Relazione Progetto "Kalk"

Federico Omodei, mat. 1126500

28/08/2018

Progetto di Programmazione ad oggetti A.A. 2018-2019 a cura di Federico Omodei e Luca Violato.

Indice

1 Introduzione

1.1 Comandi di compilazione

La compilazione del progetto richiede un file di progetto (.pro) differente da quello "standard" ottenibile dall'invocazione del comando *qmake -project*. Il file DarkCalc.pro corretto è già fornito nella consegna[1], è dunque sufficiente assicurarsi della sua effettiva presenza dentro la directory "Codice C++/" ed invocare in sequenza i comandi
qmake
make

1.2 Ambiente di sviluppo

- Sistema Operativo: Manjaro Linux
- Compilatore c++: GCC 8.1
- Compilatore Java: javac 9.0
- Libreria Qt: 5.11
- Editor: Geany 1.33
- IDE: Qt Creator
- Version Control: git
- Repository hosting service: https://gitlab.com
- Testing: macchina virtuale OS del laboratorio su VirtualBox

1.3 Descrizione introduttiva del progetto

La calcolatrice sviluppata ha l'obiettivo di aiutare i giocatori di videogames di ruolo nei calcoli che coinvolgono congiuntamente statistiche del personaggio ed equipaggiamenti in suo possesso. La calcolatrice permette dunque al giocatore di variare a piacimento i parametri d'interesse per vederne i risultati finali sul personaggio e di confrontare, ad esempio, l'efficacia di un'arma su un'armatura e viceversa. La calcolatrice è liberamente ispirata alla saga videoludica di Dark Souls e permette di raggiungere lo sviluppo del personaggio (e relativo equipaggiamento) in modo coerente alle proprie aspirazioni senza dover per forza provare le varie configurazioni dal gioco vero e proprio.

[1] In caso vi sia la necessità di generarlo nuovamente, è sufficiente l'invocazione del comando *qmake -project "QT+=widgets" "CONFIG+=c++11"*

2 Gerarchia dei tipi

La gerarchia della logica del programma è composta da 8 classi, disposte su 3 livelli di derivazione (sempre di tipo pubblico) e senza dichiarazioni di amicizia da segnalare. La classe *Caratteristiche* è a sé stante, ovvero concreta e senza ereditarietà di alcun tipo. La classe *Equipaggiamento* invece è astratta ed è alla base della gerarchia da cui derivano pubblicamente le classi concrete *Arma e Armamento* che danno un primo "livello di specializzazione". Da Arma e Armamento infatti derivano rispettivamente *ArmaFisica ed ArmaMagica* (da Arma) e *Armatura e Scudo* (da Armamento), andando a formare un secondo livello di specializzazione.

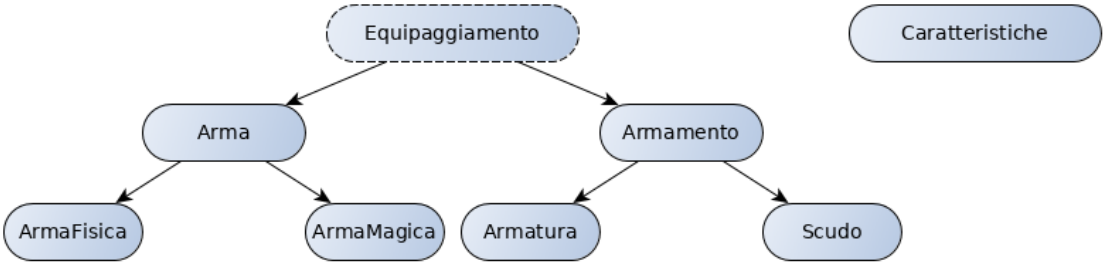


Figura 1: Rappresentazione grafica della gerarchia dei tipi utilizzati.

