RELAZIONE PROGETTO “DARK CALCULATOR”

COPPIA OMODEI / VIOLATO

A.A 2017-2018 – Luca Violato Matricola: 1127437

COMANDI DI COMPILAZIONE
La compilazione del progetto richiede un project file (.pro) diverso da quello ottenibile tramite l’invocazione di qmake -project. **Viene dunque consegnato il project file “DarkCalculator.pro”**, nella cartella “Codice C++” (ottenuto tramite il comando qmake -project “QT+=widgets” “CONFIG+=c++11”).
Per una corretta compilazione è dunque sufficiente la seguente sequenza di comandi:
qmake
make

AMBIENTE DI SVILUPPO
Sistema Operaztivo: Manjaro Linux
IDE: Geany 1.33
Compilatore C++: GCC 8.1.0
Standard di Compilazione: C++11
GUI: QT 5.11.0
Compilatore Java: javac 9.0.4

INTRODUZIONE
Il progetto in questione prende spunto da diversi calcolatori, presenti sulla rete, che consentono calcoli sull’efficacia di Armi, ed Oggetti in senso generale, per diversi Videogiochi di Ruolo.
In tal senso dunque la calcolatrice presentata in questo progetto gestisce, e consente operazioni, su tipi di oggetti complessi che rappresentano, a diversi gradi di specializzazione, equipaggiamenti di un ipotetico gioco di ruolo, il quale nello specifico è largamente ispirato alla serie videoludica di Dark Souls.
Il fine primo e principale di tale calcolatrice sarebbe dunque quello di fornire, ad eventuali giocatori, informazioni le più dettagliate possibile sull’efficacia o meno di diverse tipologie di Equipaggiamenti in diverse condizioni, o contro determinati “avversari”, oltre a fornire indicazioni precise su quello che potrebbe essere uno sviluppo intelligente delle caratteristiche del proprio personaggio virtuale per meglio adattarsi a certe tipologie di Equipaggiamento che si desidera usare.

SUDDIVISIONE DEL LAVORO PROGETTUALE
Come da titolo la coppia di studenti responsabile per questo progetto è formata da Omodei Federico matricola 1126500 e Violato Luca matricola 1127437.
In tale contesto, sebbene non sia possibile scindere in maniera netta il lavoro eseguito dal singolo individuo a causa di un confronto fondamentale e costante all’interno della coppia, il sottoscritto Luca Violato si è occupato maggiormente della definizione strutturale della gerarchia, sia nella sua versione in codice C++, sia nella versione Java, occupandosi nello specifico anche dell’implementazione della stessa e della conseguente fase di debugging. In tal senso è stato dedicato diverso tempo alla strutturazione di un codice il più polimorfo possibile e che ben interagisse con la GUI.
Lo studente Federico Omodei invece si è premurato maggiormente dell’implementazione grafica del progetto, con conseguente apprendimento della libreria Qt, progettazione e realizzazione della GUI, oltre a sovrintendere la fase di testing.

DESCRIZIONE DELLE GERARCHIE DI TIPI USATE
La principale e fondamentale Gerarchia di Tipi utilizzata in questo progetto è caratterizzata da una struttura lineare a tre livelli con derivazione pubblica.
La classe base “Equipaggiamento” è astratta e rappresenta dunque il “contratto” a cui le classi derivate danno implementazione. Al secondo livello della Gerarchia troviamo le due classi concrete “Arma” e “Armamento” le quali non presentano tra loro dichiarazioni di amicizia di sorta. Con un ulteriore livello di specializzazione troviamo le classi “ArmaFisica” e “ArmaMagica” derivate pubblicamente da “Arma” e, dall’altro lato, “Scudo” e “Armatura” derivate pubblicamente da “Armamento”. Anche in questo livello non vi sono dichiarazioni di amicizia da segnalare.

In sostanza dunque la gerarchia presentata è formata da una classe base astratta e sei classi concrete su cui la calcolatrice consente diverse tipologie di operazioni.

Accanto a questa fondamentale Gerarchia viene definita una classe separata, senza dichiarazioni di amicizia di sorta, chiamata Caratteristiche. Tale classe è concreta e non prevede ereditarietà, non vi sono dunque altre classi da lei derivate.
Lo scopo di tale classe è modellare le fondamentali (in un videogioco di ruolo) caratteristiche del personaggio che andrà ad utilizzare gli Equipaggiamenti modellati nell'apposita gerarchia precedentemente descritta. Questo, oltre a modellare in modo abbastanza preciso la realtà del videogioco di ispirazione, aggiunge un livello di profondità alla complessità delle operazioni presenti nella gerarchia principale "Equipaggiamento" le quali richiedono sempre come parametro un riferimento costante a Caratteristiche.

