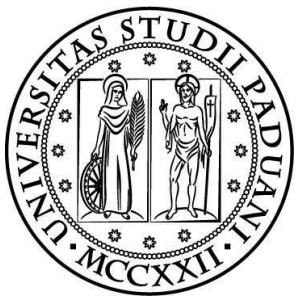


Università degli Studi di Padova

Dipartimento di matematica

"Museum Tickets Manager"

Progetto del corso di Programmazione ad Oggetti

Jordan Gottardo
Matricola: 1070703

Anno Accademico: 2015/2016

Introduzione

Il progetto è stato sviluppato utilizzando Qt Creator 3.5.1 con versione delle librerie Qt 5.5.1 in ambiente Windows 10 64 bit. Il compilatore utilizzato è MinGW 32 bit. La GUI è stata sviluppata utilizzando Qt Designer nelle prime fasi di progettazione (soprattutto per la gestione dei layout), mentre per quanto riguarda la versione finale è stata realizzata interamente via codice.

Scopo del progetto

Lo scopo del progetto è fornire un'applicazione, corredata da interfaccia grafica, per la gestione di biglietti di museo, che dia la possibilità di memorizzarli, visualizzarli, rimuoverli e modificarli e permetta di avere informazioni "riassuntive" su tutti i biglietti emessi, tra le quali il numero totale di biglietti emessi e il guadagno totale generato dalla vendita degli stessi.

Descrizione parte logica

- **Biglietto**
  La classe **biglietto** è una classe astratta da cui derivano i vari tipi di biglietto che l'applicazione è in grado di gestire. Tra i campi dati propri (privati) troviamo:
  - l'ID, caratterizzato da una QString inserita dall'utente;
  - il prezzo base del biglietto;
  - l'età del soggetto che ha acquistato il biglietto
  - la data in cui è stato acquistato (rappresentata da una QDate per utilizzare le funzionalità messe già a disposizione da QT).

  Per ogni campo dati sono presenti metodi di get e set, in modo da fornire l'accesso in lettura/scrittura ai membri privati solo in modo controllato. Dato che **biglietto** è la classe base di una gerarchia polimorfa, l'ho dotata di un distruttore virtuale, in modo che quando si andrà ad invocare la delete su di un puntatore polimorfo, verrà invocata tramite dynamic binding quella relativa al tipo dinamico del puntatore nel momento della sua cancellazione e non quella relativa al suo tipo statico. Ciò permette di evitare di lasciare garbage sullo heap.
  Biglietto è astratta in quanto al suo interno sono definiti due metodi virtuali puri, *calcolaPrezzo* e *clone*: il primo serve a calcolare il prezzo totale di un biglietto, il secondo ritorna un puntatore ad una copia del biglietto su cui è invocato. Entrambi i metodi verranno implementati dalle classi concrete derivate da biglietto.
- **BigliettoOrdinario**
  La classe **biglietto ordinario** è una classe concreta derivata da **biglietto**; ha gli stessi campi dati di quest'ultimo (ereditati pubblicamente tramite il sottooggetto) ed implementa i metodi *calcolaPrezzo* e *clone*. Il prezzo totale è semplicemente calcolato ritornando il prezzo base, mentre *clone* raffina il tipo di ritorno, specificandone uno covariante con **biglietto***, per meglio rappresentare il tipo del biglietto copiato.
- **BigliettoRidotto**
  La classe **biglietto ridotto** è una classe astratta derivata da **biglietto**. Ha un unico campo dati proprio, *riduzioneFissa*, che rappresenta la riduzione fissa applicata al prezzo base di un biglietto; come metodi propri ha metodi di get e set per il campo dati.
- **RidottoBambini**
  La classe ridotto bambini è una classe concreta derivata da biglietto ridotto. Come campo dati proprio ha un campo double riduzione variabile, che rappresenta la percentuale (0-100) di riduzione da applicare al prezzo base per ogni anno di età in meno rispetto alla soglia d'età (definita come campo dati statico), oltre la quale il soggetto non è più considerato un bambino. L'implementazione del metodo *calcolaPrezzo* è la seguente:
  PrezzoBase-RiduzioneFissa-[(soglia-Età_visitatore)*RiduzioneVariabile%]*PrezzoBase.
  Il metodo clone è implementato come usuale.

