Music vs Robots

Cognome	Nome	Matricola
Pellegrinelli	Nicolò	2034334
Rosso	Carlo	2034293

"Music vs Robots" è un programma scritto interamente in c++ che si ispira al famoso gioco mobile Plants vs Zombie. Il gioco si basa su una griglia, chiamata Game, in cui si possono inserire, rimuovere e migliorare degli strumenti musicali (copia delle piante) che suonano per sconfiggere i robot (zombie) che arrivano dal lato destro. L'obiettivo del gioco è quello di resistere il più possibile all'attacco dei robot. Il gioco termina se almeno un robot raggiunge il lato sinistro del Playground. La scelta iniziale è stata da subito molto combattuta: l'obiettivo era realizzare un programma interessante e sfidante ma allo stesso tempo gradevole da sviluppare. La scelta si è rilevata soddisfacente e divertente. Lo sviluppo si è rivelato estenuante in alcuni punti, ma siamo riusciti a portare a termina il progetto rispettando le nostre aspettative.

Descrizone del modello

Il modello si suddivide in due parti:

- La gestione delle entità del gioco, ovvero i robot e gli strumenti musicali che producono i danni;
- La gestione del gioco, ovvero la griglia in cui si muovono le entità e la gestione del tempo e l'aumento della difficoltà ad esso legato.

Gestione delle entità

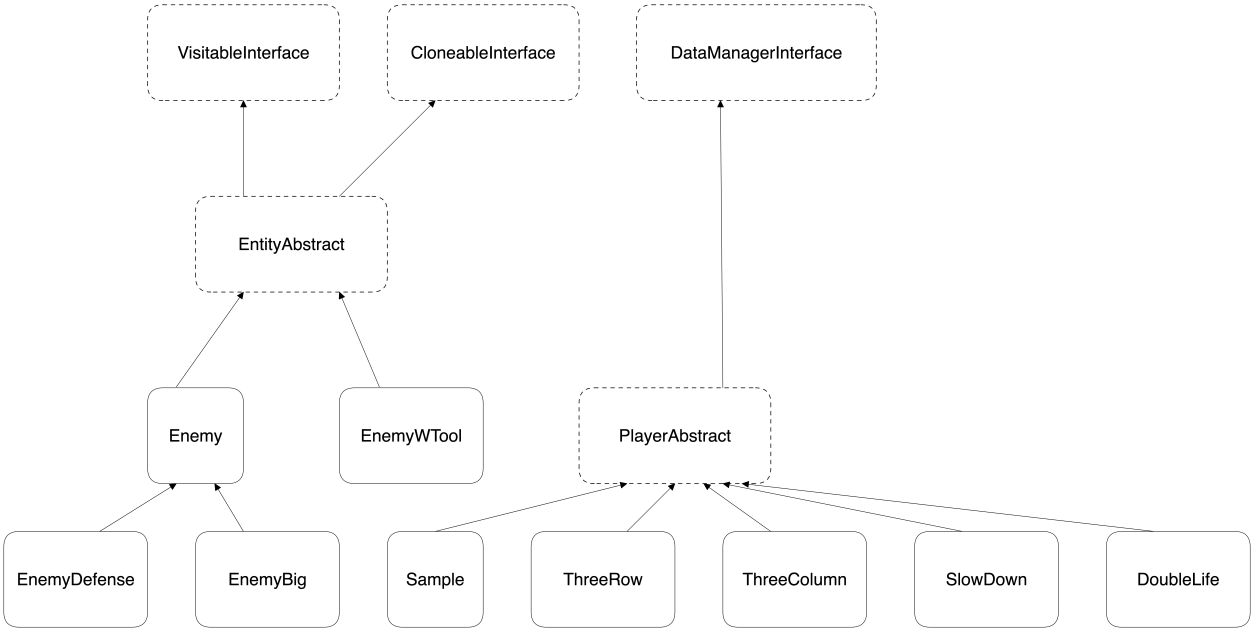

Figure 1: Gerarchia delle entità

La gerarchia comincia con tre classi astratte:

- VisitableInterface: interfaccia che permette di visitare un oggetto mediante il pattern Visitor, attraverso il metodo accept. Questa classe è utilizzata dalla classe VisitorInterface per accedere alle immagini delle entità;

- CloneableInterface: interfaccia che permette di clonare un oggetto mediante il pattern Prototype, attraverso il metodo clone. Per esempio, abbiamo definito una classe `ptr<T>`, si tratta di uno smart pointer classico, con l'aggiunta che la classe T deve implementare CloneableInterface;

- DataManagerInterface: tutti gli oggetti che implementano questa interfaccia possono essere salvati e caricati da un file;

Le prime due classi sono ereditate da EntityAbstract, una classe astratta che gestisce le interzioni tra le entità. Le entità possono effettuare due azioni principali: possono attaccare e ricevere danno.
Le classi che ereditano da EntityAbstract sono:

- PlayerAbstract: è una classe astratta perché non implementa alcuni metodi delle interfacce e della classe EntityAbstract; inoltre definisce metodi virtuali puri che devono essere implementati dalle classi che ereditano da essa. Le classi che derivano da PlayerAbstract sono gli strumenti musicali che sono utilizzati per difendersi dai robot;

- Enemy: è una classe concreta che rappresenta i robot che attaccano l'utente. Questa classe rappresenta il robot più semplice. Da Enemy derivano le classi EnemyDefense e EnemyBig che rappresentano dei robot con caratteristiche diverse;

- EnemyWTool: utilizza la relazione "has-a" e rappresenta un robot con un Tool. I tool migiorano qualche proprietà del robot. Enemy e tool sono generati casualmente utilizzando il pattern Factory;

Le entità interagiscono tra di loro generando i danni o subendoli.

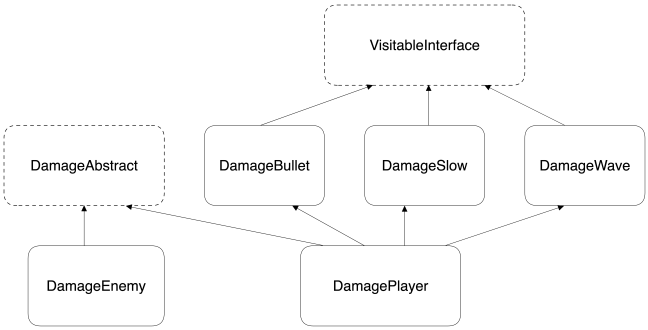

Figure 2: Gerarchia dei danni

La classe DamageEnemy rappresenta i danni che sono generati dai robot e sono inflitti agli strumenti musicali.
La classe DamagePlayer rappresenta i danni che sono generati dagli strumenti musicali e sono inflitti ai robot. Eredita da VisitableInterface perché sono visibili. Eredita da tre classi concrete, anche se non sono mai istanziate singolarmente, perché gli strumenti musicali possono generare diversi tipi di danni. DamageBullet rappresenta i danni che si infrangono sul primo robot; DamageSlow non infligge alcun danno, ma rallenta i robot; DamageWave infligge la stessa quantità di danno su tutti i robot per un certo numero di colonne.

