Università degli Studi di Padova

Relazione per la calcolatrice

KalkRpg

per giochi di ruolo di fantasia

sviluppata da:

Giovanni Cavallin, matricola 1148957

in collaborazione con:

Mirko Gibin, matricola 1102450

Presentazione

La calcolatrice

KalkRpg è una calcolatrice che simula delle operazioni tra oggetti di un gioco di ruolo di fantasia. L'utente può interagire con la calcolatrice tramite dei pulsanti, con i quali può creare degli oggetti, impostandone i valori tramite una procedura guidata, e selezionare le operazioni che desidera fare. Un display indicherà le operazioni e gli oggetti selezionati e il risultato delle operazioni.

Installazione

qmake untitled.pro
make
./KalkRpg

Guida all'uso

La calcolatrice offre sei tipi di oggetto su cui effettuare nove operazioni.

Oggetti

Tutti gli oggetti hanno in comune il `Livello`, la `Rarità` e una statistica: lo `Spirito`. Tre sono tipi base, tre sono tipi da loro derivati, come si può apprezzare in questa lista:

Erba Spirito, vitalità	=>	**Unguento** Spirito, vitalità, energia
Pietra Spirito, durezza	=>	**Cristallo** Spirito, durezza, magia
Osso Spirito, attacco, difesa	=>	**Amuleto** Spirito, attacco, difesa, fortuna

Il gioco di ruolo di fantasia ha alcune regole che abbiamo seguito:

a. Tutti gli oggetti possono avere `Livello` e `Rarità` da 1 a 10 compresi;
b. Ogni statistica, alla creazione, non può superare 150. Tuttavia, può superare questo limite in seguito ad alcune particolari operazioni;
c. La somma dei valori di tutte le statistiche deve essere inferiore a questa formula:
   `Livello*150*(numero di statistiche dell'oggetto)`
   Qualora questo limite venisse superato, la calcolatrice procede automaticamente a normalizzare e/o correggere le statistiche affinché seguano il limite.
d. Il `mana` è l'energia necessaria per creare o potenziare un oggetto e dipende dal suo `Livello`, dalla sua `Rarità` e dalle sue statistiche.

Operazioni

Le operazioni disponibili possono essere unarie, binarie o speciali, ovvero sono disponibili una ciascuna per ogni oggetto foglia. Per poter utilizzare le prime, bisogna cliccare l'operazione, quindi l'operando. Per le seconde bisogna cliccare l'operando, quindi l'operazione, poi l'operando. Per le terze, che sono binarie, il primo operando deve essere l'oggetto per cui esse funzionano. Ogni operazione deve essere confermata con l'apposito tasto di conferma.
Le operazioni disponibili sono le seguenti, e ne verrà descritto il comportamento nella relativa sezione del Model.

1. Crea: operazione unaria.
2. Ricicla: operazione unaria.
3. Combina: operazione binaria.
4. Estrai: operazione binaria.
5. Potenzia: operazione binaria.
6. Trasforma: operazione binaria.
7. Distribuisci: operazione binaria e speciale per Cristallo, come primo operando.
8. Ripara: operazione binaria e speciale per Unguento, come primo operando.
9. Duplica: operazione binaria e speciale per Amuleto, come primo operando.

Pulsanti di gestione

La calcolatrice offre tre pulsanti di gestione:

1. Backspace: permette di ritornare allo stato precedente all'ultimo click che non sia "Conferma" o "Cancella". È disponibile in ogni momento.
2. Cancella: pulisce il display della calcolatrice.
3. Conferma: esegue l'operazione e ne visualizza il risultato.

Progettazione

Per progettare la calcolatrice si è seguito il pattern Model View Controller, del quale si discuterà nelle rispettive sezioni.

