RELAZIONE PROGETTO PROGRAMMAZIONE AD OGGETTI

ANNO ACCADEMICO 2017/2018

PROGETTO: DotaKalk

Progetto svolto dalla seguente coppia:

STUDENTE: Riccardo Dario MATRICOLA: 1123773
STUDENTE: Enrico Trinco MATRICOLA: 1121850

INDICE

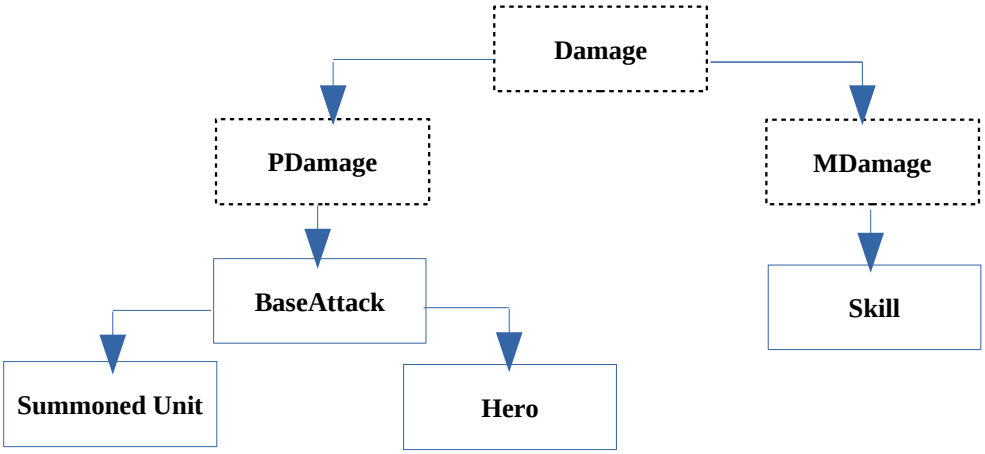
ABSTRACT:

DotaKalk nasce con lo scopo di offrire una calcolatrice che permetta agli utilizzatori di effettuare operazioni di calcolo attraverso dei tipi di dato ispirati dal videogioco DOTA 2.
In ogni calcolatrice offerta, l'utente troverà una scelta di operazioni che solitamente vengono effettuate dai giocatori per capire come utilizzare al meglio un determinato eroe o abilità.

I tipi di dato supportati da questa calcolatrice sono:
- BaseAttack: rappresentazione degli attacchi base;
- Skill: rappresentazione di una singola abilità
- Hero: rappresentazione di un singolo eroe
- Summoned Unit: rappresentazione di un unità diversa dall'eroe e solitamente evocata.

GERARCHIA E DESCRIZIONE CLASSI

La gerarchia utilizzata nel modello è la seguente



CLASSI ASTRATTE:

- Damage: Classe astratta la quale offre dei metodi virtuali puri che verranno utilizzati dalle classi concrete;
- PDdamage e MDamage: queste due classi astratte sono utlizzate allo solo scopo di differenziare i due tipi di danno che andranno ad essere trattati (P = Phisical e M = Magical). Possibile estensione: danno puro

CLASSI CONCRETE:

- BaseAttack: classe concreta che rappresenta un attacco fisico derivata da PDamage. Un base attack è composto da 3 elementi: danno (ereditato da Damage), animatione, raggio d'azione e velocità d'attacco. Abbiamo deciso di non dare un nome agli attacchi all'interno della gerarchia poiché il nome non è una carateristica caraterizzante di tale tipo.
  METODI IMPLEMENTATI
  La classe implementa i metodi virtuali puri della classe Damage e ridefinisce gli operatori di somma, differenza e uguaglianza. Infine sono presenti i metodi get per l'ottenimento dei dati, utilizzabili dalle classi esterne a BaseAttack. I metodi implementati sono:
  - bool IsCastable(double time,unsigned int distance): ritorna un booleano dopo aver controllato che l'attacco base sia utilizzabile con i valori di tempo e distanza passati;
  - unsigned int HitByTime(double time,unsigned int distance): ritorna il numero di colpi che l'attacco fa in un certo lasso di tempo e da una determinata distanza;
  - double DPS(unsigned int distance): ritorna un double che equivale al danno effettuato in un secondo da un attacco base;
  - double DamageByTime(double time,unsigned int distance): ritorna un double con il valore del danno fatto in un determinato tempo e da una determinata distanza dell'attacco base;

