiamo già ampiamente discusso agiorna lo stato attuale dell' "aniTick" e dell' "aniIndex". Momento in cui una animazione finisce secondo una logica che non abbiamo ancora mostrato, finisce anche uno stato del nemico, questo viene impostato a quello di partenza quando finisce una animazione. Illustreremo in un paragrafo successivo come questo meccanismo funziona.
- "**restoreState()**": preso in input un EnemyMemento riporta allo stato registrato nel memento il nemico che la invoca.
- "**Getters e Setters**" classici per ottenere informazioni riguardanti variabili stati eccetera.

Logica di loop di un Nemico

Implementare una IA per la gestione dei nemici non è un obbiettivo di questo software pertando ci si è dovuti ingegnare per implementare una qualche logica uguale a tutti i nemici che gli permettesse di essere perfettamente responsivo agli eventi di gioco. Se il player aveva le flag booleane **per implementare gli stati, i nemici hanno degli interi descritti nella classe "Costants" che per la Render, scelgono quale riga della matrice di sprite andare a disegnare, e per la update invece gestiscono tramite uno switch case i comportamenti del nemico.**

Il "**Behavioral Loop**" (così ci è piaciuta chiamarla) è la logica applicata per dare vita al nemico e segue questo schema.



Il codice generale è implementato tramite uno switch case e fa uso delle funzioni descritte precedentemente per rispondere a quelle domande ed impostare i nuovi stati di conseguenza. Analizzeremo i singoli "behavioralLoop" dei nemici quando parleremo delle loro classi poichè implementazioni del Template Method.

Cannon (Estensione di AbstractObject):

La classe cannon implementa il nodo dei cannoni. Ecco come si presentano nel gioco:

La loro implementazione in codice è semplicissima, vivono essensialmente del loro costruttore.

Come variabili di ambiente possiedono:

- “cannonTyleY”: la posizione verticale nella mappa.
- “cannonBall”: una classe estensione di proiettile che non abbiamo ancora osservato (utilizzata per il Prototype Design Pattern)
- “direction”: la direzione grazie alla wuale il cannone ed i suoi proiettili saranno posizionati/lanciati verso la destra o la sinistra.

Il costruttore è come dicevamo il cuore del nodo, esso:

1. Richiama il costruttore della classe madre inviando gli argomenti necessari.
2. Calcola la sua posizione in Y e la conserva in “cannonTyleY”
3. Inizializza la sua hitbox (Se pur non interagibile serve a posizionarlo all’interno del mondo di gioco) e la riposiziona in modo che lo spritre che verrà disegnato al di sopra di questa combaci con il livello del terreno.
4. La variabile “direction” viene impostata a –1 se l’ ”objectType” in ingresso è uguale a “CANNON_LEFT”.
5. Viene creata una istanza della classe “CannonBall” associandola all’oggetto prima creato sulla base delle coordinate del Cannone e della sua direzione.

La sua funzione update avviene solo quando arriva un determinato segnale, la classe madre ricordiamo che possiede la variabile “doAnimation” in questo caso utilizzata per far partire la funzione di “updateAnimationTick()”. Ricordiamo che se questa funzione nota che l’objType è un “CANNON” imposta la flag di animazione “doAnimation” a false quando l’animaizone finisce, l’effetto ottenuto è che quando il player si trova davanti al cannone, calcolo fatto grazie al “cannonTyleY” ed dall’tyleY di Player all’interno dell’ObjectManager, questo comincia ad eseguire la sua animazione che si fermerebbe solo quando il PLayer si sposterà dalla sua traiettoria.

Metodi Getters e Setters per ottenere informazioni

CannonBall (Estensione di AbstractProjectile, implementazione di ProjectileInterface):

Ach'essa è una classe che vive del proprio costruttore e che implementa il "Prototype design patter". Nel mondo di gioco si mostra nel seguente modo:

Il **costruttore** richiama la superclasse per istanziare il tipo di proiettile, calcola poi gli offset per un corretto posizionamento della hixbot rispetto al cannone che la spara, se la durezione è quella di destra, l'offset della coordinata X viene aumentato per renderlo coerente col cannone girato verso la destra, la hitbox viene inizializzata con le coordinate così calcolate, ed usanco Costanti di dimensione dalla classe Costants.

Vie è in particolare il metodo implementante l'interfaccia ProjectileInterface "makeClone()". Questo metodo implementa il Prototype Design Pattern sfruttando la capacità della superclasse ereditata "Cloneable" per creare un clone di se stessa e restituirlo come output del metodo. Questa caratteristica è utilizzata nell'ObjectManager per creare nuovi proiettili sulla base di esempi istanziati in ogni Cannone di cui abbiamo discusso nel paragrafo dedicato.

LootBox (Estensione di AbstractObject):

Le LootBoxes sono oggetti di gioco con il quale il Player è in grado di interagire, queste possono contenere pozioni di vario tipo quando distrutte e di cui parleremo successivamente. Ecco come si presentano in gioco le due tipologie di lootBoxes:

Possiedono una sola variabile boolean "canSpawnPotion" che indica se può far spawnare una pozione.

Il costruttore prende in input delle coordinate di partenza ed un "objType", esegue il costruttore della classe madre "AbstractObject" che ne inizializza le caratteristiche basi dopodichè, come accadeva per il cannone, viene inizializzata una hitbox stavolta in base all'objType (Il barile è poco più grande della cassa) usando lo stesso approccio di costruzione offsets per il corretto allineamento di hitbox e sprite con il terreno di gioco che usava il cannone.

Possiede due funzioni:

- **Update** ragiona esattamente come ragionava quella del cannone, possiede un controllo sulla flag "doAnimation", se questa è attiva viene eseguito l' "**updateAnimationTick()**" altrimenti viene skippata. L'unica differenza si trova nel come questa flag venga attivata, nel caso del cannone ciò avveniva se il Player era in traiettoria per essere sparato, qui invece **l'ObjectManager** effettua un **controllo di collisione con la AttackBox del Player**, per osservare se quando questo attacca, stia colpento una di questecasse. In tal caso verrà spawnata la pozione e l'animazione di questo oggetto verrà fatta partire, al termine la "updateAnimationTick()" porterà il suo stato su "inattivo" impostano la flag "isActive" a false.

Potion (Estensione di AbstractObject):

Le Pozioni spawnano quando una cassa viene distrutta in base a delle percentuali, queste possedendo “objType” differenti triggereranno eventi nel Player differenti quando verranno raccolte, ecco gli esempi di pozione blu e rossa presenti nel gioco:

Il costruttore ormai l’abbiamo capito, esegue le stesse istruzioni del Cannone e delle Lootbox.

Quello che è interessante osservare invece è la funzione Update sempre accessibile dal momento che quando le pozioni vengono spawnate sono subito interagibili. Troviamo al suo interno oltre all’ “updateAnimazionTick()” (che in questo caso non imposta lo stato ad inattivo quando una animazione finisce ma fa ricominciare il loop dell’aniIdex) la funzione “updateHovering()” una funzione semplicissima ma che regala a questo oggetto un effetto unico nel suo genere quello di “Hovering”. La pozione fluttua nell’aria essendo un oggetto raccoglibile molto piccolo ed avendo quindi bosogno di essere notata, spostandosi prima verso l’alto e poi verso il basso in un evento ciclico per il quale prima si somma un valore costaten alla coordinata Y e raggiunto un certo limite si inizia a sottrarre (vale il contrario).

Spikes (Estensione di AbstractObject):

Le “**Spikes**” o “Spine” vivono di un semplice costruttore richiedente delle coordinate spaziali e che richiama quello della classe estesa, inizializza una hitbox e la riposiziona sommandone degli offset calcolati in loco. La sua semplicità è grande quanto la sua letalità, di fatto sono l’unico oggetto interagibile del gioco capace di invocare direttamente la funzione “**die()**” del Player, portandolo ad una morte indolore ed instantanea (ps. Se non si fosse ancora capito il gioco è crudele dal punto di vista della dificoltà 😐). Ecco come si presentano nel gioco:

<u>Osservazione importante. Oggetti inamovibili di gioco:</u>

Fanno riferimento a questa categoria di oggetti tutti quelli che non possiedono movimento spaziale e che rimangono fermi nel mondo di gioco. **Ricadono in questa categoria i Cannon, le Lootboxes e le Spike.**

Ma come mai si sente la necessità di fare questa osservazione?

Gli oggetti inamovibili sono spawnati osservando l’immagine del “levelData” e non possiedono la capacità di muoversi nel mondo di gioco, ma solo di interagire con esso, le coordinate spaziali di spawn sono direttamente la scalatura secondo il “Game.TILE_DEFAULT_SIZE” delle coordinate matriciali della componente RGB corrispondente.

MenuButtons (Estensione di AbstractButton):

La classe MenuButtons rappresenta tutti i bottoni che si trovano all'interno del menù di gioco, si presentano in questo modo:

Analizziamone le principali caratteristiche.

Variabili id ambiente:

- "xPos, yPos": Rappresentano le coordinate x e y del punto in alto a sinistra del bottone.
- "xOffsetCenter": Rappresenta lo spostamento orizzontale dal centro del bottone. Viene utilizzato per centrare il bottone.
- "buttonHitbox": Un oggetto "Rectangle" che rappresenta l'area sensibile del bottone per le interazioni.
- "imgs": Un array di immagini che rappresentano gli sprite del bottone per diversi stati (normal, hover, cliccato).
- "GameState state": Rappresenta lo stato del gioco da applicare associato al bottone.

Il costruttore accetta le coordinate x e y del bottone, un indice di riga "rowIndex", e lo stato del gioco associato al bottone. Carica le immagini del bottone ed inizializza la hitbox.

Metodi:

- "**initButtonHitbox()**": Inizializza la hitbox del bottone utilizzando le coordinate x e y e le dimensioni del bottone.
- "**loadImages()**": Carica gli sprite del bottone da un'immagine contenente tutti gli sprite dei pulsanti del menu tramite la classe "LoadSave".
- "**draw()**": Disegna il bottone sulla schermata grafica utilizzando l'immagine corrispondente allo stato corrente del bottone seguendo la "L*ogica di disegno di una entità*".
- "**applyGameState()**": Applica lo stato del gioco associato al bottone al GameState generale.
- "**getHitbox()**": Restituisce l'oggetto "Rectangle" che rappresenta l'area sensibile del bottone.
- "**getState()**": Restituisce lo stato del gioco associato al bottone.

Possiede tutti gli elemmenti ereditati dalla classe mmadre pertanto non ha bisogno di implementazione della funzione Update e possiede tutti gli altri metodi utili.

PHRButtons (Estensione di AbstractButton):

L’acronimo sta per “**Pause Home Reset Buttons**” sono i tipi di bottoni che troviamo nel menù di pausa del gioco, raggiunto premente il tasto “Escape” della tastiera quando ci si trova nello stato di “Playing”. Nella gui si presentano in questo modo:

Analizziamone le principali caratteristiche.

Variabili di ambiente:

- "**imgs**": Un array di immagini che rappresentano gli sprite dei bottoni per diversi stati (normal, hover, cliccato).

Il costruttore accetta le coordinate x e y del punto in alto a sinistra del bottone, la larghezza e l'altezza del pulsante, e un indice di riga "rowIndex" utilizzato dalla funzione “Draw()” per secidere quale riga della matrice di sprite andare a disegnare.

Metodi:

- "**loadImages()**": Carica gli sprite dei bottoni da un'immagine contenente tutti gli sprite del gruppo di bottoni.
- "**draw()**": Disegna il bottone adeguato in base al “rowIndex” mandato in ingresso ed allo stato corrente dei bottone identificato dalle variabili della classe madre che abbiamo già descritto.

SoundButtons (Estensione di AbstractButton):

Gli audio buttons sono due, uno gestisce la presenza o assenza dei suoni del gioco ed uno invece quella della musica di sottofondo. Si presentano uguali nella gui pertanto ne verrà mostrato uno solo in entrambi gli stati:

Variabili di ambiente:

- "BufferedImage[][] soundImgs": Una matrice di immagini che rappresentano gli sprite del pulsante del suono per diversi stati (normale, hover, cliccato), organizzati in righe per rappresentare lo stato del volume (mute o non mute).
- "boolean muted": Una flag che indica se il suono è attualmente spento (muted) o meno.

Il costruttore accetta le coordinate x e y del punto in alto a sinistra del pulsante, la larghezza e l'altezza del pulsante. Richiama inoltre il metodo per caricar la matrice di Sprite.

Metodi:

- "**update()**": Aggiorna gli indici della matrice degli sprite in base allo stato del pulsante, considerando se il suono è attualmente spento, se il mouse è sopra il pulsante o se il pulsante è stato premuto.
- "**loadSoundImages()**": Carica gli sprite del pulsante del suono da un'immagine contenente tutti gli sprite del pulsante del suono, organizzati in una matrice in base allo stato del volume.
- "**draw()**": Disegna il pulsante del suono nella gui utilizzando l'immagine corrispondente allo stato corrente del pulsante.
- "**setMuted()**": Imposta lo stato del volume in base alla flag "muted".
- "**getMuted()**": Restituisce lo stato del volume (muted o non muted).

VolumeButtons (Estensione di AbstractButton):

Viene creata e gestita una classe per questo tipo di bottone ma viene creato un solo oggetto riutilizzato in tutte le interfaccie che ne hanno bisogno.

Variabili di ambiente:

- **"imgs"**: Un array di immagini che rappresentano gli sprite del pulsante del volume per diversi stati (normale, hover, cliccato).
- **"slider"**: Un'immagine che rappresenta lo slider del pulsante del volume.
- **"index"**: Un indice utilizzato per determinare quale immagine del pulsante del volume deve essere disegnata (normale, hover, cliccato).
- **"buttonX, minX, maxX"**: Coordinate x del pulsante del volume, del limite minimo e del limite massimo per il movimento del pulsante lungo la barra del volume.
- **" floatValue"**: Il valore effettivo del volume, normalizzato tra 0 e 1.