Suddivisione del lavoro progettuale

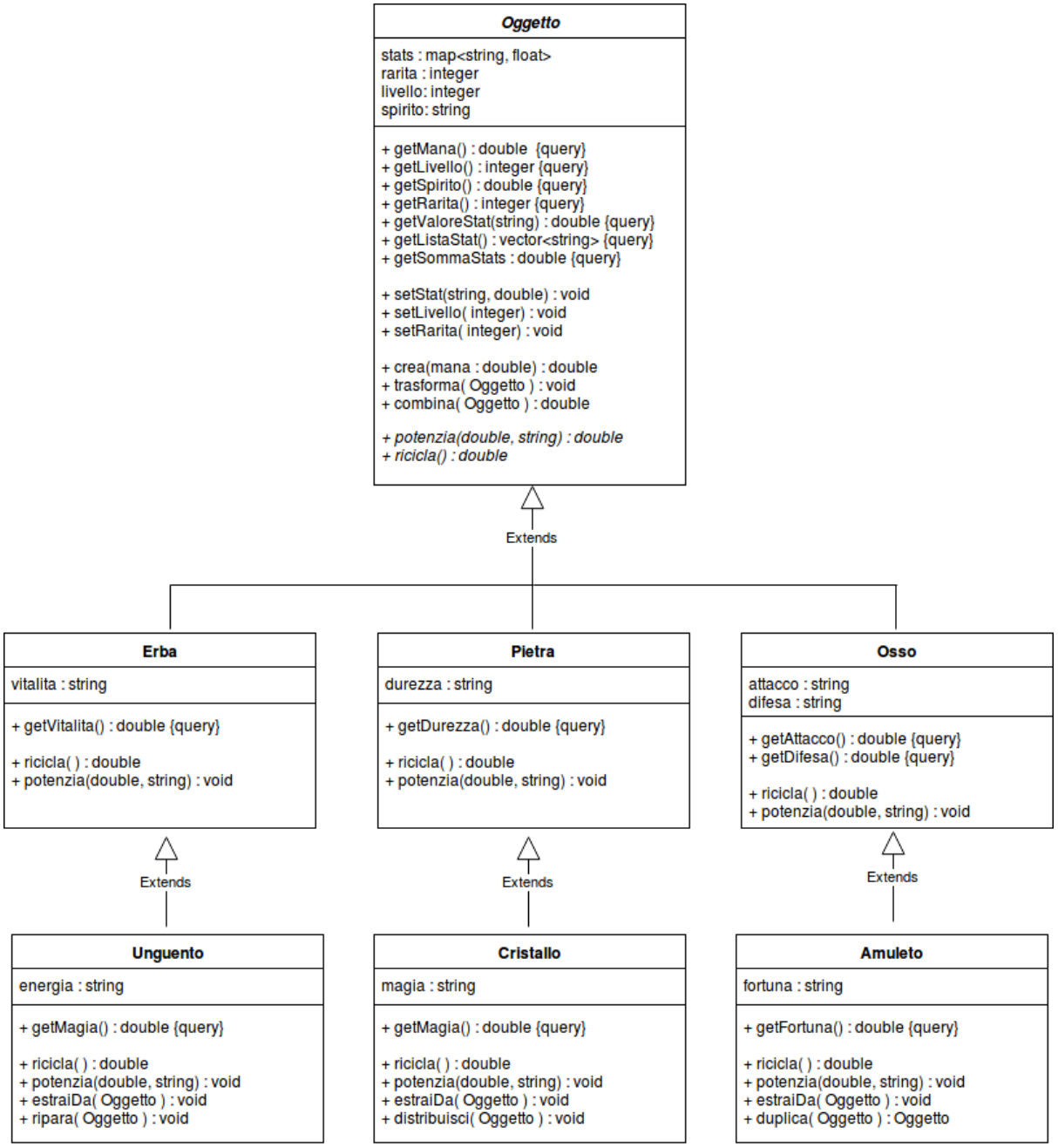
Il lavoro è stato suddiviso in maniera piuttosto equa ma specializzata: io mi sono occupato principalmente della logica d'uso della calcolatrice e della sua interazione con la gerarchia, Mirko ha predominantemente implementato la gerarchia e i suoi metodi e la grafica della calcolatrice. Il lavoro è stato svolto insieme, non senza molti interventi nelle rispettive parti altrui.

Model

La parte Model si occupa di preparare gli oggetti e le operazioni della gerarchia, quindi di creare un'interfaccia tra la gerarchia e il Controller.

Gerarchia: class Oggetto e figli

Nel diagramma qui sotto sono presenti le classi, la loro gerarchia e le operazioni disponibili per ogni classe:



Sono state seguite alcune scelte progettuali, che si sono rivelate vincenti poi nell'utilizzo effettivo della gerarchia:

1. La gerarchia è derivata tutta da una classe astratta, chiamata `Oggetto`. Questa ha come campi privati:
   a. Due valori interi, che contraddistinguono il suo `Livello` e la sua `Rarità`.
   b. Una stringa contenente il nome della variabile particolare dell'`Oggetto` (in cima alla gerarchia è `Spirito`)
   c. Una `map<string, double>` contenente come `key` il nome del parametro, come `value` il suo valore. Si è scelto di usare `double` per facilitare le operazioni sui parametri.
2. Tutte le classi della gerarchia che derivano da `Oggetto` sono concrete e, come campo privato, aggiungono solamente una stringa con il nome della variabile. Della differenziazione si occupa il costruttore: questo infatti aggiunge alla mappa, costruita in `Oggetto`, tutte le statistiche particolari che poi caratterizzano ogni oggetto della classe.
3. È stata creata una classe di eccezioni `OperationException` che gestisce le operazioni non permesse.