USO DI CODICE POLIMORFO

- **File "Codice C++/Model/Arma.cpp":** Metodo "ConfrontaDanno" riga 106, viene richiamato, per i due oggetti in confronto, il metodo virtuale "DannoEffettivo" che, attraverso override nelle classi derivate, consente un confronto sui tipi dinamici senza che il metodo "ConfrontaDanno" sia reso virtuale.
- **File "Codice C++/Model/Arma.cpp"**: Metodo "Sopravvivenza" riga 209, viene richiamato il metodo virtuale "Efficacia" sul parametro "a". Si sfrutta covarianza tra tipi base e derivati.
- **File "Codice C++/Model/Arma.cpp":** Metodo "Sopravvivenza" riga 229. Nuovamente viene sfruttata la covarianza tra i tipi nella chiamata del metodo "Efficacia" su "temp"
- **File "Codice C++/Model/Scudo.cpp":** Metodo "SpezzaGuardia" riga 180. Per calcolare la variabile locale "consumo" viene richiamato il metodo "DannoEffettivo" che sfrutta la covarianza dei tipi base e derivati.
- **In tutti i seguenti files: dalla cartella "Codice C++/View" i fles "TabArma", "TabArmaFisica", "TabArmaMagica", "TabArmamento", "TabArmatura", "TabScudo":** per la creazione dei corrispondenti campi dati "OperazioniArma(/ArmaFisica/ArmaMagica/….) di tipo Qwidget* viene sfruttata la covarianza tra i tipi QWidget e OperazioniArma, OperazioniArmaP2, OperazioniArmaFisica, OperazioniArmaFisicaP2 (ecc…) per creare dinamicamente un campo dati che abbia tipo dinamico, e dunque rappresentazione, distinta nel caso in cui si riferisca alla sezione di sinistra oppure di destra della MainWindow.
- **EquipMap mantenuta dalla GUI:** Per implementare in modo efficace tutte le operazioni che necessitano di un secondo oggetto oltre a quello di invocazione viene mantenuta una std::map di Equipaggiamento*. Tale mappa viene sfruttata tramite safe dynamic_cast(s) nella costruzione di ogni tab (TabArma, TabArmaFisica, TabArmaMagica, TabArmamento, TabArmatura, TabScudo) e soprattutto nelle operazioni del "Personaggio2", ovvero files quali OperazioniArmaP2, OperazioniArmaFisicaP2, OperazioniArmaMagicaP2 ecc….
- **File "Codice C++/Model/Armatura.cpp":** Viene fatto RTTI con dynamic_cast nei metodi: "ConfrontaDifesa" riga 100, "Sopravvivenza" riga 213 e "HyperArmor" riga 246. Per sfruttare le funzionalità proprie dei tipi derivati
- **File "Codice C++/Model/Scudo.cpp":** Viene fatto RTTI con dynamic_cast nei metodi: "ConfrontaDifesa" riga 84, "SpezzaGuardia" riga 178. Per sfruttare le funzionalità proprie dei tipi derivati
- **File "Codice C++/Model/ArmaFisica.cpp":** Viene fatti RTTI con dynamic_cast nel metodo: "Frantuma" riga 187 e 188.

Classe CARATTERISTICHE

Per prima cosa si fa notare che l'importanza di tale classe, sebbene totalmente separata a livello logico dalla gerarchia "Equipaggiamento", è fondamentale all'interno del modello presentato. Questo perché gli oggetti di tale classe vengono richiesti come parametro in tutte le operazioni propriamente definite delle classi della gerarchia principale.
La classe è caratterizzata da sette campi dati di tipo intero, i quali rappresentano nel modo più classico (a livello videoludico) le principali caratteristiche presenti in quasi ogni videogioco di ruolo attualmente presente in commercio. Ogni caratteristica rappresentata in questo modo modella un aspetto del tipico personaggio virtuale, come ad esempio la forza, la destrezza o la stamina ecc.. ecc… Si fa notare che non vi sono caratteristiche puramente "ornamentali", ogni campo dati viene usato in modo effettivo nell'implementazione di almeno un'operazione della gerarchia principale.

Nella parte pubblica della classe sono presenti setter e getter per ogni campo dati, i primi per consentire di variare con facilità la singola caratteristica da parte dell'utente, i secondi per consentire alle operazioni della gerarchia di Equipaggiamento di accedere ai campi della classe senza la necessità di dichiarazioni di amicizia.
Sono inoltre presenti due metodi detti "CalcolaSalute" e "CalcolaStamina" che ritornano valori interi e la cui funzionalità è esemplificata dal nome stesso dei metodi. Questi vengono richiamati da alcune operazioni delle classi più basse della gerarchia di "Equipaggiamento".