Il costruttore accetta le coordinate x e y del punto in alto a sinistra del pulsante, la larghezza e l'altezza del pulsante. Posiziona il pulsante del volume esattamente al centro della barra del volume quando il gioco parte. Carica le immagini del pulsante del volume.

Metodi:

- **"loadImages()"**: Carica gli sprite del pulsante del volume e lo slider da un'immagine contenente tutti gli sprite del pulsante del volume.
- **"update()"**: Aggiorna l'indice per determinare quale immagine del pulsante del volume deve essere disegnata in base allo stato (normale, hover, cliccato). Aggiorna la posizione della hitbox del pulsante del volume.
- **"draw()"**: Disegna lo slider e l'immagine del pulsante del volume nella gui.
- **"changeX()"**: Cambia la coordinata x del pulsante del volume, assicurandosi che rimanga all'interno dei limiti. Aggiorna il valore normalizzato del volume in base alla nuova posizione.
- **"updateFloatValue()"**: Aggiorna il valore normalizzato del volume in base alla posizione corrente del pulsante del volume.
- **"getFloatValue()"**: Restituisce il valore normalizzato del volume.

Menu (Estensine di State ed implementazione di StateMethods):

Il Menu se ricordiamo bene, è uno degli stati principali del nodo Game che è di fatto il nodo padre, quì dentro vi troviamo i MenuButton prima descritti ed è il responsabile del cambio di stato riguardandi lo State Design Pattern. Si presenta così:

Variabili di ambiente:

- **"buttons"**: Un array di oggetti "MenuButton", che rappresentano i bottoni nel menu. Ogni bottone è associato a uno stato di gioco.
- **"backgroundImage, actualBackgroundImage"**: Immagini di sfondo del menu e l'immagine effettiva di sfondo utilizzata per il rendering.
- **"menuX, menuY, menuWidth, menuHeight"**: Coordinate e dimensioni del contenitore del menu.

Il costruttore accetta un oggetto "Game" e inizializza le immagini di sfondo del menu e dei bottoni. Carica i bottoni e l'immagine di sfondo.

Metodi:

- **"loadBackground()"**: Carica l'immagine di sfondo del menu e imposta le dimensioni del contenitore del menu.
- **"loadButtons()"**: Carica i bottoni del menu e li posiziona nelle posizioni specificate.
- **"update()"**: Chiama il metodo "update" per ciascun bottone del menu.
- **"draw()"**: Disegna l'immagine di sfondo effettiva e il contenitore del menu, quindi chiama il metodo "draw" per ciascun bottone.
- **"mouseClicked(MouseEvent)"**: Gestisce l'evento di clic del mouse.
- **"mousePressed(MouseEvent)"**: Gestisce l'evento di pressione del mouse. Imposta il flag "mousePressed" per il bottone corrispondente se è stato premuto.
- **"mouseMoved(MouseEvent)"**: Gestisce l'evento di movimento del mouse. Imposta il flag "mouseOver" per il bottone corrispondente se è stato fatto l'hover.
- **"mouseReleased(MouseEvent)"**: Gestisce l'evento di rilascio del mouse. Applica lo stato del gioco associato al bottone se è stato premuto, e riproduce una canzone se il bottone è associato allo stato "PLAYING". Riporta i bottoni allo stato normale.
- **"keyPressed(KeyEvent)"**: Gestisce l'evento di pressione del tasto. Se viene premuto il tasto "ENTER", imposta lo stato di gioco a "PLAYING".
- **"keyReleased(KeyEvent)"**: Metodo vuoto per gestire l'evento di rilascio del tasto.
- **"resetButtons()"**: Resetta i flag dei bottoni, riportandoli allo stato normale.

GameOptions (Estensine di State ed implementazione di StateMethods):

Questo è l'ultimo grande stato di gioco che contiene le opzioni di audio personalizzabili prima di iniziare effettivamente a giocare. Si presenta in questo modo:

Variabili di ambiente:

- **"MenuButton[] buttons"**: Un array di oggetti "MenuButton", che rappresentano i bottoni nel menu. Ogni bottone è associato a uno stato di gioco.
- **"BufferedImage backgroundImage, actualBackgroundImage"**: Immagini di sfondo del menu e l'immagine effettiva di sfondo utilizzata per il rendering.
- **"int menuX, menuY, menuWidth, menuHeight"**: Coordinate e dimensioni del contenitore del menu.

Il costruttore accetta un oggetto "Game" e inizializza le immagini di sfondo del menu e dei bottoni. Carica i bottoni e l'immagine di sfondo effettiva.

Metodi:

- **"loadBackground()"**: Carica l'immagine di sfondo del menu e imposta le dimensioni del contenitore del menu.
- **"loadButtons()"**: Carica i bottoni del menu e li posiziona nelle posizioni specificate.
- **"update()"**: Chiama il metodo "update" per ciascun bottone del menu.
- **"draw()"**: Disegna l'immagine di sfondo effettiva e il contenitore del menu, quindi chiama il metodo "draw" per ciascun bottone.
- **"mouseClicked(MouseEvent)"**: Gestisce l'evento di clic del mouse.
- **"mousePressed(MouseEvent)"**: Gestisce l'evento di pressione del mouse. Imposta il flag "mousePressed" per il bottone corrispondente se è stato premuto.
- **"mouseMoved(MouseEvent)"**: Gestisce l'evento di movimento del mouse. Imposta il flag "mouseOver" per il bottone corrispondente se è stato fatto l'hover.
- **"mouseReleased(MouseEvent)"**: Gestisce l'evento di rilascio del mouse. Applica lo stato del gioco associato al bottone se è stato premuto, e riproduce una canzone se il bottone è associato allo stato "PLAYING". Riporta i bottoni allo stato normale.
- **"keyPressed(KeyEvent)"**: Gestisce l'evento di pressione del tasto. Se viene premuto il tasto "ENTER", imposta lo stato di gioco a "PLAYING".
- **"resetButtons()"**: Resetta i flag dei bottoni, riportandoli allo stato normale.

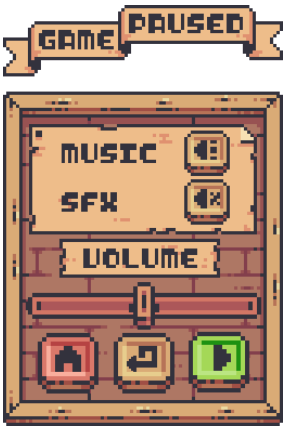
I Successivi nodi sono figli del nodo Playing e vengono mostrati a schermo sotto determinati eventi di gioco. Essi rappresentano il menù di pausa, quello di completamente del livello e quello di Gam Over.

PauseOverlay (Implementazioni di MenusOverlayInterface):

La classe "PauseOverlay" costituisce un componente chiave all'interno del gioco, progettata per gestire lo schermo di pausa ed offrire all'utente un insieme di opzioni durante lapartita. Questo overlay fornisce un'interfaccia utente interattiva, presentando una serie di pulsanti per consentire all'utente di eseguire azioni specifiche mentre il gioco è temporaneamente in pausa.

All'interno di questa classe, vengono gestite diverse funzionalità, tra cui la visualizzazione di un'immagine di sfondo, la creazione e la gestione di pulsanti per operazioni come tornare al menu principale, ripetere il livello e riprendere il gioco. Inoltre, l'overlay offre un'opzione per regolare le impostazioni audio, consentendo all'utente di personalizzare l'esperienza sonora del gioco.

Attraverso l'utilizzo di eventi del mouse, la classe reagisce alle interazioni dell'utente, garantendo che le azioni desiderate vengano eseguite in risposta a clic, trascinamenti o rilasci del mouse. Si presenta in questo modo:



Variabili di ambient:

- **"BufferedImage backgroundImg"**: Immagine di sfondo per lo schermo di pausa.
- **"int bgX, bgY, bgWidth, bgHeight"**: Coordinate e dimensioni dell'immagine di sfondo.
- **"PRHButtons homeB, replayB, unpauseB"**: Oggetti che rappresentano i bottoni per le azioni di "Home", "Replay" e "Unpause".
- **"AudioOptions audioOptions"**: Oggetto per gestire le opzioni audio nella schermata di pausa.
- **"Playing playing"**: Riferimento all'istanza della classe "Playing" per accedere al game state e alle sue funzionalità.

Il costruttore accetta un'istanza della classe "Playing" e inizializza l'immagine di sfondo e i bottoni per "Home", "Replay" e "Unpause" button.

Metodi:

- **"createPRHButtons()"**: Crea e inizializza i bottoni per "Home", "Replay" e "Unpause" in posizioni specifiche.
- **"loadBackground()"**: Carica l'immagine di sfondo per lo schermo di pausa.
- **"update()"**: Chiama il metodo "update" per i bottoni e le opzioni audio.
- **"draw()"**: Disegna l'immagine di sfondo e i bottoni sulla schermata di pausa.
- **"mouseDragged(MouseEvent)"**: Aggiorna la posizione del mouse quando viene trascinato, utilizzato per gestire il trascinamento nelle opzioni audio.
- **"mousePressed(MouseEvent)"**: Gestisce l'evento di pressione del mouse, impostando i flag "mousePressed" per i bottoni e gestendo l'evento nelle opzioni audio.
- **"mouseReleased(MouseEvent)"**: Gestisce l'evento di rilascio del mouse, eseguendo azioni specifiche in base al bottone premuto o rilasciato e gestendo l'evento nelle opzioni audio.
- **"mouseMoved(MouseEvent)"**: Gestisce l'evento di movimento del mouse, impostando i flag "mouseOver" per i bottoni e gestendo l'evento nelle opzioni audio.

- **"mouseHovering(AbstractButtons, MouseEvent)"**: Metodo di supporto per verificare se il mouse è sopra un determinato bottone.

LevelComplitedOverlay (Implementazioni di MenusOverlayInterface):

La classe "LevelCompletedOverlay" rappresenta un overlay visualizzato quando un livello del gioco è completato con successo. Questo componente fornisce all'utente un'interfaccia per scegliere tra diverse opzioni, come passare al livello successivo o tornare al menu principale. Ecco come si presenta:

Variabili di ambient:

- **"playing":** Una istanza della classe "Playing" che rappresenta lo stato di gioco corrente.
- **"menuButton":** Un oggetto della classe "**PRHButtons**" che rappresenta il pulsante per tornare al menu principale.
- **"nextButton":** Un oggetto della classe "**PRHButtons**" che rappresenta il pulsante per passare al livello successivo.
- **"img":** Un'immagine rappresentante l'overlay visuale quando il livello è completato.
- **"bgX", "bgY", "bgWidth", "bgHeight":** Coordinate e dimensioni dell'immagine di sfondo.

Il costruttore accetta un oggetto "Playing" inizializza l'immagine di sfondo e i bottoni per "Home" "GoNext" button.

Metodi:

- **"initButtons()":** Inizializza i pulsanti del menu principale e del livello successivo.
- **"loadImages()":** Carica l'immagine di sfondo dell'overlay.
- **"draw(Graphics g)":** Disegna l'overlay con l'immagine di sfondo e i pulsanti.
- **"update()":** Aggiorna lo stato dei pulsanti.
- **"mouseHovering(AbstractButtons button, MouseEvent e)":** Verifica se il mouse sta passando sopra a un pulsante.
- **"mouseMoved(MouseEvent e)":** Gestisce gli eventi quando il mouse si muove sopra l'overlay.
- **"mousePressed(MouseEvent e)":** Gestisce l'evento di pressione del mouse, impostando i flag "mousePressed" per i bottoni e gestendo l'evento nelle opzioni audio.
- **"mouseReleased(MouseEvent e)":** Gestisce gli eventi quando il mouse viene rilasciato sopra l'overlay andando ad impostare lo stato dei bottono considerati.

GameOverOverlay (Implementazioni di MenusOverlayInterface):

La classe "GameOverOverlay" rappresenta l'overlay visuale che appare quando il giocatore perde una partita. Questo overlay fornisce all'utente opzioni come ritornare al menu principale o giocare nuovamente. Eccone l'esempio:

Attributi:

- **"playing":** Una istanza della classe "Playing" che rappresenta lo stato di gioco corrente.
- **"img":** Un'immagine che costituisce lo sfondo dell'overlay di game over.
- **"imgX", "imgY", "imgW", "imgH":** Coordinate e dimensioni dell'immagine di sfondo.
- **"menu", "play":** Oggetti della classe "PRHButtons" che rappresentano i pulsanti per tornare al menu principale e giocare nuovamente.

Il costruttore accetta un oggetto "Playing" inizializza l'immagine di sfondo e i bottoni per "menu" e "play" button.

Metodi:

- **"createButtons()":** Inizializza i pulsanti per il menu principale e per giocare nuovamente.
- **"loadImages()":** Carica l'immagine di sfondo dell'overlay di game over.
- **"draw()":** Disegna l'overlay di game over con sfondo, testo e pulsanti.
- **"keyPressed(KeyEvent)":** Gestisce gli eventi quando un tasto della tastiera viene premuto, in particolare il tasto "Escape" per tornare al menu principale.
- **"update()":** Aggiorna lo stato dei pulsanti.
- **"mouseHovering(AbstractButtons , MouseEvent)":** Verifica se il mouse sta passando sopra a un pulsante.
- **"mouseMoved(MouseEvent)":** Gestisce gli eventi quando il mouse si muove sopra l'overlay.
- **"mousePressed(MouseEvent)":** Verifica se il mouse sta premendo un pulsante per applicarne l'effetto di "pressed".
- **"mouseReleased(MouseEvent e)":** Verifica se il mouse sta premendo un pulsante per applicarne gli stati.
- **"mouseDragged(MouseEvent e)":** Gestisce gli eventi quando il mouse viene trascinato sopra l'overlay.

