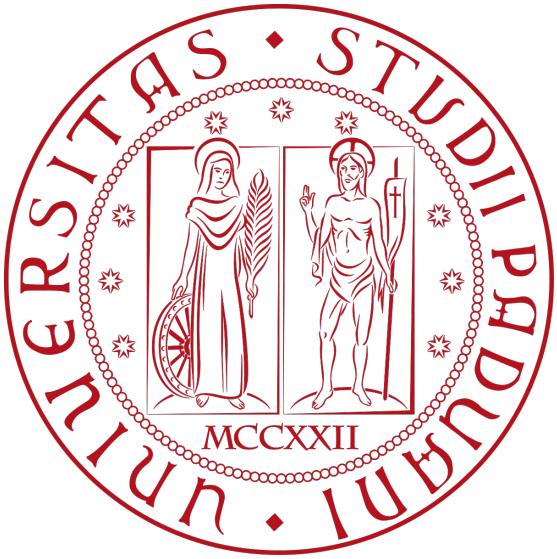
Università degli Studi di Padova



Relazione per:

DamageTrack

Per Simulare un:

Sensore Danno Videogiochi

Realizzata da:
Gabriele Di Pietro, matricola 2010000

GitHub Repository:
`https://github.com/SerpenTaki/Sensore-Videogiochi-P2.git`

1 Introduzione

DamageTrack è un'applicazione che consente di gestire tre tipi diversi di sensori danno riguardanti videogiochi: **Sensore fisico**, **Sensore magico** e **Sensore sacro**. Il programma permette la creazione, la modifica, la ricerca e l'eliminazione di un sensore tramite un'interfaccia grafica intuitiva realizzata tramite il framework QT. **DamageTrack** permette di avviare una simulazione di una partita immaginaria (stile *Dungeons Dragons* o giochi di carte - l'ispirazione principale è stata il gioco *"Iscriptyon"*) di massimo 30 turni e mostrare i danni inflitti ad un ipotetico avversario. Le condizioni del danno variano in base al tipo di attacco utilizzato: il sensore fisico registra danni di tipo fisico che dipendono dall'affilatura dell'arma utilizzata, mentre un danno di tipo magico dipenderà dal livello di magia e dagli eventuali status negativi dell'avversario. Infine il sensore sacro, oltre a registrare danni di tipo sacro molto difficili da mettere a segno, mostra anche il valore di una *Limit*, una barra che si riempie a seconda di ogni attacco andato a segno e che dipende dal livello di fede selezionato. Come anticipato il programma permette di creare un sensore da interfaccia grafica, tuttavia è anche possibile salvare i dati di un sensore con tanto di simulazione e ricaricarla successivamente importando il file in formato strutturato XML, che viene generato dal programma in caso decidessimo di salvare il sensore. In questo modo si ha un supporto alla persistenza dati. Ho scelto questo progetto in quanto permette di separare nettamente il funzionamento dei vari sensori e di sperimentare i concetti di ereditarietà e polimorfismo.

2 Descrizione del modello

Il modello (figura 1) parte dalla classe astratta **sensoreDanno** che contiene tutte le informazioni comuni di tutti i sensori e quindi le sottoclassi: **nome** (in questo caso identificatore unico di sensore non è possibile infatti avere sensori con lo stesso nome), **danno**, **numero di turni** e **numero di attacchi per turno**. Per ciascuno di essi sono stati definiti i metodi *getter* e *setter* che permettono di interagire con i campi privati del sensore danno, riducendo l'impatto che una modifica alla progettazione della classe base causerebbe alle classi derivate. Per lo stesso scopo non si è deciso di usare un accesso di tipo protetto. In **sensoreDanno** sono anche stati definiti tre metodi che non possono essere modificati dalle sottoclassi concrete: `generaValoriRandomGrafico()`, `incrementaHit(), incrementaMiss()` le quali possiedono la keyword `final`. Questi metodi non cambiano il loro comportamento nelle sottoclassi pertanto è inutile ri-definirle. La simulazione di ogni sensore cambia a seconda del comportamento dei metodi `getHit()` e `calcolaDanno()` che devono essere ri-definiti nelle classi e di conseguenza vengono ri-definiti, introducendo nuovi elementi specifici per ogni classe, ad esempio nel sensore di tipo **fisico** la probabilità di colpire è elevata e il valore del danno dipende dall'affilatezza un campo privato di questa classe. Stesso discorso vale per gli altri sensori. **Magico** introduce il valore di status e il livello di magia permettendo quindi di fare numerosi danni. Per il sensore **Sacro** viene introdotto il valore *limit*, una volta superato questo

permette di infliggere numerosi danni. Quindi, per rompere la *limit* bisogna raggiungere un certo valore e queste aggiunte sono calcolate in maniera pseudo casuale, in quanto dipendono dal livello di fede. Come tutte le sottoclassi, il **sensoreDanno** implementa 2 tipi diversi di costruttore. Il primo si riferisce alla creazione di un sensore in-app, quindi noi non abbiamo una vera e propria simulazione gi'a avviata ma poniamo noi da interfaccia utente le regole della partita. Il secondo invece è pensato principalmente per importare un sensore già esistente e mostrare i dati del sensore salvato e la sua ultima simulazione. Questo avviene grazie al metodo `toXML()`, ridefinito in ogni sottoclasse della gerarchia, che permette il salvataggio del sensore in un formato strutturato.

