GALLERIA VIRTUALE

Progetto realizzato da **Alessio Berton (1229137)** per lo svolgimento della parte orale del corso di Programmazione ad oggetti dell' anno accademico 2020/2021

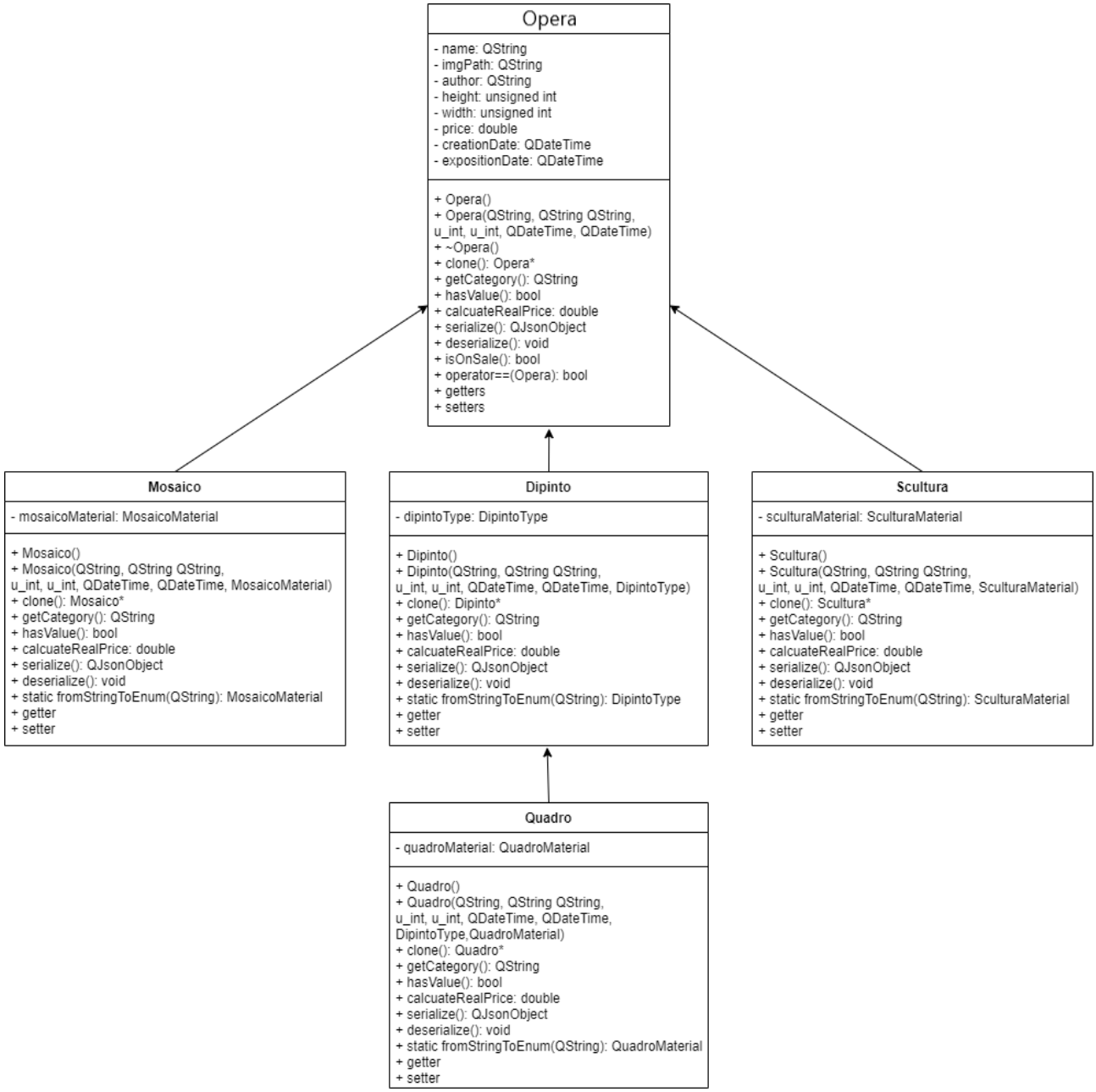
Primo tentativo di consegna per il secondo appello d'esame

Progetto creato singolarmente

INTRODUZIONE

Galleria Virtuale è un progetto nato per contrastare la riduzione di afflusso alle gallerie fisiche, dovute allo scoppio della pandemia di covid che ha colpito tutto il mondo. L'applicativo permette di vedere ogni opera, le caratteristiche della stessa e dell' autore. Inoltre permette di filtrare e cercare in base a dei valori. I dati vengono caricati e salvati in appositi file json.

GERARCHIA



La gerarchia ha altezza 2 con Opera come classe base astratta (contiene numerosi metodi virtuali puri descritti sotto) che vengono implementati dalle classi derivate.

