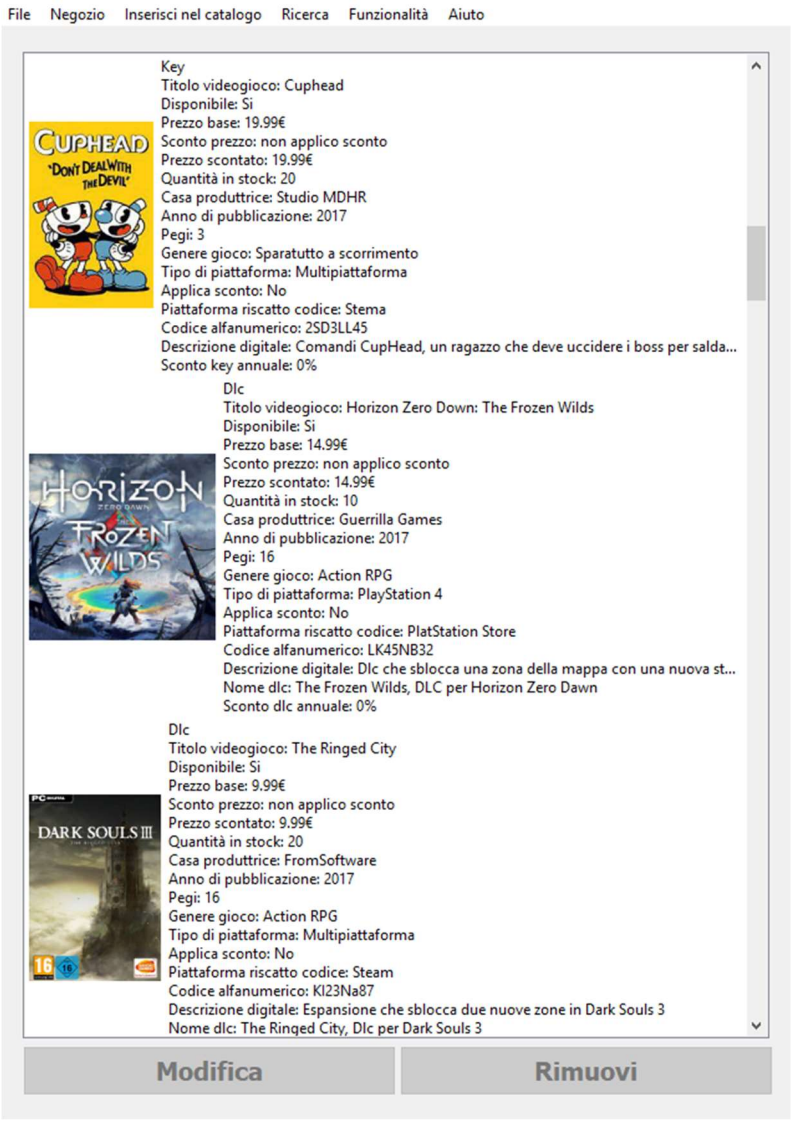
Nome: Daniele
Cognome: Larosa
Matricola: 1097821
A.A: 2018-2019

Progetto di programmazione ad oggetti “Videogame”

Sommario



Introduzione

Il progetto consiste nel creare un programma che permetta di gestire un negozio di videogiochi, in cui sarà presente un catalogo di prodotti visualizzabili sotto forma di elenco con immagine e informazioni relative ad ognuno.

Modalità di consegna

Il progetto è stato consegnato con il comando “consegna progetto-pao-2019” come rischiesto nelle specifiche. Sono presenti tutti i file del progetto compreso il file .pro generato automaticamente da Qt all’avvio del progetto. Questo perchè la compilazione richiede un file .pro per qmake differente da quello ottenibile traminte l’invocazione di “qmake-project” da terminale, che permetta la generazione automatica tramite qmake del Makefile.