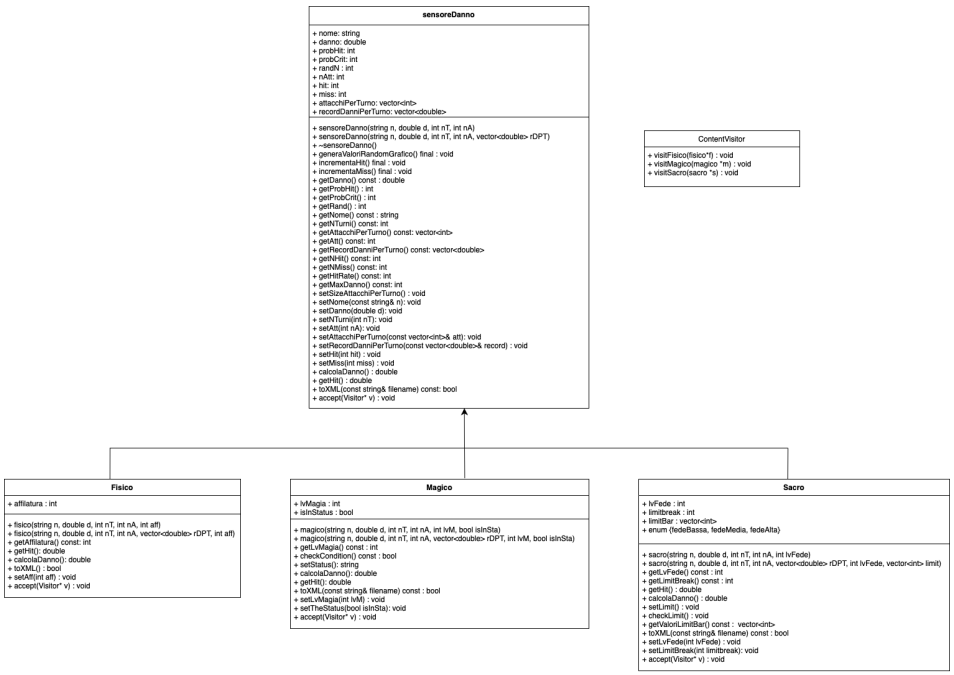


Figura 1 : Diagramma delle classi

3 Polimorfismo

L'utilizzo del polimorfismo riguarda la gerarchia delle classi che derivano da **sensoreDanno**. Quest'ultima dichiara infatti tre metodi virtuali puri `getHit(), calcolaDanno(), toXML()`, ed è inoltre presente un metodo `accept()` per permettere l'integrazione del pattern **visitor** nel programma. Il metodo `getHit()` ci permette di calcolare il danno solo se l'avversario viene colpito. Per ogni sensore c'è una probabilità differente di colpire l'avversario. Ad esempio in sensore **Sacro** la probabilità è molto bassa (20%), quindi con partite più lunghe e livelli di fede più alti sarà possibile fare molti danni. Questa separazione tra la

probabilità di colpire e il calcolo danno è pensata per un'eventuale espansione futura del programma. Difatti inizialmente nel progetto il Sensore Magico era diviso in altre tre sottoclassi concrete dove a seconda del tipo di magia c'era una diversa percentuale di successo e calcoli danno differenti. Un uso migliore del polimorfismo si ha con il metodo `calcolaDanno()`. Sempre in sensore sacro il metodo implementa **limitBar** il quale, raggiunto un certo valore, permette di fare dei danni molto alti. Naturalmente questi dati vengono mostrati attraverso un grafico apposito. In sostanza, nel programma il polimorfismo permette di gestire in maniera flessibile le operazioni comuni tra i sensori. La superclasse **sensoreDanno** è quella che si occupa di definire le operazione comuni e di creare un supporto con la parte grafica del programma (*ad esempio generando i valori del grafico*), mentre alle classi derivate viene lasciato il compito di modificare le operazioni sui sensori, con la possibilità di aggiungere nuovi componenti come la sopracitata **limitBar** (*sacro*) o il **livello di magia** (*magico*) così come **l'affilatura** (*fisico*). Questo approccio migliora e facilita l'aggiunta di nuovi tipi di sensori in futuro, come l'aggiunta di nuove funzionalità o nuove implementazioni da mostrare. Potremmo pensare di introdurre ad esempio in futuro magie di tipo diverso o aggiungere nuovi elementi o altre caratteristiche.

