Relazione progetto di Programmazione ad Oggetti

Jessica Carretta, mat. 2034312

Titolo del progetto: CUDLE

1 Introduzione

CUDLE è un piccolo gioco ispirato a *Wordle*, tuttavia si caratterizza per essere molto più personalizzabile. Infatti, si ha la possibilità di scegliere quanti elementi indovinare, ossia il numero di round, ma anche quanti tentativi per round avere. Inoltre, vera particolarità del gioco, si ha la possibilità di creare dei set particolari su cui giocare. [1]

1.1 Cos'è un set e come è composto

Un set è un contenitore di elementi che riguardano un certo argomento (ad es. se il set riguarda i numeri da 1 a 50, i vari elementi appartenenti a questo set saranno 1, 2, ..., 50). Inoltre, il set contiene una serie di attributi che sono le proprietà che caratterizzano tutti gli elementi nel set. Pertanto, per ogni elemento, dovremmo dare un particolare valore per ciascuno di questi attributi (ritornando all'esempio dei numeri da 1 a 50, un attributo che li caratterizza tutti potrebbe essere dire se sono pari o dispari, quindi ad es. il numero 27 avrà come valore "dispari" per quanto riguarda questo attributo).

Si noti inoltre che possiamo avere diversi tipi di elementi nel set:

- elementi base, in cui semplicemente abbiamo il nome dell'elemento (la parte da indovinare) e i valori per ogni proprietà/attributo,
- elementi con immagine, in cui oltre ai campi dell'elemento base si ha anche un'immagine che riguarda l'elemento da indovinare,
- elementi con citazione, in cui oltre ai campi dell'elemento base si ha anche una citazione che riguarda l'elemento da indovinare.

[1] Si noti che è comunque presente un set base chiamato *sample_set*, il cui obiettivo è trovare un numero casuale da 1 a 50.

1.2 Come si svolge il gioco

Praticamente, per ogni round viene scelto un elemento a caso dal set selezionato che i giocatori cercano di indovinare entro i tentativi prefissati. Dopo ogni tentativo, vengono visualizzati i valori dell'elemento relativi a ciascun attributo. Ogni valore verrà quindi contrassegnato come verde, giallo o rosso:

- il verde indica che il valore è completamente corretto,
- il giallo significa che il valore è una parte della risposta o la contiene,
- il rosso indica che il valore è completamente sbagliato.

L'obiettivo è quindi trovare l'elemento i cui valori sono tutti verdi, ossia l'elemento estratto randomicamente per quel round.

2 Descrizione del modello

Il modello logico è così costituito:

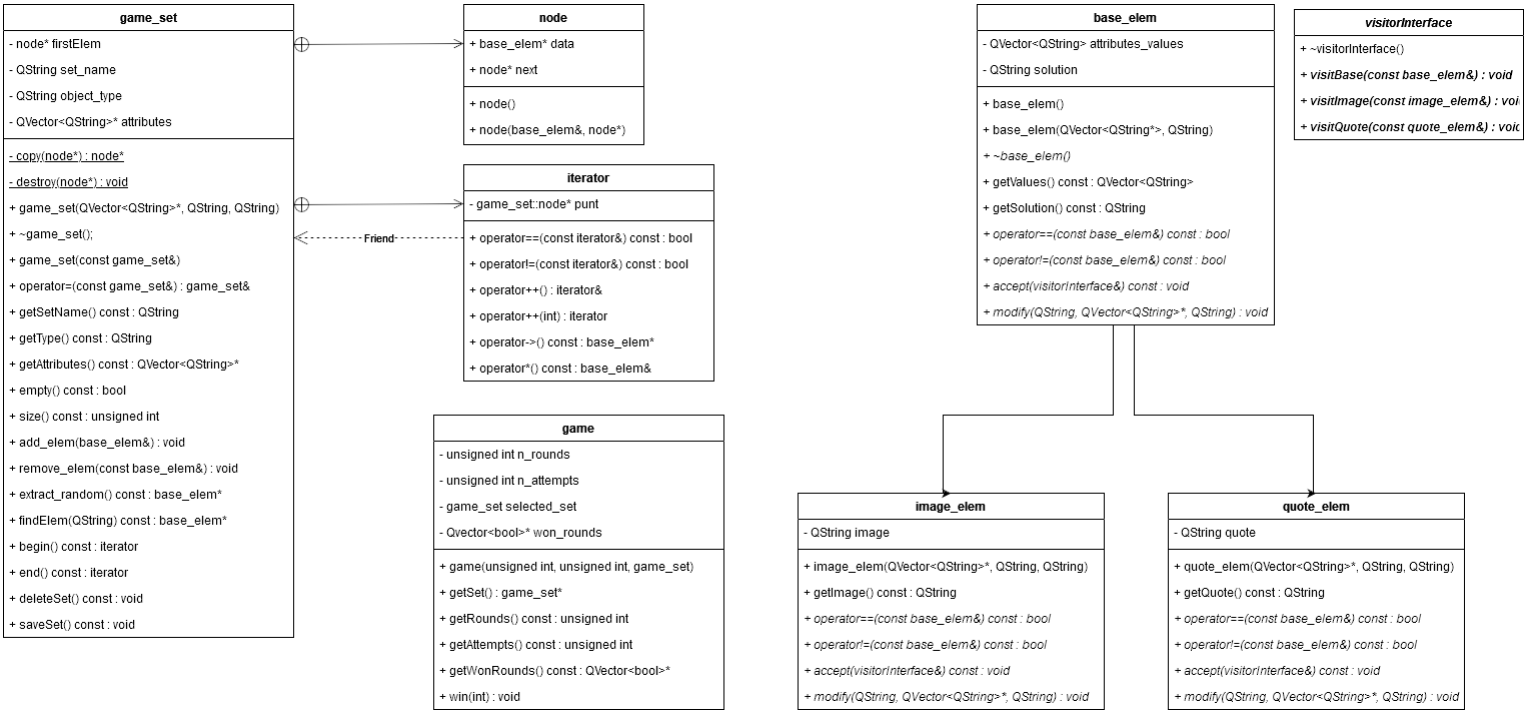

Figure 1: *Diagramma delle classi del modello*

Nello specifico la gerarchia contiene:

- base_elem

- image_elem
- quote_elem

usate per rappresentare i corrispettivi elementi del set (vedi 1.1). In particolare, il modello parte dalla classe base concreta *base_elem* che rappresenta le informazioni comuni a tutti gli elementi, ovvero nome e valori degli attributi (dei quali sono implementati i relativi metodi *getter* e un metodo virtuale per la modifica), e contiene anche alcuni metodi virtuali per il confronto tra i diversi elementi (usati per verificare che l'elemento indovinato sia lo stesso di quello estratto per quel round).

Da questa classe base, derivano altre due classi concrete, ossia *quote_elem* e *image_elem* in quanto aggiungono altre informazioni specifiche all'elemento, ossia rispettivamente un campo citazione e un campo contente il percorso di un'immagine (di cui anche qui sono implementati i relativi metodi *getter*). Ovviamente si fa l'override dei metodi virtuali di confronto di *base_elem* per confrontare anche le informazioni aggiuntive.

Si è scelto di utilizzare il design pattern Visitor per consentirne l'arricchimento in maniera dinamica, nello specifico per creare dei widget (usati nell'area di gioco) diversi a seconda del tipo di elemento. A tal fine sono state realizzate la classe astratta *visitorInterface* e la relativa classe derivata *visitorInfo*, che memorizza il widget. Di conseguenza, nelle classi *base_elem*, *quote_elem* e *image_elem* sono stati inseriti i metodi virtuali *accept* per accettare i visitor.

Nel modello logico, abbiamo poi una classe *game_set* che è la classe contenitore di questi elementi e rappresenta il set. Questa classe viene identificata con un nome (che sarà anche il nome del file .json che ne memorizzerà le informazioni) e contiene anche un campo che rappresenta l'argomento e un vettore con gli attributi (in quanto questi valori non sono specifici di un elemento, bensì comuni a tutti gli elementi contenuti), oltre ovviamente ad un puntatore al primo elemento contenuto (di tutti questi campi sono implementati i relativi metodi *getter*). Il contenitore di per sè è implementato come fosse una lista singolarmente concatenata, nello specifico gli elementi sono rappresentati da una classe annidata *node* che contiene un puntatore a *base_elem* e un puntatore all'elemento successivo nella lista. Per scorrere questa lista si utilizza un iteratore (grazie alla classe annidata *iterator* di *game_set*). Si è scelto di usare una lista in quanto l'accesso casuale non è la priorità, infatti la maggior parte delle volte è necessario visualizzare tutti gli elementi (per scriverli/leggerli dal file .json, per visualizzarli nella modifica/creazione del set e nella visualizzazione del gioco stesso). Se invece si vuole effettuare operazioni in particolari elementi di questa lista (aggiungere o rimuovere un elemento, oppure semplicemente cercare o estrarre a caso un elemento) ci sono dei metodi appositi. Infine, abbiamo anche dei metodi usati per scrivere e eliminare i file .json che memorizzano tutte le informazioni del *game_set* in questione (vedi 4).