Descrizione del Qontainer

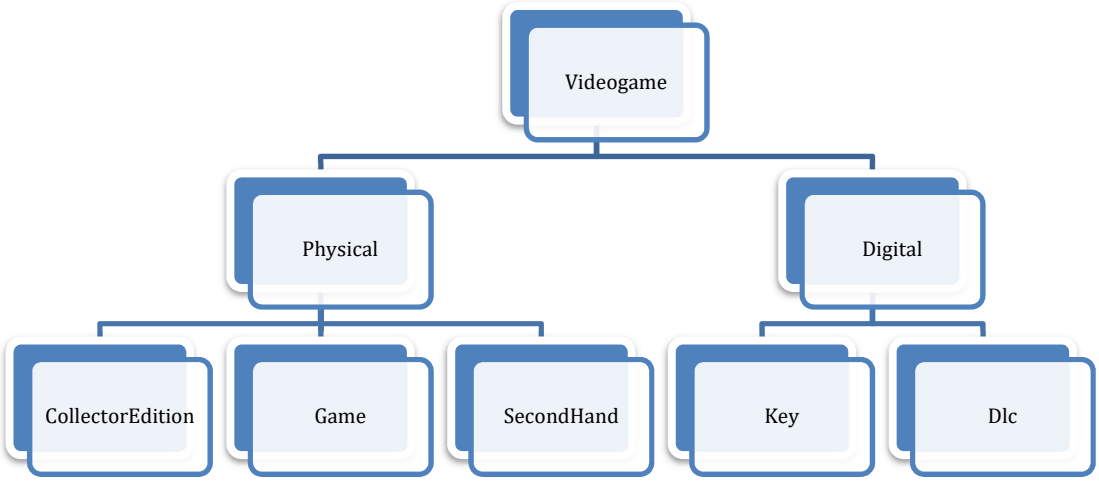
Nel progetto è presente il “Qontainer” con funzionalità richieste quali:

- inserimento all’inizio e alla fine;
- ricerca;
- rimozione;

Il Qontainer è una lista di SmartPointer che puntano a Nodi, si è scelto l’uso degli SmartPointer perché servono ad ottimizzare al meglio l’uso della memoria e prevengono la cancellazione involontaria di elementi che sono ancora in uso ad esempio da diverse parti del programma.

È stato scelto l'utilizzo di una doppia lista linkata poiché, pensando che si trattasse di un gestore, la probabilità che vengano cancellati oggetti presenti in mezzo alla lista (ad esempio quando si finiscono le scorte di un determinato prodotto) è alta, quindi risulta più comoda una lista dato l'uso frequente di rimozioni.
Nel Qontainer è stato aggiunto un metodo statico chiamato "**AnnoCorrente**" che serve a prendere l'anno corrente nella macchina in cui viene runnato, l'utilità verrà spiegato nella sezione successiva "**Polimorfismo ed estendibilità del codice".**

Descrizione delle classi e della gerarchia



La gerarchia del progetto consta di una classe base astratta "**Videogame**" e due classi derivate astratte "**Physical**" e "**Digital**" che servono a distinguere i videogiochi presenti in copia fisica nel negozio da quelli digitali non presenti fisicamente in negozio. La divisione è pensata dal fatto che esistono appunto due macrocategorie di videogiochi acquistabili, da cui a loro volta derivano i prodotti effettivi che saranno acquistabili dall'utente finale.
Dalla classe "**Digital**" derivano due classi concrete:

- "**Key**" che sono delle chiavi con codice alfanuemrico riscattabile presso numerose piattaforme online,dopo l'acquisto si inserisce il codice e se risulta valido è possibile scaricare all'interno della propria console il videogame;
- "**Dlc**" che sono digitali e sono i così detti contenuti aggiuntivi, anch'essi scaricabili presso piattaforme online.

A sua volta da "**Physical**" derivano tre classi concrete:

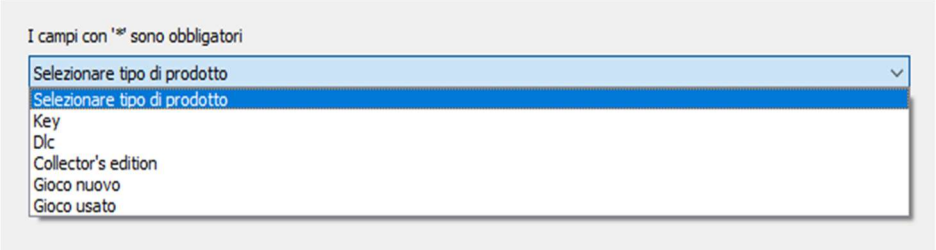
- "**CollectorEdition**" che racchiude tutte le collector's edition, ossia le edizioni a tiratura limitata, contenenti oggetti come action figure dei personaggi, cd con colonne sonore del videogioco, sketchbook con i disegni preparatori dei designer ecc. Sono varie e spesso ogni casa produttrice può decidere a sua scelta quante produrne e cosa mettere dentro;
- "**Game**" che è il gioco che si compra, e che si comprava prima della diffusione in larga scala di internet, in negozio dove è presente la scatola del videogame e all'interno il CD. Possono essere presenti delle edizioni limitate, anche queste in poche copie, dove solitamente cambia la scatola ed è presente qualche contenuto aggiuntivo scaricabile con all'interno della scatola un codice;
- "**SecondHand**" che sono i giochi usati che spesso gli utenti portano in negozio per farsi valutare e ottenere uno sconto sui giochi nuovi e/o altro presente all'interno del catalogo del negozio. L'usato è stato messo in una classe a parte perchè solitamente gli unici giochi che vengono valutati dai negozi (e che hanno una facile rivendibilità) sono i "**Game**", ossia videogame con semplice disco ed eventualmente scatola integra. È difficile che vengano ri-vendute collector's edition in quanto vi è una difficoltà nel valutare le condizioni/usura di tutti i prodotti presenti al loro interno. Né è possibile ri-vendere "**Key**" o "**Dlc**" acquistati in quanto il codice è univoco e non può essere ri-utilizzato.