    - Damage* sum(Damage*s): funzione di somma polimorfa che somma il danno e prende l'animazione più alta tra i due che è il tempo perché entrambi colpiscano;
    - BaseAttack* clone()const: ritorna una copia profonda del BaseAttack chiamante.
- Skill: classe concreta che rappresenta un' abilità utilizzabile derivata da MDamage. Una skill è composta da 8 elementi: danno (ereditato da Damage), nome, animazione, raggio d'azione, velocità, costo in mana, tempo di ricarica e livello dell'abilità;
  METODI
  In questa classe vengono implementati i metodi virtuali puri presenti in Damage quali operatori e metodi per il calcolo dei danni .I metodi implementati sono:
    - bool IsCastable(double time,unsigned int distance): ritorna un booleano dopo aver controllato che la Skill sia utilizzabile con i valori di tempo e distanza passati;
    - double totalTime(): ritorna un double che corrisponde a quanto una Skill ci mette dal momento in cui viene utilizzata fino al momento in cui colpisce l'eroe nemico;
    - double DPS(unsigned int distance): ritorna un double che equivale al danno effettuato in un secondo dalla Skill;
    - double DamageByTime(double time,unsigned int distance): ritorna un double con il valore del danno fatto in un determinato tempo e da una determinata distanza dalla Skill chiamante;
    - bool Heal(): ritorna vero se la Skill chiamante è una cura e non un danno;
    - void EditedValues(…): aggiorna tutti i campi di una skill(funzioni edit GUI);
    - Damage* sum(Damage*s): funzione di somma polimorfa che somma danno, tempo di animazione e mana di due skill.
- Hero: classe figlia di Baseattack che rappresenta un eroe , ossia il personaggio principale del gioco nonché classe più importante della gerarchia.
  E' caratterizzato da:
    - BaseAttack;
    - Nome;
    - Attributi di Forza (Str), Agilità (Agl) e Intelligenza (Int);
    - Salute base: la salute completa si basa sulla salute base e sulla forza dell'eroe;
    - Mana base: il mana totale si basa sul mana base e sull'intelligenza dell'eroe;
    - Armatura base: l'armatura totale si basa sull'armatura base e sull'agilità dell'eroe;
    - Resistenza magica;
    - Skills: vettore che memorizza le abilità possedute dall'eroe
    - Livello: il livello influenza le abilità che l'eroe può possedere.

    METODI IMPLEMENTATI:
    - GetArmor(),GetHP(),GetMP():calcolano i valori di armatura,vita e mana dell'eroe in base agli attributi;
    - getLongestReadySkill(..): ritorna il valore dell'animation più lunga tra le skill dell'eroe in base alla distanza;
    - vector<Damage*> MaxDamageByTime(double time,unsigned int mana): restituisce un vector con le skill e gli attacchi usati per massimizzare il danno in quel lasso di tempo, tenendo conto anche del mana.
    - double MaxDamageByTimeDMG(double time,unsigned int mana): restituisce un damage che corrisponde al totale di tutti i Damage* presenti nel vettore risultante dalla chiamata MaxDamageByTime(time,mana);
    - char Fight(Hero* h): fa scontrare due eroi, facendo vincere chi per primo uccide l'altro. Per ogni tick di tempo(0.2 secondi) si controlla se uno dei due eroi fa più danno della vita dell'altro(con dovute riduzioni basate su armatura e resistenza magica) e ritorna 'w' se vince il chiamante,'l' se vince l'eroe passato,altrimenti 't' se è un pareggio;
    - vector<Damage*> MaxPower(double time,unsigned int distance): metodo che memorizza tutte le fonti di danno di un eroe in un vector di Damage* e fa chiamate polimorfe ai metodi DPS() e DamageByTime() per vedere quale tra questi ha il DPS più alto e quale il DBT(DamageByTime) più alto. Memorizzati quali sono inserisce una loro copia rispettivamente in posizione 0 e 1 di un vector di Damage* e lo ritorna;
    - void InsertSkill(Skill*s): inserisce una copia della Skill puntata da s nelle skill dell'eroe;
    - Skill getSkill(unsigned int i): ritorna la skill puntata dal puntatore iesimo del vettore skills di eroe;
    - void resetSkill(): elimina tutte le skill puntate dal vettore skills di eroe;
    - void ~Hero(): ridefinizione del distruttore di hero.