Gestione del gioco

ObservablePlaygroundInterface è l'interfaccia utilizzata per comunicare alla grafica le modifiche che avvengono nel gioco.
Ci sono tre classi da cui eredita Game oltre a ObservableGame:

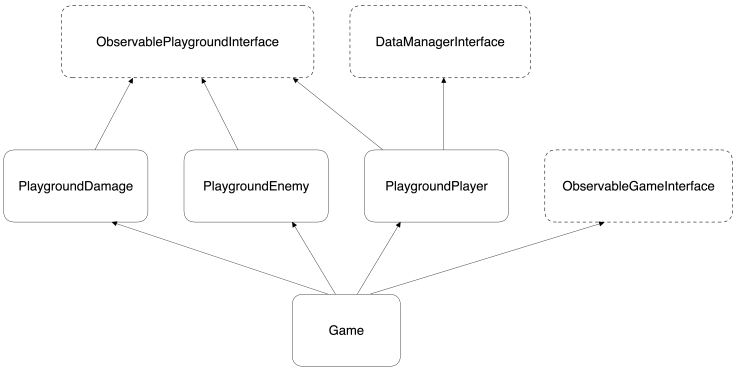

Figure 3: Gerarchia del gioco

- PlaygroundPlayer: è composta da una griglia di `ptr<PlayerAbstract>`. Questa classe gestisce gli strumenti musicali che attaccano i robot;
- PlaygroundEnemy: è composta da una griglia di `deque<EnemyWTool>`, perchè ci possono essere più robot sulla stessa cella. Questa classe gestisce i robot che attaccano gli strumenti musicali;
- PlaygroundDamage: è composta da una griglia di `ptr<DamagePlayer>`. Questa classe gestisce i danni che sono generati dagli strumenti musicali e sono inflitti ai robot;

La classe Game gestisce le interazioni tra le tre classi Playground e sfrutta il pattern Singleton per mantenere un'unica istanza del gioco accessibile da qualsiasi punto del programma.

Utilizzo del polimorfismo

Il polimorfismo è stato utilizzato in EnemyWTool, per quanto riguarda le entità. Infatti i vari robot ed i vari tool sono gestiti in modo interscabiabile dalla classe EnemyWTool.
Il polimorfismo è sfruttato in modo più ampio nella gestione del gioco. Infatti, la classe Game gestisce le interazioni tra le entità utilizzando i metodi virtuali puri definiti in EntityAbstact e in PlayerAbstract.
Le principali funzioni virtuali della gerarchia sono:

- `bool sufferDamage(DamageAbstract &damage)`: questa funzione viene chiamata quando un'entità subisce un danno. Ritorna true se l'entità è stata distrutta, false altrimenti;
- `DamageAbstract *attack() const`: questa funzione viene chiamata quando un'entità attacca. Ritorna un puntatore al danno che viene generato dall'entità;
- `CloneableInterface *clone() const`;
- `void accept(VisitorInterface &visitor) const`: il metodo è utilizzato per accedere ad un istanza di VisitableInterface. È chiamato da VisitorInterface, in particolare da imageVisitor, per accedere all'immagine dell'entità;
- `void levelUp()`: viene utilizzata per aumentare il livello di uno struemnto musicale;
- `u32 getCost()`: ritorna il costo per posizionare uno strumento musicale sulla griglia, oppure il costo per effettuare l'upgrade di uno strumento musicale già presente sulla griglia;
- `u32 move()`: ritorna il numero di celle che uno robot può percorrere in un turno;
- `u32 damage()`: ritorna il danno che un oggetto DamageAbstract infligge;
- `bool slow()`: ritorna true se un oggetto DamagePlayer rallenta gli EnemyWTool, false altrimenti;
- `void oneWave()`: indica all'oggetto DamagePlayer che ha percorso una cella.

Persistenza dei dati

La persistenza dei dati è gestita attraverso la classe DataManagerInterface. Infatti, tutte le classi che devono essere salvate o caricate da un file JSON ereditano da essa; ovvero: PlayerAbstract, PlaygroundPlayer, Cash e Timer. Le classi che ereditano da DataManagerInterface devono implementare i seguenti metodi:

- `std::string toString() const`: viene utilizzata per convertire la classe in una stringa, in modo da poterla salvare su un file;
- `DataManagerInterface *fromString(std::string)`: viene utilizzata per convertire una stringa in una classe, in modo da poterla caricare da un file;

Il metodo `static void saveAll()` viene invocato alla chiusura del programma o al ritorno alla schermata principale e chiama il metodo `std::string toString() const` su tutti gli oggetti che sono da salvare. Il metodo `static bool loadAll()` viene invocato per caricare l'ultima partita salvata, sfruttando i metodi `DataManagerInterface *fromString(std::string)`. Il file example.json contiene un esempio di salvataggio.