La gerarchia è stata pensata in maniera aggiornabile, infatti sia da "**Physical**" che da "**Digital**" è possibile aggiungere nuove categorie di giochi fisici e digitali senza andare ad intaccare le classi già pre-esistenti. Inoltre in ogni classe è stato inserito un campo comune a tutte, ossia il campo "sconto". Questo campo non è stato inserito nella base poiché,

dato che per ogni classe concreta della gerarchia è fissato uno sconto annuale massimo statico, personalmente ho preferito avere il campo dati direttamente sulla classe concreta in modo che se un domani la gerarchia dovesse essere aggiornata non sarà necessario toccare la base dovendo aggiungere nuovi campi dati.

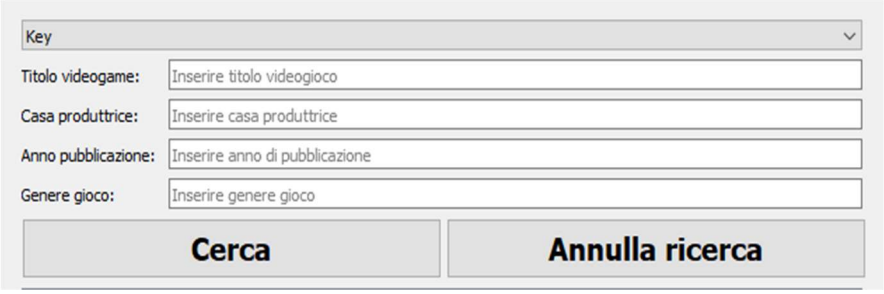
Descrizione della GUI

La GUI è la parte grafica, anch'essa pensata per essere aggiornata senza andare per forza ad intaccare parti già esistenti. Infatti per ogni classe (sia astratta che concreta) è stata creato un QWidget con le QLineEdit in modo che se un domani si dovesse creare una nuova classe si potranno aggiungere le due classi astratte e la parte nuova senza dover riscrivere tutte le QLineEdit.
E' presente l'interfaccia "**Inserisci**" dove è possibile inserire gli oggetti all'interno del nostro catalogo, si seleziona l'oggetto che bisogna inserire e dopo di che appaiono le varie QLineEdit da riempire con i parametri. Sono stati resi obbligatori i campi dati marchiati con "*" come "Titolo videogame", "Casa produttrice", "Anno di pubblicazione" e "Genere gioco" in quanto questi campi insieme alla scelta del tipo servono a rendere univoco l'oggetto creato. Non possono esistere due oggetti delle stesso tipo (ad esempio due "Key") che abbiano questi 4 campi dati identici, nel caso succeda ci penserà il programma (mentre si esegue l'inserimento) a cercare se l'oggetto è già presente nel catalogo fornendo un avviso che l'oggetto non è stato creato ma eventualmente è stata aggiornata la quantità in stock.