2.1 Classe Caratteristiche

Nonostante questa classe non presenti ereditarietà di alcun tipo e sia quindi separata dalla gerarchia principale del progetto, ha un ruolo di primaria importanza per la calcolatrice. La classe modella le caratteristiche tipiche di un personaggio di un gioco di ruolo; ne deriva quindi che ogni operazione definita nella gerarchia principale richieda un riferimento costante all'istanza di questa classe che descrive il/i personaggio/i su cui si effettua l'operazione. Il variare di campi dati quali *Forza, Vigore, Destrezza ecc..* di un'istanza della classe Caratteristiche determinerà il variare del risultato delle operazioni che coinvolgono tali parametri. Si noti che i campi dati (privati) di questa classe sono 7 interi ed ognuno di essi è implicato ed utilizzato come parametro in almeno un'operazione della calcolatrice. La parte pubblica della classe comprende metodi utili per far impostare all'utente, tramite gui, i valori dei campi dati (*setter*) e metodi per prelevare i valori dei campi dati e sfruttarli nelle operazioni della gerarchia principale (*getter*). Vi sono inoltre 2 metodi *int CalcolaSalute e int CalcolaStamina* utilizzati all'interno della gerarchia di Equipaggiamento in alcune operazioni.

2.2 Classe Equipaggiamento

Questa è la classe astratta che sta alla base della gerarchia di tipi del calcolatore. Ha due campi dati primitivi di tipo double, *peso* e *usura*. Vi è un costruttore a due parametri che permette la costruzione di default settando entrambi i campi dati a 0, e controlla inoltre che i valori passati come parametri, qualora ve ne fossero, siano accettabili (peso>0 e 0<=usura<100). Nel caso venissero passati valori fuori range accettabile viene mostrato un messaggio d'errore e la costruzione procede settando i valori di default ai campi dati dell'oggetto. La classe presenta una parte dichiarata protetta contenente un unico metodo costante *Moltiplicatore(char)* che ritorna un double. Questo metodo è stato messo come protetto poiché deve poter essere utilizzato dalle classi derivate in varie

operazioni, ma non deve essere reso disponibile all'utente finale in quanto di natura strettamente implementativa per la logica del programma. Vi sono inoltre, nella parte pubblica, i consueti *setter* e *getter* per permettere la modifica da parte dell'utente dei parametri desiderati, cambiando di fatto, tramite Gui che richiama questi metodi, l'oggetto su cui si opera. Vengono ridefiniti e resi virtuali gli operatori di uguaglianza e disuguaglianza per rendere possibile un confronto effettivo tra gli oggetti del calcolatore. L'ultimo metodo di questa classe è il metodo virtuale puro *VerificaUsabilita(const Caratteristiche&)* che ritorna un valore booleano che rappresenta l'usabilità o meno di un dato oggetto d'invocazione da parte di un dato personaggio (con le proprie caratteristiche). L'importanza di questo metodo deriva dal fatto che, oltre ad essere utilizzabile liberamente dall'utente come operazione di verifica, viene richiamato anche da ogni operazione in ogni classe derivata; questo perché l'usabilità da parte del personaggio di un dato oggetto è prerequisito obbligato per poterci effettuare delle operazioni.

2.3 Sottoclasse Arma

La classe Arma deriva pubblicamente dalla classe Equipaggiamento ed aggiunge 3 campi dati privati: *"danno"* di tipo double, *"forzaRichiesta"* e *"intelligenzaRichiesta"* di tipo intero. Il costruttore segue il comportamento del precedente, ovvero chiede 5 parametri con controlli di consistenza, che se non rispettati, portano ad una costruzione con valori di default. L'implementazione del metodo virtuale puro della classe Equipaggiamento *verificaUsabilità* avviene verificando che le caratteristiche del personaggio di forza e intelligenza siano maggiori o uguali a quelle richieste nei campi *forzaRichiesta* ed *intelligenzaRichiesta*. Oltre alle ridefinizioni delle operazioni di uguaglianza, disuguaglianza e somma, vi sono anche 3 metodi pubblici che vanno a definire le operazioni proprie della classe; *DannoEffettivo* e *Efficacia*, che sono metodi virtuali, e *ConfrontaDanno* che non è marchiato virtuale ma fa uso al suo interno del metodo *DannoEffettivo* su entrambi gli oggetti coinvolti, rendendo consistente il controllo di tipo dinamico a run-time.

