UNIVERSITÀ DEGLI STUDI DI PADOVA, INFORMATICA, PROGETTO PER IL CORSO DI PROGRAMMAZIONE AD OGGETTI

MONSTERKALK

di

Piazzetta Samuele Giuliano matricola: 1144219

in collaborazione con

Lain Gianluca matricola: 1144212

MONSTERKALK: CALCOLATRICE POLIMORFA

Introduzione

Abbiamo deciso di progettare una calcolatrice ispirata ai famosi giochi di combattimento a turni in cui i giocatori si sfidano con i loro mostri digitali. I mostri possono combattere tra di loro, usare degli strumenti ed altro, in condizioni differenti. La nostra calcolatrice simula questi combattimenti permettendo all'utente di emulare particolari situazioni per studiarne i risultati.

Descrizione della gerarchia

La nostra gerarchia è basata su una classe astratta pura, *Oggetto*, che ci permette di trattare tutte le nostre classi in modo polimorfo nell'applicazione. Inoltre mette a disposizione due metodi astratti puri, *clone* e *toMap*. Clone permette la clonazione dell'oggetto , mentre toMap permette di ottenere una mappa <attributo, valore> di stringhe, per rendere l'oggetto più facilmente stampabile a schermo. Questa classe contiene inoltre degli attributi statici usati più o meno in tutte le classi, che sono le efficacie dei tipi (es.: fuoco, acqua, erba) e le statistiche (attacco, difesa, attacco speciale, difesa speciale).

La classe principale che utilizza proprio i concetti di tipi e statistiche è *mostro.* Mostro rappresenta la creatura che nei giochi da cui abbiamo preso ispirazione è al centro delle azioni. Un mostro è caratterizzato da uno o più tipi e da un valore per ogni caratteristica presente. Ha inoltre un campo energia, un rango, che ne determina la potenza complessiva ed infine lo stato di megaevoluzione. I metodi della classe mostro sono i seguenti:

- *void Mostro::evolvi()* : aumenta il rango e le statistiche del mostro, questo argomento verrà trattato nella classe pietre;
- *Mostro* fusione(Mostro* mostro)*: permette di fondere due mostri creandone uno di nuovo, più potente dei materiali da fusione;
- *void attacca(vector<Mostro*> difensori, Ambiente* ambiente) const*: permette di attaccare un secondo mostro o una serie di mostri, sottraendo salute ai bersagli in base alle loro statistiche e ai loro tipi (viene usata la mappa dei tipi di oggetto per calcolarne l'efficacia totale). Nel calcolo del danno influisce anche l'ambiente in cui avviene l'attacco, ma l'argomento verrà trattato negli ambienti;
- *void megaMossa(vector<Mostro*> difensori)*: una attacco che può essere sferrato solamente da un mostro nello stato di mega-evoluzione, infligge il quintuplo del danno da attacco base.
- *void sacrificio(Mostro* sacrificio)*: un mostro può essere sacrificato ad un altro mostro che guadagna così dell'energia.

Un'altra classe che deriva direttamente da oggetto è *ambiente*, che rappresenta le caratteristiche del luogo in cui avvengono le azioni. In particolare gli ambienti possono potenziare e/o depotenziare alcuni tipi negli attacchi e nelle mosse usati dai mostri ed influire su alcuni strumenti, come nella raffinatura di una pietra o nell'esposizione di una bacca/caramella. I metodi di ambiente sono:

- *void inverti()*: scambia tipi potenziati e depotenziati;
- *Ambiente* fusione(Ambiente* ambiente)*: crea un terzo ambiente a partire da due ambienti;

Sempre da oggetto deriva una classe base astratta, *Cura*, che include tutti gli oggetti che possono essere utilizzati su di un mostro per alterarne le caratteristiche. Da un lato derivano *bacca* e *caramella*, dall'altro gli strumenti. Bacca e caramella hanno un gusto, delle calorie, e nel caso della caramella anche una qualità. Queste hanno i seguenti metodi:

- *virtual void usa(Mostro* mostro) const =0*: applicato ad un mostro ne varia le caratteristiche;
- *Pozione* spremi()*: produce una pozione se l'oggetto spremuto è una bacca, un etere se è una caramella;
- *virtual Caramella* cucina(Cura *cura)*: permette di cucinare una bacca/caramella con un qualsiasi strumento, ed in base all'affinità del gusto con il formato dello strumento ritorna una caramella di qualità bassa o alta;

- *virtual void esponi(Ambiente* ambiente)*: una bacca o una caramella se esposte ad un ambiente favorevole aumentano le loro proprietà benefiche.

