# RELAZIONE PROGETTO P2

ANNO ACCADEMICO 2017/2018

*PROGETTO: DotaKalk*


Progetto svolto dalla seguente coppia:

**STUDENTE: Enrico Trinco MATRICOLA: 1121850**

STUDENTE: Riccardo Dario MATRICOLA: 1123773

## INDICE

## 1. ABSTRACT:

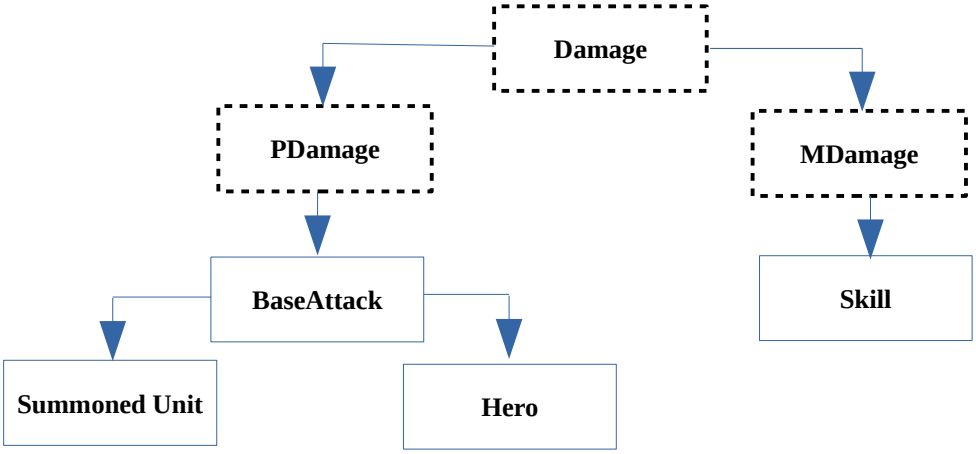
DotaKalk nasce con lo scopo di offrire una calcolatrice che permetta agli utilizzatori di effettuare operazioni di calcolo attraverso dei tipi di dato ispirati dal videogioco Dota2.
Tramite questa applicazione, l'utente potrà creare i dati che gli servono ed avrà accesso ad una scelta di operazioni che 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.

## 2. MODELLO: GERARCHIA E DESCRIZIONE CLASSI

Il modello contiene la parte logica del progetto, ossia la definizione delle classi e implementazione dei metodi.

La gerarchia adottata nel modello è la seguente




In questo modo è possibile estendere il modello con la creazione di nuovi tipi di danno o derivati (come ad il danno puro oppure l'inserimento degli oggetti come strumento da danno magico).

***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 state create allo solo scopo di differenziare i due tipi di danno che andranno ad essere trattati (P = Physical e M = Magical).

***CLASSI CONCRETE:***

- *BaseAttack*: classe concreta che rappresenta un attacco base con danno fisico. Un BaseAttack è composto da 3 elementi: danno (ereditato da Damage), animazione, raggio d'azione e velocità d'attacco.

  *METODI*
  Nella classe sono presenti i metodi virtuali ereditati da "Damage" per il calcolo dei danni e dei colpi eseguiti. Presenta poi le ridefinizioni degli operatori di somma, differenza e uguaglianza. Infine sono presenti i metodi get per l'ottenimento dei dati dalle classi che non derivano da BaseAttack.

- *Skill*: classe concreta che rappresenta un'abilità utilizzabile con classificazione magica. Una skill è composta da 8 elementi: danno (ereditato da Damage), nome, animazione, raggio d'azione, velocità, mana (costo per utilizzare l'abilità), tempo di ricarica e il livello dell'abilità;

  M*ETODI*
  Anche in questa classe sono presenti i metodi virtuali ereditati da "Damage" per il calcolo dei danni, dei colpi eseguiti e le ridefinizioni degli operatori. Presenta inoltre dei metodi "IsCastable" che restituiscono un valore boolean e indicano se è possibile eseguire o meno l'abilità.

- *Hero*: classe figlia di Baseattack che rappresenta un eroe , ossia il personaggio principale utilizzato nelle partite.
  E' caratterizzato da:
  - BaseAttack;
  - Nome
  - Tre diversi attributi: Forza (Str), Agilità (Agl) e Intelligenza (Int);
  - Salute base: la salute completa (HP = health point) si basa sulla salute base e sulla forza dell'eroe;
  - Mana base: il mana totale (MP = mana point) 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*:
  Presenta due metodi che permettono di calcolare il danno massimo e il massimo numero di colpi che un eroe riesce ad effettuare in un determinato arco di tempo, tenendo in considerazione sia il base attack che le skills possedute dall'eroe.
  Viene implementato poi il metodo `char Fight(Hero* s):` dati due eroi effettua una simulazione di uno scontro fra i due, valutando ogni 0.2 secondi (o tick) se gli eroi posso o meno fare un'azione contro l'avversario. Alla fine dello scontro verrà restituito una lettera che riporta l'esito dello scontro (vittoria dell'eroe chiamante, sconfitta dell'eroe chiamante oppure pareggio)
  Infine è stato implementato il metodo `std::vector<Damage*> MaxPower:` questo metodo memorizza tutte le fonti di danno di un eroe in un vector di Damage* ed effettua due chiamate ai metodi polimorfi DPS e DBT. Restituisce alla fine un vettore che avrà come primo valore il DPS e come secondo valore DBT.