Classe Base Astratta: EQUIPAGGIAMENTO
La struttura della classe base che funge da contratto è semplice ed elementare. Vi sono due campi dati primitivi double "peso" e "usura" che indicano rispettivamente il peso e la percentuale di usura (0<= usura <100) dell'Equipaggiamento in questione. La costruzione è consentita tramite costruttore a due parametri con valori di default settati a 0, il quale controlla che il peso inserito sia positivo e che l'usura sia compresa nell'intervallo [0,100) . In caso contrario, i valori "non corretti" vengono settati al valore di default 0.
Il metodo virtuale puro di cui si richiede l'implementazione alle classi derivate è chiamato "VerificaUsabilita", prende come parametro un riferimento costante a "Caratteristiche". Tale metodo è costante e ritorna un booleano che indica se l'oggetto di invocazione in quanto Equipaggiamento sia o meno utilizzabile da un personaggio avente le Caratteristiche indicate.
Si noti che, vista l'importanza intrinseca di tale metodo, esso verrà preventivamente richiamato da ogni operazione propriamente definita (quindi esclusi operatori) in ogni classe derivata. Tale metodo risulta dunque essere, in modo concorde alla natura del progetto stesso ovvero una "guida" per il giocatore, una sorta di prerequisito all'esecuzione delle altre operazioni.
Per quanto riguarda la parte protetta della classe in questione è presente un metodo Moltiplicatore, che prende come parametro un char e ritorna un double. Tale metodo è stato inserito nella parte protetta poiché è puramente implementativo, serve alle classi derivate per il calcolo di numerose operazioni. Non deve essere dunque messo nella parte pubblica, a disposizione dell'utente, il quale ovviamente non è e non deve essere interessato alla logica interna che funge da implementazione per la gerarchia. Tuttavia deve essere accessibile da tutte le classi derivate e dunque la scelta di inserirlo nella parte protetta della classe ci è sembrato più ragionevole rispetto ad un inserimento nella parte privata con conseguenti dichiarazioni di amicizia. Si noti che per tale metodo non viene usato codice polimorfo, questa è una scelta coerente con la natura "implementativa" del metodo che non può, e non deve, variare nelle sottoclassi in quanto elemento cardine della Gerarchia.
Per quanto riguarda la parte pubblica della classe invece: il distruttore virtuale della classe è impostato alla versione di default e la classe mette a disposizione come metodi propri due getter e due setter per i campi dati. La scelta dell'implementazione di setter per i campi dati, che sarà mantenuta in tutta la gerarchia, nasce dalla volontà di consentire all'utente un'agevole modifica dei "parametri" degli oggetti su cui ha interesse ad eseguire operazioni. Vista la natura del progetto, i suoi obiettivi ideali, ed essendo, soprattutto nelle classi derivate, gli oggetti costituiti da numerosi campi dati, si è pensato che questa strategia fosse più "user friendly" consentendo quindi all'utente di cambiare velocemente e in modo non macchinoso diversi campi dati per poter, di fatto, "cambiare oggetto" in modo agevole.
Vengono inoltre ridefiniti e resi virtuali gli operatori fondamentali di uguaglianza e disuguaglianza, questo ovviamente per garantire un confronto effettivo degli oggetti che non sarebbe possibile in assenza di codice polimorfo.

Sottoclasse Concreta ARMA
Questa classe deriva direttamente da Equipaggiamento, ed è costituita, nella sua parte privata, da tre capi dati: un double "danno", e due interi "forzaRichiesta" e "intelligenzaRichiesta". Questi ultimi due campi sono fondamentali per l'implementazione del metodo virtuale puro "verificaUsabilita" di cui si richiede l'implementazione. Il campo dati "danno" invece è quello che caratterizza principalmente questa classe all'interno della gerarchia.
Seguendo lo stesso trend della classe base, il costruttore è caratterizzato da 5 parametri con valori di default, vengono controllati i valori inseriti nei campi dati propri in modo che non siano negativi, in caso contrario vengono settati ai rispettivi valori di default.
Oltre all'ovvia, e già accennata, definizione del metodo "verificaUsabilita" questa classe ridefinisce tramite override gli operatori di uguaglianza e disuguaglianza nella loro versione "classica".
Vengono aggiunte in questa classe tre funzionalità proprie, delle quali due sono metodi virtuali ("DannoEffettivo" e "Efficacia"), mentre una ("ConfrontaDanno") pur non essendo marchiata virtuale fa

comunque uso di codice polimorfo richiamando, per il calcolo del danno, il metodo virtuale "DannoEffettivo". La scelta di non rendere virtuale "ConfrontaDanno", che richiede come parametri un riferimento costante a Caratteristiche e un puntatore ad Arma, nasce proprio dall'inutilità di inserire late binding in tale contesto. Grazie infatti alle due chiamate a "DannoEffettivo" sia per quel che riguarda l'oggetto di invocazione, sia per quanto riguarda il puntatore ad Arma passato come parametro, verrà fatta una corretta valutazione sul tipo dinamico a run-time. Si noti infine che tale metodo ha tipo di ritorno double, il quale indica la differenza di danno tra le armi.
Per quanto riguarda le due operazioni citate ("DannoEffettivo" e "Efficacia"), la loro utilità è facilmente intuibile dal nome stesso del metodo. "DannoEffettivo" richiede come paramentro un riferimento costante a Caratteristiche, e ritorna il vero output di danno, sotto forma di double, tenendo ad esempio conto del grado di usura dell'arma. "Efficacia" richiede come parametri un riferimento costante a Caratteristiche e un puntatore ad ArmamentoDifensivo e ritorna, sotto forma di double, l'effettivo output di danno che avrebbe procurato un colpo dell'arma, che funge da oggetto di invocazione, tenendo conto della difesa garantita dall'Armamento Difensivo passato per parametro.
Viene inoltre ridefinito l'operatore di somma che restituisce la somma del danno tra due Armi.
Si noti che tutti i metodi, tranne ovviamente i setter, sono definiti come costanti.