- **RidottoDisabili**
  La classe **ridotto disabili** è una classe concreta derivata da biglietto ridotto. Come campo dati proprio ha un booleano accompagnatore, che indica se il visitatore è accompagnato da qualcuno oppure no. I disabili entrano sempre gratis, quindi *calcolaPrezzo* ritorna sempre 0, e il metodo *clone* è implementato come usuale.
- **ListaBiglietti**
  La classe **lista biglietti** è il contenitore da me definito: è una lista che memorizza puntatori a biglietto polimorfi con funzionalità di aggiunta in testa e in coda, in modo da permettere all'utilizzatore del programma di far "crescere" la lista in entrambi i versi (ad esempio nel caso si volessero memorizzare i biglietti seguendo l'ordine cronologico di emissione e ci si fosse dimenticati di registrarne uno emesso prima di tutti gli altri, è possibile inserirlo in testa mantenendo l'ordinamento invariato). Sono anche disponibili i duali dei due metodi appena descritti, *popFront e popBack*, che permettono rispettivamente di togliere il primo e l'ultimo biglietto. Altri metodi presenti sono: *svuota*, che permette di svuotare l'intera lista, *vuota* che ritorna true se e solo se la lista non contiene nessun elemento, *bigliettiEmessiTot* che restituisce il numero di biglietti presenti, *redditoTotale* che restituisce la somma di tutti i prezzi dei biglietti emessi, *giornoPiuRedditizio* che restituisce un intero (1=lunedì..7=domenica) relativo al giorno della settimana che ha generato più reddito*, alaPiuRedditizia* che restituisce l'intero relativo all'ala del museo che ha generato più reddito e infine *redditoFasciaEtà* che restituisce la somma dei prezzi dei biglietti acquistati da soggetti con età compresa in un dato range.
  Il contenitore è dotato di una classe iteratore, che racchiude al suo interno un puntatore polimorfo a biglietto, che permette di scorrere l'intera lista tramite l'operatore ++ e di accedere ad un dato elemento tramite l'operatore[], dei quali sono definiti gli usuali overloading.
  La gestione della memoria è gestita tramite smart pointers per garantire estensibilità all'applicazione (vedi sezione "scelte progettuali").

Descrizione parte grafica

- **Mainwindow**
  E' una classe eredita da QMainWindow e presenta dei QButtons che permettono di aprire le altre finestre e delle QLineEdit/QSpinBox che permettono di visualizzare i risultati restituiti dai metodi di listaBiglietti. Il menù della Mainwindow presenta tre QAction:
  - *apri*, che permette di riaprire la lista salvata in una sessione precedente, leggendola dal file dati.xml (che si presume sia presente nella directory di build e che sia stato creato dal programma stesso);
  - *salva*, che permette di salvare la lista presente in memoria all'interno dello stesso file xml, sovrascrivendo lo stesso con i nuovi dati;
  - *esci*, che permette di uscire dal programma.

  Mainwindow, come le altre finestre, ha degli slot da me definiti che ho poi collegato ai signal generati dai vari widget tramite le connect; inoltre, ho definito il signal *sigAggiornaTutti* che ho poi collegato agli slot che si occupano di visualizzare i risultati dei vari metodi riassuntivi di listaBiglietti. Il signal serve per aggiornare i risultati contemporaneamente dopo che l'utente ritorna da una delle finestre che potrebbero aver modificato la lista.
- **Aggiungiwindow**
  E' una classe che, come tutte le altre finestre del progetto (tranne la Mainwindow), deriva pubblicamente da QDialog: questa scelta è motivata dal fatto che ho utilizzato il metodo *setModal* per ottenere una finestra che avesse il focus esclusivo e permettesse all'utente di concentrarsi su ciò che ha davanti al momento, senza avere la possibilità di cliccare le finestre sottostanti. La *Aggiungiwindow* permette di inserire nuovi biglietti alla lista; l'interazione dell'utente con il QComboBox per la scelta del tipo attiva uno slot che si occupa di attivare/disattivare i campi di input relativi al tipo scelto. Viene anche offerta la possibilità di inserire il biglietto in coda oppure in testa tramite dei *QRadioButton*.
- **Rimuoviwindow**
  E' la finestra che permette di rimuovere i biglietti. Dopo che un biglietto è stato rimosso, lo slot *slModificaUltimo* provvede a visualizzare le informazioni relative ad esso.