- *Summoned Unit*: classe concreta che rappresenta un'unità diversa da un eroe, caratterizzata da un nome, una salute (HP = Health point), un attacco base e un moltiplicatore (valore da inserire per adattare la vita e l'attacco al livello).

  *METODI:*
  Presenta anch'essa due metodi per il calcolo dei danni che effettua in base ai dati inseriti e un metodo Fight per la simulazione di uno scontro tra le due unità.

## 3. UTILIZZO DEL POLIMORFISMO

Il progetto possiede nella classe Damage alcuni metodi virtuali, che vengono implementati in modo diverso nelle classi derivate, in modo da avere del codice polimorfo. Questi metodi sono:

- `virtual double DPS(unsigned int);`
- `virtual double DamageByTime(double, unsigned int);`
- `virtual unsigned int HitByTime(double ,unsigned int);`
- `virtual double operator+(Damage*) const;`
- `virtual double operator-(Damage*) const;`
- `virtual bool operator==(const Damage&) const;`
- `virtual bool IsCastable(double,unsigned int distance=0);`
- `virtual Damage* sum(Damage*);`

Questi metodi verranno ereditati ed implementati successivamente dalle classi BaseAttack, Skill e 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);`

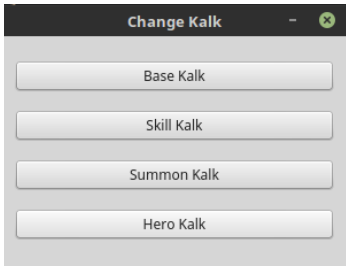
## 4. GUI: DESCRIZIONE E MANUALE

Per la codifica della GUI è stato usato del codice scritto a mano per la maggior parte delle classi. Una leggera parte è stata creata con QtCreator in modo da migliorare il posizionamento dei bottoni e renderle visivamente migliori.

Tutte le finestre derivano da Qwidget, QMainWindow o da QDialog.

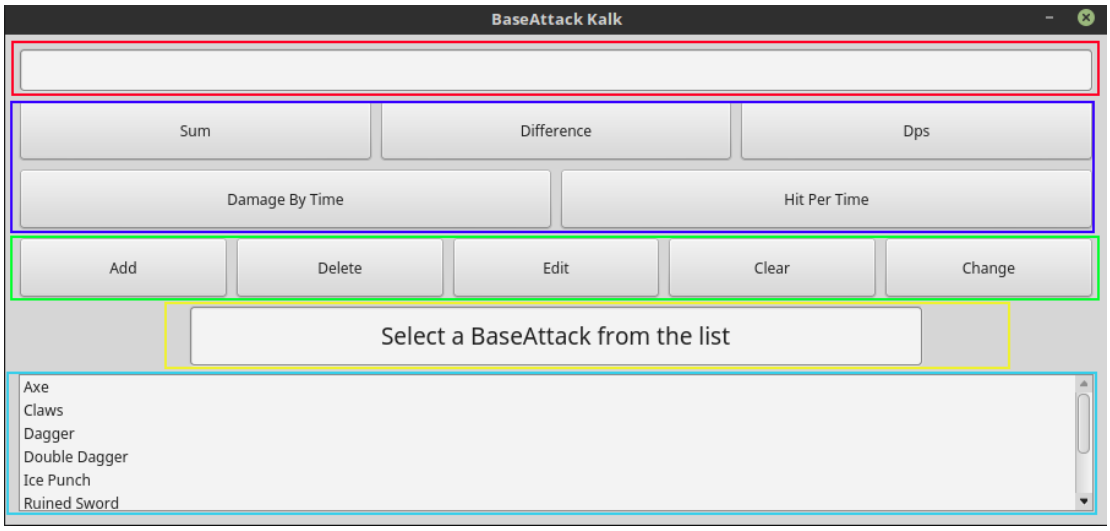
Inoltre la GUI è stata realizzata in modo da evitare che durante l'esecuzione del codice si presentino dei problemi. Ciò è stato fatto attraverso l'utilizzo di:
- *QValidator* (sia Int che Double): creati in modo che l'utente inserisca dati che rientrano nei parametri ufficiali del gioco, come il livello massimo dell'eroe (fissato a 25), oppure per evitare che usi valori inappropriati, come per esempio l'inserimento di un'abilità che ha velocità (projectile speed) pari a 0 oppure l'inserimento di un range negativo.
- *QMessageBox*: utilizzati per indicare all'utente eventuali errori come la rimozione di un elemento inesistente o per dare delle informazioni, come la creazione di un abilità al livello 0 (possibile ma non utilizzabile).