Sottoclasse Concreta ARMAMENTO
Questa classe deriva direttamente da Equipaggiamento, e risulta quasi essere speculare rispetto alla classe "sorella" Arma, questo perché va a modellare la realtà di tutti quegli "Equipaggiamenti" votati alla difesa anziché all'attacco.
Nella parte privata troviamo due campi dati: double "difesa" e int "vigoreRichiesto", come nella classe sorella il primo campo caratterizza la classe, mentre il secondo svolge un lavoro del tutto analogo ai campi "forzaRichiesta" e "intelligenzaRichiesta" nell'implementazione che questa classe dà al metodo "verificaUsabilita". Troviamo anche un campo dati statico di tipo double "pesoMinimoEquilibrio" che è fondamentale per parametrizzare la funzionalità garantita dal metodo "Equilibrio".
La costruzione avviene in modo del tutto analogo a quanto accade in "Arma", così come risultano essere analoghi gli operatori ridefiniti di uguaglianza e disuguaglianza, come anche l'operatore di somma.
Le funzionalità introdotte in questa classe sono due, entrambe fanno uso di codice polimorfo. La prima è garantita dal metodo "Equilibrio" precedentemente accennato. Tale metodo richiede come parametro un riferimento costante a Caratteristiche e ritorna un double che rappresenta il valore di equilibrio garantito dall'equipaggiamento in questione. Tale Equilibrio dipende grandemente dal peso dell'oggetto di invocazione. Questo valore non è puramente fine a sé stesso ma verrà sfruttato dalle sottoclassi di "Armamento".
Il secondo metodo è "ConfrontaDifesa" il quale ha implementazione differente rispetto alla controparte nella classe sorella. Qui infatti si tratta di un metodo virtuale, che richiede come parametro un riferimento costante a Caratteristiche e un puntatore ad "Armamento". La scelta di usare codice polimorfo, a differenza di "ConfrontaDanno", è dettato in questo caso dalla struttura stessa della classe e delle conseguenti sottoclassi. Anche a livello logico, mentre è possibile valutare il danno di un'arma, la difesa di un Equipaggiamento difensivo dipende dal tipo di danno che si trova a dover fronteggiare. Ecco quindi che manca un corrispondente di "DannoEffettivo" in questa sottogerarchia e questa realtà viene modellata, in modo efficiente per quanto si desidera rappresentare, usando codice polimorfo.
Come per Arma anche in questa classe, a parte ovviamente i setter, tutti i metodi descritti sono costanti.

Sottoclasse di Arma: ARMA FISICA
Questa classe deriva direttamente da Arma ed è caratterizzata, nella sua parte privata, da tre campi dati aggiuntivi tutti di tipo char denominati "tipoDanno", "scalingForza" e "scalingDestrezza".
La costruzione eredita l'idea dell'intera gerarchia di controllare che i valori ricevuti in input siano coerenti con il dominio stabilito per i campi dati. In tal senso mentre le superclassi controllavano soprattutto la non negatività dei campi qui viene controllato che il valore inserito per il campo dati "tipoDanno" sia presente nell'insieme {"T", "A", "C"} che stanno rispettivamente per "Taglio", "Affondo" e "Contundente". Viene poi controllato che i valori inseriti per gli altri due campi siano appartenenti all'insieme {"S", "A", "B", "C", "D", "E"}. In tutti i casi anche qui si mantiene la logica che, se i valori in input non sono corretti, il campo dati viene settato al valore che avrebbe in una costruzione di default. In questo caso "T" per "tipoDanno" e "E" per gli altri campi.
Mentre il significato del campo "tipoDanno" risulta abbastanza ovvio, la componente denominata "scaling" viene modellata all'interno della gerarchia seguendo lo stesso spirito che la caratterizza nel videogioco di

