Qontainer – Museo

Relazione sul progetto di Programmazione ad oggetti

Lorenzo Dei Negri – 1161729

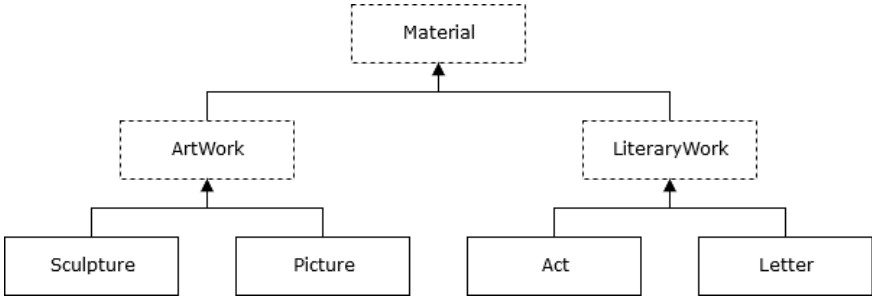
A.A. 2018/2019

INDICE

SCOPO DEL PROGETTO

Il progetto ha lo scopo di realizzare un'applicazione munita di GUI che permetta la completa gestione di materiali in dotazione ad un museo, tramite l'uso di contenitori. Ogni oggetto mantenuto dal programma è definito da delle caratteristiche comuni e da altre peculiari per la specifica tipologia di materiale.
Tramite l'interfaccia grafica è possibile aggiungere dei nuovi materiali, visualizzare e modificare le informazioni dei singoli oggetti già memorizzati, effettuare delle ricerche in tempo reale specifiche basate sulle loro caratteristiche particolari e rimuovere oggetti precedentemente inseriti.
Inoltre l'applicazione permette anche di effettuare delle procedure di restauro e di prestito dei materiali disponibili, mantenendo traccia di tutte le spese e i ricavi accumulati con queste operazioni, e gestisce anche il valore intrinseco dei vari oggetti e il suo mutamento in base agli interventi effettuati.
Infine è possibile, in ogni momento, salvare lo stato del contenitore attualmente in uso su file, oppure caricare dei materiali da un salvataggio precedente.

GERARCHIA DI TIPI



La gerarchia di tipi utilizzata dall'applicazione è rappresentata dallo schema sovrastante ed è formata dalla seguenti classi:

- ***Material*:** è la classe base della gerarchia ed è una classe astratta polimorfa, infatti contiene dei metodi virtuali puri che verranno descritti nell'apposita sezione. Rappresenta un generico materiale che può essere in dotazione ad un museo ed è caratterizzata dai seguenti attributi che permettono di mantenere le informazioni proprie del materiale oltre a quelle di interesse per il museo: autore, titolo, luogo di realizzazione, data di realizzazione e un attributo booleano per segnalare un eventuale danneggiamento per il materiale in se; settore, valore di base, numero di restauri, numero di prestiti, ricavi accumulati, spese accumulate e due attributi booleani che indicano se il materiale è di proprietà di un privato e è al momento disponibile. Inoltre è presente un attributo che mantiene il percorso, all'interno del file system, della foto del materiale.
- ***ArtWork*:** è una sottoclasse diretta della classe base ed è una classe astratta polimorfa, in quanto non implementa tutti i metodi virtuali puri dichiarati nella classe *Material*, ma solamente alcuni di essi. Rappresenta una generica opera d'arte in dotazione ad un museo ed è caratterizzata dai seguenti attributi, che vanno ad aggiungersi a quelli ereditati da *Material*, in modo da mantenere le informazioni generali su opere artistiche: materiale col quale è stata realizzata, tecnica di realizzazione, movimento artistico a cui appartiene e il soggetto che essa rappresenta.
- ***LiteraryWork*:** è una sottoclasse diretta della classe base ed è una classe astratta polimorfa, in quanto non implementa tutti i metodi virtuali puri dichiarati nella classe *Material*, ma solamente alcuni di essi. Rappresenta una generica opera letteraria (intesa come qualsiasi tipo di produzione scritta) in dotazione ad un museo ed è caratterizzata dai seguenti attributi, che vanno ad aggiungersi a quelli ereditati da *Material*, in modo da mantenere le informazioni generali sulle scritture: lingua e stile con cui è stata scritta, e due attributi booleani che danno l'indicazione se l'opera è completa e se è stata scritta a mano.
- **Sculpture:** è una sottoclasse diretta della classe *ArtWork* ed è una classe concreta istanziabile, in quanto implementa tutti i metodi virtuali puri non definiti dalla sua superclasse. Rappresenta una scultura in dotazione ad un museo ed oltre agli attributi ereditati aggiunge l'indicazione sulla forma, ossia come è stata realizzata.
- **Picture:** è una sottoclasse diretta della classe *ArtWork* ed è una classe concreta istanziabile, in quanto implementa tutti i metodi virtuali puri non definiti dalla sua superclasse. Rappresenta un quadro in dotazione ad un museo ed oltre agli attributi ereditati aggiunge un indicatore booleano per segnalare se si tratta di un quadro disegnato e dipinto, oppure se è una fotografia.