Dall'altro lato, sempre da cura, deriva una classe astratta pura, *strumento*, che aggiunge un attributo, il formato dello strumento. Viene aggiunto inoltre un metodo, *unisci*, che fonde due strumenti creando una pietra con tanta più lucentezza quanto i formati degli strumenti sono pregiati e compatibili. Da strumento derivano le tre classi: *pozione, revitalizzante ed etere*. Tutte e tre dispongono del metodo *clone*, mentre differiscono nel metodo *usa*:

- *pozione*, caratterizzata dalla salute che fa recuperare ad un mostro, se usata su di un mostro ne aumenta la salute di questa quantità, fino alla salute massima. Se il mostro è morto o se è al pieno della vita l'utilizzo della pozione non ha alcun effetto;
- *revitalizzante*, caratterizzato dall'attributo percentuale vita, riporta in vita un mostro debilitato, ovvero con 0 di vita, e guarisce la vita della percentuale indicata;
- *etere* invece se utilizzato su di un mostro, ne aumenta le caratteristiche di una certa percentuale, entrambe le informazioni sono degli attributi dell'oggetto stesso.

Un altro ramo che deriva direttamente da oggetto è quello di *pietra* e della derivata *mega-pietra*. Una pietra è caratterizzata dalla lucentezza, da un rischio di rottura e dallo stato di rottura (booleano che indica se è rotta). Una pietra può essere *lucidata*, eventualmente con l'ausilio di una pozione, e ciò ne aumenta la lucentezza. Durante il processo di lucidatura rischia la rottura, ed una pietra rotta non può essere ulteriormente lucidata. Una volta che la lucentezza della pietra assume dei valori positivi può essere *raffinata* a megapietra. Una megapietra acquisisce un tipo randomico normalmente, o uno dei tipi favoriti se viene lucidata in presenza di un ambiente. Una megapietra inoltre può essere frammentata producendo un revitalizzante o raffinata ulteriormente per variare il suo tipo. Delle pietre e megapietre non rotte possono essere assegnate durante l'evoluzione di un mostro per trarne dei bonus: una pietra raddoppia l'evoluzione, consumando l'energia solo per la prima evoluzione, ed una megapietra aggiunge pure lo stato di megaevoluzione in caso il suo tipo sia lo stesso del mostro preso in considerazione.

Infine c'è il ramo delle mosse. Da Oggetto deriva la classe *mossa*, con attributi tipo e precisione. Mossa è una classe astratta pura, che aggiunge due metodi virtuali puri che sono "*esegui*" su un vettore di mostri e il relativo overloading in presenza di un ambiente. Dunque da mossa deriva direttamente la mossa *stato*, che è caratterizzata dal modificare un insieme di statistiche di una certa percentuale. Questi dati sono ovviamente salvati come degli attributi ed utilizzati dalle implementazioni di usa. Visto che esistono delle mosse che incrementano le statistiche del mostro che la utilizza, le mosse stato forniscono un attributo chiamato "self", che se settato a true esegue la

mossa sul mostro stesso, senza necessitare di bersagli. Dall'altro lato abbiamo invece un'altra classe astratta pura, *danno*, che introduce due nuovi attributi, potenza e area. La potenza influisce nel calcolo dei danni da mossa, mentre area è un booleano che stabilisce se la mossa colpirà solo il primo bersaglio che troverà o tutti quelli presenti. Le due concretizzazioni di danno sono *danno fisico e danno speciale*. Danno fisico calcola il danno utilizzando l'attacco e la difesa dei mostri, mentre danno speciale usa attacco e difesa speciali, come suggerisce il nome stesso. Inoltre danno fisico può comportare un danno collaterale al mostro attaccante, mentre danno speciale comporta un consumo di energia.

Eccezioni

E' presente anche una gerarchia di eccezioni, relative alle classi sopra nominate. La classe base è *Eccezione*, che mette a disposizione un costruttore ed un metodo di stampa del messaggio di errore salvato. Tutte le eccezione derivate eventualmente modificano il formato del messaggio tornato dal metodo *getMessaggio*. E' presente anche una classe di eccezioni dedicate agli errori logici commessi dall'utente, *EccezioneGestione,* che segnala eventuali errori e ne suggerisce l'adeguato utilizzo.