- **Confermawindow**
  E' una semplice finestra che chiede conferma all'utente se è sicuro di voler eliminare l'intera lista.
- **Modificawindow**
  E' la finestra che permette di modificare un biglietto dato un certo ID. Quando la QLineEdit relativa all'ID perde il focus, viene fatta la ricerca all'interno della lista: se è stato trovato il biglietto con quel dato ID, i campi dati vengono attivati/disattivati a seconda del tipo scelto e popolati con le informazioni relative al biglietto
- **Visualizzawindow**
  E' la finestra che contiene una QPlainTextEdit, la quale visualizza l'intera listaBiglietti.
- **Sogliawindow**
- E' la finestra che permette di modificare il campo dati statico *sogliaEta* della classe ridottoBambini.

Scelte progettuali

Contenitore: ho scelto di utilizzare una lista in quanto dovrebbe avere una maggiore efficienza per quanto riguarda le rimozioni dal "centro" della struttura dati, funzionalità che è data tramite il "rimuovi biglietto da ID" all'interno di **Rimuoviwindow**. Inoltre viene anche fornita la possibilità di aggiungere in testa alla lista, oltre che in coda, anche questa operazione dovrebbe avere una maggior efficienza rispetto ad un array.

Eccezioni: ho cercato di ridurre al minimo l'utilizzo di eccezioni per evitare di complicare la logica del flusso di esecuzione, mantenendo comunque il controllo dei casi limite tramite controlli preventivi prima di richiamare determinati metodi. Ad esempio, all'interno della **Mainwindow**, gli slot che vanno ad invocare i metodi riassuntivi di **listaBiglietti** fanno un controllo preventivo per identificare quando la lista è vuota, e solamente quando non lo è invocano il metodo.

Gestione della memoria: gli oggetti di tipo biglietto (e relativi oggetti di classi derivate) sono allocati esclusivamente sullo heap. Il distruttore virtuale si occupa di invocare il distruttore corretto a seconda del tipo dinamico del puntatore a biglietto, in modo da non lasciare garbage distruggendo esclusivamente il sotto-oggetto. Malgrado non fossero fondamentali per il mio progetto, **listaBiglietti** implementa degli smart pointers per motivi di estendibilità dell'applicazione, ad esempio in caso in cui si volessero tenere più liste in contemporanea e si volesse fare condivisione di memoria di parte di queste, gli smart pointers renderebbero tutto ciò possibile. Per quanto riguarda questo progetto, essendoci una sola lista, quando viene invocata la delete sul primo smartp il campo riferimenti del nodo puntato viene messo a 0 e ricorsivamente viene invocata la delete sul next (che è uno smartp): tutto ciò comporta una distruzione profonda dell'intera lista.
Per quanto riguarda la parte grafica, ho lasciato la gestione della memoria a Qt: assegnando un parent ad un widget, quando il padre viene distrutto allora viene distrutto anche il figlio. Molteplici esempi di ciò si possono trovare nei costruttori delle varie finestre: tutti i widget all'interno della finestra hanno come parent la finestra stessa e al momento dell'invocazione del metodo *quit* o *close* sulla finestra viene invocato il distruttore di tutti i widget figli. Stessa cosa vale per i layout.

Utilizzo di Qt Creator: ho utilizzato Qt Creator per avere una prima bozza di interfaccia e per la gestione dei layout, successivamente ho proceduto a realizzare anche le classi grafiche interamente via codice.