- **Act:** è una sottoclasse diretta della classe *LiteraryWork* ed è una classe concreta istanziabile, in quanto implementa tutti i metodi virtuali puri non definiti dalla sua superclasse. Rappresenta un atto ufficiale in dotazione a un museo e oltre agli attributi ereditati aggiunge l'indicazione dell'oggetto del documento.
- **Letter:** è una sottoclasse diretta della classe *LiteraryWork* ed è una classe concreta istanziabile, in quanto implementa tutti i metodi virtuali puri non definiti dalla sua superclasse. Rappresenta lettera in dotazione a un museo, oltre agli attributi ereditati aggiunge il nome del destinatario a cui è stata spedita.

Tutte le classi della gerarchia, oltre ai metodi virtuali e virtuali puri necessari per sfruttare il polimorfismo, forniscono anche dei get e dei set per gli attributi privati. Inoltre, sono presenti, in ogni classe, dei campi dati statici che forniscono un prezzo di base oppure un sovrapprezzo, fisso per tutti i materiali di un certo tipo, che si applica alle operazioni di restauro e di prestito.
Infine, la gerarchia rappresenta solo un sottoinsieme ridotto dei possibili materiali che un museo espone, quindi la si può facilmente estendere ereditando dalla classe base astratta o dalle sottoclassi e implementando opportunamente i metodi virtuali, per aggiungere nuovi tipi di materiali.

METODI POLIMORFI

Metodi protetti:

- **virtual float calculateRestorationCost() const:** questo metodo calcola e ritorna il costo di un restauro in base alle caratteristiche del materiale da restaurare e in base al suo tipo, in quanto ci sono i campi dati statici che forniscono il costo di base o un eventuale sovrapprezzo. Ogni sottoclasse, infatti, lo può ridefinisce per tenere conto delle caratteristiche statiche.
- **virtual float calculateLoanProceed() const:** questo metodo calcola e ritorna il ricavo di un prestito in base alle caratteristiche del materiale da prestare e in base al suo tipo, in quanto ci sono i campi dati statici che forniscono il ricavo di base o un eventuale sovrapprezzo. Ogni sottoclasse, infatti, lo può ridefinisce per tenere conto delle caratteristiche statiche.

I due precedenti metodi sono definiti con modificatore d'accesso protetto in quanto non sono funzionalità pensare per essere usufruite direttamente da un cliente della classe, ma sono utilizzate solamente all'interno dei metodi *void restore()* e *void lend()* che effettuano concretamente il restauro o il prestito del materiale d'invocazione, e quindi hanno bisogno di calcolare l'ammontare delle operazioni che effettuano.
Anche se questi due metodi non sono virtuali, la chiamata effettuata al loro interno è comunque polimorfa, in base al tipo dinamico del materiale di invocazione.