Classi grafiche

Abbiamo costruito diverse classi per implementare al meglio la view:

- la classe base pop-up, da cui derivano tutti i pop-up specializzati per la creazione dei diversi mostri;
- display, che implementa gli schermi;
- type-button e image-button che implementano dei particolari bottoni;
- alcuni widget che raccolgono bottoni o immagini usati in diversi pop-up come *widget-formato, widget-gusto, widget-stats* e *widget-tipo*;
- il widget slider-value che viene usato per gestire l'inserimento delle diverse caratteristiche dell'oggetto.

Polimorfismo

Polimorfismo nella gerarchia

Svariate funzione sono polimorfe:

- le due funzioni virtuali di oggetto, clone (che sfrutta anche la covarianza nel tipo di ritorno) e toMap sono state sfruttate nell'implementazione del pattern MVC. Nel modello tutti gli

oggetti vengono salvati in un vettore di Oggetto*. Laddove sia necessario creare una copia o stampare i dati relativi all'oggetto per rappresentarlo graficamente, non è necessario fare il type check;

- in bacca accettiamo una qualunque Cura sulla funzione Cucina, e dunque distinguiamo due diversi casi a seconda che la cura sia uno strumento o una bacca;
- il metodo virtuale esponi permette una chiamata polimorfa per distinguere l'esposizione di una Bacca da una Caramella, sua classe derivata;
- evolvi di mostro accetta una qualsiasi pietra*, e poi esegue operazioni differenti a seconda che sia una pietra o una mega-pietra;
- usa di cura permette di usare una cura qualsiasi su un mostro, essendo virtuale la funzione chiamata sarà quella del tipo dell'oggetto considerato;
- il metodo virtuale esegui, di mossa, permette di eseguire una mossa senza sapere di che tipo di mossa si tratti;
- il metodo spremi di bacca e caramella non è virtuale in quanto i tipi ritornati non sono compatibili tra di loro.

Polimorfismo nella parte grafica

La classe display, che usiamo tre volte nella nostra view (nella schermata di calcolo, nella lista dei risultati e in quella degli oggetti salvati), accetta dei DisplayObject*. Nel display principale sono usati nella loro forma base (widget definito per rappresentare gli oggetti), mentre negli altri due display vengono usati dei DisplayObjectButton*, una classe derivata a cui sono stati aggiunti dei pulsanti per la gestione.

Polimorfismo nelle eccezioni

La funzione virtuale getMessaggio di Eccezioni permette di stampare il messaggio personalizzando eventualmente l'errore da mostrare a schermo.

Comandi per l'esecuzione

Per eseguire la nostra applicazione, dalla cartella del progetto eseguire i seguenti comandi:

1. qmake monsterKalk.pro
2. make
3. ./monsterKalk

Risoluzione consigliata: 1920x1080px

Ambiente di sviluppo

- Editor: Qt 5.11.0
- Qt Creator: 4.6.1
- SO: Linux Manjaro Deepin 17.1.10
- Compilatore: gcc 8.1.0
- Eclipse: Oxygen.3a Release (4.7.3a)

Guida per l'uso

MonserKalk è suddivisa in sei blocchi. Al centro è presente il display principale, attraverso il quale gli oggetti vengono inseriti e dove vengono svolte le operazioni. I risultati delle operazioni vengono mostrati su un secondo display sottostante, ed ogni oggetto risultato è accompagnato da due bottoni che permettono di salvare l'oggetto o di utilizzarlo nuovamente ponendolo sul display. Gli oggetti salvati vengono mostrati sul display in basso, dove ogni oggetto ha la possibilità di essere rimesso sul display o di essere rimosso.

A sinistra sono presenti i bottoni che servono per creare gli oggetti su cui monsterKalk può svolgere le operazioni, i cui pulsanti sono localizzati sulla parte destra della finestra.

Appena sotto le funzioni ci sono tre pulsanti di gestione, annulla, reset e conferma.

Gli oggetti vengono creati tramite degli appositi pop-up che compiano una volta premuto il relativo pulsante. Una volta selezionati i parametri e cliccato conferma l'oggetto viene inserito nel display principale. Le funzioni disponibili per ogni set di oggetti sono evidenziate da alcune etichette che rendono più semplice l'utilizzo.

Per effettuare ogni operazione è necessario scegliere gli operandi dell'operazione e l'operazione da eseguire, dunque premere conferma. Se il pulsante conferma viene premuto senza che nessuna operazione sia stata inserita, allora la calcolatrice stamperà tra i risultati tutti gli oggetti nel display, funzione che è particolarmente utile nelle fasi iniziali se abbinata al salvataggio degli oggetti nella pila dei salvati attraverso l'apposito pulsante.