Con questo abbiamo finalmente finito con le classi di primo livello. Anche qui c'è voluto parecchio ma ci siamo riusciti. La bella notizia è che le classi di secondo livello sono soltanto 3, i nemici effettivi del gioco. Andiamo a vederle ...

Capitolo 2.3: Commento alle Classi di Secondo livello

In questo capitolo andremo ad osservare quelle che nel codice sono le:

- ***Classi di secondo livello***: sono quegli elementi del progetto che estendono o implementano una o più *classi di primo livello ed in aggiunta classi o interfaccie indipendenti.* Arrivati a questo tipo di solito si hanno gli oggetti concreti, i nodi effettivi dell'albero che possiedono i metodi Render ed Update implementati e funzionanti.

Esse sono:

1. NightBorne
2. HellBound
3. Ghost

NightBorne “Il Nato dalla Notte” (Estensione di AbstractEnemy):

Siamo giunti al primo nemico vero a proprio dopo questa lunga carrellata di classi sottoclassi nodi ed interfaccie.

Il ***NightBorne*** è stato il primo nemico ad essere aggiunto nel gioco, sul quale sono stati fatti i test ed è stata ideato il “**Behavioral loop**” dei nemici di cui abbiamo parlato durante la illustrazione della classe madre. Diamo uno sguardo al look:

In questi sprite osserviamo dei frame riguardanti lo stato di Idle, due frame dello stato di attacco, ed un frame dell’animazione di morte.

Con il NightBorne introduciamo una prima implementazione dell’idea di “Behavioral Loop” perchè alla fine i nemici sono questo, implementazioni singolari del Template Method richiesto dalla classe madre “AbstractEnemy”. Di fatto lei offre come ricordiamo tutti i metodi necessari a rendere il nemico responsivo e che verranno, come vedremo, utilizzati nel loop *MA*, lascia nella funzione “**act**” una blocco di codice da implementare che gestisce l’intelligenza propria del nemico.

Il **costruttore** solito richiede una posizione in coordinate di partenza (gestita dall’EnemyManager tramite il levelData), e richiama il costruttore della classe madre al quale vengono inviati elementi in ingresso aggiuntivi che sono definiti nella classe “Costants” vediamone un esempio una volta per tutte, valido per ogni nemico:

```
public NightBorne(float x, float y) {
        super(x, y, NIGHT_BORNE_WIDHT, NIGHT_BORNE_HEIGHT, NIGHT_BORNE);
        initHitbox(x, y, (int)(45 * Game.SCALE), (int)(40 * Game.SCALE));
        initAttackBox();
}
```

Vi è anche l’inizializzazione della “attackBox” e della “hitBox”.

Caratteristiche di Gameplay:

- **Coriaceità**: elevata, è duro da tirare giù.
- **Tipo di Movimento**: lineare, non può cadere nei burroni, è relativamente lento nei movimenti causa la sua pesante spada.
- **Tipo di Attacco**: “Fendente di luce”, il NightBorne infonde la sua spada di elettricità e luce lunare per poi scagliare un potente fendente capace di respingere il Player ad una breve distanza.
- **Debolezze**: è molto lento nella sua animazione di attacco, questo apre a delle finestre per scagliare un colpo.

La funzione “Update”:

Effettua un controllo sulla flag “**active**” e se questa è settata su true la funzione “**act()**” viene richiamata, il template method utilizzante la funzione da implementare “**makeMovement()**” (la logica del loop vera e propria). La funzione “**updateAttackBoxDirection()**” posiziona la hitbox del nemico in relazione alla sua direzione di movimento. In entrambi i valori della variabile “active” l’ “**updateAnimationTick()**” viene eseguita per la modifica dell’aniIndex e quindi del rendering dello sprite corretto.

Metodo “makeMovement()”:

Iniziamo col definire i 5 stati possibili del NightBorne:

1. ***IDLE***: stato in di partenza del nemico che comporta a questo lo stare fermo sul posto.
2. ***RUN***: stato di movimento in cui il NightBorne inizia a correre in una specifica direzione.
3. ***ATTACK***: stato in cui il NightBorne esegue l’animazione di attacco e si effettuano i controlli di collisione con la hitbox del Player
4. ***HURTED***: stato contrario a quello precedente in cui il NightBorne entra se il player l’ha appena colpito

5. ***DYING***: quando il player porta la vita di questo nemico a valori minori o uguali a 0, questo stato viene impostato cominciando l'animazione di morte.

Gli stati hanno valori interi partendo dallo 0 fino al 5, così da inviare questa variabile come indice di riga al metodo Render, che prenderà dalla matrice degli sprite la riga corrispondente all'attuale stato.

Tramite uno "switch" viene gestita la casistica in base allo stato:

- Caso "***IDLE***": viene automaticamente impostato lo stato su quello di "RUN".
- Caso "***RUN***": l'animationSpeed viene settata a 20 per rendere il movimento più lento (è inversamente proporzionale, più è alto il valore, più lenta sarà l'animazione) ed un controllo if richiama la funzione "**canSeePlayer()**" della classe madre, se questa restituisce vero verrà invocata la funzione "**turnTowardsPlayer()**" che porterà la direzione della camminata a rivolgersi verso quella del player se non si trova già in quello stato. Un controllo if interno invoca la funzione "**isPlayerCloseForAttack()**", nel caso ritorni vero, viene impostato lo stato su "***ATTACK***". In tutti quanti i casi, alla fine di questo stato viene richiamata la funzione "**move()**" che fa muovere il NightBorne nella sua attuale direzione di camminata.
- Caso "***ATTACK***": l'animationspeed viene decrementata a 13 e se l'animazione si trova nel decimo frame e se non è ancora stato effettuato il controllo dell'applicazione danno (Anche qui indicato dalla variabile "attackChecked" come nel caso del Player), la funzione "**checkEnemyHit()**" viene invocata.
- Caso "***HURTED***": l' "**updateAnimationTick()**" farà incrementare l'animation index fino al concludersi dell'animazione per poi impostare lo stato su "***IDLE***".
- Caso "***DYING***": l' "**updateAnimationTick()**" farà incrementare l'animation index fino al concludersi dell'animazione per poi impostare la flag "**active**" su false. **Questa assegnazione porterà ad inviare un segnale all'EnemyManager per il quale questo nodo smetterà di eseguire l'update e la render su qesta specifica istanza di questo NightBorne appena sconfitto.**

HellBound “Il cane infernale” (Estensione di AbstractEnemy):

L’HellBound è un demone di fuoco molto aggressivo e difficile da gestire. Come pre il precedente osserviamone degli sprite:

Le animazioni corrispondono al salto, corsa, “sliding” e morte.

Caratteristiche di Gameplay:

- **Coriaceità**: bassa, questo tipo di nemico è molto facile da sconfiggere, bilanciamento dovuto al suo tipo di attacco.
- **Tipo Di Movimento**: lineare, inizialmente passeggia sereno ma entra in allerta non appena il player gli si avvicina abbastanza, inizierà di fatto a correre verso di lui.
- **Tipo di Attacco**: “Istinto Animale”, dopo aver puntato la sua preda ed aver iniziato a correre verso il Player, l’HellBound eseguirà un balzo verso di esso nel tentativo di ferirlo e fidatevi “*ci riuscirà, e farà male*”. Nel caso dovesse copire tenete ben presente che scotta, per un periodo potreste subire ustioni.
- **Debolezze**: una volta atterrato dopo un balzo l’HellBound sarà costretto a rallentare la sua corsa e a riposizionarsi affondando gli artigli nel terreno ed effettuando una scivolata per qualche tempo. Non riuscirà a cadere dai burroni perchè molto stabile sulle zampe anteriori ma ciò non significa che non possa farlo, infatti se dietro il Player vi troviamo un fossato, al momento del salto l’HellBound potrebbe anche rischiare di cadere e farsi molto male...

Variabili di Ambiente:

L’HellBound a differenza del precedente nemico possiede alcune variabili proprie per la gestione di determinate meccaniche.

- “jumpForce”: la variabile che simula la gravità e che serve alla meccanica del salto
- “orizzontalSpeed”: quella che definisce la velocità si spostamento orizzontale del salto
- “slidingSpeed”: la velocità dello “slide” quando atterra dopo un salto
- “slideDistanceTravelled”: una variabile che definisce la distanza massima di “slide”
- “jumping”: flag indicatrice dello stato di salto.

Il **costruttore** è uguale a quello del NightBorne.

La funzione “Update”:

Un pò come accadeva al “Nato dalla Notte” **la funzione Update effettua un controllo sulla flag “active”** per controllare che il nemico sia attivo, il metodo “**act()**” viene richiamato e verrà eseguita la funzione template “m**akeMovement()” anche qui ovviamente implementata.** Viene updatata la direzione della attackBox con la “**updateAttackBoxDirection()**” dopodichè viene fatto un controlo particolare. *Se il cangnetto si trova nello stato di salto (e quindi la flag “jumping” è attiva) se l’indice di animazione è uguale a 4 e lo stato non è quello di morte “**DIE**”* la “updateAnimationTick()” che conosciamo bene viene saltata. Il motivo è molto semplice, se mai l’animazione di salto dovesse finire, lo stato verrebbe subito impostato a quello di “***SLIDE***” mentre il cane ancora si trova in aria, portando ad una inevitabile catastrofe di glitch e bug differenti.

Metodo “makeMovement()”:

Come prima andiamo a definire gli stati di questo nemico:

- ***WALK***: stato di partenza del nemico dal quale inizia il “Behavioral Loop”.
- ***RUN***: Stato di corsa in cui enntriamo se il cane vede il player.
- ***JUMP***: praticamente lo stato di attacco.
- ***SLIDE***: stato successivo a quello di attacco in cui il cane scivola sul terreno.
- ***HIT***: stato che definisce quando queto nemico viene colpito.
- ***DIE***: stato di morte.

Gli stati hanno valori interi partendo dallo 0 fino al 5, così da inviare questa variabile come indice di riga al metodo Render, che prenderà dalla matrice degli sprite la riga corrispondente all’attuale stato.

Tramite uno “switch” viene gestita la casistica in base allo stato:

- Caso “***WALK***”: la velocità di movimento viene impostata a 0.4 per rendere il nemico molto lento, anche l’aniSpeed viene aumentata per rallentare l’animazione. “***If (canSeePlayer())***” la funzione “**turnTowardsThePlayer()**” viene invocata così come l’assegnazione dello stato su quello di “*RUN*”. In tutti quanti i casi la funzione “**move()**” dalla classe madre viene invocata per aggiornare la posizione X.
- Caso “***RUN***”: la velocità di movimento viene quasi quintuplicata rendendo questo nemico il più veloce tra i nemici normali aumentandone di tanto la pericolosità. “***if(isPlayerCloseForAtack())***” lo stato viene impostato su quello di “*JUMP*” altrimenti la funzione “**move()**” viene di nuovo invocata eseguendo però uno spostamento più rapido basandosi sulla “walkSpeed” qui diverse volte modificata.
- Caso “***JUMP***”: il caso più complicato di tutti. La flag “**jumping**” viene impostata a true e viene invocata la funzione peculiare “***jumpAttack()***” per la quale questo stato assume l’accezione di complicato. Con la stessa logica di “**attackChecked**” usata già in precedenza il metodo “**checkAttack()**” che già conosciamo viene richiamato. In questo caso il cambio di stato non avviene direttamente all’interno ma nella funzine “jumpAttack()” di cui discuteremo a breve.
- Caso “***SLIDE***” al quale ci si arriva dalla funzione “jumpAttack()” invoca semplicemente un’altra funzione peculiare “**slide()**” della quale ancora, discuteremo a breve. Anch'essa gestirà il cambio di stato di questo caso.
- Caso “HIT”: gestisce alcuni controlli atti solo all’evitare la presenza di alcuni bug.
- Caso “DIE”: viene rallentata l’ ”aniSpeed” per rendere l’animazione un pochino più lenta delle altre.

Funzione “JumpAttack()”:

Questo metodo permette all'hellbound di saltare per attaccare il player.

Vengono fatti due calcoli in modo parallelo, **si aggiorna la componente orizzontale e quella verticale**. Per la direzione destra o sinistra viene fatto un controllo sul possibile movimento sfruttando la variabile “**orizzontalSpeed**” per spostare l’entità nella direzione di movimento, viene aggiunta una componente alla coordinata se non ci sono muri in traiettoria, controllo eseguito grazie alla funzione “**canMoveHere()**” proveniente dalla classe “**HelpMethods**”. **In caso contrario il cagnetto darà una bella testata al muro**, eppure non può di certo continuare a volare, la componente in Y di fatto continuerà ad essere aggiornata fin qando il terreno sotto i suoi piedi viene percepito come aria (la variabile per effettuare lo spostamento verticale è la “jumpForce” di cui abbiamo parlato prima che viene aggiunta alla component Y se il movimento è permesso. Essa aumenta di volta in volta aggiungendole il valore della gravità del gioco, ricordiamo anche che è inversamente proporzionale). **Fatti i controlli per le ordinate viene sempre effettuato un return se il movimento vi è stato permesso**, questo per continuare ad aggiornare il moto parabolico del nemico. **A momento in cui questo controllo fallisce** si prosegue per la parte successiva di codice in cui la jump force viene resettata al suo valore di

partenza,un veloce richiamo alla funzione “**updateInAir()**” viene seguito per posizinare il cane a livello del terreno e se l’entità è finalmente atterrata viene cambiato lo stato in “***SLIDE***” triggerando la prossima funzione.

Metodo “Slide()”:

Una variabile booleana “maxDistance” viene dichiarata a falso.

Due controlli uguali effettuano dei controlli su movimento direzionale verso destra e verso sinistra sfruttando la funzione “**isSolid()**” dalla classe “**HelpMethods**” per osservare de il movimento in quella direzine è permesso, ciò richiamerebbe la funzione propria “**moveSlide()”, proprietaria anch’essa**, della quale assiociamo il sisultato alla variabile “**maxDistance**”.