RTTI: l'unico operatore di RTTI che ho utilizzato è il dynamic_cast, soprattutto quando ho avuto la necessità di stampare i vari campi dati che dipendono dal tipo dinamico a cui punta il puntatore polimorfo: per poter accedere a tali campi dati, ho avuto necessità di fare downcasting per far coincidere il tipo statico con il tipo dinamico del puntatore.

Descrizione dell'uso di codice polimorfo

La classe **biglietto** possiede tre metodi virtuali, di cui due virtuali puri, quindi è una classe base polimorfa. Ciò significa che avendo un puntatore alla classe base e andando ad invocare un metodo virtuale su quel puntatore, verrà invocato il metodo relativo al tipo dinamico del puntatore invece che relativo al tipo statico. Il late binding viene sfruttato in vari punti del mio progetto, tra cui:

- Nei metodi *redditoTotale*, *redditoFasciaEta* e *giornoPiuRedditizio* di listaBiglietti: vado a scorrere il contenitore di puntatori polimorfi alla base e invoco il metodo *calcolaPrezzo*, virtuale puro nella base e

avente implementazioni diverse nelle classi derivate concrete. Il metodo che verrà realmente invocato sarà quello relativo al tipo dell'oggetto puntato dal puntatore polimorfo alla base, che quindi effettuerà il calcolo del prezzo in modo corretto;