2.4 Sottoclasse Armamento

La struttura di questa sottoclasse di Equipaggiamento risulta essere praticamente speculare a quella della classe Arma, con la differenza che definisce una tipologia di Equipaggiamento dedicato alla difesa piuttosto che all'attacco. I campi dati privati della classe sono: *"difesa"* di tipo double e *"vigoreRichiesto"* di tipo intero, e svolgono un ruolo analogo a "danno" e "forzaRichiesta" "intelligenzaRichiesta" della classe Arma, con l'aggiunta di un campo statico di tipo double *"pesoMinimoEquilibrio"*. Anche il modello seguito per l'implementazione del costruttore e del metodo virtuale *verificaUsabilità* è assimilabile a quello seguito per la classe Arma. Le funzionalità aggiunte in questa classe sono fondamentalmente due: il metodo virtuale *Equilibrio* che restituisce un double sfruttato anche nelle classi derivate, e un metodo *ConfrontaDifesa*. Quest'ultimo è anch'esso virtuale, a differenza della sua controparte *ConfrontaDanno* per la classe Arma: la scelta è dovuta alla struttura stessa della classe, in quanto la difesa, a differenza del danno di un Arma, viene calcolata anche in base alla tipologia di danno che dovrà fronteggiare.

2.5 Sottoclasse di Arma: ArmaFisica

Questa classe deriva direttamente da Arma e la amplia, nella sua parte privata, con 3 campi dati aggiuntivi *tipoDanno, scalingForza, scalingDestrezza*, tutti di tipo char. La costruzione degli oggetti segue il pattern delle precedenti, con controlli sui valori passati al costruttore e possibilità di costruzione di default. Il campo dati *tipoDanno*, abbastanza intuitivamente, rappresenta la tipologia di danno procurabile da un'ArmaFisica e può assumere i valori "T","A","C", che stanno rispettivamente per "Taglio", "Affondo" e "Contundente". Meno intuitivo è invece il ruolo dei campi dati *scalingForza* e *scalingDestrezza*, che possono assumere i valori "S", "A", "B", "C", "D", "E"; questi caratteri, esposti in ordine decrescente di valore, sono una sorta di moltiplicatore di efficacia dell'ArmaFisica in questione in base alle caratteristiche del personaggio. Ad esempio, se un'ipotetica ArmaFisica ha valore "A" per lo scaling della Forza significa che è molto affine ad essere utilizzata da un personaggio forzuto: la sua efficacia quindi cresce di molto se brandita da un personaggio con la caratteristica "Forza" di valore alto. Per valori di scaling più bassi, come "D" o "E", questo moltiplicatore quindi si abbassa, nonostante il personaggio possa avere un valore anche alto di Forza nelle sue caratteristiche, rendendo l'ArmaFisica meno efficace di una con valore di scaling sulla Forza più alto. Collegata all'introduzione di questi campi dati di scaling vi è anche la ridefinizione dei metodi *Efficacia* e *DannoEffettivo* ereditati dalla classe madre Arma, che in questa classe tengono conto anche di questi particolari moltiplicatori per calcolare i risultati. Vengono aggiunte inoltre 4 funzionalità proprie della classe attraverso metodi non virtuali[2]. In particolare 3 di questi metodi, *Raffina, Riforgia* e *Cristallizza*, hanno tipo di ritorno void, infatti non mostreranno nessun risultato nell'apposito spazio in output, bensì fanno side-effects sull'oggetto di invocazione modificandone alcune caratteristiche. Si noti, come verrà riportato anche in seguito nella apposita sezione del documento, che i metodi *Efficacia* e *Frantuma* si servono di RTTI nella loro implementazione.