4 Persistenza dei dati

La persistenza dati è implementata attraverso l'uso di un formato strutturato XML. Ogni sensore può essere salvato attraverso la pressione del pulsante *"salva sensore"* collocato nella finestra in basso a sinistra sotto la lista dei sensori. Il file salva i dati del sensore comuni come il nome, il danno di base, il numero di turni, il numero di attacchi per turno, gli attacchi andati a segno e quelli mancati e anche il danno massimo effettuato nella simulazione. Dopo il salvataggio un sensore può essere successivamente importato nel programma. Quest'ultimo mostrerà all'utente tramite GUI, come mostrato nella *figura 2*. Come già accennato, il metodo `toXML()` viene implementato, in tutte le sottoclassi di **sensoreDanno**. Il vantaggio è che possiamo usare il polimorfismo per salvare tutte le caratteristiche che differenziano i sensori. Ad esempio in sensore **Sacro** vengono anche salvati i valori della ***limitBar***. Nella cartella (`sensori_xml`) sono stati lasciati dei sensori di prova.

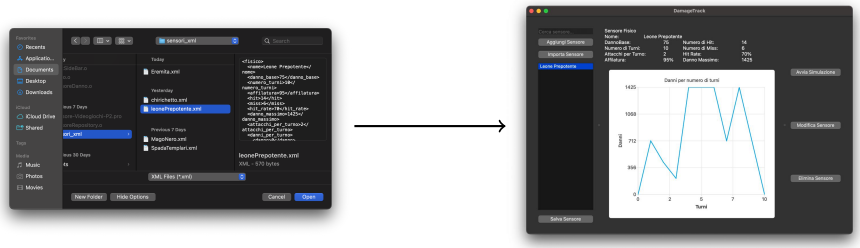


Figura 2: Finestre per importare il sensore

5 Funzionalità implementate

DamageTrack implementa le seguenti funzionalità:

- Creazione di sensori tramite GUI con controllo univocità sul nome
- Importazione di sensori da file in un formato strutturato XML
- Esportazione di sensori in un file in formato strutturato XML
- Possibilità di modificare le caratteristiche di un sensore tramite GUI
- Possibilità di eliminare un sensore
- Possibilità di avviare una simulazione per tutti i tipi di sensore e di salvare i relativi dati
- Possibilità di riprendere una simulazione salvata precedentemente

Le funzionalità *grafiche* sono le seguenti:

- Elenco laterale nel quale vengono mostrati tutti i sensori istanziati
- Pulsante per aggiungere un sensore tramite una finestra pop-up di dialogo
- Pulsante per importare un sensore da un formato XML
- Pulsante per avviare una simulazione
- Pulsante per modificare un sensore
- Pulsante per eliminare un sensore
- Lista laterale per selezionare i tipi di sensore da un elenco
- Grafico posizionato al centro della schermata che permette di visualizzare i dati relativi alla simulazione
- Grafico aggiuntivo per **sensore sacro** per visualizzare il valore delle *limit* nella simulazione
- Informazioni di ogni sensore mostrate sopra il grafico e aggiornate alla simulazione corrente

6 Rendicontazione ore

Le ore in eccesso sono dovute a piccoli problemi riscontrati con la libreria *QT*. La risoluzione ha richiesto del tempo supplementare per poter approfondire il loro funzionamento. Altre ore sono state impiegate nel debug, specialmente per errori a run-time. Inoltre ho riscontrato problemi nell'utilizzo della libreria *QPixMap* su *MacOS*. Ho quindi preferito non implementare la gestione delle immagini, lasciando nella cartella (`assets`) solo l'immagine per l'icona del programma.

Attività	Ore Previste	Ore Effettive
Studio e progettazione	10	13
Sviluppo Del Codice Del Modello	10	11
Studio Del Framework Qt	10	12
Sviluppo Del Codice Della GUI	10	15
Ricerca E Editing Degli Assets	1	2
Test E Debug	5	8
Stesura Della Relazione	4	4
TOTALE	50	65

Tabella 1: Ore di sviluppo

7 Ambiente Di Sviluppo

Per Testare l'applicazione e le varie funzioni è stata utilizzata la macchina virtuale messa a disposizione dal corso. Ho lasciato le disposizioni su come avviare il programma in un file `readme.md`.

SO:	macOS 14.6.1 23G93 arm64
Editor:	Visual Studio Code
Compilatore:	Apple clang version 15.0.0 (clang-1500.3.9.4)
QT:	Qt version 6.7.0

Tabella 2: Ambiente di sviluppo