Music vs Robots

Sviluppatori: Nicolò Pellegrinelli 2034334 e Carlo Rosso 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 sinistro.
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 *Game*.

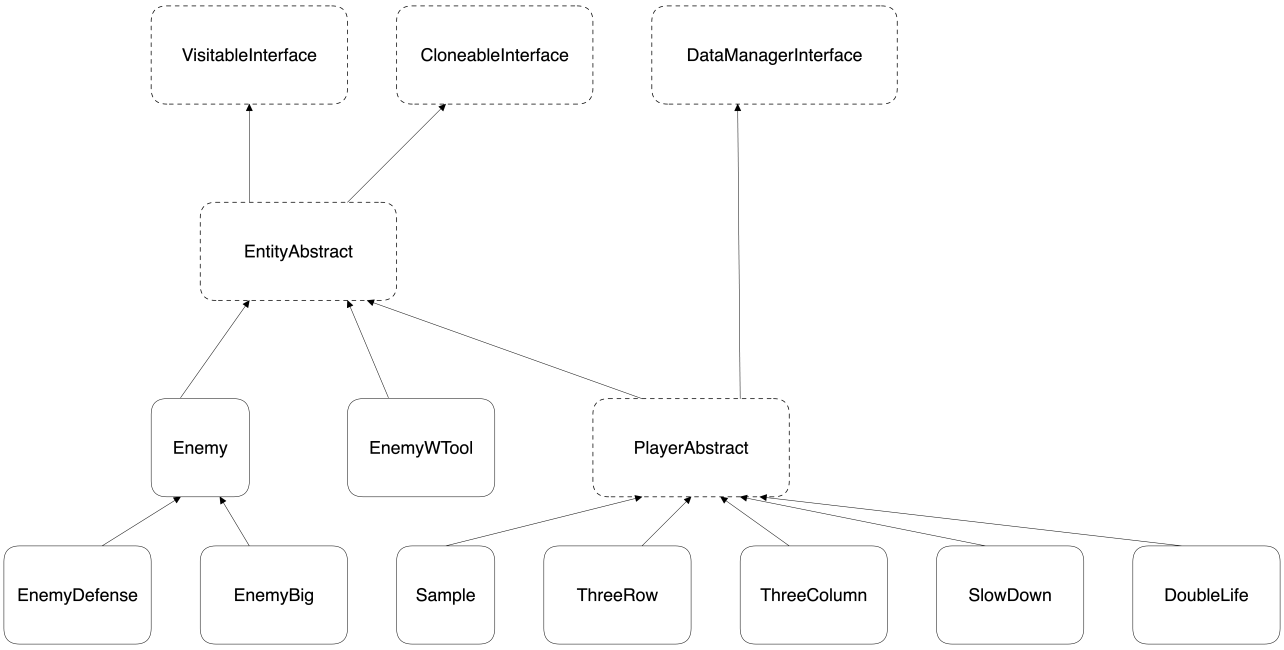
La scelta iniziale è stata da subito molto combattuta: l'obiettivo era realizzare un programma interessante e sfidante ma allo stesso tempo gradevole.
La scelta si è rivelata molto entusiasmante e soddisfacente, anche se lo sviluppo è stato, a tratti, più impegnativo del previsto.

Nota: questa è una riconsegna ma siccome molte parti del progetto sono cambiate ho deciso di riscrivere anche la relazione.

Descrizione del modello

Il modello si suddivide in due parti:
- La gestione delle entità del gioco, separate in strumenti musicali e robot
- La gestione del gioco, ovvero la gestione del posizionamento e combattimento nel *Game*

Gestione delle entità



Gerarchia delle entità

Tre interfacce rappresentano la punta della gerarchia:

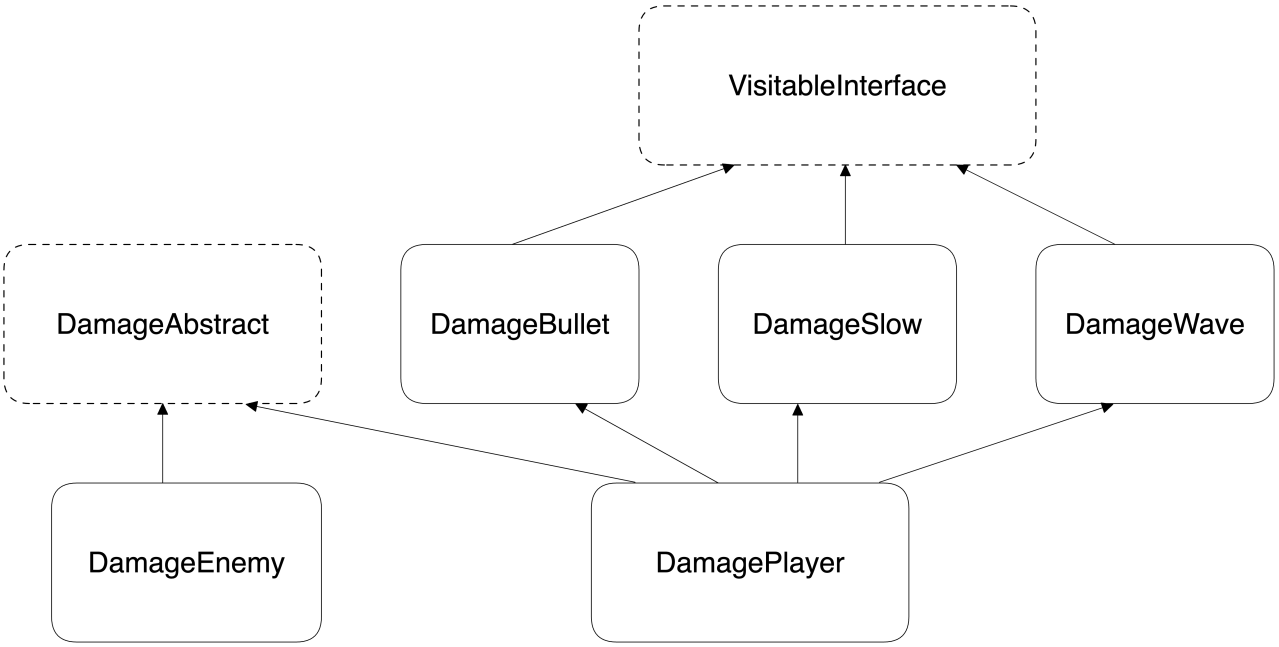
- *VisitableInterface* contiene i metodi per la visita degli oggetti attraverso il visitor pattern
- *CloneableInterface* permette alle entità di avere un metodo clone. Lo smart pointer che abbiamo implementato ptr<T> richiede che T implementi *CloneableInterface*
- *DataManagerInterface* garantisce la permanenza dei dati

Al livello successivo troviamo la classe *EntityAbstact* che rappresenta la base delle entità del gioco; questa classe permette l'attacco e il ricevimento dei danni.
Da *EntityAbstract* derivano *Enemy*, *EnemyWTool* e *PlayerAbstract*.

Enemy è la classe che rappresenta i robot più semplici e da essa derivano due tipi di robot ovvero *EnemyDefence*, che possiede una difesa ai danni maggiore, e *EnemyBig* che invece ha una velocità di movimento inferiore. Queste classi, nonostante siano concrete, vengono rappresentate nel *Game* solamente tramite *EnemyWTool* che, tramite una relazione "has-a", contiene uno dei tre tipi di enemy e un *Tool* che ne migliora le qualità.

Gli strumenti musicali sono invece derivazioni della classe PlayerAbstract, che rappresenta uno strumento generico, ma essendo astratta non è istanziabile.

La gestione dei danni è affidata a una gerarchia di classi distinta.