Un controllo su di questa viene effetuata e se risulta essere settata su true, le variabili di “slidingDistance” e “slidingSpeed” vengono resettate ai valori di partenza, la direzione di movimento viene cambiata ed un **nuovo stato** viene applicato, quello di “***WAK***”.

Metodo “MoveSlide()”:

Questa viene invocata dalla funzione precedente e gestisce il movimneto verso la direzione indicata. Ancora usando la funzione “**canMoveHere()**”, viene osservata la possibilità del movimento previsto, se risulta possibile il movimento viene applicato. La variabile “**slidingSpeed**” **viene diminuita** di una certa quantità costante (potremmo definirlo un coefficiente di attrito) mentre la “**slideDistanceTravelled**” **viene aumentata** del valore assoluto della quantià di spostamento applicata in questa iterazione. **Momento in cui la distanza di slide è maggiore di 4 blocchi oppure se la velocità di slide è diminuita sotto una certa quantità viene ritornato vero** e segnalato alla funzione precedente che la distanza massima di slide è stata applicata. **Se questo return della flag true non viene eseguito viene ritornato falso**, la variabile “maxDistance” di cui prima rimane sullo stato di falso **ed il nuovo stato di “WALK” non verrà di conseguenza applicato.**

Altri Metodi:

Questa classe possiede ovviamente altri metodi classici come la “updateAttackBoxDirection()” che imposta la nuova posizione della attackbox in base alla direzione del movimento.

La funzione “hurt()” che applica il danno inviato in ingresso al nemico.

L’implementazione delle funzioni “flipX()” e “flipW”.

Ghost “Padrone dei Ghiacci” (Estensione di AbstractEnemy):

Il Ghost è una entità eterea capace di controllare la temperatura attorno a sé generando una ondata di gelo imporvvisa e letale. Osserviamone l’aspetto:

Animaizione di: Idle, teletrasporto, danno e morte.

Caratteristiche di Gameplay:

- **Coriacetà**: il “Ghost” non è un tipo di nemico che è difficile da abbattere, ma non è neanche così debole poichè la sua essenza ectoplasmatica gli permette di subire poco i danni. Una via di mezzo diciamo.
- **Tipo di movimento**: il ghost ha la capacità di teletrasportarsi. Esso infatti non si muove ma studia il terreno circostante anche tenendo conto dei dislivelli per teletrasportarsi in una nuova posizione dopo aver attaccato. All’inizio della partita inoltre i Ghost sono dormienti e nascosti alla vista del Player, questo non li vedrà fin quando non sarà troppo tardi ...
- **Tipo di Attacco**: “Tempesta di gelo”. Il Ghost concentra le sue energie per generare un’area attorno a lui in cui la temperatura per pochi istanti raggiunge temperature bassissime, causando danno e rallentando i movimenti del Player se questo si trova al suo interno. *Attenzione, **al ghost basta essere abbastanza vicino per attaccare ed i muri non lo fermeranno, il danno verrà applicato anche dietro di questi.***
- **Debolezze**: Il Ghost non attacca se il player non si trova nel suo raggio di azione e per subire danno il questo deve trovarsi nella tempesta quando inizia ad essere abbastanza grande e carica. Si può pertanto spingere il Ghost ad attaccare avvicinandosi per poi scappare nella direzione opposta. Questo dovrà poi ricaricare le sue energie prima di riattaccare aprendo a finestre molto lunghe ed in cui è parecchio vulnerabile. Ricordiamo però che si teletrasporta poco tempo dopo un attaco, siate quindi veloci e letali.

Variabili di ambiente:

Come l’HellBound il Ghost possiede alcune variabili di ambiente utili a gestire le sue meccaniche:

- “attackBoxOffset”: un intero che serve a riposizionare la attackbox del ghost centrata su di lui, dovendo attaccare in una zona circolare questa variabile conta come raggio della circonferenza.
- “teleportTimer, timeToTeleport e attackTimer” sono 2 variabili che gestiscono per quanto riguarda le prime due, il tempo del teletrasporto, e per quanto riguarda la seconda il tempo tra un attacco e l’altro.
- “spawnPoints” è una lista di “Point2D” (classe nativa di Java) che rappresenta set di coortinate spaziali. Servirà per contenere tutti i punti di teletrasporto possibili.
- “canTeleport” è la variabile booleana che indica se può effettuare un teletrasporto.
- “firstSpawn” indica se il ghost non è ancora spawnato e quindi è ancora nascosto nell’ombra in attesa che il Player sia abbastanza vicino. Potrbbe rappresentare lo stato di “è inattivo ma non è morto”.
- “random”, una istanza della classe “Random” del java che useremo per scegliere punti randomici di spawn.

Il costruttore è uguale a quello solito dei nemici con l’unica differenza che la lista di punti viene inizializzata ad una nuova lista.

La funzione “Update”:

La funzine update del Ghost esegue 4 funzini se la flag “active” è impostata su true.

Come prima cosa incova la funzione “act()”, richiamante poi il Template Method “makeMovement()” che studieremo. Viene fatto un update della attackbox per riposizionarla con il suo centro suquello del Ghost, e le funzioni “flipWP()”

“flipXP()” eseguono un update dei moltiplicatori per la funzione Render in base alla posizione corrente del Player, per far girare lo sguardo del Ghost verso la sua direzione. L’ “updateAnimationTick()” viene richiamata a prescindere dall’*if.*

Metodo “makeMovement()”:

Come preannunciavamo il metodo “makeMovement()” è molto peculiare, non fa utilizzo della funzinoe della classe madre “move()” di cui abbiamo parlato ed osservato degli usi. Ma utiilzza una funzione propria chiamata “teleport()”.

Iniziamo con l’analizzare gli stati possibili:

- ***NOT_SPAWNED:*** stato in cui il Ghost è dormiente e non interagisce con niente, in attesa che il player si avvicini.
- ***SPAWN***: è praticamente lo stato che gestisce il teletrasporto al momento di comparsa in azione.
- ***TELEPORT***: stato in cui il Ghost si teletrasporta.
- ***IDLE***: il ghost rimane in attesa che il player si avvicini per attaccare con la sua tempesta, oppure attende del tempo prima di teletrasportarsi in una nuova posizione.
- ***ATTACK*** è lo stato in cui si gestisce l’attacco del nemico.
- ***HIT*** rappresenta lo stato dopo essere stato colpito dal Player.
- ***DIE*** quello di morte.

Gli stati hanno valori interi partendo dallo 0 fino al 5, così da inviare questa variabile come indice di riga al metodo Render, che prenderà dalla matrice degli sprite la riga corrispondente all’attuale stato.

Tramite uno “switch” viene gestita la casistica in base allo stato:

- Caso “***NOT_SPAWNED***”: questo caso come abbiamo detto porta il ghost a non agire e a rimanere invisibile. La funzione “**updateAnimazionTick()**” nel caso il nemico sia un Ghost di fatto effettua semplicemente dei return e continua ad impostare la flag “firstSpawn” su falso. Nel caso intuibile ed esclusivo in cui il PLayer “isInRangeOfVision()” il nuovo stato verrà impostato su “***SPAWN***”.
- Caso ”***SPAWN***”: se la flag “**canTeleport**” risulta **vera** e “**firstSpawn**” invece **falsa** (questo per impedire che il ghost effettui la prossima azione al primo spawn) viene invocata la funzione propria “teleport()”. Che gestirà il cambio di stato ed il movimento nella mappa di questo nemico.
- Caso “***IDLE***”: è lo stato in cui il ghost, dopo essere spawnato o dopo essersi teletrasportato, finisce. Incrementa ad ogni iterazione l’ “**attackTimer**”, posiziona la “**aniSpeed**” a 20 tick Di aggiornamento, l’invulnerabilità che quì gica un ruolo importante, viene impostata su falso così come la logica di attacco con la flag “**attackChecked**”. La funzione proprietaria “**updateTeleportTick()**” viene invocata. Un ulteriore controllo avviene successivamente per controllare che l’ “**attackTimer**” abbia misurato abbastanza tempo di attesa per eseguire uno nuovo e che il player sia abbastanza vicino per essere attaccato, se queste condizioni dovessero essere vere un nuovo stato viene impostato su “***ATTACK***” e l’attackTimer viene resettato.
- Caso “***ATTACK***”: viene comunque aggiornato il tempo per il teletrasport con la “**updateTeleportTick()**”, per tutto lo stato di attacco viene eseguito un controllo con la funzione “**checkEnemyHitEllipse()**” (uguale alla funzione “**checkEnemyHit()**” ma per i paramteri di aree geometriche ellittiche”.
- Caso ”***HIT***” e “***DIE***” impostano l’ “aniSpeed” ai loro valori desiderati e gestiscono un loop singolo di animzione.

Metodo “teleport()”:

Il metodo teleport è molto semplice, quando invocato richiama a sua volta una funzione che dopo osserveremo, “findTeleportPoints()” che seleziona in una area di 20 blocchetti per 20 con centro l’attuale posizione del ghost, i punti di teleport possibili, genera quindi un nuovo numero randomico e seleziona dalla lista di punti di spawn possibili da quella posizione “spawnPoints” la posizione randomica scelta andando ad impostare le coordinate della “hitBox” a quelle del punto così scelto. L’array di “spawnPoints” viene poi svuotato.

Metodo “findTeleportPoints()”:

Questo metodo trova i punti possibili di spawn in una area 20 x 20 attorno al ghost.
Inizialmente converte le attuali coordinate del ghost in relazione alla grandezza del “levelData” in questo modo:

```
int Xposition = (int)(hitbox.x / Game.TILES_SIZE);
int Yposition = (int)(hitbox.y / Game.TILES_SIZE);
```

Caocola poi i punti di inizio e fine area sulla base di quest:

```
int Xstart = Xposition - 10;
int Ystart = Yposition - 10;

int Xend = Xposition + 10;
int Yend = Yposition + 10;
```

E li regola se superano i bordi della mappa in modo da non dare una area impossibile da ragiungere.

Con due cicli for innestati la matrice del levelData viene analizzata e se un blocco di terreno possiede al di sopra di esso due blocchi di aria (viene fatto largo uso della funzione “isSolid()” della classe “HelpMethods”) questo set di coordinate viene riscalata in relazione a quelle di gioco e vengono aggiunte sotto forma di punto 2D alla lista di possibili punti di spawn usata dal metodo precedente “teleport()”.

Metodo “updateTeleportTick()”:

Gestisce il reset tel tempo per il teletrasporto, andandolo a resettare a 0 se questo diventa magiore del “**timeToTeleport**”. Vi è una caratteristica agiunta secondo la quale il “timeToTeleport” viene riassegnato ad un nuovo intero tra 700 e 1000, così da reglare al ghost una sorta di randomicità nel movimento.

Un controllo viene effettuato. Se il ghost non si trova in stato di attacco allora lo stato viene settato su quello di “**TELEPORT**”, non presente nella funzione “**makeMovement()**” e che serve solo per gestire l’animazione di teletrasporto della Render. *Piccola osservazione:* ***la variabile di immortalità “invulnerability” viene settata a true quano il ghost inizia il teletrasporto, rendendolo pertanto invulnerabile durante questa animazione****. Ritornerà ad essere bersagliabile una volta finita l’animazione di “****SPAWN****”*

Altri Metodi ed “@Overriding”:

Due altri metodi sono “**isPlayerCloseForAttack()**” che **viene aggiornata in modo da calcolare se il player si trovi nel *raggio* di attacco in base a**l raggio de**lla circoferenza che descrive questa area** di attacco, ed ***“isPlayerInRangeOfVision()” che a differenza di tutti gli altri nemici effettua solo calcoli sulla base della distanza dal Player ignorando tutti i possibili muri burroni oppure ostacoli che si possono trovare nel percorso*** triggherando l’attacco anche da dietro i muri.

Conclusioni Capitolo 2 “Commento alle Classi”:

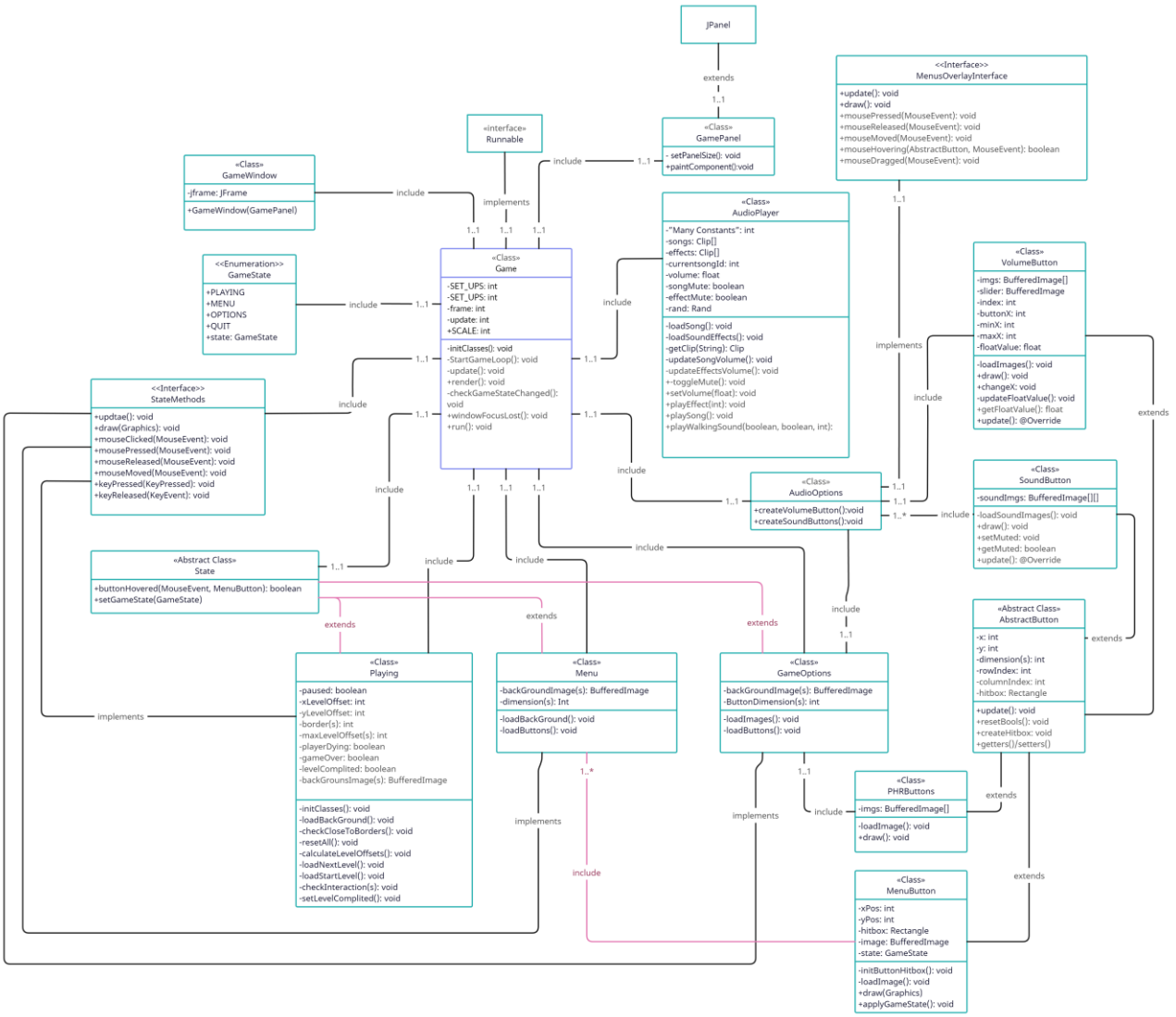
Il viaggio alla scoperta delle classi si conclude qui, ci è voluto tanto, è stato complesso, ma abbiamo descritto il gioco sotto ogni suo punto di vista base, illustrando tutte le coponenti base avanzate e non che compongono le classi ed i nodi di WOP, aggiungendo anche illustrazione per mostrare le componenti del mondo di gioco eppure, il racconto non è ancora finito.

Dovremmo ora proseguire sotto punti di vista molto più tecnici ai fini di illustrare componenti come i “Design Patter” scelti ed implementati per gestire componenti di gioco e l’illustrazione del diagramma UML delle classi. Dopodichè potremmo definirci esperti programmatori di giochi creati da 0, e senza utilizzo di framwork, in Java.

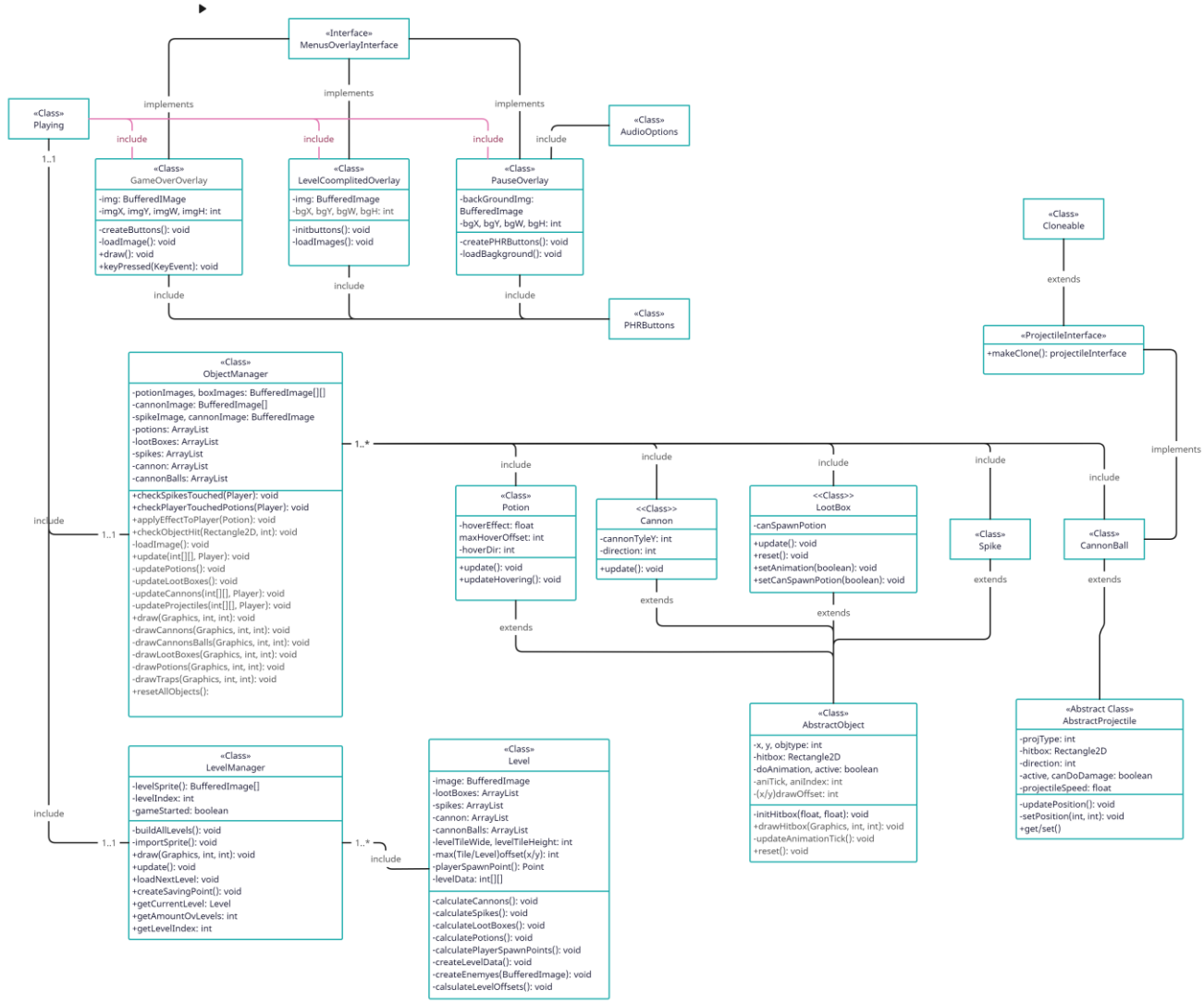
Capitolo 3: Diagramma UML delle classi

In questo capitolo verrà semplicemente illustrato il diagramma delle classi descritte nel capitolo precedente. Tenendo in considerazione il numero delle classi in gioco, il diagramma verrà diviso in più immagini per facilitarne la lettura, ogni qual volta ci sarà bisogno di riferirsi ad una classe che non è presente nella attuale immagine questa sarà sostituita da una sua “mini-classe” o “sample”.

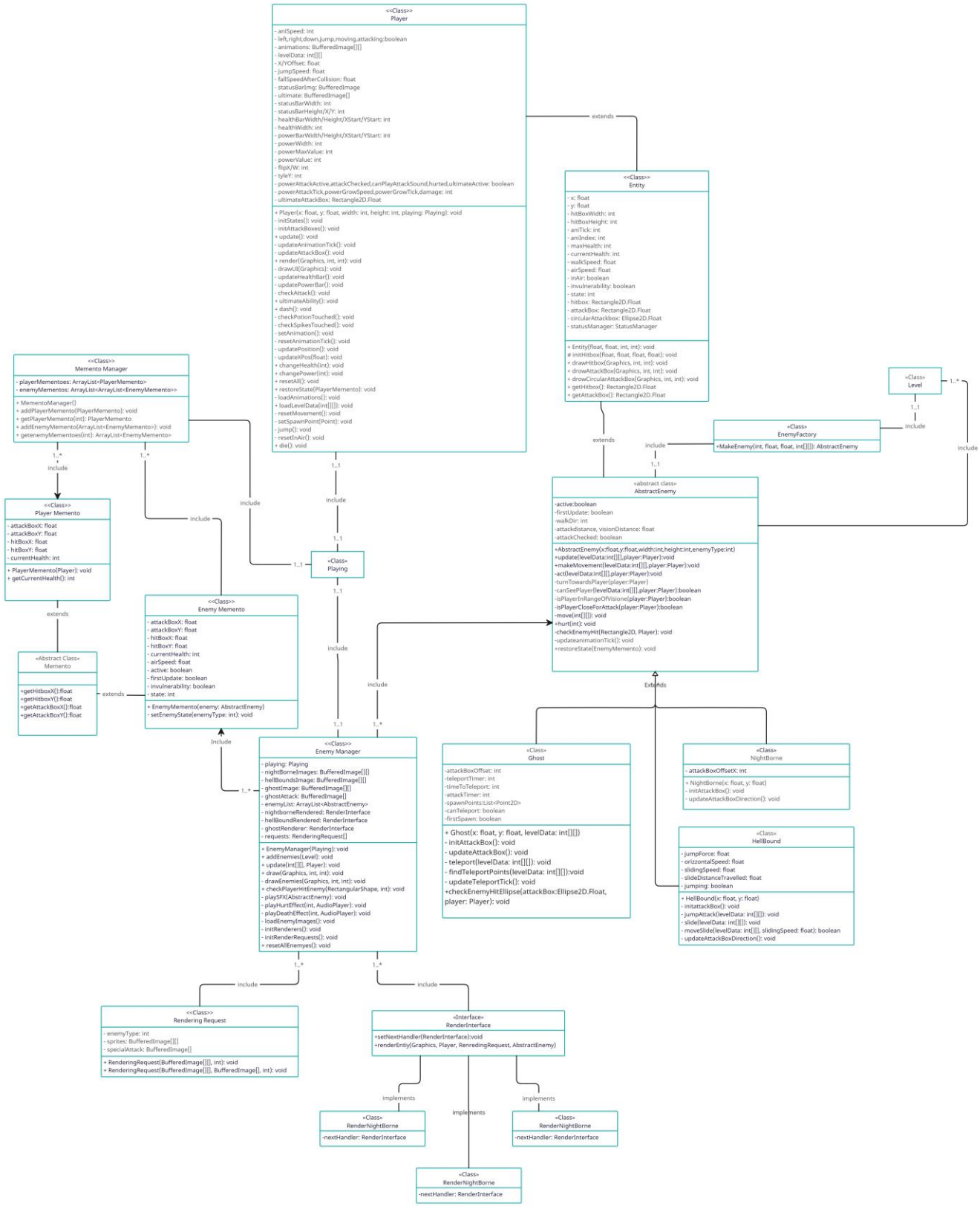
Parte 1:



Parte 2:


«Interface»
MenusOverlayInterface
implements
implements
implements
«Class»
Playing
include
include
include
include
«Class»
AudioOptions
1..1
«Class»
GameOverOverlay
-img: BufferedIMage
-imgX, imgY, imgW, imgH: int
-createButtons(): void
-loadImage(): void
+draw(): void
+keyPressed(KeyEvent): void
«Class»
LevelCoomplitedOverlay
-img: BufferedImage
-bgX, bgY, bgW, bgH: int
-initbuttons(): void
-loadImages(): void
«Class»
PauseOverlay
-backGroundImg: BufferedImage
-bgX, bgY, bgW, bgH: int
-createPHRButtons(): void
-loadBagkground(): void
include
include
include
«Class»
PHRButtons
«Class»
Cloneable
extends
«ProjectileInterface»
+makeClone(): projectileInterface
implements
«Class»
ObjectManager
-potionImages, boxImages: BufferedImage[][]
-cannonImage: BufferedImage[]
-spikeImage, cannonImage: BufferedImage
-potions: ArrayList
-lootBoxes: ArrayList
-spikes: ArrayList
-cannon: ArrayList
-cannonBalls: ArrayList
+checkSpikesTouched(Player): void
+checkPlayerTouchedPotions(Player): void
+applyEffectToPlayer(Potion): void
+checkObjectHit(Rectangle2D, int): void
-loadImage(): void
+update(int[][], Player): void
-updatePotions(): void
-updateLootBoxes(): void
-updateCannons(int[][], Player): void
-updateProjectiles(int[][], Player): void
+draw(Graphics, int, int): void
-drawCannons(Graphics, int, int): void
-drawCannonsBalls(Graphics, int, int): void
-drawLootBoxes(Graphics, int, int): void
-drawPotions(Graphics, int, int): void
-drawTraps(Graphics, int, int): void
+resetAllObjects():
1..*
include
include
include
include
include
include
include
1..1
«Class»
Potion
-hoverEffect: float
maxHoverOffset: int
-hoverDir: int
+update(): void
+updateHovering(): void
<<Class>>
Cannon
-cannonTyleY: int
-direction: int
+update(): void
<<Class>>
LootBox
-canSpawnPotion
+update(): void
+reset(): void
+setAnimation(boolean): void
+setCanSpawnPotion(boolean): void
«Class»
Spike
«Class»
CannonBall
extends
extends
extends
extends
extends
«Class»
AbstractObject
-x, y, objtype: int
-hitbox: Rectangle2D
-doAnimation, active: boolean
-aniTick, aniIndex: int
-(x/y)drawOffset: int
-initHitbox(float, float): void
+drawHitbox(Graphics, int, int): void
-updateAnimationTick(): void
+reset(): void
«Abstract Class»
AbstractProjectile
-projType: int
-hitbox: Rectangle2D
-direction: int
-active, canDoDamage: boolean
-projectileSpeed: float
-updatePosition(): void
-setPosition(int, int): void
+get/set()
include
1..1
«Class»
LevelManager
-levelSprite(): BufferedImage[]
-levelIndex: int
-gameStarted: boolean
-buildAllLevels(): void
-importSprite(): void
+draw(Graphics, int, int): void
+update(): void
+loadNextLevel: void
+createSavingPoint(): void
+getCurrentLevel: Level
+getAmountOvLevels: int
+getLevelIndex: int
1..*
include
«Class»
Level
-image: BufferedImage
-lootBoxes: ArrayList
-spikes: ArrayList
-cannon: ArrayList
-cannonBalls: ArrayList
-levelTileWide, levelTileHeight: int
-max(Tile/Level)offset(x/y): int
-playerSpawnPoint(): Point
-levelData: int[][]
-calculateCannons(): void
-calculateSpikes(): void
-calculateLootBoxes(): void
-calculatePotions(): void
-calculatePlayerSpawnPoints(): void
-createLevelData(): void
-createEnemyes(BufferedImage): void
-calsulateLevelOffsets(): void