2.6 Sottoclasse di Arma: ArmaMagica

Questa classe aggiunge, rispetto alla classe madre Arma, i seguenti 6 campi dati privati: *fuoco, magico, elettrico, oscuro, scalingIntelligenza, scalingFede*. Mentre molte caratteristiche della classe, come gli scaling e la costruzione, hanno un funzionamento facilmente riconducibile a quelli di ArmaFisica, la grande differenza sta nella tipologia dei danni. Infatti in ArmaFisica è sufficiente un unico campo dati di tipo char per identificare il tipo di danno, in ArmaMagica vi sono 4 distinti campi dati di tipo double, dove ognuno rappresenta la **percentuale** di danno di quel tipo. Un esempio può essere un'ArmaMagica con danno=300, fuoco=50, magico=0, elettrico=25 e oscuro=25: tale arma risulterà avere un danno base di 300 ripartito in 150 danni da fuoco, 0 magici, 75 elettrici e 75 oscuri. Questa differenza nelle tipologie di danno introduce un problema di consistenza dei dati: essendo espressi in percentuale, è necessario verificare che la somma dei 4 campi dati sia **sempre ed esattamente 100(%)**. La prima, e più semplice, verifica viene fatta nel corpo del costruttore per verificare i parametri ad esso passati. Il problema restante sono i setter della classe; mettere dei controlli al loro interno avrebbe, a nostro parere, invalidato gran parte dell'esperienza utente con continui messaggi di errore **prima ancora** che l'utente abbia

[2]In caso di una futura possibile espansione in profondità della gerarchia, se si volessero ereditare tali metodi in una classe figlia sarebbe sufficiente marchiarli virtuali.

completato di settare tutti e 4 i campi[3]. La soluzione adottata per mantenere contemporaneamente "user-friendly" il programma e consistenti i dati è quella di ridefinire il metodo virtuale puro *VerificaUsabilita* affinché controlli anche che questa somma sia uguale a 100. In questo modo, essendo *VerificaUsabilita* richiamato prima di effettuare ogni operazione sull'oggetto, il suo risultato funge da prerequisito determinando la consistenza (o meno) dell'ArmaMagica, e quindi l'esecuzione o meno dell'operazione voluta. In caso di esito negativo l'utente viene avvisato dell'errore ed invitato a correggere le imprecisioni nei dati precedentemente immessi. Vi sono inoltre 5 operazioni proprie della classe, riconducibili a "Raffina", "Riforgia" e "Cristallizza" di ArmaFisica: *Incanta, Infuoca, Benedici, Maledici* e *ArteMistica*, tutti facenti side-effect sull'oggetto d'invocazione.

2.7 Sottoclasse di Armamento: Armatura

Spostandosi sull'altro ramo della gerarchia si trova la classe Armatura, derivata dalla classe Armamento. La parte privata contiene i 7 campi dati double che racchiudono le difese dell'Armatura da tutti i tipi possibili di danno (sia quelli di ArmaFisica che quelli di ArmaMagica). Si noti che a differenza dei tipi di danno di un'ArmaMagica i valori delle difese sono intesi come valori puri, **non percentuali**. Vi è anche un campo dati statico di tipo double *pesoMinimoHyperArmor* che viene sfruttato nel metodo *HyperArmor*. Vengono ridefiniti i metodi *ConfrontaDifesa* ed *Equilibrio* della classe madre Armamento, in particolar modo il metodo *ConfrontaDifesa* ora fa RTTI per verificare il tipo dinamico del parametro passato (puntatore ad Armamento). Vengono aggiunte inoltre 5 operazioni proprie della classe: *Appesantisci, Alleggerisci, CottaDiMaglia, HyperArmor* e *Sopravvivenza*. Mentre i primi 3 metodi cambiano la natura dell'oggetto di invocazione, cambiandone secondo logiche implementative interne i valori, il metodo *HyperArmor* è costante e restituisce un double. Il metodo *Sopravvivenza* ritorna invece un intero positivo che rappresenta il numero di colpi che il giocatore, indossando l'Armatura oggetto d'invocazione, può sopportare dall'Arma dell'altro giocatore prima che l'Armatura stessa diventi inutilizzabile o il giocatore che la indossa muoia.