- Summoned Unit: classe concreta derivata da BaseAttack che rappresenta un'unità evocata caratterizzata da nome, HP (Health point) ,livello e moltiplicatore(per moltiplicatore si intende il valore da moltiplicare per il livello per aumentare hp e danno dell'unità stessa, all'aumentare del livello).

METODI IMPLEMENTATI:

- double DPS(unsigned int distance): utilizza il metodo DPS(distance)di BaseAttack incrementando il valore di ritorno utilizzando i campi di SummonedUnit;
- double DamageByTime(unsigned int distance): utilizza il metodo DamageByTime(distance)di BaseAttack incrementando il valore di ritorno utilizzando i campi di SummonedUnit;
- char Fight(SummonedUnit*s): simile a Fight di Hero ma semplificato dal fatto che SummonedUnit non possiede nessuna Skill e non ha armor o resistenza fisica.

UTILIZZO DEL POLIMORFISMO

I metodi della classe Damage sono tutti virtuali(fatta eccezione per getters e setters), e vengono implementati in modo diverso nelle classi derivate, in modo da poter effettuare chiamate polimorfe. Questi metodi sono:

- `virtual double DPS(unsigned int);`
- `virtual double DamageByTime(double, unsigned int);`
- `virtual unsigned int HitByTime(double ,unsigned int);`
- `virtual bool IsCastable();`
- `virtual Damage* sum(Damage *);`

Inoltre vengono forniti gli operatori di somma, differenza e ugualianza in forma virtuale.
Questi metodi verranno ereditati ed implementati successivamente dalle classi BaseAttack, Skill e in parte da Summoned Unit.

Inoltre, nella classe Hero il metodo `std::vector<Damage*> Hero::MaxDamageByTime` utilizza le due chiamate polimorfe:

- `DPS(unsigned int);`
- `DamageByTime(double, unsigned int);`

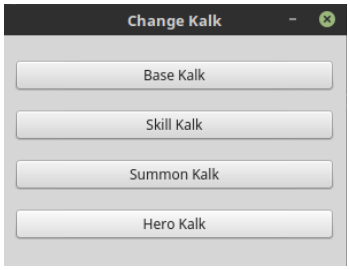
GUI: DESCRIZIONE E MANUALE

La GUI è stata realizzata per la maggior parte con codice scritto a mano. Una leggera parte è stata creata con QtCreator in modo da rendere le calcolatrici più presentabili e rendere più intuitivo il loro utilizzo.

Ogni finestra deriva da Qwidget, QMainWindow o da QDialog e non presente nessun tipo di gerarchia.