Parte 2:


<<Class>>
Player
- aniSpeed: int
- left,right,down,jump,moving,attacking:boolean
- animations: BufferedImage[][]
- levelData: int[][]
- X/YOffset: float
- jumpSpeed: float
- fallSpeedAfterCollision: float
- statusBarImg: BufferedImage
- ultimate: BufferedImage[]
- statusBarWidth: int
- statusBarHeight/X/Y: int
- healthBarWidth/Height/XStart/YStart: int
- healthWidth: int
- powerBarWidth/Height/XStart/YStart: int
- powerWidth: int
- powerMaxValue: int
- powerValue: int
- flipX/W: int
- tyleY: int
- powerAttackActive,attackChecked,canPlayAttackSound,hurted,ultimateActive: boolean
- powerAttackTick,powerGrowSpeed,powerGrowTick,damage: int
- ultimateAttackBox: Rectangle2D.Float
+ Player(x: float, y: float, width: int, height: int, playing: Playing): void
- initStates(): void
- initAttackBoxes(): void
+ update(): void
- updateAnimationTick(): void
- updateAttackBox(): void
+ render(Graphics, int, int): void
- drawUI(Graphics): void
- updateHealthBar(): void
- updatePowerBar(): void
- checkAttack(): void
+ ultimateAbility(): void
+ dash(): void
- checkPotionTouched(): void
- checkSpikesTouched(): void
- setAnimation(): void
- resetAnimationTick(): void
- updatePosition(): void
- updateXPos(float): void
+ changeHealth(int): void
+ changePower(int): void
+ resetAll(): void
+ restoreState(PlayerMemento): void
- loadAnimations(): void
+ loadLevelData(int[][]): void
- resetMovement(): void
- setSpawnPoint(Point): void
- jump(): void
- resetInAir(): void
+ die(): void
extends
<<Class>>
Entity
- x: float
- y: float
- hitBoxWidth: int
- hitBoxHeight: int
- aniTick: int
- aniIndex: int
- maxHealth: int
- currentHealth: int
- walkSpeed: float
- airSpeed: float
- inAir: boolean
- invulnerability: boolean
- state: int
- hitbox: Rectangle2D.Float
- attackBox: Rectangle2D.Float
- circularAttackbox: Ellipse2D.Float
- statusManager: StatusManager
+ Entity(float, float, int, int): void
# initHitbox(float, float, float, float): void
+ drawHitbox(Graphics, int, int): void
+ drowAttackBox(Graphics, int, int): void
+ drowCircularAttackBox(Graphics, int, int): void
+ getHitbox(): Rectangle2D.Float
+ getAttackBox(): Rectangle2D.Float
<<Class>>
Memento Manager
- playerMementoes: ArrayList<PlayerMemento>
- enemyMementos: ArrayList<ArrayList<EnemyMemento>>
+ MementoManager()
+ addPlayerMemento(PlayerMemento): void
+ getPlayerMemento(int): PlayerMemento
+ addEnemyMemento(ArrayList<EnemyMemento>): void
+ getenemyMementoes(int): ArrayList<EnemyMemento>
«Class»
Level
«Class»
EnemyFactory
+MakeEnemy(int, float, float, int[][]): AbstractEnemy
«abstract class»
AbstractEnemy
-active:boolean
-firstUpdate: boolean
-walkDir: int
-attackdistance, visionDistance: float
-attackChecked: boolean
+AbstractEnemy(x:float,y:float,width:int,height:int,enemyType:int)
+update(levelData:int[][],player:Player):void
+makeMovement(levelData:int[][],player:Player):void
-act(levelData:int[][],player:Player):void
-turnTowardsPlayer(player:Player)
-canSeePlayer(levelData:int[][],player:Player):boolean
-isPlayerInRangeOfVisione(player:Player):boolean
-isPlayerCloseForAttack(player:Player):boolean
-move(int[][]): void
+hurt(int): void
-checkEnemyHit(Rectangle2D, Player): void
-updateanimationTick(): void
+restoreState(EnemyMemento): void
<<Class>>
Player Memento
- attackBoxX: float
- attackBoxY: float
- hitBoxX: float
- hitBoxY: float
- currentHealth: int
+ PlayerMemento(Player): void
+ getCurrentHealth(): int
«Class»
Playing
<<Class>>
Enemy Memento
- attackBoxX: float
- attackBoxY: float
- hitBoxX: float
- hitBoxY: float
- currentHealth: int
- airSpeed: float
- active: boolean
- firstUpdate: boolean
- invulnerability: boolean
- state: int
+ EnemyMemento(enemy: AbstractEnemy)
- setEnemyState(enemyType: int): void
«Abstract Class»
Memento
+getHitboxX():float
+getHitboxY():float
+getAttackBoxX():float
+getAttackBoxY():float
<<Class>>
Enemy Manager
- playing: Playing
- nightBorneImages: BufferedImage[][]
- hellBoundsImage: BufferedImage[][]
- ghostImage: BufferedImage[][]
- ghostAttack: BufferedImage[]
- enemyList: ArrayList<AbstractEnemy>
- nightborneRendered: RenderInterface
- hellBoundRendered: RenderInterface
- ghostRenderer: RenderInterface
- requests: RenderingRequest[]
+ EnemyManager(Playing): void
+ addEnemies(Level): void
+ update(int[][], Player): void
+ draw(Graphics, int, int): void
- drawEnemies(Graphics, int, int): void
+ checkPlayerHitEnemy(RectangularShape, int): void
- playSFX(AbstractEnemy): void
- playHurtEffect(int, AudioPlayer): void
- playDeathEffect(int, AudioPlayer): void
- loadEnemyImages(): void
- initRenderers(): void
- initRenderRequests(): void
+ resetAllEnemyes(): void
«Class»
Ghost
-attackBoxOffset: int
-teleportTimer: int
-timeToTeleport: int
-attackTimer: int
-spawnPoints:List<Point2D>
-canTeleport: boolean
-firstSpawn: boolean
+ Ghost(x: float, y: float, levelData: int[][])
- initAttackBox(): void
- updateAttackBox(): void
- teleport(levelData: int[][]): void
- findTeleportPoints(levelData: int[][]):void
- updateTeleportTick(): void
+checkEnemyHitEllipse(attackBox:Ellipse2D.Float, player: Player): void
«Class»
NightBorne
- attackBoxOffsetX: int
+ NightBorne(x: float, y: float)
- initAttackBox(): void
- updateAttackBoxDirection(): void
«Class»
HellBound
- jumpForce: float
- orizzontalSpeed: float
- slidingSpeed: float
- slideDistanceTravelled: float
- jumping: boolean
+ HellBound(x: float, y: float)
- initattackBox(): void
- jumpAttack(levelData: int[][]): void
- slide(levelData: int[][]): void
- moveSlide(levelData: int[][], slidingSpeed: float): boolean
- updateAttackBoxDirection(): void
<<Class>>
Rendering Request
- enemyType: int
- sprites: BufferedImage[][]
- specialAttack: BufferedImage[]
+ RenderingRequest(BufferedImage[][], int): void
+ RenderingRequest(BufferedImage[][], BufferedImage[], int): void
«Interface»
RenderInterface
+setNextHandler(RenderInterface):void
+renderEntity(Graphics, Player, RenredingRequest, AbstractEnemy)
«Class»
RenderNightBorne
-nextHandler: RenderInterface
«Class»
RenderNightBorne
-nextHandler: RenderInterface
«Class»
RenderNightBorne
-nextHandler: RenderInterface
extends
include
implements
1..1
1..*

Osservazione importante:

Alcune classi potrebbero non essere incluse all'interno del diagramma poiché di largo uso all'interno di una moltitudine di classi sotto forma di richiami statici "Static" oppure perché stand alone o ancora potrebbe trattarsi di classi peculiari dei design pattern. Queste classi sono:

- Constants.
- HelpMethods.
- LoadSave.
- KeyboardInputs, di cui non abbiamo parlato. Questa è una classe che implementa l'ascolto dei tasti da tastiera e che manda i segnali captati ai vari Stati del gioco.
- MouseInputs, che, come la precedente, non è mai stata nominata e che svolge lo stesso compito di ascolto e ricezione dei segnali ma ovviamente, dal mouse e non dalla tastiera.

Capitolo 4: Illustrazione dei Design Patter usati

Nel seguente capitolo verranno trattati i design pattern utilizzati per la realizzazione del progetto. I design pattern implementati sono:

- **Factory Method**
- **Chain of Responsability**
- **State**
- **Memento**
- **Template Methods**
- **Prototype**

Factory Method:

IL factory method coinvolge le classi Enemy Factory, AbstractEnemy, Level, HellBound, Night Born e Ghost.

È utilizzato per rendere la creazione ed aggiunta di nuovi nemici nel modo più semplice possibile. L'aggiunta di altri nemici del gioco avviene descrivendo nuove classi che estendono la classe "AbstractEnemy" ed aggiungendo questa come caso per la classe "£nemyFactrory".

L'Enemy Factory come si può dedurre dal nome funge da fabbrica fornendo il metodo MakeEnemy che prende come parametri:

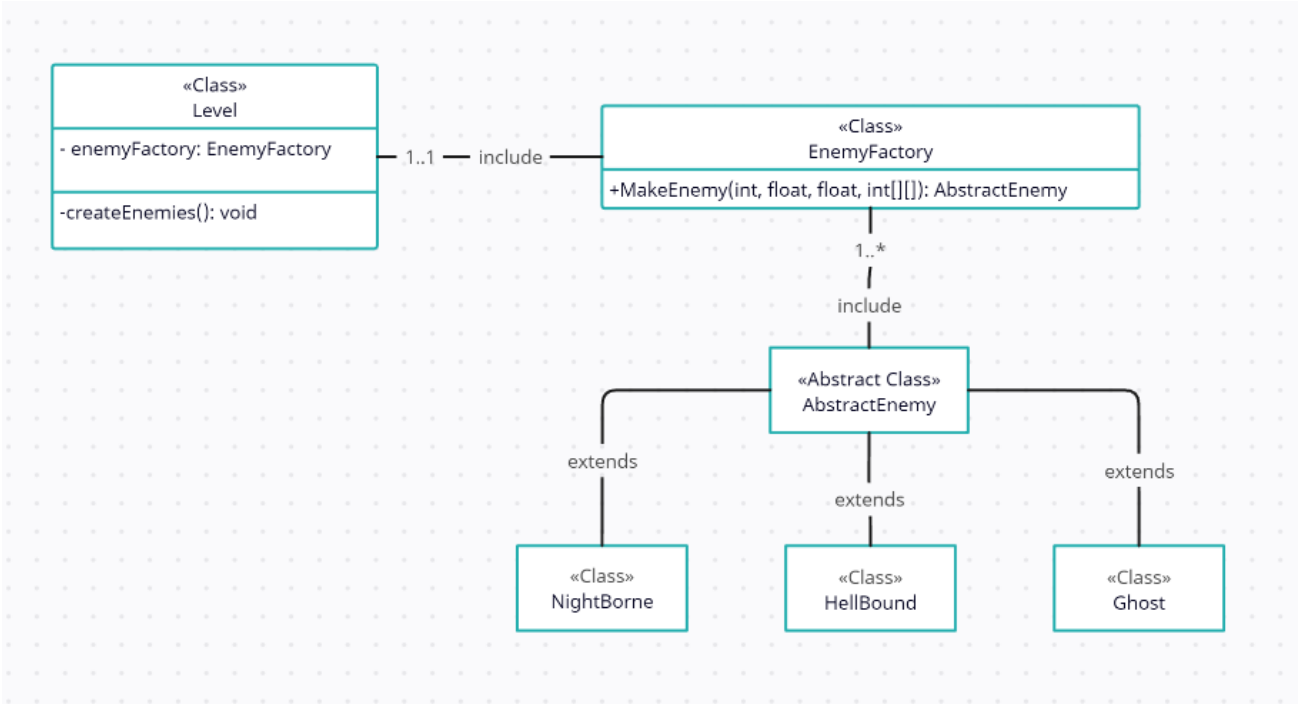
- **enemyType**: è un intero che serve a decidere il tipo di nemico che vogliamo creare
- **x,y**: Sono le coordinate in cui vogliamo far spawnare il nemico
- **levelData**: è la matrice contenente i dati del livello. Questa viene utilizzato effettivamente solo dal ghost

Il metodo implementa uno switch case che in base al tipo di nemico in ingresso (la variabile "enemyType") ritorna un nuovo nemico creando una nuova istanza di quel tipo di nemico desiderato.

Tutti i nemici ereditano dalla classe astratta **Abstract Enemy** e quindi implementano ognuno in modo diverso i metodi di questa classe.

Un esempio di utilizzo della classe "**EnemyFactory**" per generare un'istanza specifica di nemico è presente nella classe "**Level**" dove nel metodo "**createEnemy()**" l'istanza "**enemyfactory**" della classe "EnemyFactory" viene utilizzata per avere in output tramite il suo metodo "**makeEnemy()**" prima descritto, il tipo di nemico sulla base della variabile "value" (componente RBG Blu del levelData).

Ecco la sua rappresentazione in UML scelta per l'implementazione:



Chain Of Responsability:

Il design pattern chain of responsibility coinvolge le classi contenute nella cartella RenderChain in Entities. Viene utilizzato per gestire la parte “Render” dei nemici. La richiesta del tipo di disegno viene inviata al così detto “Handler” e se questo non sa gestirla la passerà al prossimo ad ecco collegato.