CHIAMATE POLIMORFE

La gerarchia contiene numerosi metodi virtuali elencati qui sotto.

virtual ~***Opera***() = *default*
Distruttore virtuale puro alla base della gerarchia.

virtual Opera ****clone***() *const* = 0
Metodo virtuale costante puro che ritorna un double che rappresenta *il prezzo dell' Opera. Il prezzo varia in base a numerosi parametri, quali l' essere un opera di valore valore, se è in vendita, dal suo materiale e dalle dimensioni.*

virtual QString ***getCategory***() *const* = 0
Metodo virtuale costante puro che ritorna il nome della classe istanziata. Utile per capire, post rtti, quale classe della gerarchia venga rappresentata.

virtual bool ***hasValue***() *const* = 0
Metodo virtuale costante puro che ritorna vero se l'opera ha valore. Dipinto ha valore se dipinta di olio/cera/pastelli. Scultura ha valore se scolpita in marmo/terracotta/pietra/avorio. Un quadro ha valore se la cornice è in oro/ceramica/rame.

virtual double ***calcuateRealPrice***() *const*
Metodo virtuale costante che ritorna un double che rappresenta *il prezzo dell' Opera. Il prezzo varia in base a numerosi parametri, quali l' essere un opera di valore valore, essere in vendita o meno, dal suo materiale e dalle dimensioni.*

virtual QJsonObject ***serialize***() *const*
Metodo virtuale costante che ritorna un oggetto QJsonObject che serve a trasformare un oggetto nel suo rispettivo json corrispondente.

virtual void ***deserialize***(*const* QJsonObject&)
Metodo costante che non ritorna niente. Dato un QJsonObject, il metodo costruisce il relativo oggetto istanziabile della gerarchia.

CONTENITORE SCELTO

E' stato scelto di utilizzare un semplice array come contenitore per l'applicazione, perché il più semplice e duttile da implementare e usare per un progetto che contiene, anche a regime, pochissimi oggetti istanziati. Avendo un monte d'ore contenuto, non sono state implementate tutte le funzionalità avanzate (per esempio il for-each), ma solo quelle basiche, utili e significative (anche) viste a lezione.

FORMATI FILE

Per il caricamento e salvataggio delle opere esposte dall' applicazione viene usato il formato json, scelto perché personalmente più parlante e facile da usare dei formati xml o csv. Al momento del boot, l'applicazione cerca nella cartella data il file data.json per caricare le opere. Durante l'uso di Galleria Virtuale l'utente può caricare nuovi file dati, cancellare opere e successivamente ri-salvare il nuovo stato della galleria, creando un nuovo file o sovrascrivendo il file originale.

Un ipotetico snippet di file json, dopo aver invocato il metodo serialize() dall'applicazione, è il seguente:


```
[ {
    "author": "Alessio Berton",
    "category": "Quadro",
    "creationDate": "26/12/2015",
    "expositionDate": "10/05/2019",
    "height": 100,
    "imgPath": ":/img/quadroArgento.jpg",
    "name": "TestQaudro",
    "quadroMaterial": "Legno",
    "dipintoType": "Olio",
    "width": 100
  }, {
    "author": "Marco Mataracco",
    "category": "Quadro",
    "creationDate": "16/07/2012",
    "expositionDate": "11/12/2018",
    "height": 130,
    "imgPath": ":/img/quadroOro.jpg",
    "name": "MilanQuadrro",
    "quadroMaterial": "Oro",
    "dipintoType": "Olio",
    "width": 110
 } ]
```


MODUS OPERANDI DI SVILUPPO

Il progetto è stato sviluppato su ambiente windows ed è stato versionato tramte git per evitare problemi di inconsistenza dopo una modifica al codice (es: ad ogni implementazione di funzione GUI o modifica modello).

Alla fine dello sviluppo, ho impiegato qualche ora su ambiente linux per verificare l'applicativo e perché presente valgrind in maniera nativa, che mi ha permesso di trovare e risolvere dei memory leaks.

COMPILAZIONE ED ESECUZIONE

Per compilare il progetto è necessaria l'esecuzione di questi 3 comandi all' interno della cartella ottenuta dopo la decompressione del file .zip:

1. **qmake GalleriaVirtuale.pro**
2. **make**
3. **./GalleriaVirtuale**

ORE EFFETIVAMENTE RICHIESTE E CONSIDERAZIONI SUL PROGETTO

Il progetto in se non è stato troppo costoso in termini di tempo (anche se 50 ore sono veramente poche), avrei preferito sapere a priori se spendere x ore per implementare una nuova funzionalità utente o se fosse meglio utilizzare quel tempo per rendere più pulito e funzionale il sw (spesso le aziende preferiscono far uscire più funzionalità possibili a discapito di possibili problemi successivi. Degli esempi recenti possono essere Cyberpunk2077 o il rilascio degli aggiornamenti di windows 10). Ho quindi preferito rendere l'applicativo il più bug-free possibile, al costo di togliere la possibile funzionalità di inserimento di un'opera da GUI (la modifica ovviamente non avrebbe avuto senso, quella è e quella rimane). Ovviamente comprendo che è impossibile definire a priori una scelta di valutazione universale, quindi spero che la mia scelta sia stata corretta.

Le ore indicate sono spannometriche. La libreria qt l' ho "scoperta" mentre risolvevo problemi o necessitavo di funzionalità, mentre l'analisi preliminare del problema è stata pressoché inesistente (ho contato il download e l'apprendimento di qt creator, preparare la repository su github e l'idea di fare questa applicazione con il pattern visto al tutorato).

Analisi preliminare del problema: 2 Ore

Progettazione modello e GUI: 5 Ore

Apprendimento libreria Qt: 3 Ore

Codifica modello e GUI: 30 Ore

Testing e debugging: 14 Ore