Metodi pubblici:

- **virtual ~Material() = default:** questo è il distruttore in versione standard che è stato marcato come metodo virtuale per fare in modo che la distruzione degli oggetti della gerarchia venga fatta in modo polimorfo, richiamando il distruttore corretto in base al tipo di materiale che lo invoca.
- **virtual Material * clone() const = 0:** questo è il metodo di clonazione che effettua la copia profonda polimorfa del materiale di invocazione, ritornandone un puntatore. Esso viene ridefinito solo nelle classi concrete, in quanto richiede che l'oggetto di cui si vuole fare la copia sia di una classe istanziabile, infatti è dichiarato come metodo virtuale puro; inoltre le ridefinizioni sfruttano la covarianza sul tipo di ritorno in modo da sfruttare l'informazione di che tipo di materiale si sta clonando.
- **virtual std::string getType() const = 0:** questo metodo ritorna una stringa che identifica il tipo dell'oggetto di invocazione (in modo univoco). Questa informazione è utile per la serializzazione, in quanto indica all'interno del file di salvataggio il tipo di materiale salvato, in modo da poterlo ricaricare nell'applicazione in un secondo momento.
  Inoltre può essere usata per effettuare il controllo del tipo dinamico di un materiale durante la normale esecuzione del programma; in questo modo, al posto di utilizzare il dynamic_cast, si sfruttano le v_tables dei diversi oggetti per fare l'identificazione del tipo dinamico degli stessi attraverso dynamic binding, operazione meno onerosa, dal punto di vista computazionale, rispetto al dynamic_cast.
- **virtual std::string getMaterialType() const = 0:** questo metodo ritorna una stringa che identifica il supertipo astratto dell'oggetto di invocazione (in modo univoco). Questa informazione è utile per la serializzazione, in quanto indica all'interno del file di salvataggio il supertipo di materiale salvato, in modo da poterlo ricaricare nell'applicazione in un secondo momento.
  Inoltre può essere usata per effettuare il controllo del tipo dinamico di un materiale durante la normale esecuzione del programma; in questo modo, al posto di utilizzare il dynamic_cast, si sfruttano le

v_tables dei diversi oggetti per fare l'identificazione del tipo dinamico degli stessi attraverso dynamic binding, operazione meno onerosa, dal punto di vista computazionale, rispetto al dynamic_cast.

- **virtual std::string getInfo() const:** questo metodo ritorna una stringa contenente tutte le informazioni del materiale di invocazione in formato coppia chiave-valore composta da nome-valore degli attributi. In questo modo si può facilmente ottenere una descrizione testuale del materiale d'invocazione. Non è marcato virtuale puro in quando nella classe base sono presenti degli attributi valorizzabili e quindi stampabili, ogni classe derivata può ridefinire il metodo per aggiungere, alla stringa ottenuta richiamando il metodo della sua superclasse, le informazioni proprie.
- **virtual float calculateValue() const:** questo metodo calcola e ritorna il valore intrinseco di un materiale in base alle sue caratteristiche e in base al suo valore di base. Ogni sottoclasse, infatti, lo può ridefinisce per tenere conto delle caratteristiche peculiari

FORMATO DEL FILE DI CARICAMENTO E SALVATAGGIO

La scelta progettuale per effettuare il caricamento e il salvataggio degli oggetti del contenitore gestito dall'applicazione è stata quella di utilizzare il linguaggio XML. È stata presa questa decisione in quanto XML è uno standard mondiale per l'archiviazione e la condivisione di dati tra applicativi software e, inoltre, è un linguaggio di facile manipolazione ed interpretazione anche per un essere umano. Infine la libreria Qt mette a disposizione delle classi apposite per la completa gestione di file XML.
La lettura e scrittura di file XML sono state implementate in una classe apposita del modello *fileHandler*, che si occupa solamente di questo. In tal modo si garantisce una netta separazione tra modello e vista, per quanto riguarda la gestione di file XML, con la sola classe sopracitata che funge da interfaccia tra i due.
Le funzioni per la lettura e la scrittura sono le seguenti:

- **Container<DeepPointer<Material>> read() const:** il metodo read seleziona il file indicato nell'apposito campo dati della classe, effettua i controlli di esistenza e di utilizzabilità, lo apre e inizia ad effettuare il parsing del contenuto istanziando un nuovo oggetto, di tipo opportuno, in base ai dati che legge dal file all'interno del tag che identifica il materiale corrente, e inserendolo in un nuovo contenitore locale di cui ritornerà una copia a fine lettura.
  Se durante la sua esecuzione si presentano degli errori causati dalla errata formattazione del file o dall'impossibilità nell'aprirlo, allora viene lanciata un'opportuna eccezione di tipo *FileException*, un'apposita classe per la gestione delle eccezioni in *FileHandler*.
  Il contenitore ritornato a fine esecuzione conterrà tutti i materiali che erano stati correttamente salvati sul file.
- **void write(const Container<DeepPointer<Material>> &) const:** il metodo write seleziona il file indicato nell'apposito campo dati della classe, effettua i controlli di esistenza e di utilizzabilità, lo apre e inizia ad effettuare la scrittura, materiale per materiale, di tutti gli oggetti presenti nel contenitore che riceve come parametro. Per ogni materiale apre un apposito tag **<Material>** nel quale aggiunge come attributi l'indicazione del supertipo astratto e del tipo dello stesso tramite i metodi polimorfi visti in precedenza, successivamente procede, campo dati per campo dati, a scriverne i valori, creando un nuovo tag, per ciascuno di essi, del tipo **<attributo>valore</attributo>**.

Un esempio, generico, di formattazione del file XML è il seguente:

```
<?xml version="1.0" encoding="UTF-8"?>
<! --File generato automaticamente! Non modificare manualmente!-- >
<Materiali>
        <Materiale tipo="tipo_concreto" tipoMateriale="tipo_astratto">
                ...
                <attributo>valore</attributo>
                ...
        </Materiale>
        ...
        <Materiale tipo=”tipo_concreto” tipoMateriale=”tipo_astratto”>
        ...
        </Materiale>
</Materiali>
```

MODELLO

Per la realizzazione del modello è stato implementato un template di classe *container*<*T*> che rende disponibile un contenitore sotto forma di array dinamico. Il template *container*<*T*>, oltre a fornire un contenitore utilizzabile come quelli della libreria standard STL, implementa, al suo interno, anche le classi iterator e const_iterator che realizzano degli iteratori allo stesso modo della libreria sopracitata. La classe fornisce quindi le tipiche operazioni di inserimento, eliminazione, ricerca e accesso in posizione arbitraria, oltre a diversi overload degli operatori previsti da i contenitori standard.
La scelta progettuale di preferire un array dinamico ad una lista doppiamente linkata è stata effettuata in base a diverse motivazioni che vengono di seguito riportate:

1. L'applicazione richiede molti accessi in posizione arbitraria agli elementi del contenitore per effettuare operazioni di modifica e di ricerca, e quasi sempre questi accessi vengono effettuati tramite indice e non tramite iteratore. Di conseguenza mentre un array dinamico permette di effettuare accessi tramite indice in tempo costante O(1), una lista impiegherebbe tempo lineare O(n) in quanto deve scorrere tutti i nodi fino a quello richiesto.
2. Per lo scopo dell'applicazione, memorizzare e gestire i materiali in dotazione ad un museo, l'inserimento di nuovi oggetti può essere effettuato sempre in coda al contenitore, operazione effettuata in tempo ammortizzato costante O(1) da un array dinamico, quindi equivalente ad una lista.
3. L'applicazione prevede la possibilità di effettuare eliminazioni in posizione arbitraria, operazione eseguita in tempo lineare O(n) da un array dinamico, mentre in tempo costante O(1) da una lista. Tuttavia, per quanto la lista risulterebbe più efficiente per le eliminazioni, il numero di volte che queste vengono eseguite è significativamente minore rispetto a quello delle operazioni di inserimento e accesso in posizione arbitraria.
   Inoltre, la maggiore efficienza di una lista è solo apparente, in quanto l'elemento da eliminare viene identificato tramite indice, e non tramite iteratore, perciò prima di poter effettuare la rimozione in tempo costante O(1) risulta necessaria una ricerca in tempo lineare O(n).

Per questi motivi quindi si è preferito adottare un array dinamico al posto di una lista doppiamente linkata.

Inoltre, è stato realizzato anche un template di classe *DeepPointer*<*T*> che implementa dei puntatori smart a tipo T, per automatizzare la gestione della memoria in modalità profonda. Quindi il modello utilizza un contenitore di classe *container*<*DeepPointer*<*Material*>>, ossia istanziato con puntatori smart polimorfi alla classe base astratta della gerarchia di tipi.
In questo modo, pur mantenendo il polimorfismo del puntatore a *Material*, si automatizza la gestione della memoria in modalità profonda degli stessi.