L interfaccia RenderInterface contiene il metodo “setNextHandler()” per impostare l’Handler successivo a cui verrà inviata la richiesta ed il metodo “renderEntity()” per gestire la richiesta effettiva. Quest’ultimo prende come parametri:

- Graphics g: richiesto dal JPanel per effetuare il Rendering.
- Player: una reference del Player che serve esclusivamente al ghost per effettuare determinati controlli legati al suo Rendering.
- Requests: un array di “RenderingRequest” classe il cui scopo è memorizzare le matrici di sprite di un nemico ed altre informazioni peculiari (motivo per il quale generiamo un array, RenderingRequest contiene le informazioni di un singolo nemico, non di tutti).
- AbstractEnemy: è il nemico che si vuole Renderizzare
- xlevelOffset e yLevelOffset: che servono alla logica di rendering per disegnare il nemico anche in relazione alla telecamera che si sposta seguendo il Player.

Nella classe “EnemyManager” viene fatta una istanza per ogni possibile Handler atto al rendering dei nemici. Tramite il costruttore della stessa classe questi vengono inizializzati e collegati l’uno all’altro formando la “Catena di Rendering” o “RenderChain”.

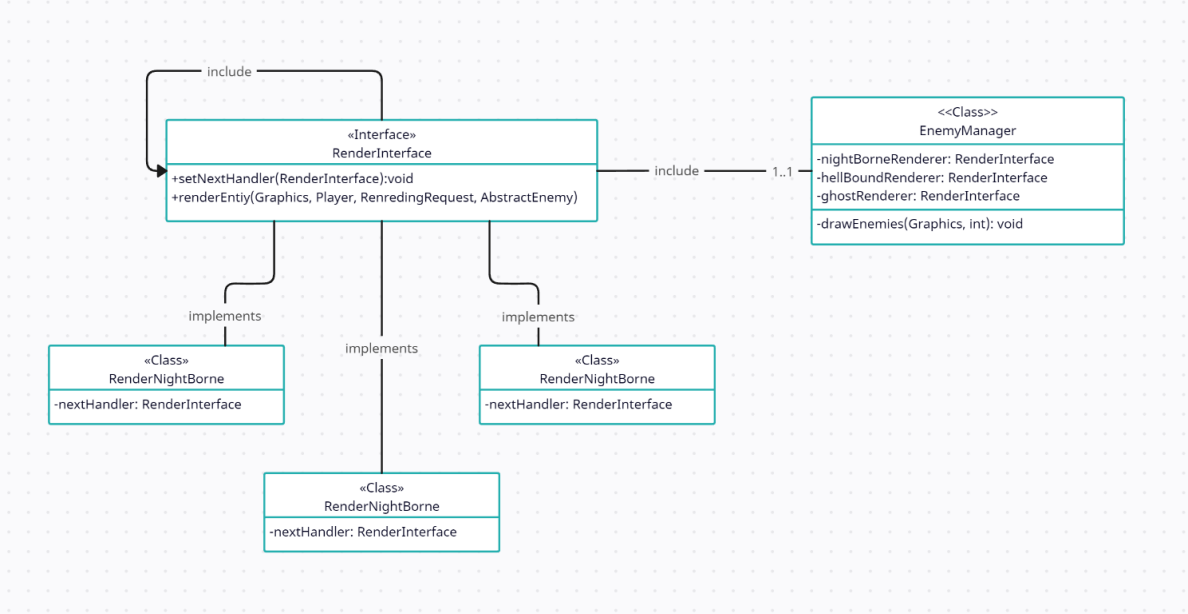
Durante la funzione Render della classe “EnemyManegr” un richiamo al metodo “RenderEntity” del primo Handler “NightBorneRenderer” viene seguito inviando tutti i parametri necessari.

All’interno della renderEntity viene fatto un controllo per verificare che il tipo di nemico sia quello che l’Handler si aspetta. Nel caso fosse così, viene gestita la richiesta ed il nemico viene Renderizzato nel suo stato corrente e nell suo sprite di animazione corrente. Ogni Handler ad eccezione di quello dedicato al “Ghost” se non sa gestire la richiesta la manderà quindi al successivo.

RenderingRequest contiene la richiesta effettiva con il tipo di nemico che si vuole creare e tutti gli sprite di quest’ultimo. Ha anche un eventuale attacco speciale. Questa classe ha due costruttori,il primo che prende l’immagine e il tipo di nemico e il secondo oltre a questi anche un attacco speciale nel caso dovesse essere presente (Prendiamo ad esempio il Ghost).

IL Metodo “initRenderRequests()” inizializza un array di RenderingRequest di grandezza 3 e inserisce in questo una richiesta personalizzata per ogni nemico.

Eccone il diagramma UML:


include
«Interface»
RenderInterface
+setNextHandler(RenderInterface):void
+renderEntiy(Graphics, Player, RenredingRequest, AbstractEnemy)
include
1..1
<<Class>>
EnemyManager
-nightBorneRenderer: RenderInterface
-hellBoundRenderer: RenderInterface
-ghostRenderer: RenderInterface
-drawEnemies(Graphics, int): void
implements
implements
implements
«Class»
RenderNightBorne
-nextHandler: RenderInterface
«Class»
RenderNightBorne
-nextHandler: RenderInterface
«Class»
RenderNightBorne
-nextHandler: RenderInterface

STATE:

Il design pattern dello “State” implementato comprende le classi “Game” “Playing” “GameOptions” “Menu” e l’interfaccia “StateMethods”:

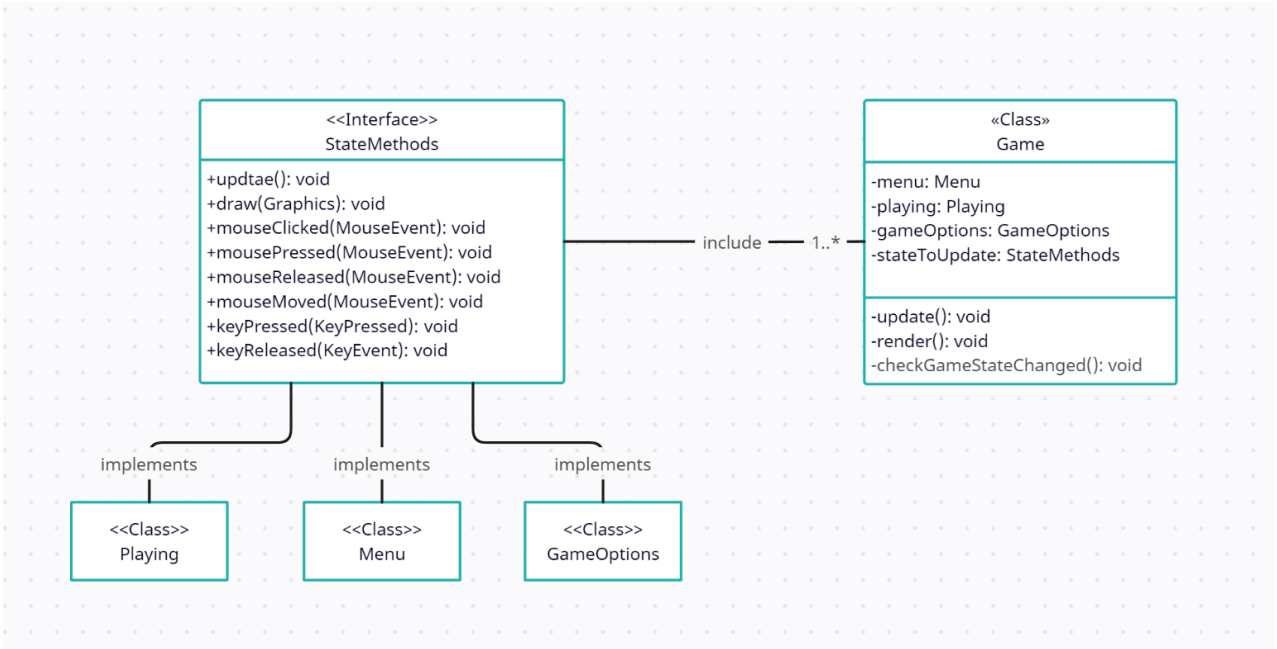
Le classi Menu, Playing e GameOptions sono implementazioni della interfaccia “StateMethods” che richiede di due importantissimi metodi da implementare che già conosciamo :“Draw” ed “Update” oltre ad altri metodi per l’oservazione degli input di gioco. rappresentano gli stati che può assumere il gioco a *runtime*.

L’utilizzo del pattern State avviene con il metodo “checkGameStateChanged()” della classe Game ed è bene introdurre un’altro elemento importante ai fini della spiegazione, l’enumerazione “GameState”. Questa non contiene altro che gli elementi della enumerazione che rappresentano gli stati di gioco (PAUSE, OPTIONS, PLAYING, QUIT) ed una variabile , chiamata “state”, capace di contenerli. Sono accesibili globalmente ed assegnati alla variabile di appoggio “state” di cui prima sotto determinati eventi di gioco, come di solito accade per la pressione di un bottone.

La classe Game contiene istanze dei suoi nodi figli “Playing” “Menu” “GameOptions” ed una istanza dell’interfaccia “StateMethds” quindi polimorfa.

Il controllo per osservare in che stato si trova il gioco può essere gestito tramite uno switch case nel metodo sopra nominato (“checkGameStateChanged()”) che, comportandosi di conseguenza, assegnarà caso per caso all’interfaccia polimorfa “StateMethods” una delle sue implementazioni “Playing” “Menu” o “GameOptions”. Game eseguirà i metodi “Update” e “Draw” andando a richiamare gli stessi metodi della interfaccia implementata applicando così il pattern e permettendo di aggiungere facilmente nuovi stati di gioco e di gestire quelli esistenti con una estrema favilità ed intuibilità.

Eccone il diagramma UML:



Memento:

Il design pattern del Memento è usato per creare i punti di salvataggio all'interno della partita atti al riavvio di un livello nel caso lo si voglia ricominciare per via di "un brutto start".

Utilizza principalmente le classi "EnemyMemento", "Memento", "MementoManager", "PlayerMemento" per essere implementato ed è utilizzato da varie classi, in particolare dalla classe "Level" per gestire i punti di salvataggio registrati o per crearne di nuovi.

"EnemyMemento" e "PlayerMemento" estendono "Memento" che ha al suo interno i metodi per ottenere le cordinate della hibox e della attackbox relative allo stato di partenza del livello.

PlayerMemento contiene al suo interno le seguenti variabili:

- "AttackBoxX" ed "attackBoxY": le coordinate dell' attacbox del player
- "hitBoxX" ed "hitBoxY": le coordinate del hitbox del player
- "currentHealt": La salute attuale del player.

I metodi getters sono implementati per ottenere informazioni su questi campi ed il costruttore li richiede in ingresso per effeettuare una inizializzazione.

EnemyMemento oltre alle variabili in comune con PlayerMemento possiede:

- "**airspeed**": la velocità di caduta all'inizio della partita (i nemici spawnano ad altezza di un blocco"
- "**active**": la flag booleana che ci indica lo stato di un nemico (se è vivo o morte per intenderci)
- "**firstUpdate**": la flag booleana che serve all' "AbstractEnemy" per eseguire gli update aerei allo spawn di un nemico nel livello.
- "**invulnerability**": stato di immortalità di un nemico.
- Il metodo "**setEnemyState()**": preso in input l'enemyType, ne conserva nella variabile "state" locale, quello che nella logica del "behavioral loop" di un nemico è il suo stato di partenza.

Il costruttore come nel caso del PlayerMemento utilizza il reference del nemico inviato per ottenere tutte le informazioni necessarie ad inizializzare le variabili sopra descritte. I metodi Getters non mancano di certo per poter riottenere indietro le informazioni quando necessario.

MementoManager gestisce la creazione dei memento.

Ha al suo interno:

- "**PlayerMementoes**": un' "ArrayList" di "PlayerMemento" che ci permetti di salvare gli stati legati al Player come illustrato sopra.
- "**EnemyMementos**": un' "ArrayList" di "ArrayList" di tipo "PlayerMemento" che ci permetti di salvare lo stato di N nemici di un determinato livello, (Ecco il perchè di ArrayList di ArrayList, sono gli stadi di M livelli contenenti N nemici).