ispirazione per questo progetto. Si tratta quindi di una meccanica volutamente “oscura” e non chiara a livello implementativo all’utente, che però non ha bisogno di comprenderne appieno l’implementazione per intuirne, in modo naturale, il funzionamento. Tutti i valori della gerarchia conteneti la parola “scaling” sono caratteri che indicano quanto è “affine” l’arma ad una determinata caratteristica con una situazione in cui, a livello intuitivo, E<D<C<B<A<S. Si ha dunque una situazione in cui, ad esempio, un’arma con valore “A” per lo scaling della forza avrà un notevole incremento nell’efficacia se brandita da un personaggio le cui caratteristiche comprendono un valore molto elevato nella “forza”.
In tal senso quindi, oltre ovviamente agli operatori di uguaglianza e disuguaglianza, trovano ridefinizione tramite overriding in questa classe i metodi “Efficacia” e “DannoEffettivo” introdotti nella superclasse “Arma”.
Nello specifico, il metodo “DannoEffettivo” ora, oltre al funzionamento della classe base, tiene conto del precedentemente descritto sistema di scaling per aumentare il danno in base ai campi dati “scalingForza” e “scalingDestrezza” sfruttando queste due caratteristiche recuperate dal riferimento costante a “Caratteristiche” passato per parametro. Per far questo viene richiamato due volte il metodo “Moltiplicatore” ereditato dalla parte protetta della classe base “Euipaggiamento”, tale metodo a seconda del char passato per parametro (rispettivamente il valore di scalingForza e scalingDestrezza) ritorna un moltiplicatore sotto forma di double che viene moltiplicato per il valore della conseguente caratteristica.
Il metodo “Efficacia” invece in questa ridefinizione fa RTTI per stabilire il tipo dinamico del puntatore ad ArmamentoDifensivo passato per parametro. In base al risultato agisce in modo differente per calcolare l’effettivo output di danno che si sarebbe verificato. In questo senso, nel caso in cui il tipo dinamico sia un puntatore ad “Armatura” (Sottoclasse di ArmamentoDifensivo) viene sfruttato il campo dati “tipoDanno” per calcolare in modo corretto la difesa dell’equipaggiamento passato per parametro.
Vengono inoltre aggiunte quattro funzionalità, garantite attraverso metodi non virtuali. La scelta di rendere tale metodi non virtuali deriva dall’oggettiva difficoltà nel pensare, anche come estensione futura, una sottoclasse di ArmaFisica che abbia senso di esistere.
Tre di questi metodi (“Raffina”, “Riforgia” e “Cristallizza”) hanno tipo di ritorno void e non richiedono parametri. Sono marchiati non costanti perché il loro scopo è quello di modificare, in modo sistematico e seguendo una certa logica, le caratteristiche fondamentali dell’oggetto di invocazione, andando ad agire sia su campi dati propri, sia eventualmente sul sottoggetto. L’idea alla base di queste funzioni è quella di modellare la realtà, comune a tutti i videogiochi di ruolo e in particolare a quello di ispirazione, di poter modificare il proprio equipaggiamento ricorrendo a quelli che sono fabbri o artigiani similari. Risulta essere dunque molto utile per il giocatore che si approccia a tale software modificare tramite tali metodi le proprietà dell’arma e poi verificarne l’efficacia tramite gli altri metodi per vedere quale tipologia di equipaggiamento risulta essere più efficace sulla base delle caratteristiche del proprio personaggio.
L’ultimo metodo introdotto è detto “Frantuma”, richiede come parametri un riferimento costante a Caratteristiche e un puntatore ad Equipaggiamento, ritorna un double che indica la percentuale (compresa tra 0 e 100) di usura che un attacco dell’oggetto di invocazione, sulla base delle caratteristiche indicate, causa all’equipaggiamento passato per parametro. Anche questo sistema si serve di RTTI nella sua implementazione.

Sottoclasse di Arma: ARMA MAGICA
Questa classe deriva direttamente da Arma ed è caratterizzata, nella sua parte privata, da sei campi dati: “fuoco”, “magico”, “elettrico”, “oscuro”, “scalingIntelligenza” e “scalingFede”. La struttura di tale classe è in buona parte analoga alla sorella “ArmaFisica”: molti metodi, come “DannoEffettivo” ed “Efficacia” vengono ridefiniti in modo analogo a quanto avviene nella classe precedentemente descritta, sfruttando il sistema di scaling grazie ai due campi dati “scalingIntelligenza” e “scalingFede”. Tuttavia la natura della caratteristica fondamentale di questa sottogerarchia (il danno) è qui molto diversa rispetto alla classe sorella. Non vi è infatti un campo dati che indica il tipo di danno, vi sono invece i quattro campi dati “fuoco”, “magico”, “elettrico” e “oscuro” che indicano ciascuno la **percentuale** di danno dell’arma legato all’elemento in questione. Per fare un esempio pratico un’ArmaMagica con campo danno=200, fuoco=25, magico=25, elettrico=50 e oscuro=0 avrà un danno base pari a 200, costituito da 50 danni da fuoco, 50 danni magici, 100 danni elettrici e nessun danno di tipo oscuro.
Naturalmente occorre che i valori di questi quattro campi siano monitorati e controllati, in quanto la loro somma deve sempre essere pari al 100%. Per far questo in fase di costruzione, se si riscontrano anomalie, il costruttore setta tutti i campi al default 25%.
Si è scelto anche in questo caso, in maniera coerente con quanto fin’ora deciso, di lasciare massima libertà all’utente, che è in grado di modificare a piacimento il valore di tali campi grazie agli appositi setter. Questa