2.8 Sottoclasse di Armamento: Scudo

Questa classe aggiunge 4 campi dati privati a quelli ereditati da Armamento: *assorbimentoFisico* e *assorbimentoMagico* (in percentuale) di tipo double, *stabilita* di tipo intero e *scalingVigore* di tipo char. La differenza fondamentale con la classe Armatura, che provvede a fornire un valore di difesa puro per ogni tipo di danno, è che lo Scudo assorbe il danno sia magico che fisico facendolo diminuire in percentuale, ammesso che lo Scudo sia in uso. Anche qui, come per Armatura, i metodi *ConfrontaDifesa* e *Equilibrio* della classe madre Armamento vengono ridefiniti e adeguati alle caratteristiche della classe Scudo. Vengono inoltre aggiunte due operazioni proprie tramite i metodi costanti *Parata* e *SpezzaGuardia*. Il primo ritorna un double che rappresenta la probabilità (espressa in percentuale) di eseguire una parata completa, che azzera i danni subiti in un attacco di un'Arma e sbilancia l'avversario. Il metodo *SpezzaGuardia* invece ritorna un intero senza segno che rappresenta il numero di colpi, inferti dall'avversario che brandisce una data Arma, sopportabili dal giocatore che ha lo Scudo in posizione di guardia, prima che quest'ultima si rompa. Questo metodo

[3] Ad esempio, se l'utente richiama il setter partendo dall'impostare il campo fuoco=50 e gli altri campi sono ancora di default a 25 la somma risulta !=100 e lancia un errore. Così via per ogni altro campo fino ad aver settato l'ultimo.

si avvale di RTTI per verificare se l'Arma da "affrontare" sia un'ArmaFisica o un'ArmaMagica e comportarsi di conseguenza.

3 Uso di codice polimorfo

3.1 Polimorfismo nel Model

Tutti i file menzionati di seguito sono reperibili nella **directory "Codice C++/Model/":**