Queste tre scelte ci hanno permesso una gestione semplificata delle operazioni tra gli oggetti, rendendoci disponibili tre tipologie di approccio alle operazioni disponibili:

1. Approccio generico, ideando operazioni che si basano sul numero di statistiche presenti negli oggetti (quindi nella mappa), potendo quindi incapsularle nella classe base virtuale `Oggetto` e renderle disponibili per tutte le sottoclassi della gerarchia ed eventuali espansioni. Di questi metodi fanno parte `Crea`, `Combina` e `Trasforma` che sono implementate in `Oggetto`.
2. Approccio specifico, con operazioni disponibili in tutta la gerarchia ma ogni volta implementate in maniera differente a seconda dell'oggetto su cui sono invocate, dipendenti dalla natura particolare dell'oggetto in questione. Di queste fanno parte `Ricicla` e `Potenzia`, che sono definite virtuali pure all'interno di `Oggetto` e poi implementate in tutte le classi della gerarchia con comportamenti differenti. Per queste due classi si è usato il polimorfismo, scelta che verrà giustificata nella discussione della `class Model`.
3. Approccio particolare: alcune operazioni sono state ideate solo per vivere in una determinata classe, poiché molto legate al significato dell'oggetto in questione. Abbiamo quindi:
   a. `EstraiDa`, implementata in ogni classe foglia, che prende come parametro il proprio padre.
   b. `Distribuisci`, disponibile solo in `Cristallo`, che aumenta le statistiche del parametro a seconda di quelle dell'oggetto di invocazione.
   c. `Ripara`, disponibile solo in `Unguento`, che modifica le statistiche del parametro a seconda di quelle dell'oggetto di invocazione.
   d. `Duplica`, disponibile solo in `Amuleto`, che crea una copia del parametro con statistiche diverse a seconda dell'oggetto di invocazione

Alcune operazioni logiche sulle mappe sono state incapsulate in un namespace nel file `mathOp`, poiché non strettamente legate alla gerarchia e possibilmente disponibili per altre classi o gerarchie.

Implementazione metodi

Segue una breve descrizione della firma e del comportamento di ogni metodo.
In oggetto:

1. `void crea(double mana, int livello, int rarita, string statistica ="");`
   Riceve come input il `mana` da utilizzare, il `Livello` e la `Rarità` desiderati, in più si può inserire il nome di una statistica che si desidera più alta delle altre. Inserisce le statistiche quindi create nell'oggetto di invocazione, provocando side effect.
2. `void trasformaDa(Oggetto*);`
   Facendo side effect sull'oggetto di invocazione, questo metodo modifica le statistiche dell'oggetto di invocazione solo se il numero di statistiche dell'oggetto di invocazione ha un numero inferiore o uguale di statistiche. Altrimenti solleva un'eccezione di tipo `OperationException`.
3. `void combina(Oggetto*);`
   Questo metodo fa side effect sull'oggetto di invocazione. Viene fatta la media delle statistiche uguali tra questo e il parametro. Invece, viene fatta la somma delle statistiche non comuni del parametro e viene distribuita nell'oggetto di invocazione.
4. `virtual double ricicla() const`:
   Metodo virtuale puro in `Oggetto`, restituisce il `mana` ottenuto dal riciclo dell'oggetto di invocazione. Dipende in particolar modo dal `Livello`, dalla `Rarità` e dal numero di statistiche dell'oggetto, ma viene implementato in maniera differente in ogni classe della gerarchia.
5. `virtual void potenzia(double mana, string parametro =" ");`
   Facendo side effect sull'oggetto di invocazione, incrementa le statistiche dell'oggetto di invocazione grazie al `mana` e al paramentro.
6. `void estraiDa(Oggetto*);`

può essere invocata su qualsiasi oggetto, e cerca di estrarre un oggetto derivato da uno base. Quindi, ha successo solo se eseguita su un oggetto erba, per estrarre un unguento, su un oggetto pietra per estrarre un cristallo, su un oggetto osso per estrarre un amuleto. Restituisce l'oggetto desiderato con statistiche impostate in base alle statistiche dell'oggetto base. Se l'estrazione viene utilizzata su oggetti non corretti, lancia un'eccezione di tipo `OperationException`. Modifica le statistiche dell'oggetto di invocazione a partire da quelle del parametro, "estraendo" quindi un oggetto più specializzato da uno meno specializzato.