scelta mantiene il software sufficientemente “user friendly” ma aggiunge un problema ovvio di consistenza dei dati all’interno di questa classe. Per ovviare a questo problema, in sede di costruzione, il costruttore a più parametri controlla che la somma dei campi sia pari al 100%, altrimenti avvisa l’utente e setta tutti i campi al default 25%. Inoltre avviene, in questa classe, una ridefinizione del metodo virtuale puro “verificaUsabilita”, che ora controlla anche la consistenza di questi campi. Si noti che tale metodo, essendo richiamato preventivamente in ogni altro metodo definito (quindi escludendo gli operatori) di ogni classe della gerarchia, funge da **prerequisito** per l’esecuzione della funzione richiesta, e dunque non consente alcun tipo di operazione effettiva su un oggetto di questa classe avente valori non consistenti.
Le funzionalità introdotte vengono anche qui, come in “ArmaFisica”, implementate tramite metodi non virtuali.
Vi sono quattro metodi: “Incanta”, “Infuoca”, “Benedici” e “Maledici” che seguono lo stesso spirito di “Raffina”, “Riforgia” e “Cristallizza” di ArmaFisica, consentono quindi di manipolare, in modo logico e sistematico, le caratteristiche dell’oggetto di invocazione, modificandone quindi la “natura” per consentire all’utente una rapida valutazione di oggetti differenti sulla base delle medesime caratteristiche del personaggio, e manipolando quella che è la realtà dei fabbri nei videogiochi.
L’ultima funzionalità introdotta prende il nome di “ArteMistica”, si tratta di un metodo non costante che prende come parametro un riferimento costante a caratteristiche e ritorna un double. Tale metodo non è marchiato come costante perché fa side effect sul campo “usura” dell’oggetto di invocazione, e per tale ragione la chiamata può non andare a buon fine se tale valore è troppo alto.
Di nuovo ispirandosi al videogioco Dark Souls questo metodo rappresenta, agli occhi del giocatore, l’attacco speciale dell’arma. Una sorta di jolly molto rischioso che però consente un danno drasticamente alto in alcune condizioni. In questo senso dunque il metodo ritorna un double che rappresenta l’output di danno di questo attacco speciale, un po’ come “dannoEffettivo” rappresenta l’output di danno di un attacco normale.

Sottoclasse di Armamento: ARMATURA
Spostandoci dall’altro lato della gerarchia, questa classe deriva direttamente da Armamento. La parte privata è costituita di sette campi double (“difesaTaglio”, “difesaAffondo”, “difesaContundente”, “difesaFuoco”, “difesaMagia”, “difesaElettricita”, “difesaOscurita”) e un campo dati statico double “pesoMinimoHyperArmor” che viene sfruttato nel metodo proprio “HyperArmor”.
La costruzione avviene mantenendo lo stesso spirito delle altre classi: il costruttore controlla che tutti i valori in input siano positivi, se qualcuno non lo è, lo setta al default “0”.
Questa classe specializza la superclasse Armamento inserendo dei valori di difesa aggiuntivi (fissi, e quindi non percentuali) a seconda del tipo di danno che si riceve.
Trovano qui ridefinizione, oltre ovviamente agli operatori di uguaglianza e disuguaglianza, i metodi virtuali “ConfrontaDifesa” ed “Equilibrio” introdotti in Armamento.
Il metodo “ConfrontaDifesa” ora esegue RTTI per verificare il tipo dinamico del puntatore ad Armamento passato per parametro. Se questo è a sua volta un puntatore ad Armatura allora il confronto risulta esaustivo, vengono paragonati nel dettaglio i due oggetti e fatte valutazioni sulla rispettiva utilità in differenti situazioni. Se invece il tipo dinamico tra oggetto di invocazione (che è ovviamente un’Armatura) e parametro attuale non coincide allora l’utente viene avvisato e avviene un confronto generale richiamando la versione di Armamento del metodo.
Il metodo “Equilibrio” invece tiene ora conto della natura dell’oggetto “Armatura” e vengono dunque considerati, nel computo del valore da restituire, alcuni valori incrementali basati sui campi propri dell’oggetto di invocazione.
Vengono inoltre aggiunte 5 funzionalità attraverso metodi propri non virtuali. Si è scelto nuovamente di non rendere virtuali tali metodi per l’oggettiva difficoltà nello scendere ulteriormente nella gerarchia.
Tre di questi metodi (“Appesantisci”, “Alleggerisci” e “CottaDiMaglia”) sono nuovamente funzionalità che consentono di cambiare, in modo strutturato e sistematico, la natura dell’oggetto di invocazione, andando ad agire sia sui campi dati propri sia sul sottoggetto, seguendo lo spirito già delineato nelle altre classi della gerarchia.
Il metodo costante HyperArmor richiede come parametro un riferimento costante a Caratteristiche e un puntatore ad Arma e ritorna un double che indica la quantità di danni sopportabili, senza essere interrotti, dal giocatore avente le caratteristiche indicate, nel caso in cui indossi l’armatura che funge da oggetto di invocazione mentre esegue un attacco pesante con l’arma passata per parametro. Si noti che tale metodo sfrutta diverse caratteristiche di differenti classi della gerarchia. Viene infatti considerato non solo il peso degli oggetti in questione ma anche l’Equilibrio dell’Armatura, il tipo dinamico dell’Arma e il suo scaling.

