Relazione sul progetto di Programmazione ad Oggetti a.a. 2020/2021

QtAssicurator

Mazzucato Marco

Matricola:1193113

Introduzione

Ho deciso di sviluppare il progetto QtAssicurator che consiste in un gestionale di assicurazioni. Esso permette di scorrere le assicurazioni già presenti, eliminarle oppure inserirne una nuova. Si possono inserire assicurazioni di tre tipi: assicurazioni sulla vita, assicurazioni su immobili e assicurazioni RCA. Ho pensato che il prodotto sia destinato a un dipendente di qualche compagnia assicurativa.

Descrizione della gerarchia, metodi polimorfi e classe contenitore

La gerarchia rappresenta delle assicurazioni, la classe base è quindi Assicurazione ed è astratta. Da essa derivano pubblicamente AssVita e AssBeni, la prima istanziabile e la seconda astratta. Da AssBeni derivano poi pubblicamente AssImmobili e RCA, entrambe istanziabili.

La classe base Assicurazione contiene campi dati comuni a qualsiasi polizza: *nome*, *cognome*, *età*, *codice fiscale*, *data di sottoscrizione* e *codice della polizza*. Nella sottoclasse astratta AssBeni è presente il *costo del bene* che verrà assicurato. Le tre classi istanziabili contengono dati specifici per il tipo di assicurazione che si sta sottoscrivendo come i *metri quadrati* per AssImmobili o *cilindrata* per RCA.

Nella classe base Assicurazione sono presenti due metodi virtuali puri *CalcolaPremio()* e *CalcolaMassimale()*, mentre nella sottoclasse astratta AssBeni è presente il metodo virtuale puro *CalcolaFranchigia(),* dato che la franchigia interessa solo le polizze che assicurano dei beni materiali. Tutti questi metodi verranno implementati nelle classi istanziabili in base ai propri campi dati. Ho inoltre dotato tutte le classi del metodo virtuale *clone()*.

Spiegazione di alcuni campi dati e metodi

A parte alcuni campi dati dei quali si capisce bene il loro scopo (es. Nome, Codice Fiscale, Cilindrata) ce ne sono alcuni dei quali non si potrebbe capire bene il significato:

-*CodPolizza* in Assicurazione: ogni polizza presente è identificata da un intero univoco, questo campo è inoltre usato per l'eliminazione.

-*InizioContratto* in Assicurazione: indica la data dalla quale avrà validità la polizza, se non viene modificata durante l'inserimento viene presa la data corrente. La minima data selezionabile durante l'inserimento è di 365 giorni prima rispetto alla data corrente.

-*Fascia* in AssVita: ogni assicurato viene assegnato ad una fascia in base al reddito, dal più facoltoso (fascia A) al meno facoltoso (fascia D).

-*ImportoVersato* in AssVita: al momento della sottoscrizione della polizza il cliente effettua un versamento che stabilirà il premio da pagare e il massimale.

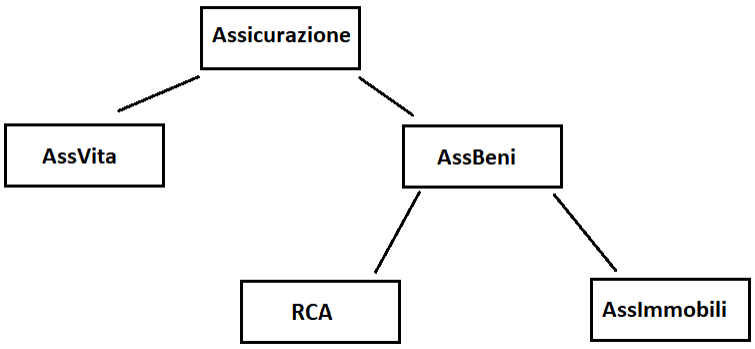
-*Edificio* in AssImmobili: se true allora si tratta di una casa/appartamento altrimenti di un terreno.