7. `void ripara(Oggetto*)`:
   Questo metodo è presente solo in `Unguento`, e serve per "curare" l'oggetto passato per parametro. Nell'oggetto di invocazione l'energia viene decrementata. Quindi fa side effect sull'oggetto di invocazione.
8. `Oggetto* duplica(Oggetto*)`:
   Specifico di `Amuleto`, crea una copia perfetta o con statistiche diminuite dell'oggetto passato per parametro, in base alla `Rarità`. Se la `Fortuna` non è pari alla media delle statistiche, la `Fortuna` dell'Amuleto scende a 1 e tutti i parametri della copia vengono impostati a 1. Quindi può fare side effect.
9. `void distribuisci(Oggetto*)`:
   Questo metodo si trova solo in `Cristallo`; facendo side effect sull'oggetto di invocazione, se la `Durezza` è almeno la metà della `Magia`, dimezza la propria `Magia` per potenziare il parametro.

Installazione gerarchia: class Model

Con installazione gerarchia si intende la `class Model`, che include tutte le classi presenti nella gerarchia, ma che eredita anche le funzionalità offerte da Qt.
La `Model` ha come campi privati:

1. `QList<Oggetto*> memoria`, che è una vera e propria collezione di oggetti della gerarchia.
2. `QMap<Unsigned int, QImage*>` immagini, che è una mappa di un intero e un'immagine di Qt.
3. `unsigned int counter`, che contiene il numero di elementi in memoria.

Questi tre campi costituiscono la memoria statica della calcolatrice: ad ogni Oggetto* è associata una `QImage*` e un `unsigned int` come contatore. La consistenza è garantita dai metodi della `Model` e del `Controller`. La scelta di rendere i due campi separati è stata dettata dal fatto che la `QList` contiene oggetti della gerarchia, la `QMap` oggetti della `View`, che quindi abbiamo tenuto opportuno separare in caso di implementazioni future.

La necessità di avere del codice polimorfo è qui palese: avendo una lista di puntatori ad `Oggetto`, è possibile richiamare tutti i metodi descritti in `Oggetto` i relativi overrides nelle classi derivate senza dover fare dei casts. Così facendo, ogni qual volta è necessario richiamare i metodi `ricicla`, `potenzia` e di clonazione su un `Oggetto*` con tipo dinamico di una classe derivata, allora il metodo selezionato sarà quello della rispettiva classe.

La memoria è da intendersi "a pila", quindi le operazioni in generale avverranno nell'ultimo e/o penultimo elemento.
`Model` ha inoltre una gestione completa degli errori attraverso la classe di eccezioni `MemoryException`, che viene lanciata ogni qual volta in memoria non ci siano abbastanza elementi per consentire l'operazione richiesta. Poi, gestisce le eccezioni `OperationException` della gerarchia in questa maniera: ogni volta che cattura questo tipo di eccezione, rilancia a sua volta un'eccezione di tipo `ViewException` al chiamante, ricopiando il messaggio d'errore della prima.
`Model` offre alcuni metodi di get e di set, per leggere e settare ed eventualemente eliminare gli elementi in memoria. In particolare, si annotano i metodi:

a. `QMap<QString, int> getLastObj(int i=0) const;`
   Che si occupa di restituire una mappa con tutte le statistiche – quindi comprensive di `Livello` e `Rarità` – e con i rispettivi valori dell'oggetto `i`-esimo. La conversione da `double` della gerarchia a `int` della `Model` è di default.
b. `bool setStatByName(QString stat, unsigned int value) const;`
   Si occupa di settare la statistica `stat` col valore `value` nell'ultimo oggetto presente in memoria. Essendo un metodo di interfaccia tra il controller e i metodi della gerarchia, si occupa di invocare diversi metodi della gerarchia a seconda della statistica: uno per il `Livello`, uno per la `Rarità`, un altro per le statistiche della mappa dell'oggetto.
c. `void deleteLast();`
   Si occupa di eliminare l'ultimo elemento presente in memoria. Per fare questo, prima rimuove l'oggetto dalla memoria, poi ne invoca il distruttore; quindi, viene rimossa l'immagine dalla `QMap` rispettiva al `counter`-elemento, che viene diminuito, e la dealloca. Poi, se l'oggetto rimanente in memoria è un `Cristallo`, `Unguento` o `Amuleto`, emette il `Signal` corrispondente che poi verrà rilanciato dal `Controller` alla `View`.
d. `void combina();`