Sono presenti, al di fuori della classe, anche alcune funzioni che sono utili per la gestione dei *game_set* in relazione ai file .json (e.g. una funzione che restituisce tutti i nomi dei *game_set* di cui esiste il file .json, una funzione che mi crea un oggetto *game_set* a partire dal file .json, etc..).

Nel modello logico è presente anche una classe *game* che è utile per riassumere le specifiche della partita selezionate in precedenza (quindi quale set si è selezionato, ma anche numero di round e tentativi), ma anche informazioni sullo stato corrente della partita (implementato come un vettore di booleani *won_rounds* che per ogni round, rappresentato da una cella del vettore, indica se è stato vinto (true) o perso (false)). Di tutto ciò sono ovviamente disponibili i relativi metodi *getter* oltre che un metodo *win* usato per aggiornare il vettore di booleani *won_rounds* quando un round è vinto.

Abbiamo anche un file, *memoryManagement*, che ci fornisce delle funzioni per la lettura e scrittura di file.

3 Polimorfismo

CUDLE utilizza dei Visitor per estendere dinamicamente le funzionalità delle proprie classi. In particolare, esso viene usato per creare un widget usato in *gameArea* per mostrare le diverse tipologie di elementi. La visualizzazione prevede quindi tre diverse rese grafiche (una per ogni tipo di elemento):

- per elementi di tipo *base_elem*, un widget che dà alcuni indizi su come trovare la soluzione
- per elementi di tipo *image_elem*, un widget che visualizza l'immagine dell'elemento
- per elementi di tipo *quote_elem*, un widget che visualizza la citazione dell'elemento, inserita in un speciale layout ornamentale

implementate dalla classe *visitorInfo* che, per l'appunto, costruisce questi widget in base al tipo concreto dell'oggetto visitato.

Si preferisce, invece, usare dei dynamic cast in *editSetElement* per selezionare il corretto layout della finestra in quanto le modifiche da effettuare sono minime. Si utilizza comunque anche il visitor per visualizzare le informazioni extra (ad es. il path dell'immagine se il tipo concreto dell'elemento è *image_elem*, oppure la citazione se il tipo concreto dell'elemento è *quote_elem*).

4 Persistenza dei dati

Per la persistenza dei dati viene utilizzato il formato JSON, in particolare si ha un file per ogni set (da cui prende il nome). Questo file contiene tre oggetti chiave-valore, in particolare si ha che due di questi oggetti (*attributes* e *elements*) hanno come valore dei vettori che memorizzano rispettivamente tutti gli attributi e tutti gli elementi. Gli elementi in sè sono rappresentati perlopiù come una serie di semplici associazioni chiave-valore, e la serializzazione delle sottoclassi viene gestita aggiungendo una associazione "tipo" : "info aggiuntiva per quel tipo". Un esempio della struttura dei file è dato dai .json forniti assieme al codice in modo da illustrare brevemente le diverse strutture.

5 Funzionalità implementate

Le funzionalità implementate sono, per semplicità, suddivise in due categorie: funzionali ed estetiche. Le prime comprendono:

- salvataggio in formato JSON
- funzionalità di ricerca (con auto-completamento nella parte grafica) dell'elemento
- salvataggio, eliminazione, modifica e selezione dei contenitori (oltre che degli elementi)
- implementazione di un iteratore per scorrere la lista di elementi nel *game_set*

Le funzionalità grafiche:

- gestione del ridimensionamento
- uso di messaggi di avvertimento nel caso di problemi
- ogni tipologia di elemento ha una propria visualizzazione (sia nella creazione/modifica che durante il gioco)
- utilizzo di icone nei pulsanti e messaggi
- utilizzo di colori e stili grafici
- effetti grafici come cambio del colore alla selezione dei pulsanti

Le funzionalità elencate sono intese in aggiunta a quanto richiesto dalle specifiche del progetto.

6 Rendicontazione ore

Attività	Ore Previste	Ore Effettive
Studio e Progettazione	10	10
Sviluppo del codice del modello	10	12
Studio del framework Qt	10	13
Sviluppo del codice della GUI	10	15
Test e Debug	5	5
Stesura della relazione	5	3
Totale	50	58

Il monte ore è stato leggermente superato per il codice del modello (in particolare per quanto riguarda l'uso dei file .json) e per il codice della GUI in quanto è stato necessario più tempo del previsto per familiarizzare con il framework Qt e per risolvere alcuni problemi a esso relativi, comportando dunque anche un ulteriore studio del framework stesso per risolverli.