L’ultimo metodo introdotto è denominato “Sopravvivenza”, richiede come parametri due riferimenti costanti a Caratteristiche e un puntatore ad Arma. Ritorna un intero senza segno che indica il numero di colpi che il giocatore, con le caratteristiche indicate nel primo riferimento a caratteristiche passato, indossando l’armatura che è l’oggetto di invocazione, può ricevere dall’Arma indicata, brandita da un giocatore aventi per caratteristiche il secondo riferimento a Caratteristiche passato, prima di morire o prima che l’armatura stessa si rompa. Tale metodo esegue RTTI sul puntatore ad Arma passato, per vedere se questa può, ad ogni colpo, aumentare l’usura dell’Armatura. Si noti che, essendo comunque una situazione ipotetica, il metodo viene marchiato come costante, in quanto non viene in alcun caso alterato il valore di usura dell’oggetto di invocazione. L’utente viene solamente avvisato, qualora fosse il caso, che l’armatura si sarebbe rotta in una situazione simile.

Sottoclasse di Armamento: SCUDO

Questa classe deriva direttamente da Armamento, ed aggiunge nella propria parte privata quattro campi dati: due di tipo double “assorbimentoFisico” e “assorbimentoMagico” che indicano il valore percentuale (tra 0 e 100 estremi inclusi) di assorbimento del danno nel caso di guardia sollevata. Un campo intero “stabilita” e un char “scalingVigore”.

Si noti che a differenza della classe “Armatura”, la quale in base al tipo di danno ricevuto “attiva” difese aggiuntive non percentuali, la natura di questa classe è molto differente. L’assorbimento del danno riduce in percentuale il danno che si sarebbe subito, a patto ovviamente che lo scudo sia sollevato. Ad esempio se si volesse calcolare la funzionalità “Efficacia” di un’Arma Magica avente DannoEffettivo pari a 200 contro uno scudo avente difesa 50, usura 0% ed assorbimento magico 80% il risultato sarebbero 30 danni (200-50*((100-80)/100)).

La costruzione avviene in maniera del tutto analoga con quanto descritto nelle precedenti classi, con controlli analoghi sia per campi dati di natura numerica, sia per quelli char.

In maniera del tutto analoga alla classe “sorella” Armatura trovano qui ridefinizione, adattata alla differente natura della classe, tramite overriding gli operatori di uguaglianza e disuguaglianza e i metodi “ConfrontaDifesa” ed “Equilibrio”.

Vengono aggiunte due funzionalità, implementate attraverso metodi non virtuali:

Il metodo costante “Parata”, che richiede come parametro un riferimento costante a Caratteristiche e ritorna un double che indica la percentuale di probabilità di eseguire una parata perfetta (il cosiddetto “parry” nel videogioco) che azzera i danni subiti e sbilancia il nemico.

Il metodo costante “SpezzaGuardia” richiede come parametri due riferimenti costanti a Caratteristiche e un puntatore ad Arma, ritorna un intero senza segno. Tale valore indica il numero dei colpi che il giocatore, che indossa lo scudo che è oggetto di invocazione e possiede per caratteristiche il primo riferimento passato, può sopportare con la guardia dello scudo alzata quando questi colpi sono inferti da un giocatore che brandisce l’arma passata come puntatore e possiede per caratteristiche il secondo riferimento a Caratteristiche indicato. Tale metodo fa RTTI sul puntatore ad Arma passato per parametro. Nello specifico se tale puntatore risulta avere tipo dinamico ArmaFisica il metodo ne considera il DannoEffettivo, altrimenti ne considera il danno base.

MANUALE UTENTE GUI

L’interfaccia grafica del progetto potrebbe risultare dispersiva ad un primo sguardo, soprattutto a causa dei numerosi campi dati di cui sono costituite alcune delle classi più in profondità nella gerarchia. Tuttavia, una volta familiarizzato un minimo con la GUI, e la logica alla base di questa, il funzionamento non è eccessivamente complesso.

La finestra principale ha dimensione fissa, a causa della notevole grandezza di cui ha necessità. Questa è suddivisa verticalmente in due macro sezioni quasi speculari. La sezione di sinistra (Personaggio1) serve per la costruzione e la manipolazione degli oggetti **principali** su cui si desidera fare le operazioni messe a disposizione dalla calcolatrice, è la sezione principale con cui si interfaccia l’utente. Quella di destra (Personaggio2) invece viene utilizzata per la costruzione di oggetti **secondari**, i quali vengono richiesti come parametro formale da diverse operazioni eseguite sugli oggetti principali.