Prendiamo questa operazione come esempio del comportamento della `Model` per quanto riguarda le operazioni. Il metodo `Combina` prevede un oggetto di invocazione e un oggetto passato come parametro. Il primo è logicamente il penultimo in memoria, mentre il secondo è l'ultimo in memoria. `Combina` fa side effect sull'oggetto di invocazione, da gerarchia, quindi qui nella `Model` si fa una copia profonda del penultimo elemento (sia dell'`Oggetto`, che della sua immagine); viene invocato su questo il metodo combina della gerarchia, e come parametro viene passato l'ultimo elemento della memoria. In questo step, alcune operazioni possono sollevare delle eccezioni, che vengono eventualmente catturate e rilanciate al chiamante. Quindi, il nuovo oggetto clone, su cui è stata richiama l'operazione, viene opportunamente inserito in memoria con la sua immagine.

e. `void createErba();`
   Prendiamo questo metodo come esempio per tutti gli altri, che si occupano di creare uno specifico `Oggetto`. Questo metodo invoca la creazione di un nuovo `Oggetto` di tipo `Erba`, che viene costruito con i suoi parametri di default, e viene inserito in memoria. Il `counter` viene coerentemente aumentato.
f. `signals: void nothingToDelete();`
   Ogni qual volta la memoria risulta essere vuota in seguito ad una operazione di eliminazione, viene emesso questo segnale.

Non ci sono accenni alla possibilità di costruire un `Oggetto` con parametri non di default. Questa è stata una precisa scelta progettuale: questa `Model` è nata in corrispondenza di una particolare interfaccia grafica, che non richiedeva questa funzionalità. Perciò l'impostazione di statistiche e immagini degli oggetti in memoria è stata demandata al `Controller`.

Controller

Il controller è sviluppato all'interno dell'omonima classe. Ha come classe accessoria `DisplayAndSlider`.

Class DisplayAndSlider

Questa eredita `QWidget` e ha il compito di fornire un oggetto composto da un `QSlider`, con range predefiniti (1:10 per `Rarità` e `Livello`, 1:150 per le statistiche, 1:6000 per il `mana`), da una `QLabel` con la descrizione del parametro che si vuole modificare, e da una `QLineEdit`, validata ad intero nel range predisposto, che permette di scrivere un valore o visualizzare quello dello slider.

Class Controller

Il Controller è invece composto da:

1. `Model* modello`, che viene istanziato alla creazione del `Controller`.
2. `QMap<QString, QSlider*> tempDataToSet`, che funge da cache temporanea e che contiene le informazioni relative al settaggio di oggetti e operazioni.
3. `QImage* image`, che mantiene la traccia dell'immagine dell'oggetto ad esso associata
4. Alcuni parametri di servizio per gestire le informazioni che poi il controller dovrà gestire.

Il compito di `Controller` è quello di coordinare le interazioni tra `View` e `Model`. In particolare:

a. `create`*`Oggetto`*`()`: quando viene premuto un pulsante Oggetto, invoca la sua creazione in `Model`.
b. `setSelectedObject(QgridLayout*)`: quando viene premuto un pulsante Oggetto che deve essere impostato, riempie un pannello di espansione della `View` di `DisplayAndSlider` corrispondenti alle statistiche dell'oggetto desiderato e ne tiene informazione all'interno di `tempDataToSet` attraverso i puntatori agli slider corrispondenti.
c. `setStatsOnObj()`, `flushControllerMemory()`: quando l'oggetto impostato viene confermato, scrive le informazioni contenute nella cache all'interno dell'oggetto corrispondente contenuto in `Model`; dopodiché, svuota la sua cache.
d. `setPotenzia()`, `setCrea()`: quando viene premuta un'operazione che ha bisogno di essere impostata (vedi `Crea`, `Potenzia`), riempie un pannello di espansione di `DisplayAndSlider` con i parametri necessari e una `QComboBox` che permetta di scegliere un eventuale parametro dalla lista dell'oggetto selezionato.
e. `new`*`Oggetto`*`()`: quando viene premuta un'operazione da eseguire, invoca l'operazione corrispondente del `Model`.
f. Tramite i metodi di gestione e di "get" possono leggere tutti gli elementi di `Model` e conoscerne il numero in memoria; fornisce inoltre dei metodi per la lettura della sua cache temporanea; può eliminare l'ultimo oggetto inserito nel `Model` oppure ordinarne lo svuotamento completo.
g. Riceve e inoltra i `signal` emessi da `Model`.