Funzionalità implementate

- Inserimento di strumenti musicali sulla griglia;
- Rimozione di strumenti musicali dalla griglia;
- Upgrade di strumenti musicali;
- Visualizzazione del livello degli strumenti musicali;
- Inserimento di un robot sulla griglia ad ogni clock. Con relativo aumento dell'attacco e della vita in funzione del tempo;
- Salvataggio automatico delle partite e caricamento dell'ultima partita salvata;
- Visualizzazione del tempo dall'inizio della partita;
- Visualizzazione del denaro disponibile e del costo di inserimento di uno strumento musicale o del suo upgrade;
- Movimento dei robot verso gli strumenti musicali;
- Inserimento dei danni generati dagli strumenti musicali;
- Movimento dei danni;
- Utilizzo di immagini per visualizzare gli strumenti musicali, i robot e i danni;

Rendicontazione delle ore di lavoro

Attività	**Ore Previste**	**Ore Effettive**
Progettazione	10	17
Sviluppo del codice del modello	25	45
Studio del framework Qt	5	5
Sviluppo del codice della GUI	5	5
Test e debug	10	25
Stesura della relazione	5	3
Totale	**60**	**100**

Le ore effettive sono state maggiori di quelle previste, perché siamo incorsi in un bug che non siamo riusciti a risolvere. Tutt'ora rimango all'oscuro del problema che si verificava. Le ore di debug sono molto maggiori delle aspettative proprio per questo motivo. Dopo diversi tentativi di risoluzione del problema, abbiamo deciso di riscrivere la logica del gioco. In effeti, credevo che la riscrittura sarebbe stata piuttosto lenta. Al contrario, avendo già un esempio da seguire si è rivelata piuttosto rapida; anzi ci ha dato modo di riorganizzare il codice per mostrare più chiaramente i pattern implementati. Inoltre, l'implementazione dei danni è potuta partire dal basso.

Suddivisione delle attività

Attività	**Sviluppatore**
Implementazione di EnemyWTool e derivati	Carlo
Implementazione di PlayerAbstract e derivati	Nicolò
Sviluppo di DamageAbstract e derivati	Nicolò
Sviluppo di PlaygroundPlayer e PlaygroundEnemy	Carlo
Sviluppo di PlaygroundDamage	Nicolò
Sviluppo di Cash e Timer	Tutti
Sviluppo di Game	Tutti
Sviluppo dell'utilità	Tutti
Sviluppo della GUI	Tutti
Salvataggio su file	Carlo
Caricamento da file	Nicolò

Considerazioni finali

Abbiamo scelto un progetto che ci permettesse di sperimentare quanto più possibile ciò che abbiamo imparato in questi anni di università. Ci siamo cimentati in un'applicazione più ampia di quanto effettivamente richiesto, perché siamo appassionati alla programmazione, in primo luogo; inoltre è nostro interesse arricchire il nostro portfolio per dimostrare le nostre capacità, alcune cose che abbiamo imparato e la nostra passione per la programmazione.
Il progetto è stato molto impegnativo, ma ho quasi voglia di continuare a lavorarci: ho diverse idee che potrebbero essere sviluppate e approfondite.
Mi è piaciuto molto lavorare con una persona che ha un approccio diverso ai problema, ma allo stesso tempo è appassionata quanto me; questo mi ha dato modo di imparare a collaborare con qualcuno che già conoscevo, ma con cui non avevo mai lavorato.
In conclusione, sono molto soddisfatto del lavoro svolto.