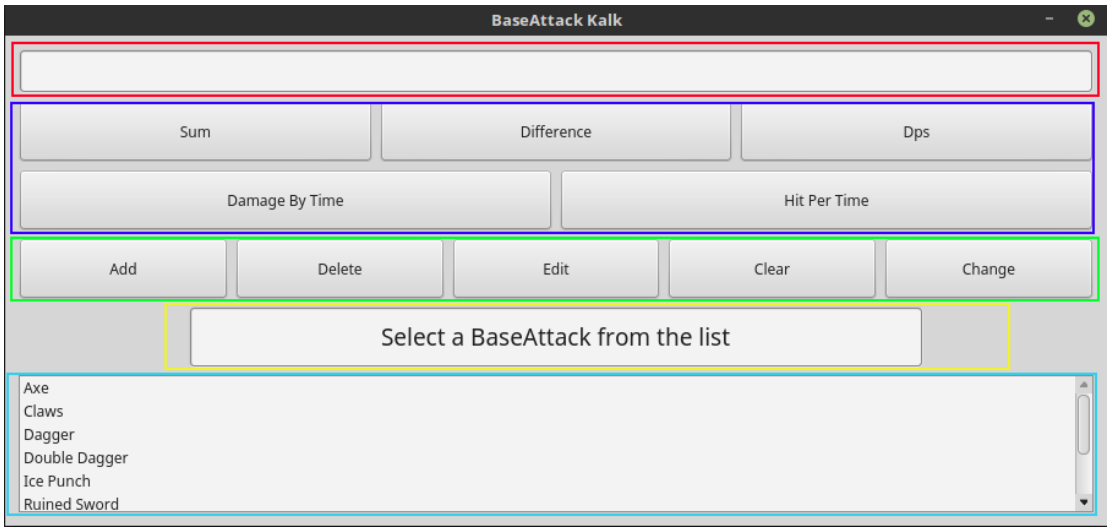
La GUI fa inoltre dei controlli sulla consistenza dei dati inseriti in modo da evitare qualsiasi tipo di problema a livello di codice (proprio per questo non è stata creata una classe per gestire le eccezioni).

FINESTRA INIZIALE:



- schermata "Change Kalk" -

L'utente all'avvio del programma si troverà davanti ad una finestra che permetterà la scelta di una delle 4 calcolatrici disponibili. Appena verrà premuto uno dei bottoni verrà aperta la calcolatrice corrispondente.



- schermata Base Kalk -

Ogni calcolatrice presenta:

- *Display per il risultato*: ogni volta che un'operazione verrà eseguita con successo, il risultato sarà mostrato sulla barra (colore rosso)

- *Operazioni di modifica*: aggiunta, edit, rimozione, pulizia del display e cambio di calcolatrice.
Il pulsante *Edit*, oltre alla modifica dell'elemento selezionato, permette anche di vederne tutte le informazioni che la caratterizzano (colore verde).

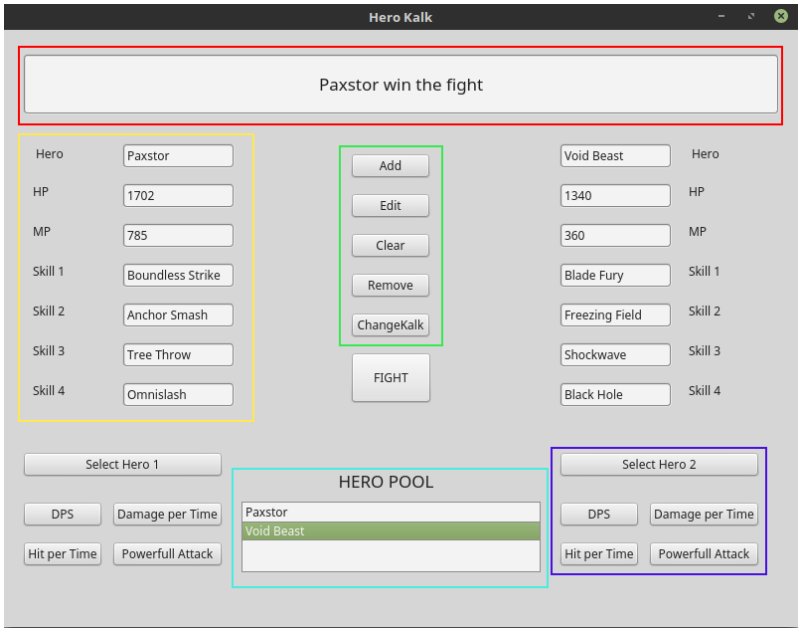
- *Operazioni unarie e binarie*: permettono di effettuare calcoli sugli elementi selezionati.Per alcune operazioni serve l'inserimento di un tempo e viene utilizzata una finestra popup utile a tale scopo. Invece quando si vorrà effettuare un'operazione binaria, appena verrà selezionato l'elemento e successivamente l'operazione, verrà visualizzato un pop-up informativo per indicare la selezione del secondo elemento per effettuare correttamente il calcolo (colore blu).