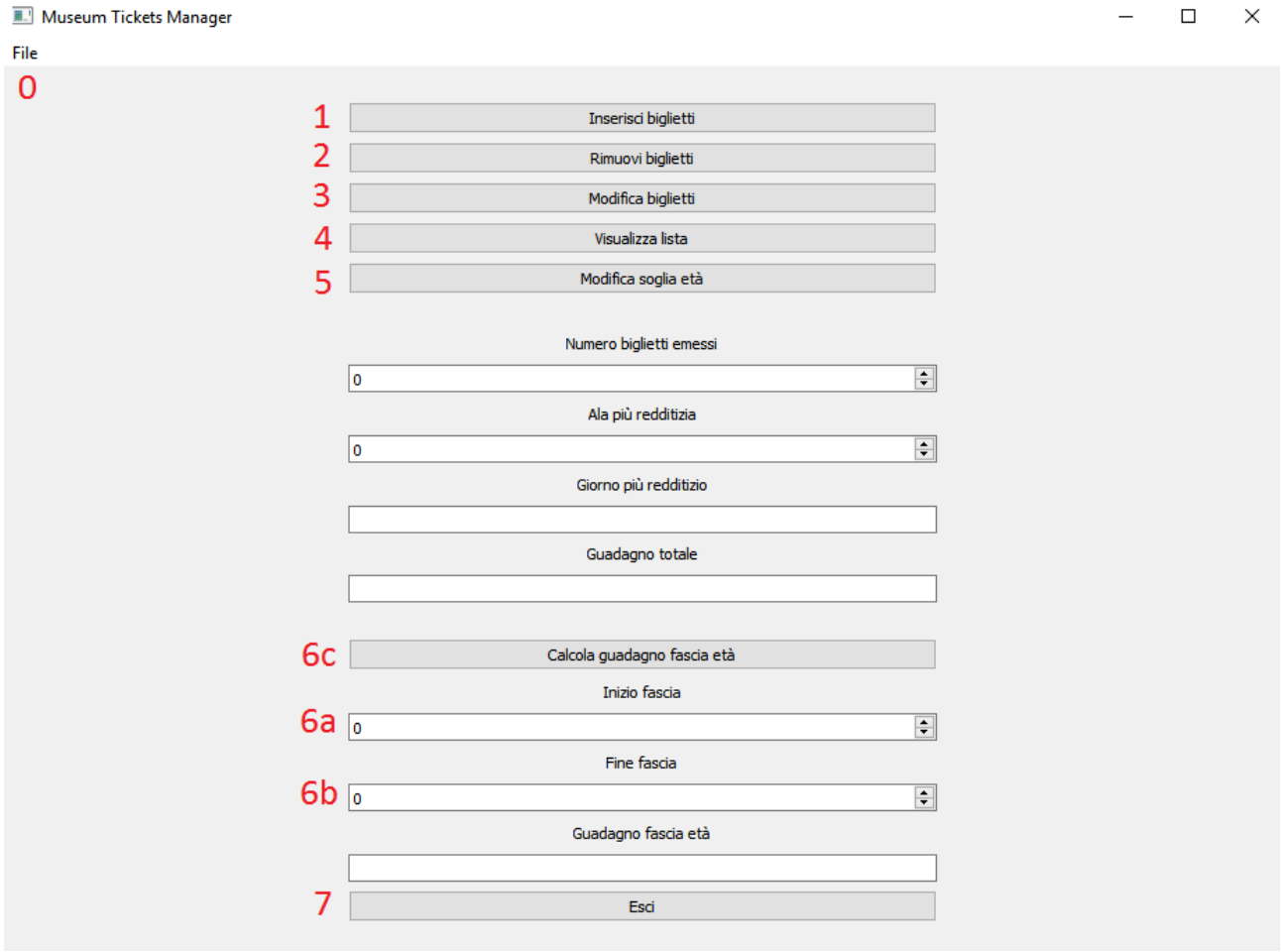
- Nei metodi *popFront*, *popBack*, *togliBiglietto* di listaBiglietti: dato che il metodo deve ritornare una copia del biglietto rimosso, prima che venga eliminato dalla lista (ad opera dell'operator= ridefinito nella classe smartp) ne faccio una copia profonda utilizzando il metodo clone(), anch'esso virtuale puro nella base e implementato nelle derivate concrete. Il suo contratto è fare una copia profonda del biglietto di invocazione e restituirne il puntatore. Le implementazioni del metodo virtuale puro sono davvero implementazioni anche se il tipo di ritorno cambia, in quanto il nuovo tipo è covariante con il tipo specificato nel metodo virtuale puro. Quindi, verrà invocato il corretto metodo di clonazione a seconda del tipo dinamico del puntatore a biglietto
- Ogni volta che viene invocata una delete, sia implicitamente che esplicitamente, su di un puntatore alla classe base, viene richiamato il distruttore corretto, sempre a seconda del tipo dinamico del puntatore: questo comportamento si ha in quanto il distruttore è dichiarato virtuale nella classe base.