GUI

Per la realizzazione dell'interfaccia grafica è stata fatta la scelta progettuale di adottare il design pattern *Model-View*, preferendolo a quello *Model-View-Controller*. È stata presa questa decisione in quanto il primo, pur non avendo il *Controller*, garantisce una completa separazione tra modello e vista, accorpando in quest'ultima le funzionalità del *Controller*. Inoltre questo design pattern è completamente implementato dalla libreria Qt, quindi, in questo modo, si possono sfruttare a pieno tutte le potenzialità del framework anche per implementare le operazioni dell'applicazione e ridurne la complessità.
Nel realizzare la visualizzazione dei dati è risultato opportuno rappresentare il contenitore tramite la classe List derivata dalla classe QListView fornita da Qt, combinata alle classi SortFilterProxyModel, derivata da QSortFilterProxyModel, e ListModelAdapter derivata da QabstractListModel, entrambe le classi base sono fornite da Qt, e si occupano di mantenere aggiornata la vista rispecchiando le modifiche effettuate al contenitore. La classe QSortFilterProxyModel permette anche di effettuare ricerche, per diverse tipologie di attributi, tra tutti gli oggetti memorizzati con visualizzazione dei risultati aggiornata dinamicamente. Inoltre gestisce in automatico la corrispondenza tra indici reali del contenitore ed indici virtuali della lista; in questo modo è possibile effettuare tutte le operazioni di modifica anche sui sottoinsiemi di materiali ottenuti dalle operazioni di ricerca.
Per favorire l'estensibilità del codice e mantenere modello e vista nettamente separati è stata fatta un'ulteriore scelta progettuale, che consiste nell'implementare le ulteriori finestre della vista ereditando la classe *QWidget* fornita da Qt, in questo modo sono state realizzate le finestre di inserimento, tramite la classe Insert, e interazione tramite la classe Interact, le quali formano la schermata principale dell'applicazione e permettono, rispettivamente, di inserire nuovi materiali inserendo i dati nell'apposito form, e di interagire con gli oggetti già

memorizzati; in particolare è possibile effettuare ricerche, modificare i dati di un singolo materiale oppure di visualizzarli in una finestra separata, e fare le operazioni previste dalla gerarchia di tipi. Le altre due finestre implementate in questo modo sono, appunto, quelle che permettono la modifica e la visualizzazione di un singolo elemento, selezionato dalla , e sono state realizzate tramite le classi Modifies e Details.
Inoltre è stato realizzato un menù ad hoc, tramite la classe Menu ereditata dalla classe QMenuBar, fornita da Qt, che permette la navigazione tra le due finestre citate in precedenza e contiene un sottomenù per la gestione del caricamento e del salvataggio dei dati tramite file XML.
Infine sono state implementate anche delle classi che realizzano delle combo box ad hoc per i filtri di ricerca e inserimento, derivando la classe QComboBox di Qt.

NOTE

Il progetto è stato realizzato interamente all'interno della macchina virtuale fornita.

- **Sistema operativo:** Ubuntu 18,04,1 LTS 64bit
- **Versione del compilatore:** GCC 7.3.0 64bit
- **Versione di Qt:** 5.9.5

Inoltre, sono state apportate delle modifiche al file progetto.pro generato automaticamente tramite il comando *qmake -project*, quindi oltre ai sorgenti e alla relazione viene consegnato anche il file progetto.pro necessario per la corretta compilazione tramite comandi: *qmake => make*

Infine, il progetto viene fornito con dei dati di esempio pre-caricati, salvati nel file XML data.xml che verrà caricato automaticamente all'avvio, e delle immagini e icone necessarie per il corretto funzionamento dell'applicazione. Questi file sono situati in apposite cartelle chiamate *Data*, *Photos* e *Icons*, presenti nella stessa directory dei file sorgenti. È opportuno che l'eseguibile venga lanciato dalla medesima directory per evitare errori con i percorsi dei file.

ORE IMPIEGATE

- **Analisi preliminare problema** 1h
- **Progettazione modello** 2h
- **Progettazione GUI** 2h
- **Apprendimento libreria Qt** 10h
- **Codifica modello** 13h
- **Codifica GUI** 14h
- **Debugging** 4h
- **Testing** 2h
- **Relazione** 4h