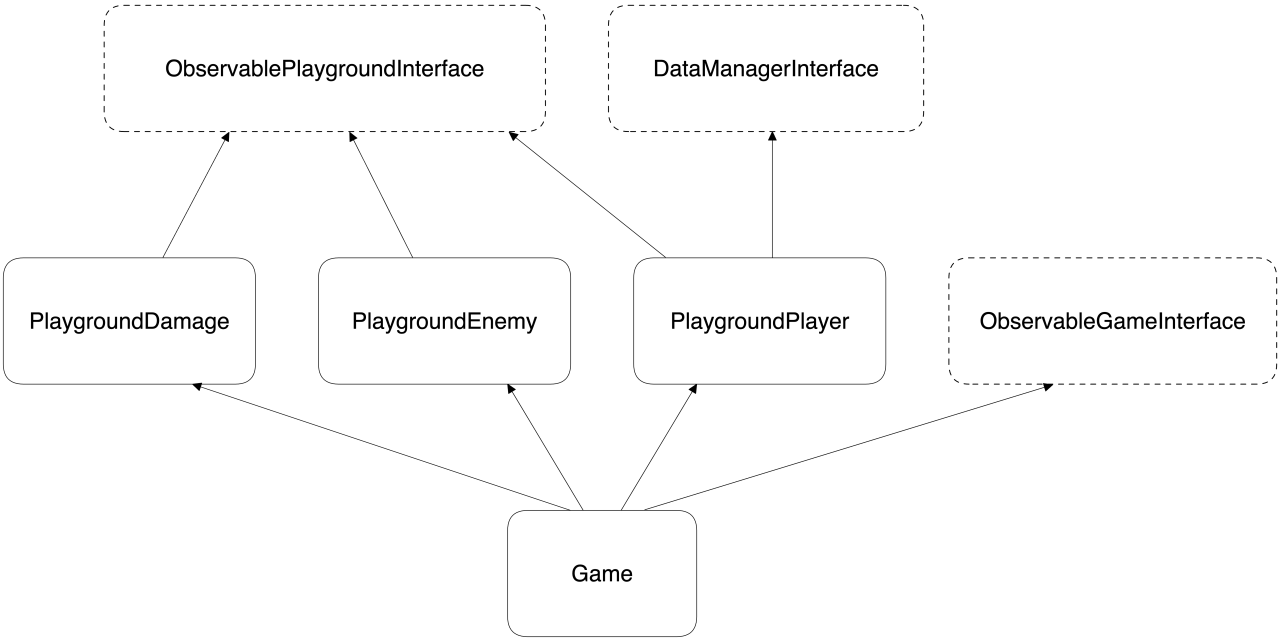


Gerarchia dei danni

DamageEnemy è la classe che si occupa dei danni inflitti dai robot, mentre *DamagePlayer* si occupa dei danni prodotti dagli strumenti musicali, divisi in *DamageBullet*, *DamageSlow* e *DamageWave* che, nonostante siano classi concrete, non vengono mai istanziate singolarmente. *DamageBullet* colpisce solo il primo robot che trova sul cammino mentre *DamageWave* infligge lo stesso danni a tutti i robot, *DamageSlow*, invece, non infligge veramente danno ma rallenta i robot. Queste tre classi inoltre derivano da *VisitableInterface* per permettere la loro rappresentazione nell'interfaccia.

Gestione del gioco

La classe principale del gioco *Game* eredita da tre tipi di playground: *PlaygroundPlayer*, *PlaygroundEnemy*, *PlaygroundDamage*.
Questa divisione permette una gestione più semplice e modulare delle meccaniche di gioco e la loro unione, con l'utilizzo del pattern singleton, garantisce un'unica istanza del motore di gioco accessibile da ogni parte del programma.



Ciascuna delle parti del *Game* deriva da un'interfaccia di tipo observable che implementa i metodi necessari alla notifica degli osservatori del gioco, utilizzati nella parte dell'interfaccia, quando lo stato del gioco cambia.

PlaygroundPlayer e *PlaygroundDamage* consistono in una matrice di ptr<>, rispettivamente *PlayerAbstract* e *DamagePlayer*; *PlaygroundDamage* invece è composto da una matrice di deque<EnemyWTool> per permettere la presenza di più robot nella stessa cella.

Utilizzo del polimorfismo

Il polimorfismo viene usato in diversi aspetti del programma, attraverso le varie interfacce e, soprattutto, per la gestione delle entità: i *RobotWTool* sono in questo modo molto dinamici e i due campi robot e tool possono essere interscambiati con i vari tipi descritti sopra per garantire una maggiore diversità e versatilità del gioco.

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 strumento musicale;
- unsigned int 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;
- unsigned int move(): ritorna il numero di celle che uno robot può percorrere in un turno;
- unsigned int 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 da una classe astratta *DataManagerInterface* dalla quale derivano tutte le classi che vengono salvate/caricate su/da JSON come *Playground*, *Entity*, *Cash* e *Timer*.

Il metodo statico "void saveAll()" viene invocato alla chiusura del programma o al ritorno alla schermata principale e si occupa di chiamare il metodo virtuale "std::string toString() const" di ogni oggetto che deve essere salvato.

Il metodo statico "bool loadAll()", invece, viene invocato per caricare l'ultima partita e si occupa di analizzare il file "data.json" e chiamare le rispettive funzioni "DataManagerInterface *fromString(std::string)" con covariante che creano gli oggetti caricati con i rispettivi attributi.
Un esempio di JSON salvato è example.json

Funzionalità implementate

- Inserimento strumenti musicali
- Rimozione strumenti musicali
- Inserimento automatico dei danni
- Aumento del livello degli strumenti musicali inseriti
- Visualizzazione del livello di ciascuno strumento
- Inserimento di zombie in modo casuale con aumento delle statistiche in base al tempo
- Salvataggio automatico e caricamento dell'ultima versione
- Visualizzazione del tempo di gioco
- Gestione dei soldi
- Movimento dei robot (funzionalità in fase di miglioramento)
- Utilizzo di immagini per la visualizzazione delle entità e dei danni

Rendicontazione ore

Attività	**Ore previste**	**Ore Effettive**
Progettazione	12	17
Sviluppo del codice del modello	30	40
Studio del framework Qt	10	10

Sviluppo del codice della GUI	12	12
Test e debug	7	20
Stesura della relazione	5	4
Totale	76	103

Le ore effettive sono risultate abbastanza superiori rispetto a quelle previste perché la scrittura del modello si è rivelata più complicata a cause di un bug che non siamo stati in grado di risolvere; abbiamo quindi deciso di riscrivere alcune parti della meccanica del gioco per migliorare la qualità del codice è rendere più chiari i pattern implementati.

Suddivisione delle attività

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

Considerazioni finali

Il progetto per il corso programmazione a oggetti è diventato da subito una sfida molto interessante che andava ben oltre al superamento di un esame, ma voleva essere inteso come un occasione per crescere e poter mostrare attraverso esso le nostre abilità non solo ai professori ma soprattutto a delle possibili occasioni lavorative.
Il progetto è stato, a tratti, più sfidante del previsto ma credo di aver imparato molto in questi mesi, in particolare a lavorare con un’altra persona.

Con questa riconsegna abbiamo riprogettato e ricostruito la parte della gestione di gioco (classi playground e game), riuscendo a risolvere il problema che avevamo incontrato nella prima fase dello sviluppo del programma e semplificando il codice per una migliore manutenibilità.