Nei casi in cui non è stato possibile implementare alcune funzionalità tramite metodi virtuali, ho utilizzato il dynamic_cast come operatore di RTTI; alcuni esempi sono:

- Metodo **Mainwindow**::*slSalvaXml,* dove ho avuto la necessità di identificare il tipo dinamico del biglietto per sapere quale elemento salvare nell'albero XML;
- Metodi **Modificawindow**::*slVisualizza,* e **Rimuoviwindow**::*slModificaUltimo*nel quale bisogna identificare il tipo dinamico in quanto da esso dipende quali widget devono essere popolati e quali no.

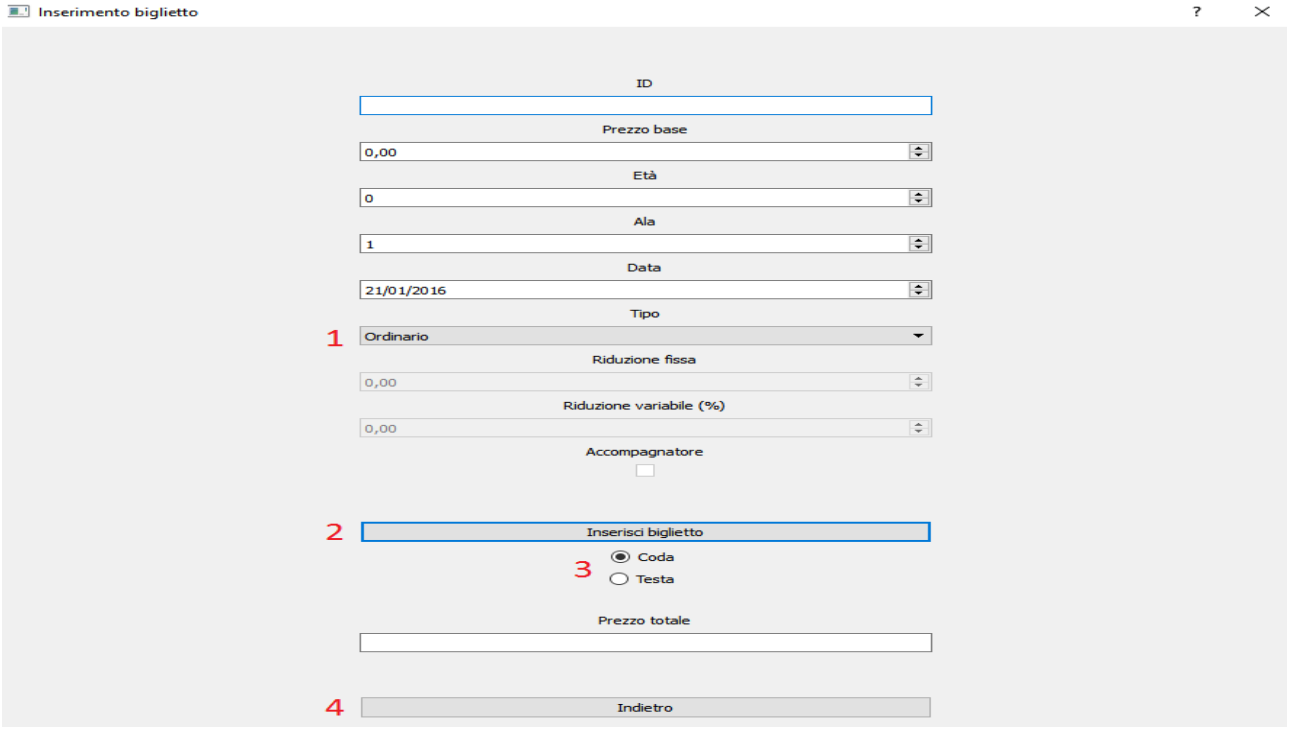
Manuale utente

Schermata principale



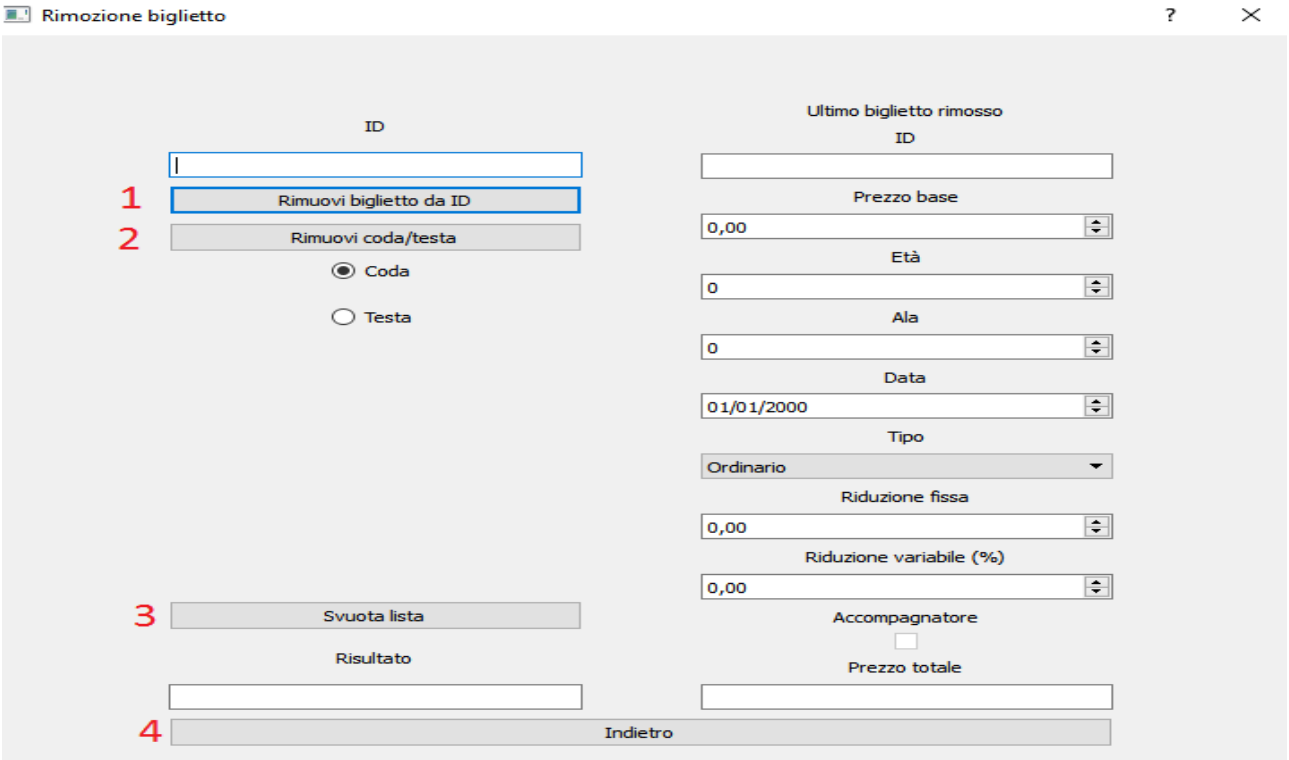
Dopo aver aperto il programma, ci si trova davanti alla schermata principale.
Se non è la prima volta che si apre il programma e si intende continuare il lavoro precedentemente salvato, fare click su (0)File->Apri: in questo modo sarà possibile caricare da file la lista di biglietti e riprendere dal punto in cui ci si era interrotti precedentemente.
Se è la prima volta che si apre il programma, procedere all'inserimento di nuovi biglietti.
Dalla schermata principale è possibile accedere a varie schermate: (1) Inserisci biglietti, (2) Rimuovi biglietti, (3) Modifica biglietti, (4) Visualizza lista, (5) Modifica soglia età. Inoltre, è possibile calcolare il guadagno generato da una certa fascia d'età di visitatori, inserendo (6a) l'età di inizio e 6(b) l'età di fine, e poi premendo il pulsante (6c) Calcola guadagno fascia d'età. Infine, è possibile chiudere il programma facendo click su (7) Esci.