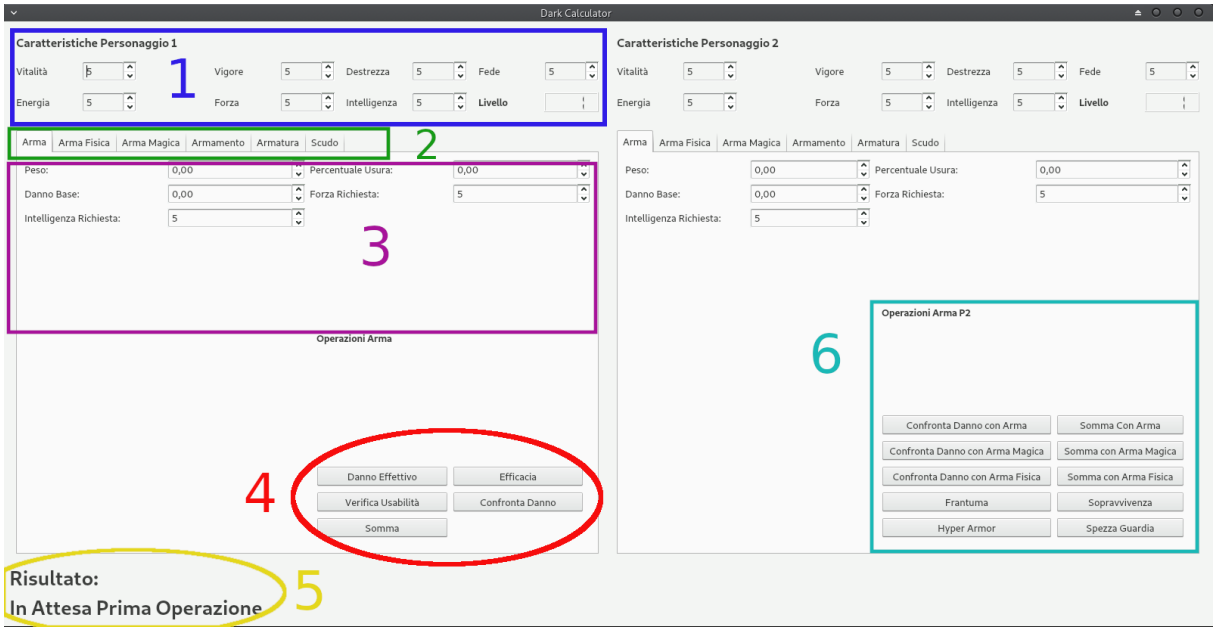
- **Arma.cpp, riga 108:** viene richiamato dai 2 oggetti in confronto[4], all'interno del metodo *ConfrontaDanno*, il metodo virtuale *DannoEffettivo* che, con l'override nelle classi sottostanti, consente di fare una verifica sui tipi dinamici senza rendere virtuale *ConfrontaDanno*.
- **ArmaFisica.cpp, righe 73, 74, 187, 188:** si effettua RTTI con dynamic_casts all'interno dei metodi *Efficacia* e *Frantuma*.
- **Armatura.cpp, riga 210:** viene richiamato il metodo virtuale *Efficacia* sul puntatore *"a"* ad Arma sfruttando covarianza tra tipi base e derivati.
- **Armatura.cpp, riga 230:** come per il caso precedente, viene sfruttata la covarianza tra tipi base e derivati nella chiamata del metodo *Efficacia* sul puntatore "temp".
- **Armatura.cpp, righe 100, 214, 247:** si effettua RTTI con dynamic_cast rispettivamente nei metodi *ConfrontaDifesa*, *Sopravvivenza*, e *HyperArmor* per sfruttare le funzionalità proprie dei tipi derivati.
- **Scudo.cpp, riga 181:** all'interno del metodo *SpezzaGuardia* viene richiamato il metodo *DannoEffettivo* sul puntatore ad Arma "a", sfruttando la covarianza tra tipi base e derivati.
- **Scudo.cpp, righe 84, 179:** si effettua RTTI con dynamic_cast rispettivamente nei metodi *ConfrontaDifesa* e *SpezzaGuardia*.

3.2 Polimorfismo nella View

Tutti i file menzionati in questa sezione possono essere reperiti nella **directory "Codice C++/View/"**. Per creare dinamicamente i widgets da inserire all'interno delle tab del QTabWidget principale si è sfruttata la covarianza tra i tipi QWidget, della libreria Qt, e i tipi definiti da utente, derivati pubblicamente da QWidget, quali *OperazioniArma, OperazioniArmaP2, OperazioniArmatura,* ecc.. Nel file **MainWindow.cpp** viene inoltre costruita una *std::map* che associa un nome, sotto forma di stringa, ad ogni (puntatore a) Equipaggiamento. Successivamente la mappa viene popolata con tutti gli oggetti utili al funzionamento del calcolatore[5] costruiti di default. Ciò permette di passare ai livelli inferiori della view solamente un puntatore a tale mappa e di estrarne tramite **safe dynamic_cast**, in base al contesto ed alle necessità, l'Equipaggiamento desiderato per collegarlo alla Gui ed alle operazioni.

[4] Puntatori ad Arma.
[5] Precisamente 2 istanze per ogni classe: 2 di tipo Arma, 2 Armamento, 2 ArmaFisica, ecc..