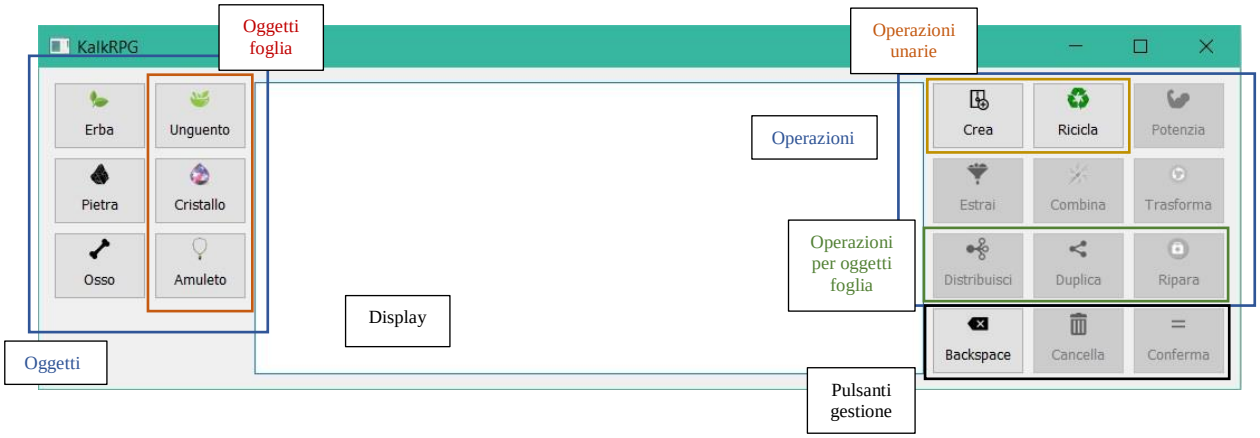
Il controller si occupa di catturare le eccezioni emesse dal `Model` in fase di lettura e scrittura in memoria. Tuttavia, come si può notare, non sono gestite in alcuna particolare maniera, perché la logica di questa particolare view non permette il verificarsi di queste situazioni.

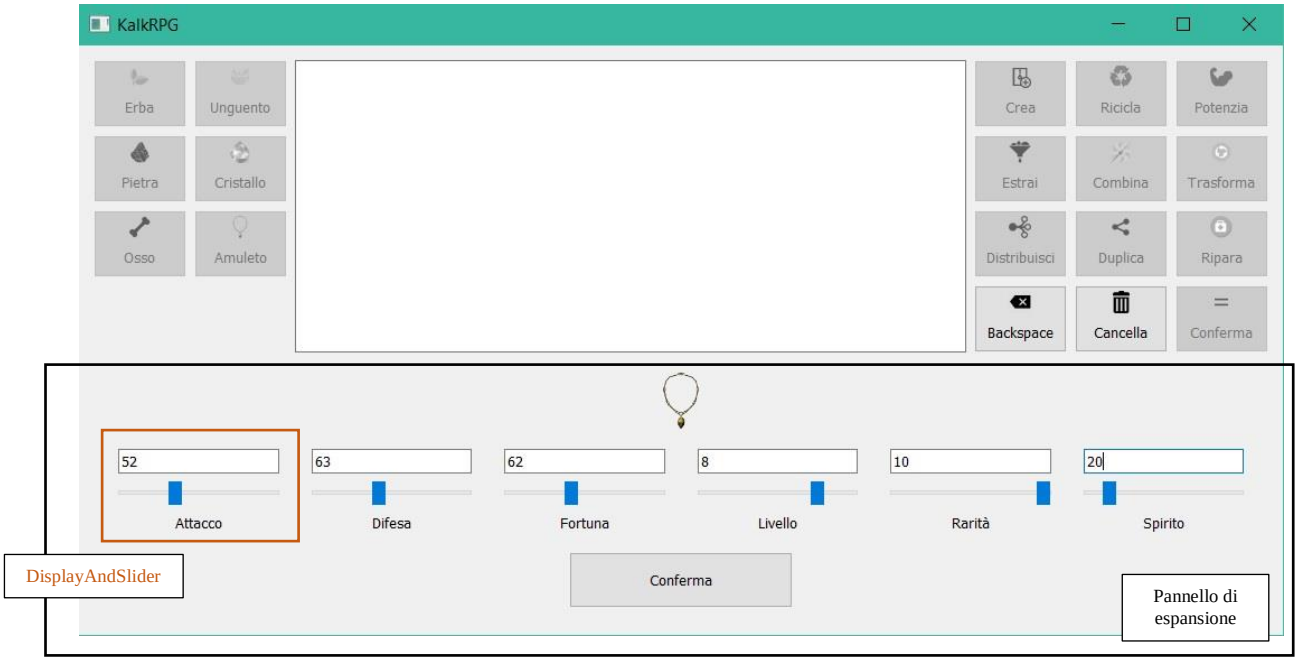
View

La `View` è sviluppata all'interno della classe `KalkRpg`, che ha come classi ausiliarie `Button` e `Display` che si occupano di definire e gestire il comportamento rispettivamente dei pulsanti e del display.
Vista la natura statica di una calcolatrice, che modifica il proprio stato solo in occasione di interventi da parte di un utente, abbiamo deciso di implementare una `View` a stati. Ogni stato è quindi caratterizzato visualmente da alcuni pulsanti che possono diventare cliccabili e non, mentre nell'implementazione ogni stato è dato dalla combinazione di tre variabili globali e di alcune specifiche, che poi vedremo nel merito.