- *Display informativo*: riporta i seguenti casi: selezione di un elemento, avvenuto inserimento di un eroe, messaggio per inviare la selezione o creazione di un elemento (colore giallo)

- *Pool degli elementi*: contenitore che permette la selezione dell'elemento sul quale eseguire un calcolo. Dopo una modifica all'elemento (aggiunta, rimozione, modifica) la lista viene aggiornata automaticamente (colore azzurro).

Summoned Kalk ed Hero Kalk presentano un layout diverso da quello di Base Kalk e Skill Kalk. Abbiamo deciso di variarlo data la quantità d'informazioni richieste da Hero e per rendere visivamente migliore la selezione di due

elementi nel caso dopo si debba effetturare l'operazione "*FIGHT*". L'unica differenza che presentano queste due calcolatrici è la presenza di spazi dove verranno inserite le caratteristiche degli elementi che verranno selezonati (colore giallo).



- Schermata "HeroKalk"-

SUDDIVISIONE LAVORO

La progettazione delle classi e la prima parte di modellazione in c++ è stata eseguita in concomitanza da entrambe le parti. In seguito abbiamo deciso di suddivedere il lavoro in questo modo:

Studente Riccardo Dario:

- codifica del modello
- codifica in java
- debug model
- revisione finale grafica

Studente Enrico Trinco:

- studio e apprendimento della libreria grafica di Qt
- codifica della parte grafica
- debug della parte grafica
- revisione finale modello

Infine le fasi di testing sono state effettuate assieme per verificare se il progetto necessitava di ulteriori aggiunte.

ORE RICHIESTE

Enrico Trinco: 49 ore
- Progettazione: 4 ore;
- Scrittura codice gerarchia tipi: 1 ora;
- Studio individuale della libreria grafica di QT (con esempi): 20 ore;
• Scrittura codice parte grafica (incluse anche le fasi di debug con correzione del codice): 20 ore;
• Scrittura relazione: 2 ore;
• Test: 2 ore;

Riccardo Dario:50 ore
-Progettazione: 4 ore;
-Scrittura scheletro gerarchia:1 ora;
-modellazione classi c++: 20 ore;
-modellazione classi java: 10 ore;
-revisione e utilizzo polimorfismo:3 ore;
-compilazione c++:3 ore;
-compilazione java:2 ore;
-revisione parte grafica:1 ora;
-debug:4 ore;
-stesura relazione:2 ore;

AMBIENTE DI SVILUPPO

Il progetto è stato effettuato sulle macchine personali del gruppo.

PC Enrico Trinco:
Sistema operativo: Mint 18.3
Compilatore: gcc version 5.4.0 20160609
QT: 5.1.1

PC Riccardo Dario:
Sistema operativo: Kubuntu 17.10
Compilatore: gcc 7.2.0
QT: 5.11

Il progetto è stato testato sulle macchine del laboratorio.

ISTRUZIONI PER COMPILARE

Versione c++
Per la compilazione in c++ viene già fornito il file .pro

1. Aprire il terminale
2. Spostarsi nella directory principale del progetto la quale contiene una cartella MODEL, GUI e il file .pro
3. eseguire da terminale **qmake**
4. eseguire da terminale **make**
5. eseguire da terminale **./DotaKalk**

Versione Java
1. aprire il terminale
2. spostarsi nella directory del progetto nel quale sono salvati i file java (MODEL/Java)
3. eseguire da terminale **Javac Use.java**
4. eseguire da terminale **java Use**

VALORI DI DEFAULT
Per facilitare il testing della calcolatrice sono stati inseriti dei valori preimpostati commentati nella classe changewindow.cpp, basta togliere il flag del commento e i valori compariranno in basso, pronti per essere utilizzati. Per vedere come sono impostati basta selezionarli e utilizzare edit nella finestra corrispondente.