Inserimento di un nuovo biglietto



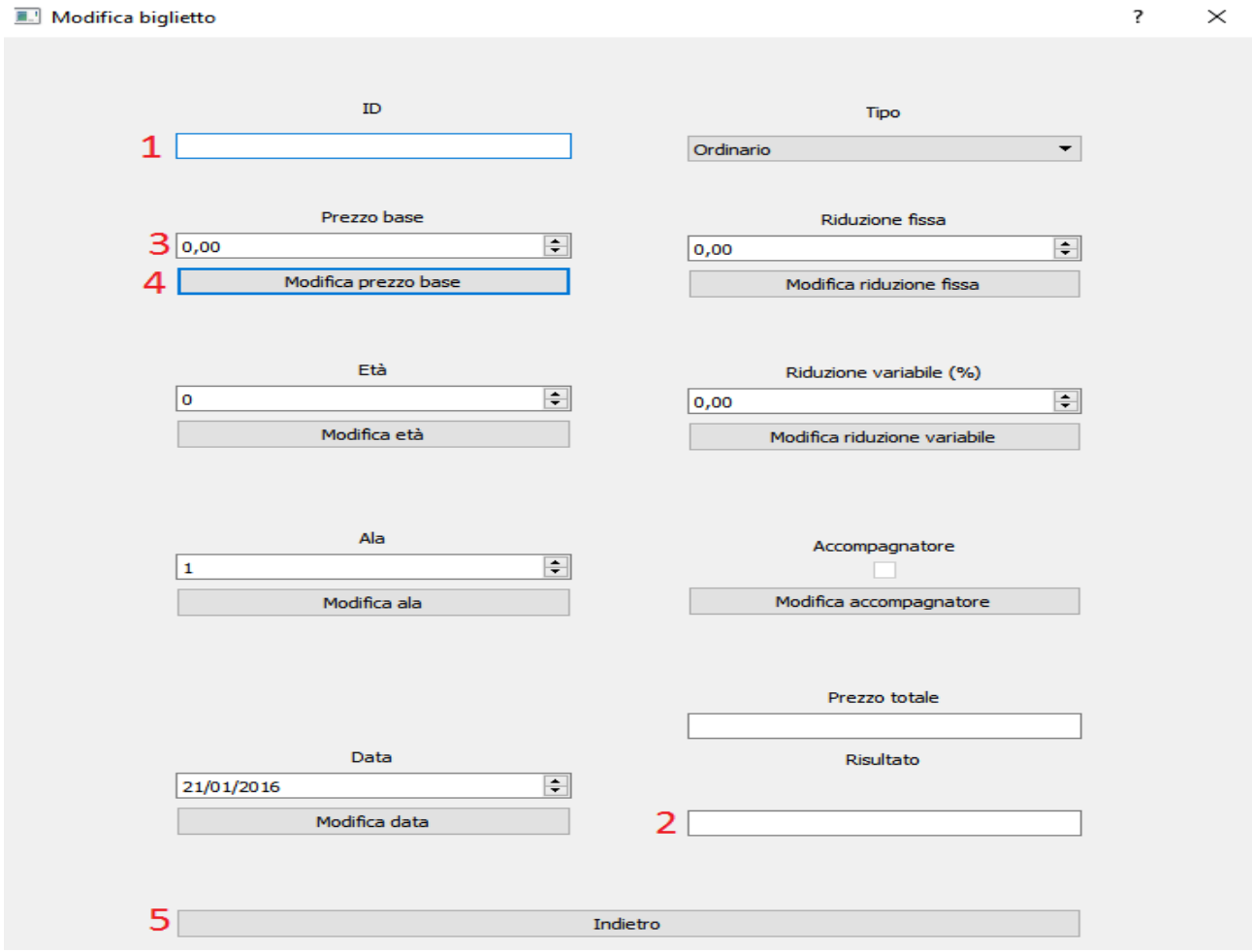
Dopo aver riempito i primi campi dati necessari alla creazione di un nuovo biglietto, scegliere il (1) Tipo: nuovi campi dati verranno resi disponibili a seconda del tipo scelto. Una volta compilati tutti i campi, scegliere (3) se inserire il biglietto in coda o in testa alla lista e infine premere il tasto (2) per l'inserimento del biglietto. E' possibile ritornare alla schermata principale con il pulsante (4) Indietro.

Rimozione di un biglietto/ cancellazione lista



Da questa schermata è possibile (1) rimuovere uno specifico biglietto dato un certo ID (se ne è presente più di uno con lo stesso ID, verrà rimosso il primo). E' anche possibile (2) rimuovere il primo/ultimo biglietto dalla lista e (3) svuotare l'intera lista (verrà chiesta la conferma prima di procedere). Sul lato destro della schermata è possibile vedere le informazioni sull'ultimo biglietto rimosso con successo. Col il pulsante (4) Indietro è possibile tornare alla schermata principale.

Modifica di un biglietto



Da questa schermata è possibile modificare i dati di un biglietto dato un certo ID (come sempre, se in lista ce ne sono più di uno con lo stesso ID, verrà modificato il primo). Per procedere alla modifica, inserire nel campo (1) ID il codice del biglietto e premere tab/fare click in un qualsiasi altro campo della schermata, il risultato della ricerca verrà visualizzato in (2) Risultato. I vari campi dati verranno attivati/disattivati a seconda del tipo di biglietto da modificare. Per procedere alla modifica, inserire il nuovo valore, ad esempio dentro al campo (3) Prezzo base, e poi premere il relativo pulsante, ad esempio (4) Modifica prezzo base.

Modifica soglia età

Per modificare la soglia d'età per il calcolo del prezzo dei biglietti ridotti per bambini, fare semplicemente click sul relativo pulsante nella schermata principale, inserire la nuova età e poi premere il pulsante ok.