Pulsanti e display

La calcolatrice è composta dalle seguenti parti:





Gestione logica dell'interfaccia

Abbiamo deciso di guidare l'utente all'utilizzo della calcolatrice in maniera naturale, disabilitando via via i pulsanti che, in un preciso stato, non hanno uno scopo definito.
Per comprendere il funzionamento della calcolatrice possiamo definire quattro macro stati:

1. `SettingObject`: in questo stato compare una griglia di espansione della `View`, all'interno della quale il controller si occupa di mettere i `DisplayAndSlider` – ed eventualmente una `QComboBox` - basati sull'oggetto o sull'operazione precedentemente cliccati per impostare o l'oggetto selezionato o l'operazione che lo richiede. Sono disponibili solo i pulsanti `Backspace`, `Cancella Memoria` e `Conferma` oggetto.
2. `ObjectIsCreated`: in questo stato la calcolatrice ha appena concluso la creazione di un oggetto, quindi rende disponibili le operazioni – coerentemente all'oggetto cliccato. Sono disponibili tutti i tasti – meno quelli delle operazioni specifiche nel caso l'oggetto selezionato non sia una foglia della gerarchia. La visualizzazione di queste operazioni particolari avviene tramite variabili booleane. Ad esempio, se si clicca

un Cristallo, una variabile cristalloObj viene settata a true, e questo permette la visualizzazione di Distribuisci.
3. ConfirmOperationToClick: siamo nello stato subito precedente alla conferma dell'operazione. È disponibile solo il tasto Conferma operazione, Cancella Memoria e Backspace.
4. OperationIsDone: siamo nello stato finale: l'operazione si è conclusa e a display è mostrato il risultato. È disponibile solamente il tasto Cancella Memoria, per poter eseguire una nuova operazione.

La seguente descrizione indica lo svolgimento dell'operazione standard, sulla quale si basano tutte le altre operazioni.

1. L'utente inizialmente ha a disposizione solo i tasti oggetto e il tasto Backspace, che però non fa nulla. Premendo un tasto oggetto qualsiasi, viene invocato il metodo showToSet(Button*), che si occupa di aprire un pannello di espansione e di invocare il metodo controller::setSelectedObject(QGridLayout*). Questo riempie il pannello di DisplayAndSlider, quindi la View aggiunge l'immagine dell'oggetto e il tasto di conferma creazione oggetto. Siamo nello stato SettingObject.
2. Quando l'utente ha completato l'impostazione dei parametri e cliccato su "Conferma", la View ordina al Controller di scrivere le informazioni raccolte nella memoria con il metodo setStatsOnObject() ed esegue il flush della cache. Quindi si occupa di nascondere ed eliminare il pannello di espansione precedentemente creato. Siamo nello stato ObjectIsCreated.
3. Possiamo scegliere di creare un nuovo oggetto, oppure di selezionare un'operazione. In questo secondo caso, la selezione del pulsante invoca la copia dello stesso in un Button* di KalkRpg, che verrà in seguito richiamato quando si confermerà l'operazione. Siamo nello stato ObjectIsCreated, con waitingOperand a true, che quindi rende disponibili solo i tasti oggetto.
4. Una volta selezionato, impostato e confermato il secondo operando, rimane disponibile solo il pulsante di conferma dell'operazione. Siamo nello stato ConfirmOperationToClick.
5. Una volta cliccato il pulsante di conferma, viene cliccato automaticamente il tasto dell'operazione, precedentemente salvato. Questo entra in un altro ramo della sua esecuzione che, effettivamente, invoca il metodo dell'operazione rispettivo del Controller, il quale a sua volta si rivolge al Model. Quando l'operazione si è conclusa, la View legge il risultato dalla memoria e lo mostra sul display. Siamo nello stato OperationIsDone.
6. Da qui si può solo cliccare Cancella Memoria, che riporta la calcolatrice allo stato di partenza.

La View si occupa di catturare l'eccezione ViewException sollevate dai metodi Estrai e Trasforma in Model: qualora l'operazione non dovesse andare a buon fine, al posto del risultato viene mostrato il testo dell'errore.