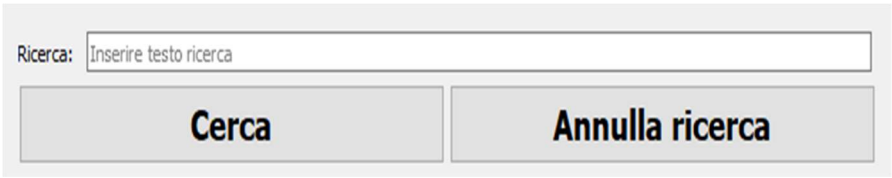


Sono inoltre presenti due ricerche:

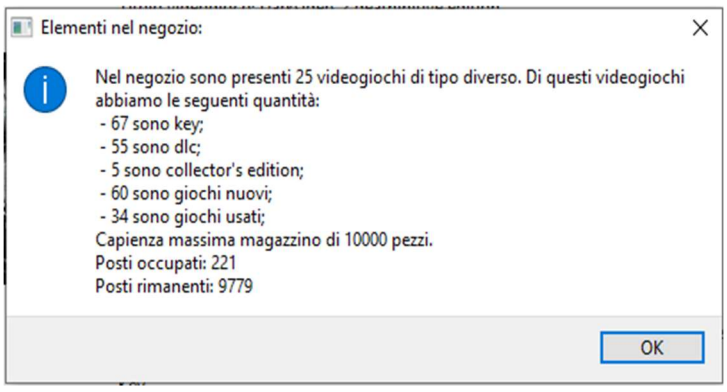
- una "**Ricerca specifica**" per oggetti che usa come campi di ricerca i campi della base astratta, in questo modo è aggiornabile a qualsiasi classe derivata concreta che sarà aggiunta in futuro. Sono stati scelti gli stessi parametri obbligatori da inserire all'interno del "**Inserisci**" in quanto appunto servono a rendere l'oggetto univoco;



- una "**Ricerca generica**" per stringa, all'interno della QLineEdit viene inserito il parametro richiesto e verrà confrontato con i vari campi dati degli oggetti presenti nel catalogo, in caso di successo verranno trovati i vari oggetti che hanno almeno un campo dati con il parametro inserito nella QLineEdit;



Un'altra funzionalità è "**Elementi in negozio**" che mi dice quanti titoli sono presenti in negozio, dopo avermi detto i titoli mi elenca quanti oggetti per tipo si trovano all'interno del mio catalogo e per finire mi vengono elencati i posti occupati e quelli rimanenti in magazzino. Funzionalità comoda per capire quanto spazio rimane in magazzino e per capire cosa ho venduto e cosa no ad una rapida occhiata senza dover aprire la lista di prodotti.
Per finire sono presenti due funzionalità "**Aiuto**" per contattare lo sviluppatore in caso di problemi.



Model Control View

Il Model Control View è uno schema di progettazzione in grado di separare la parte logica di presentazione e la parte logica applicativa, utilizzata comunemente per lo sviluppo di interfacce per l'utente. Il suo funzionamento è basato sulla separazione dei compiti in tre ruoli principali:

1. Model, che fornisce i metodi per accedere ai dati utili;
2. View, che visualizza i dati presenti nel model e si occupa dell'interazione con l'utente;
3. Controller, che riceve i comandi dall'utente e li attua modificando lo stato delle altre due componenti.

Nei file model.cpp e model.h le funzioni principali utilizzate sono quelle di Salvataggio e Caricamento che servono appunto per salvare e caricare il file .xml, in cui sono salvati i dati del gestore. Queste funzioni vengono sempre chiamate in caso di modifica, rimozione o aggiunta di un prodotto.
Nei file controller.cpp e controller.h viene gestita la maggior parte delle operazioni, ossia il flusso delle informazioni principali passa attraverso di essi. Questi due file sono i responsabili del prelievo dei dati nuovo e delle modifiche che vengono apportati all'interno del gestore.

Polimorfismo ed estendibilità del codice

Nel codice del Qontainer è presente un metodo statico "AnnoCorrente". Nel settore del videogame il ricambio annuale di giochi è molto frequente, ci sono molti titoli a cadenza annuale e quindi soggetti a forti sconti dopo un anno. In questa maniera per mezzo di un automatismo si cerca di aiutare il gestore a rendere automatico (nella maggior parte dei casi) il calcolo dello sconto per i vari prodotti inseriti. Ovviamente si è pensato ad uno sconto annuale massimo applicabile in modo da evitare possibili errori del negoziante e quindi prevenire la vendita di prodotti a prezzi troppo bassi. Sempre per evitare errori umani si è deciso di inserire dei prezzi fissi dopo tot anni (in base alla tipologia dell'oggetto) in modo tale che venisse mantenuto un prezzo fisso comunque più basso del prezzo iniziale. Non è stato inserito un controllo sull'immissione del prezzo da parte del gestore per due motivi:

- è nell'interesse del gestore inserire correttamente il prezzo la prima volta che inserisce l'oggetto e nel caso di errore vi è sempre la possibilità di modificarlo in un secondo momento;
- collegandoci al primo punto, considerando che i prezzi dell'ultimo decennio dei videogiochi dei famosi titoli AAA sono aumentati fino a stabilirsi su cifre dai 70 euro in su, ed essendo anche i giochi più venduti e quindi i principali a generare profitto, la probabilità che un gestore li acquisiti sono maggiori rispetto ad altri titoli meno famosi. Pertanto nel caso un gestore decida di acquistare un videogioco con un prezzo inferiore ad un prezzo minimo fisso stabilito basterà che dopo un tot di anni inserisca un prezzo e tolga lo sconto;

I casi in cui deciderà di non applicarlo sono come i motivi precedenti (ossia prezzo fisso dopo tot anni troppo alto) e/o decisione di non voler scontare il gioco. Lo sconto è comunque su base annuale quindi allo scoccare del nuovo anno verranno applicati gli sconti. Può essere un problema se un gioco esce il 31 dicembre 20xx starà al gestore decidere il da farsi. Sottolineo che lo sconto comunque è a piena scelta del gestore così come la scelta di applicarlo o meno selezionandolo, generando l'effetto a catena dei metodi virtuali sullo sconto.
Ovviamente è scontato che al gestore verrà fornita tutta la documentazione del caso per cercare di fargli capire le funzionalità del programma al meglio.
Essendo comunque prezzi fissi statici sarà più semplice un domani andare a modificarli essendo calcolati in automatico, nel caso i prezzi inizino ad abbassarsi e/o subire variazioni repentine.
Un'altra funzionalità presente all'interno del programma è la "capienzaMassimaMagazzino" Il numero massimo degli elementi che posso contenere in negozio sono 10000, quindi ogni volta che viene inserito un nuovo prodotto viene fatta la conta di tutti i prodotti e se il magazzino risulta pieno compariranno degli avvisi a schermo che faranno capire al proprietario del negozio che non è possibile inserire il numero totale degli oggetti.

Il polimorfismo del codice è ottenuto mediante il richiamo a funzioni e metodi virtuali:

- il distruttore virtuale;

- l'operatore di uguaglianza e disuguaglianza che sono stati ri-definiti in quanto nella ricerca non vengono controllati tutti i campi dati degli oggetti ma solo quelli di rilievo che permettono di effettuare la ricerca e quindi ottenere un risultato ottimale;
- il metodo "*prezzoVisualizzatoDopoScontoAnnuo*()" che permette di vedere a schermo nella lista il prezzo effettivo dell'oggetto dopo eventuale sconto;
- il metodo "*ottieniScontoTotale*()" dal quale si ottiene una stringa che permette di capire quanto sconto viene applicato e/o se viene messo un prezzo fisso dopo tot anni passati;
- il metodo "*getTipologiaProdotto()"* che serve ad indicare il tipo di prodotto in modo da poterlo confrontare nei vari metodi della GUI;
- il metodo "*ricercaPerStringa()*" che serve appunto per la ricerca generica che è stata pensata all'interno della GUI insieme alla ricerca per oggetto.

Approfondimento sullo sconto

É stato scelto di non mettere il campo dati "Sconto" sulla base perchè essa non può accedere ai campi delle sue sottoclassi e quindi per modificare questo campo dati in base allo sconto massimo applicabile si sarebbe dovuto usare un metodo virtuale. Per la modifica di un singolo campo dati non sembrava opportuno usare un metodo virtuale ulteriore, anche perchè al suo interno si sarebbero dovuti fare diversi controlli per poi comunque invocare il setSconto presente nella base.
Si è ritenuto più opportuno quindi ripetere il campo dati "Sconto" in ogni classe concreta, con il suo relativo massimo sconto applicabile, in modo da poter fare direttamente controlli sulla classe senza l'utilizzo di metodi virtuali e in modo da tale che se il codice venisse preso in mano da un'altra persona fosse più chiaro quello che è stato fatto.

Conteggio delle ore

- Analisi specifiche del progetto: 1 ora
- Pianificazione gerarchia e Qontainer : 3 ore
- Scrittura Qontainer e Gerarchia: 8 ore
- Apprendimento parte grafica ide Qt: 16 ore
- Scrittura parte grafica (GUI): 21 ore
- Debugging: 8 ore
- Stesura relazione: 2 ore

Totale ore sviluppo progetto: 59 ore