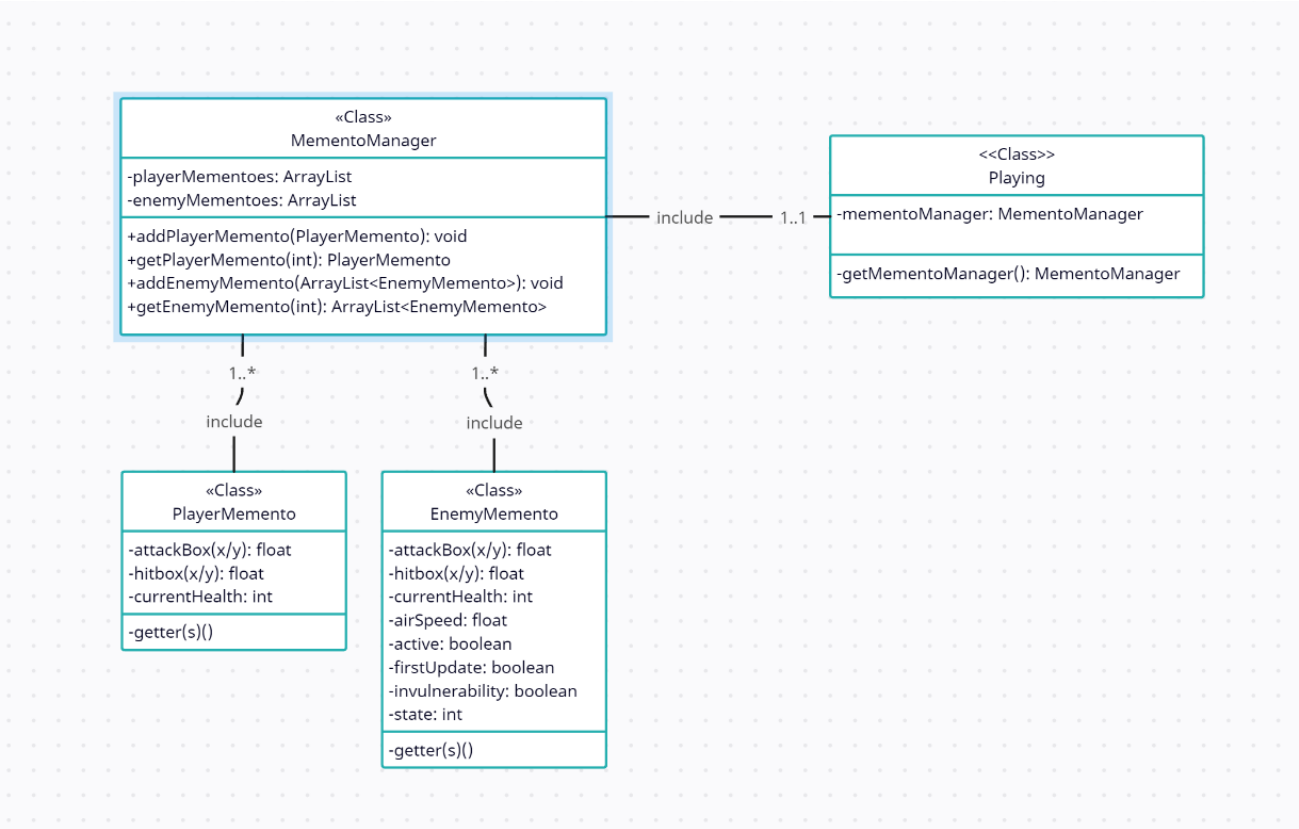
Nel costruttore vengono inizializzate le due liste e ***possiede dei metodi per ottenere indietro riferimenti ad un memento di player o a qello dei nemici di un livello scelto tramite indice, variabile in ingresso.***

Illustrata tutta la costruzione delle classi possiamo in realtà affermare che l'utilizzo del Memento Pattern è estremamente semplice.

L'istanzadi un "**MementoManager**" *la troviamo nel nodo* "**Playing**" così da renderlo accessibile a nodi figli ed in generale all'albero di nodi tramite un metodo "**getMementoMange()**".

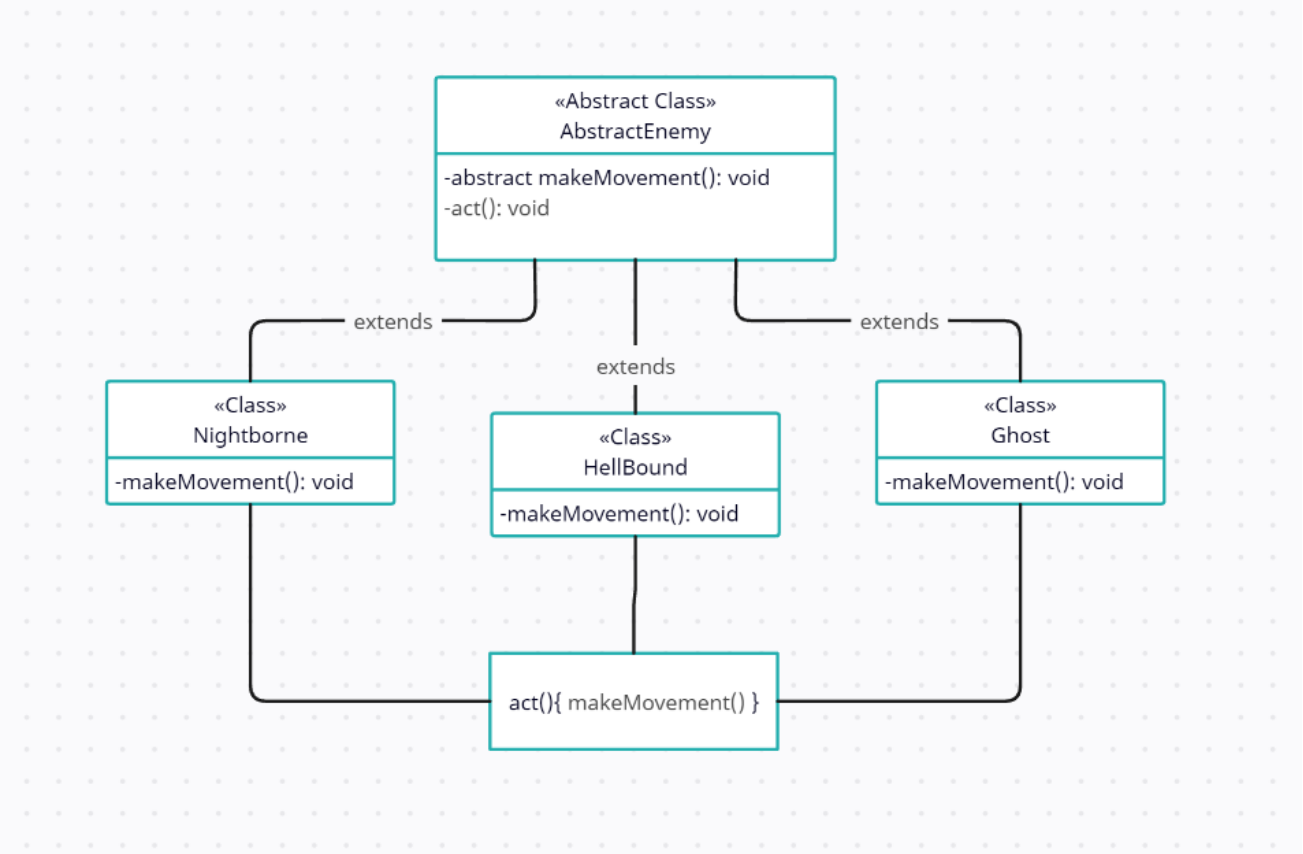
La classe ”**Level**” come annunciavamo prima ne fa largo uso al momento della creazione dei livello, il suo metodo ”**createSavingPoint()**” viene richiamato all’inizio di un livello per prendere tutti i valori di posizione e stato di ogni entità presente dentro di questo. **Tramite il riferimento al nodo padre Game, possiamo scendere l’albero e raggiungere il nodo Playing per avere accesso ai metodi del “MementoManager”**. Creati dunque i Memento del Player e l’array di memento dei nemici, possiamo richiamare il Manager ed istanziare un nuovo punto di salvataggio.

Un richiamo a questo Snapshot così creato lo possiamo vedere implementato quando viene invocato il metodo “**resetAll()**” del nodo “**Playing**”. Esso richiamerà i reset di ogni manager e del Player, i quali avendo accesso all’albero ed avendo quindi acceso anche ad un riferimento della stessa istanza di “MementoManager” appartenente al nodo “Playing”, potranno accedere tramite l’indice di livello, ai loro punti di salvatagio e ripristinare i loro stati iniziali.



Template Methods:

Il template methods coinvolge l'abstract enemy e tutti i nemici del gioco ed è implementato nel seguente modo: nella classe "**AbstractEnemy**" alla abbiamo il metodo "**act()**" che si occupa di aggiornare la posizione del nemico in aria facendolo cadere a terra. Eseguita questa operazione si occuperà di fare il richiamo alla funzioone implementante il **Design Pattern in conseiderazione, la "makeMovement()"**. Essa è un metodo dischiarato di proposito astratto in modo da obbligare le classi figlie ad implementarlo. **Questo ci permetterà di creare movimenti peculiari per ogni nemico**, prendiamo per esempio l'"HellBound" che salta ed il "Ghost" che nemmeno si muove ma is teletrasporta.



Prototype:

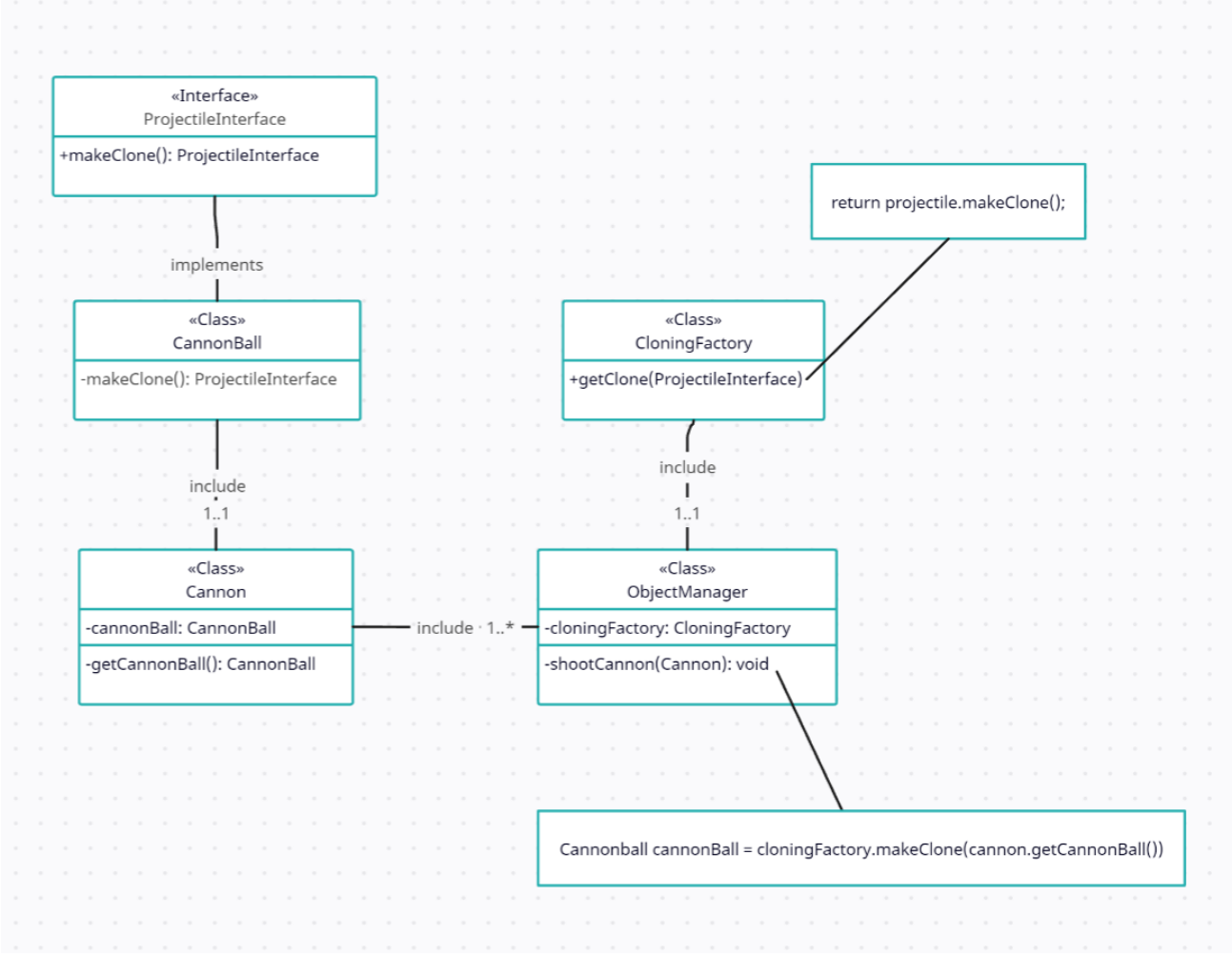
Il design pattern Prototype coinvolge le classi “Project Interface”, “CloningFactory” e “CannonBall”

Questo design pattern viene utilizzato per implementare la possibilità di creare un clone di qualsiasi proiettile nel gioco, per adesso solo le Palle di cannone.

ProjectileInterface estende la classe di java clonable e permette di implementare di conseguenza il metodo makeClone(). La classe “**CannonBall**” invece estende “**ProjectileInterface**”, gli verrà quindi richiesto di implementare il metodo “**makeClone()**” dell’intrefaccia, che in output restituià una copia esatta della classe “**CannonBall**”.

L’idea di base viene quindi implementata tramite la classe del Cannone, dentro di esso viene istanziato un oggetto “**CannonBall**” che prenderà come valori di x ed y di partenza le coordinate della bocca del cannone. Un metodo “**getCannonBall()**” ci restituisce un riferimento a quella istanziata così da poterla utilizzare.

L’”**ObjectManager**” è il nostro “**Client**” di riferimento, poichè al momento di dover far sparare un Cannone, ***richiamerà tramite il metodo “getCannonBall()” il prototipo da clonare specifico per quell’oggetto, aggiungedno poi l’istanza clonata ottenuta alla lista di proiettili da aggiornare e da gestire***. Ma di questo abbiamo già parlato.



Conclusioni:

Come Abbiamo potuto osservare WOP non è solo un gioco, o soltanto un programma scritto in Java, Wop non è stato un passatempo che il nostro team si è proposto di fare. Wop è studio, è dedizione, ingegno, fatica, amore divertimento e tantissime altre cose che ora dopo mesi di lavoro, nemmeno ci vengono in mente.

Dietro ogni singola scelta c'è stato uno studio di giorni riguardo a Level, Enemy e Player Design, c'è stato tantissimo pensiero critico sul come poter implementare una determinata meccanica, un determinato nemico, in che frame disegnato da noi a mano, bisognasse fare il controllo di applicazione del danno, dietro ogni riga di codice 10 Bug da dover risolvere ma ce l'abbiamo, siamo impazziti ma alla fine ce l'abbiamo fatta, abbiamo capito come rendere dell'inerme e puro codice, riga dopo riga, vivo...

Grazie per aver giocato con noi.

Credits:

Ariano Luigi

Biondi Morgan

Tridente Antonio