La parte più interessante della realizzazione dell'interfaccia è stata la gestione a stati del tasto Backspace. Infatti questo deve riuscire a far ritornare la calcolatrice ad uno stato precedente e sapere in quale momento sono state fatte modifiche alla memoria per poterla eventualmente far ritornare allo stato precedente. Per ottenere questo risultato sono state impiegate tre variabili booleane generali: waitingOperand, running e settingObj che, come si potrà apprezzare dalle tabelle seguenti, identificano ogni stato in maniera univoca e permettono di identificarlo univocamente. Altre variabili booleane sono state aggiunte poi, per identificare comportamenti più particolari per altre operazioni o oggetti. In particolare, per poter rendere di nuovo disponibili le operazioni speciali per gli oggetti foglia, sono stati impiegati dei connect che mettono in relazione lo stato della memoria del Model con il valore a true o false delle variabili. Quindi, se ad esempio eliminando un oggetto, Model::deleteLastObj() trova che l'ultimo oggetto rimasto è un'istanza di Amuleto, viene emesso un signal isAmuleto(bool), che viene inoltrato dal Controller alla View e che attiva un suo slot, il quale imposta a true la viariabile amuletoObj. Questa variabile, in ObjectIsCreatedState(), fa sì che il tasto operazione Distribuisci diventi cliccabile.

Backspace è sempre attivo, perché abbiamo pensato che questo tasto avrebbe aiutato l'utente a sentirsi meno "disperso" all'interno della nostra calcolatrice. Anche Cancella Memoria è sempre cliccabile, a patto che ci sia qualcosa da eliminare. Il tasto di Conferma di impostazione dell'oggetto o dell'operazione è attivo coerentemente al pannello di espansione. Le operazioni per gli oggetti foglie sono attivi solo quando il primo operando è un oggetto foglia. Queste regole sono da tenere in considerazione nella lettura delle tabelle.

Operazione binaria (esempio per Combina):

Stato		waitingObject	running	settingObj	**Azione**	**Backspace**	**Tasti disponibili**
Start	0	false	false	false	Object Clicked		Oggetti, operazioni unarie

Setting Object	1	false	fase	true	Confirm Object	Porta a 0	Nessuno
Object Is Created	2	false	false	false	Operation Clicked	Porta a 0	Tutti
Choosing Operand	3	true	true	false	Object Clicked	Porta a 2	Oggetti
Setting Object	4	true	true	true	Confirm Object	Porta a 3	Nessuno
Object Is Created	5	false	true	false	Confirm Operation	Porta a 3	Nessuno
Done	6	false	false	false	Erase		

Operazione unaria (esempio per `Crea`):

Stato		waitingObject	Running	settingObj	creaOp	**Azione**	**Backspace**	**Tasti disponibili**
Start	0	false	false	false	false	Operation Clicked		Oggetti, operazioni unarie
Choosing Operand	1	true	true	false	true	(Automatically: Confirm Object)	Porta a 0	Oggetti
Setting Operation	2	false	true	true	true	Confirm Setting	Porta a 1	Nessuno
Operation Is Ready	3	false	true	false	true	Confirm Operation	Porta a 1	Nessuno
Done		false	false	false	false	Erase		

Operazione binaria con l'inserimento dei suoi parametri (esempio per `Potenzia`):

Stato		waitingObject	running	settingObj	potenziaOp	**Azione**	**Backspace**	**Tasti disponibili**
Start	0	false	false	false	false	Object Clicked	Porta a 0	Oggetti, operazioni unarie
Setting Object	1	false	false	true	false	Confirm Object	Porta a 0	Nessuno
Object Is Created	2	false	false	false	false	Operation Clicked	Porta a 0	Tutti meno operazioni speciali
Setting Operation	3	false	true	true	true	Confirm Setting	Porta a 2	Nessuno
Operation Is Ready	4	false	True	false	true	Confirm Operation	Porta a 2	Nessuno
Done								

Ambiente di sviluppo

Giovanni	**Mirko**
C++ SO:Windows 10 Pro Editor: Qt Creator 4.4.1 Compilatore: gcc 5.4.0 Qt: 5.5.1 Java: Compilatore: jdk 1.8.3 Versione Intellij: 2017.3.4	C++ SO: Lnux Mint 18.3 Editor: Qt Creator 4.4.1 Compilatore: gcc 5.4.0 Qt: 5.5.1 Java: Compilatore: jdk 1.8.3 Versione Intellij: 2017.3.4

Ore richieste dalle fasi progettuali:

	Giovanni	Mirko
Studio	8	8
Ideazione e progettazione	7	8
Scrittura Codice	37	31
Debugging	8	7
Totale	60	54

Le ore in eccesso sono dovute a piccole problemi riscontrati con la libreria Qt, la cui soluzione non è stata immediata per via della non conoscenza del programma.