***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)

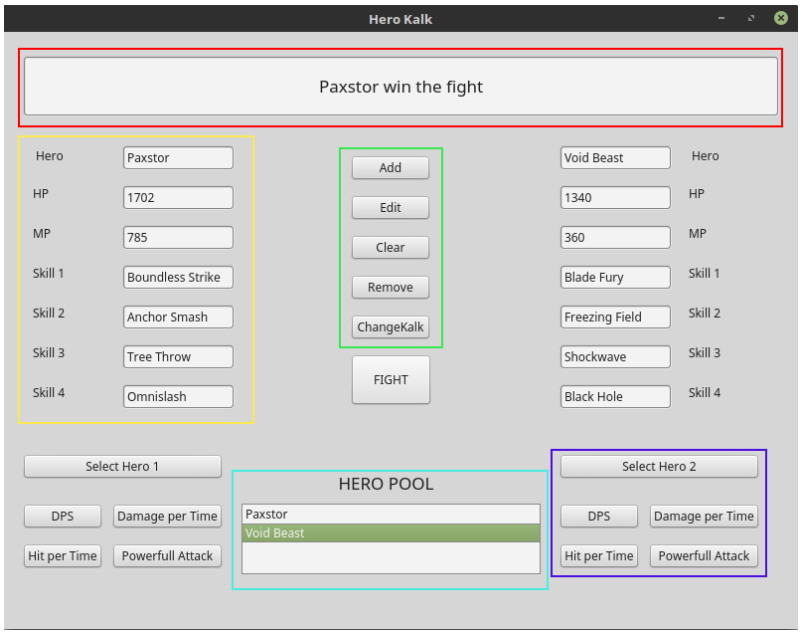
- *Operazioni di modifica*: permettono di eseguire operazioni di aggiunta, modifica , rimozione, pulizia del display e cambio di calcolatrice.
Il pulsante *Edit*, oltre alla modifica dell'elemento selezionato, permette anche di vederne tutte le informazioni che la caratterizzano (colore verde).

- *Operazioni unarie e binarie*: permettono di effettuare calcoli sugli elementi selezionati. 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 invitare 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.

*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 effettuare 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 selezionati (colore giallo).




*- Schermata "HeroKalk"-*

## 5. SUDDIVISIONE LAVORO

Nelle fasi iniziali di analisi del problema, progettazione del modello, della GUI e sulle prime fasi di codifica del modello il gruppo ha lavorato a stretto contatto in modo che entrambi fossimo d'accordo sulla strada che il progetto avrebbe intrapreso.
Successivamente ci siamo suddivisi il lavoro nel seguente modo:

Studente Enrico Trinco:
- studio e apprendimento della libreria grafica di Qt;
- codifica della parte grafica;
- debug della parte grafica;
- revisione finale modello

Studente Riccardo Dario:
- codifica del modello
- codifica in java
- revisione finale grafica

Infine le fasi di testing sono state effettuate insieme in modo da verificare se il progetto necessitava di ulteriori aggiunte.

## 6. 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: ore 50
- Progettazione: 4 ore;
- Scrittura codice gerarchia tipi: 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;
- Scrittura relazione: 2 ore;

## 7. AMBIENTE DI SVILUPPO

Il progetto è stato codificato sulle macchine personali del gruppo.

PC Enrico Trinco:
Sistema operativo: Mint 18.3
Risoluzione schermo: 1920x1080
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

Alla fine il progetto è stato testato anche sulle macchine del laboratorio: sui costruttori delle calcolatrici è stato fatto uso di QRect in quanto la risoluzione sul quale è stata codificata la GUI è maggiore rispetto a quella usata nei computer del laboratorio e dava problemi nel posizionamento delle finestre.
Per quanto riguarda la compilazione non è stato riscontrato nessun problema e il programma compila e viene eseguito correttamente.

## 8. 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:
Essendo una calcolatrice, di default NON presenta elementi inserti. Per velocizzare i test sui calcoli sono stati inseriti alcuni elementi preimpostati, quali Skill, BaseAttack e Summoned Unit.
Abbiamo deciso di lasciarli commentati in modo da offrire la possibilità al programmatore di inserirli o meno. In questo modo si può dare all'utente di già dei valori di prova.
Per utilizzarli basta togliere il commento sul costruttore nel file "changewindow.cpp".