Ogni sezione è composta da due elementi principali: un widget, posto nella parte superiore, che rappresenta e mantiene un oggetto di tipo “Caratteristiche”, il quale rappresenta per l’appunto le caratteristiche del personaggio, e una sezione inferiore che varia a seconda della tab che si seleziona. Ci sono sei tab per ogni sezione (Personaggio1 e Personaggio2), ognuna rappresenta un tipo di oggetto che è possibile istanziare, e sul quale è dunque possibile fare delle operazioni; questi sono “Arma”, “Arma Fisica”, “Arma Magica”, “Armamento”, “Scudo” e “Armatura”.

Una volta selezionata la tab di interesse (operazione che non causa la variazione del widget di “Caratteristiche”) la porzione inferiore della sezione conterrà i campi dati editabili di tale classe e le operazioni su di esse disponibili.
E’ possibile **suddividere le operazioni in due tipologie**: quelle che non richiedono l’istanziazione di un secondo oggetto, da passare per parametro, e quelle che lo richiedono.
Le operazioni della prima tipologia avvengono immediatamente, non appena si clicca il corrispondente pulsante nella sezione di sinistra (quella degli oggetti principali “Personaggio1”).
Le operazioni della seconda tipologia invece lanciano un messaggio, visualizzato sotto forma di MessageBox, atto a guidare l’utente, nel modo il più preciso possibile, su come effettuare la costruzione dell’oggetto da passare per parametro, e dunque come poter eseguire l’operazione desiderata. Nella pratica, in queste situazioni, l’utente viene invitato a spostare l’attenzione sulla sezione di destra (quella degli oggetti secondari “Personaggio2”), editare i campi dati dell’oggetto che desidera, e cliccare il corrispondente pulsante per far partire l’operazione desiderata in prima battuta.
Queste operazioni sono più macchinose delle prime, e richiedono uno sforzo aggiuntivo da parte dell’utente, il quale però viene guidato in maniera dettagliata nell’operazione e, se segue alla lettera i passi descritti, non può sbagliare. Inoltre tali operazioni sono anche le più interessanti dal punto di vista dell’utenza che potrebbe aver interesse ad interfacciarsi con un calcolatore di questo tipo, poiché consentono confronti, misure di efficacia e valutazioni dettagliate dei diversi equipaggiamenti in situazioni dinamiche.
L’utente si troverà dunque a lavorare la maggior parte del tempo unicamente interfacciandosi con la metà di sinistra della finestra, spostandosi su quella di destra solo all'occorrenza, quando e se dovrà costruire oggetti secondari per determinate operazioni.
Il risultato delle operazioni è di natura numerica (int o double) o booleana, e viene riportato in un’apposita sezione posizionata centralmente nel margine inferiore della finestra. Talvolta alcune operazioni potrebbero lanciare messaggi, di informazione o warning, che vengono visualizzati sotto forma di MessageBox.

Differenze tra Versione Java (da riga di comando) e Modello (C++) collegato alla GUI
Alcuni metodi, sopratutto quelli di confronto, risultano avere un funzionamento lievemente dissimile tra le due versioni. Questo perché la versione di tali metodi “da riga di comando” esegue diverse stampe (utili ma non necessarie), in standard output, che risultano essere esemplificative delle differenze tra i due oggetti che vengono comparati. Queste aggiunte, gradevoli ma non necessarie, sarebbero risultate eccessivamente invasive nel Modello (C++) collegato alla GUI. Si è dunque scelto di rinunciare a tali stampe (commentate nel codice C++) realizzando una versione funzionante di tali metodi, mantenendo al contempo una totale separazione tra Model e View.
Altra differenza degna di nota è quella presente nei metodi non costanti quali Cristallizza, Riforgia, Raffina di ArmaFisica e Appesantisci, Alleggerisci e CottaDiMaglia di Armatura. In tal senso si è considerata la necessità di porre un limite nella GUI a campi dati che da riga di comando risultano essere “unbounded”, quali sono “peso”, “DannoBase” e tutte le Difese di Armatura. In tal senso dunque, le operazioni non costanti che consentono di cambiare, nello specifico aumentare, tali campi sono state limitate nella versione C++ collegata alla GUI.

TEMPO IMPIEGATO
Nonostante il calcolo preciso della suddivisione del tempo in un simile progetto, portato a termine da una coppia di studenti, sia difficile e necessariamente impreciso, in linea generale l’ammontare totale delle ore dedicate alla realizzazione delle porzioni di progetto sotto la mia responsabilità è, di poco, superiore alle 50 ore, suddiviso all’incirca come segue:
Strutturazione logica della gerarchia e del suo funzionamento (5 ore), individuazione di operazioni eseguibili (3 ore), implementazione preliminare del modello C++ (15 ore), raffinamento e correzione del modello appena citato (10 ore), fase di testing e debugging C++ (7 ore), trasposizione del modello nella versione java (4 ore), testing, debugging e realizzazione del main d’esempio per java (4 ore e mezza), revisione GUI (poco più di 5 ore).