4 Interfaccia grafica e breve manuale utente



- **1. Caratteristiche del personaggio:** in questa sezione della Gui è possibile modificare le caratteristiche del personaggio tramite le QSpinBox. Il livello viene calcolato ed aggiornato automaticamente in base alle caratteristiche settate.
- **2. Selezione tipo Equipaggiamento:** qui è possibile scegliere tra i vari Equipaggiamenti, mostrati come tab di un QTabWidget, da settare e su cui eseguire le operazioni.
- **3. Parametri Equipaggiamento:** contenuto della tab selezionata, che varia da tipo a tipo di Equipaggiamento; qui si possono settare i parametri dell'Equipaggiamento voluto.
- **4. Operazioni personaggio 1:** qui si può operare sull'Equipaggiamento selezionato dopo aver settato le caratteristiche (vedi #1) ed i parametri dell'oggetto (vedi #3). Le operazioni del personaggio 1 (sezione sinistra dell'interfaccia) sono dedicate ai calcoli sull'oggetto stesso, se si clicca il pulsante di una operazione di confronto si viene indirizzati, con un messaggio, a settare l'oggetto del personaggio 2 con cui eseguire il confronto (sezione destra della Gui).
- **5. Risultati:** in questa sezione vengono mostrati tutti i risultati dei metodi che hanno un tipo di ritorno (sia booleano che numerico); per le operazioni che effettuano solamente side-effect invece, le modifiche ai parametri dell'Equipaggiamento su cui si sta operando vengono direttamente aggiornate nella sezione #3 della Gui.
- **6. Operazioni personaggio 2:** mentre il resto dell'interfaccia del personaggio 2 è speculare a quella del personaggio 1, le operazioni invece sono solamente quelle di confronto con l'Equipaggiamento del personaggio 1.

NB: si ricorda che, se non modificati dall'utente, i valori di **tutti** gli oggetti di tipo Equipaggiamento rimangono con i valori della costruzione di default. Inoltre si noti che ad ogni operazione che coinvolge gli Equipaggiamenti viene richiamato prima il metodo *VerificaUsabilita*, che può ritornare un messaggio che indica l'impossibilità di usare tale oggetto dal personaggio.

5 Java

Come richiesto dalle specifiche, il programma e la gerarchia di tipi sono stati implementati anche in Java, in una versione priva di interfaccia grafica. Vengono esposte di seguito le piccole differenze riscontrabili tra le due versioni:

- **I messaggi:** nella versione da riga di comando vengono effettuate alcune stampe in output in più per guidare meglio l'utente nelle operazioni e sono state tolte altre che facevano riferimento alla Gui, per aiutare l'utente nella versione C++.
- **Bound:** mentre i campi dati *peso, dannoBase* e tutti i tipi di difesa di Armatura sono stati limitati nella versione con Gui, nella versione Java da riga di comando si è deciso di lasciarli unbounded.

6 Organizzazione interna

6.1 Suddivisione del lavoro

La suddivisione dei compiti tra me (Federico Omodei, mat. 1126500) ed il mio collega (Luca Violato, mat. 1127437) è avvenuta seguendo il pattern di progettazione, scindendo Model e View. Da segnalare che, a rendere meno netta tale suddivisione, vi è stata una costante comunicazione, collaborazione e confronto di idee. Il mio collega si è occupato principalmente della definizione strutturale del Model, quindi della logica del programma, implementando le idee di partenza in C++ ed in Java. Si è inoltre prodigato nel rendere il codice più polimorfo possibile e nella conseguente fase di debugging. Io invece mi sono occupato prevalentemente dell'implementazione grafica del programma, con conseguente approfondimento della libreria Qt, e della fase di testing finale. La mia attenzione è andata spesso nel cercare di rendere più user-friendly possibile un'interfaccia che, a causa dei molti campi dati e operazioni da implementare, rischiava di diventare estremamente complessa per l'utente finale.

6.2 Tempo impiegato

Il tempo impiegato nella realizzazione della parte di progetto di mia responsabilità è poco meno di 52 ore, così ripartite:

- Approfondimenti preliminari della libreria Qt: 9 ore ca.
- Progettazione dell'interfaccia grafica: 3,5 ore ca. (comprensive di piccole rielaborazioni postume)
- Codifica della Gui: 23 ore ca.
- Debugging codice: 14 ore ca.
- Testing programma completo e piccoli fix finali: 2,5 ore ca.