Il pulsante "indietro" elimina l'ultimo oggetto inserito nel display, sia nel caso esso sia un operando sia che esso sia un'operazione. Non ha alcun effetto se applicato a display vuoto, dopo un messaggio di errore e dopo che si è appena conclusa un'operazione.

Il pulsante reset riporta allo stato iniziale la calcolatrice.

Le istruzioni particolari su come devono essere usate le diverse operazioni sono descritte dai messaggi che compaiono passandoci sopra con il mouse. Inoltre dei messaggi di errore vengono mostrati a schermo dopo l'errato utilizzo di una funzione, spiegandone il corretto utilizzo.

Dopo che si è premuto conferma, se nei risultati è presente un solo oggetto e si seleziona un'operazione, l'oggetto risultato verrà automaticamente posto come primo operando.

Suddivisione del lavoro progettuale

Complessivamente il lavoro è stato suddiviso un modo equo, e molte parti sono state svolte assieme, specie nella stesura della gerarchia iniziale. C'è stata una suddivisione della stesura delle classi della gerarchia: io ho implementato il ramo delle cure, pietra e megapietra, e dunque collaborato con Gianluca per la stesura delle due classi Ambiente e Mostro.

Successivamente io mi sono concentrato maggiormente nella gestione del model, controller, e del loro collegamento alle parti grafiche (gestione di tutti i SIGNAL e SLOT), mentre Gianluca si è dedicato principalmente alla creazione della parte grafica.

Model

Il model è gestito attraverso tre vector usati come pile, uno per gli oggetti attuali, ovvero quelli nello schermo della calcolatrice, uno per gli oggetti risultato dell'operazione e uno per gli oggetti salvati. Tutte le operazioni di calcolo avvengono tra oggetti presenti sulla pila degli attuali, usata come FIFO, tutti gli oggetti prodotti dalle operazioni vengono inseriti nella pila dei risultati, mentre solo gli oggetti che l'utente desidera riutilizzare vengono inseriti nei salvati.

View

La View è stata descritta in precedenza durante la descrizione generale. Ogni pulsante è collegato tramite SIGNAL e SLOT a delle funzioni del controller che settano lo stato di attività dei bottoni e garantiscono la robustezza delle operazioni.

Controller

Il controller si occupa di gestire la comunicazione tra model e view. Esso ospita tre mappe che associano ogni elemento presente nei tre vettori del model ad una QPixmap, ovvero un'immagine collegata agli oggetti e mostrata nella parte grafica per rendere l'uso più semplice e piacevole. E'

stato deciso di non mettere le QPixmap nel model in quanto strettamente legate a questa view, inutili ad esempio in un contesto come l'uso da terminale. Inoltre il controller contiene una mappa di flag che servono a gestire i bottoni e lo stato del programma. Laddove un pulsante viene premuto, la flag relativa viene settata a true nel controller, e verrà utilizzata per capire che operazione deve essere svolta una volta cliccato il pulsante di conferma che va a chiamare la funzione che restituisce i risultati, *toResult.* In particolare viene usata una flag, chiamata finale, per capire se un inserimento avviene in stato di risultato: se avviene un inserimento in questo punto allora il display principale viene ripulito, come tutte le pile degli oggetti attuali. Laddove fosse presente un solo elemento tra i risultati, questo viene considerato come primo elemento nell'operazione successiva, per marcare la possibilità di continuare a svolgere dei calcoli sui risultati senza interruzione.

Ore utilizzate

	Gianluca		Samuele	
Attività	Studio	Lavoro	Studio	Lavoro
Ideazione e prima progettazione		8		8
Stesura codice individuale		12		10
Unione codice e debugging		7		9
Studio e implementazione pattern MVC		2	14	27
Studio e implementazione grafica	20	30		3
Test		2		2
Traduzione in java	1	4	1	4
Totale	**21**	**65**	**15**	**63**

Le ore in eccesso sono causate dallo studio della libreria Qt e dall'implementazione della parte grafica e della struttura MVC. Molte problemi si sono rivelati di non semplice risoluzione ed hanno richiesto molto tempo per la ricerca di una soluzione e/o test in ambiente differente da quello progettuale. Successivamente abbiamo voluto apportare delle modifiche al progetto, in particolare alla parte grafica, per rendere più piacevole l'utilizzo.