Il metodo *calcolaPremio()* ritorna il prezzo che il cliente dovrà pagare ogni sei mesi.

I vari costi potrebbero non essere molto verosimili in quanto ho fatto delle stime in base alle mie conoscenze.

Per quanto riguarda il template del contenitore ho deciso di implementare la classe Vector, mentre il puntatore smart richiesto si chiama DeepPtr.

Di seguito viene riportato l'albero della gerarchia:



Pattern MVC

Ho deciso di adottare il pattern MVC per rendere indipendenti modello e vista. Tutto il programma è gestito dal controller che si occupa di costruire modello e vista e gestire i segnali ricevuti.

Descrizione modello

Il mio modello è la classe Assicurati, la quale contiene un `Vector<DeepPtr<Assicurazione>>` contenente tutti gli assicurati che vengono inseriti nel vettore durante la costruzione. La classe presenta i metodi *addAss(Assicurazione*)* e *removeAss(int)* per aggiungere o rimuovere un'assicurazione. Dispone infine del metodo *getNewId()* che genera un nuovo codice polizza univoco da assegnare alle nuove assicurazioni inserite.

Descrizione vista

Il progetto si apre con una schermata iniziale con una scritta e due bottoni, premendo il primo si apre la finestra di aggiunta di una nuova polizza mentre premendo il secondo si possono visualizzare i dettagli delle polizze già presenti.

Nella schermata di aggiunta sono presenti tre bottoni che se premuti fanno visualizzare le opzioni (widget e item) relative a quella determinata assicurazione. Una volta inseriti tutti i campi è possibile premere il bottone "Aggiungi assicurazione" che restituirà un messaggio di avvenuto inserimento se i campi sono corretti altrimenti un messaggio di errore. I campi *neopatentato* in RCA e *edificio* in assicurazione immobili sono due QCheckBox.

Nella schermata di visualizzazione viene mostrata una polizza alla volta che, oltre ai propri dati, indica il premio da pagare, il massimale e la franchigia (chiamate polimorfe). Sono presenti dei bottoni per scorrere avanti e indietro le polizze oltre al bottone "Rimuovi assicurazione" che permette di eliminare l'assicurazione corrente.

Sia la schermata di aggiunta che quella di visualizzazione sono dotate di un bottone "Home" che fa ritornare nella schermata iniziale.

Descrizione controller

Ogni vista ha il rispettivo controller che si occupa di gestire i segnali ricevuti.

Nella vista principale il controller si occupa semplicemente di cambiare vista in base al bottone premuto.

Nella vista di aggiunta il controller si occupa di costruire una nuova assicurazione utilizzando gli input e inserirla in coda al vettore presente nel modello. Prima di fare questo, tramite il metodo *checkParams()*, controlla che tutti gli input siano corretti.

Nella vista di visualizzazione il controller si occupa di passare il vettore dal modello alla vista. A questa vengono passati inoltre un `<Vector<DeepPtr<Assicurazione>>::iterator` e un intero `count` che servono per visualizzare l'indice corrente (es. 1/10). Gestisce inoltre l'eliminazione che viene effettuata utilizzando il codice della polizza.

Ambiente di sviluppo

Per lo sviluppo ho usato un pc con Windows 10, Qt 5.14.2 e compilatore MinGW 7.3.0 64-bit.

Compilazione ed esecuzione

Durante la fase di testing sulla macchina virtuale ho notato che mancavano delle librerie; quindi, se necessario, eseguire il comando `sudo apt-get install qt5-default`.

Nella consegna, insieme ai file richiesti è presente anche il file ProgettoP2.pro, necessario per il comando qmake in quanto aggiunge la dicitura QT += widgets.

Analisi ore di sviluppo

- Progettazione, ricerca e studio: ~ 9 ore
- Sviluppo modello: ~ 10 ore
- Sviluppo GUI: ~ 18 ore
- Sviluppo controller: ~ 9 ore
- Debugging: ~ 3 ore
- Relazione: ~ 2 ore
- Totale: ~ 51 ore