1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

CORSO DI LAUREA IN
INFORMATICA APPLICATA
SCUOLA DI
SCIENZE TECNOLOGIE E FILOSOFIA DELL'INFORMAZIONE

Programmazione ad Oggetti ed Ingegneria del Software

Sessione autunnale 2020/2021

Matteo Pulcinelli
Matricola: 293473

Made with LaTeX

Indice

1 Specifica del problema

Si richiede di realizzare un software con interfaccia grafica per sistemi Windows-based che consenta la gestione dei clienti. Il software in questione terrà traccia dei clienti e delle varie commissioni associate ad essi. Dei clienti ci interessano nome, cognome, il numero di telefono e email, mentre per ogni commissione ci interessano la data di scadenza e la descrizione.

Il software sarà composto da un menù principale con 4 tab

- **Home:** Questa è la schermata principale con cui si apre il software, all'interno troviamo la lista delle commissioni con scadenza in settimana e la possibilità di visualizzare, modificare e aggiungere commissioni. Sempre in questa schermata si ha la possibilità di salvare le modifiche apportate.
- **Rubrica:** In questa schermata troviamo tutti i contatti salvati con la possibilità di aggiungerne nuovi, modificare quelli già presenti o eliminarli (eliminando tutte le relative commissioni).
- **Commissioni:** Qui possiamo visualizzare tutte le commissioni (completate e non). Anche qui sono presenti i tasti visualizza, modifica e elimina.
- **Scadenze:** In quest'ultima tab verranno mostrate solo le commissioni non ancora completate, con la possibilità di visualizzarle.

Il software inoltre offre la possibilità di salvare i dati in locale in formato Json per un eventuale salvataggio su database.

Ogni elemento, cliente o commissione che sia, quindi, può essere visualizzato aggiunto, modificato o eliminato. Quando si richiede la modifica o la visualizzazione, verrà mostrata una schermata dedicata, mentre per quanto concerne l'eliminazione sarà richiesta una conferma prima di eliminare definitivamente l'elemento.

2 Studio del problema

2.1 Gestione dei dati

Uno dei punti critici più importanti da risolvere è sicuramente la gestione dei dati. Inizialmente si è pensato di gestire i dati attraverso una dipendenza circolare, ovvero nella classe `Cliente` era presente una lista di elementi `Commissione` mentre all'interno di `Commissione` un solo `Cliente`. Il problema è che questo tipo di approccio, oltre a rendere il codice molto più complicato, va a generare un elevato accoppiamento: entrambe le classi devono essere ricompilate ogni volta che una di esse viene cambiata. Per questo motivo si è optato per un sistema basato su eventi, ottenendo un codice debolmente accoppiato, con oggetti che rispettano il principio di singola responsabilità, separando i ruoli dei modelli (`Commissione` e `Cliente`) della memoria dei dati (`DatiLocali`). Quando avviene un cambiamento all'interno della memoria, viene inviata una notifica a tutti gli oggetti interessati. Per realizzare un tale sistema ci viene in aiuto uno dei design pattern comportamentali studiati: l'observer.

2.1.1 Classe per la gestione dei dati

La memoria dei dati quindi viene separata e incapsulata in una nuova classe `DatiLocali`, questa classe dovrà presentare una sola istanza per garantire consistenza nei dati. A primo impatto ci verrebbe naturale pensare che la classe debba essere statica, tuttavia sotto un'analisi più approfondita ci si rende subito conto che una soluzione del genere non permetterebbe l'implementazione del pattern observer, dato che c'è necessità di implementare un'interfaccia, cosa non possibile con classi statiche. Ci viene quindi in aiuto il pattern singleton, che consente di implementare interfacce vietando la creazione di più istanze.

2.2 Rappresentazione dei dati

Un altro problema da ovviare è il come verranno rappresentati i nostri dati. Si è scelto di sfruttare la classe `ListView` presente all'interno del framework Windows Presentation Foundation (WPF) dedicato all'interfaccia utente per lo sviluppo di applicazioni client desktop.
ListView è un componente specializzato alla rappresentazione di oggetti. Ogni oggetto non può essere inserito così com'è, ma, ogni volta che si aggiunge all'interno della ListView, deve essere prima convertito in un elemen-

to ListViewItem. Per tanto, per la creazione di questi, ci viene incontro il pattern **Composite** che rende possibile la composizione di oggetti che ci interessa rappresentare.

3 Analisi e progettazione

3.1 Architettura

Per lo sviluppo del software, sono state create le classi `Commissione` e `Cliente` che sono dei semplici modelli e, come già detto, al fine di avere un basso accoppiamento (loose coupling), si è introdotta una nuova classe che va a separare la parte dello storage dei dati. Per farlo, si è optato per l'uso di due dizionari:

- il primo composto da un numero intero come chiave e l'oggetto cliente come valore associato.
- l'altro definito da un intero che, oltre a funzionare come chiave, rappresenta anche il cliente attraverso il primo dizionario e come valore una lista di commissioni.

Per l'architettura sono stati usati 3 pattern:

- **Singleton**, pattern creazionale che garantisce la creazione di una e una sola istanza della classe. Il funzionamento è piuttosto semplice: viene blindato il costruttore con il modificatore di accesso **private**, rendendolo richiamabile una sola volta attraverso un altro metodo esposto. Viene usato durante la fase di salvataggio (in modo da richiamare sempre la stessa istanza) e per la gestione dei dati interni al programma. Questo pattern ci serve per poter implementare un'interfaccia o per estendere un'altra classe, avendo i vantaggi di una classe statica.
  Durante la gestione dei dati interni, implementiamo il singleton per poter poi aggiungere un altro pattern che è l'Observer.
- **Composite**, pattern strutturale che conferisce al programma flessibilità ed estensibilità. Consente di creare nuovi oggetti annidati basati su altri già esistenti. Nel nostro caso, trova la sua utilità nella fase di rappresentazione dei dati per comporre le entry all'interno delle listview. Per poter applicare il pattern abbiamo bisogno di:
  - **Component (interfaccia)**: definisce l'interfaccia degli oggetti della composizione.
  - **Leaf**: elemento atomico che implementa l'interfaccia component al fine di poterla usare per la composizione.
  - **Composite**: classe che rappresenta il gruppo di oggetti atomici leaf. Deve anch'essa implementare l'interfaccia component.

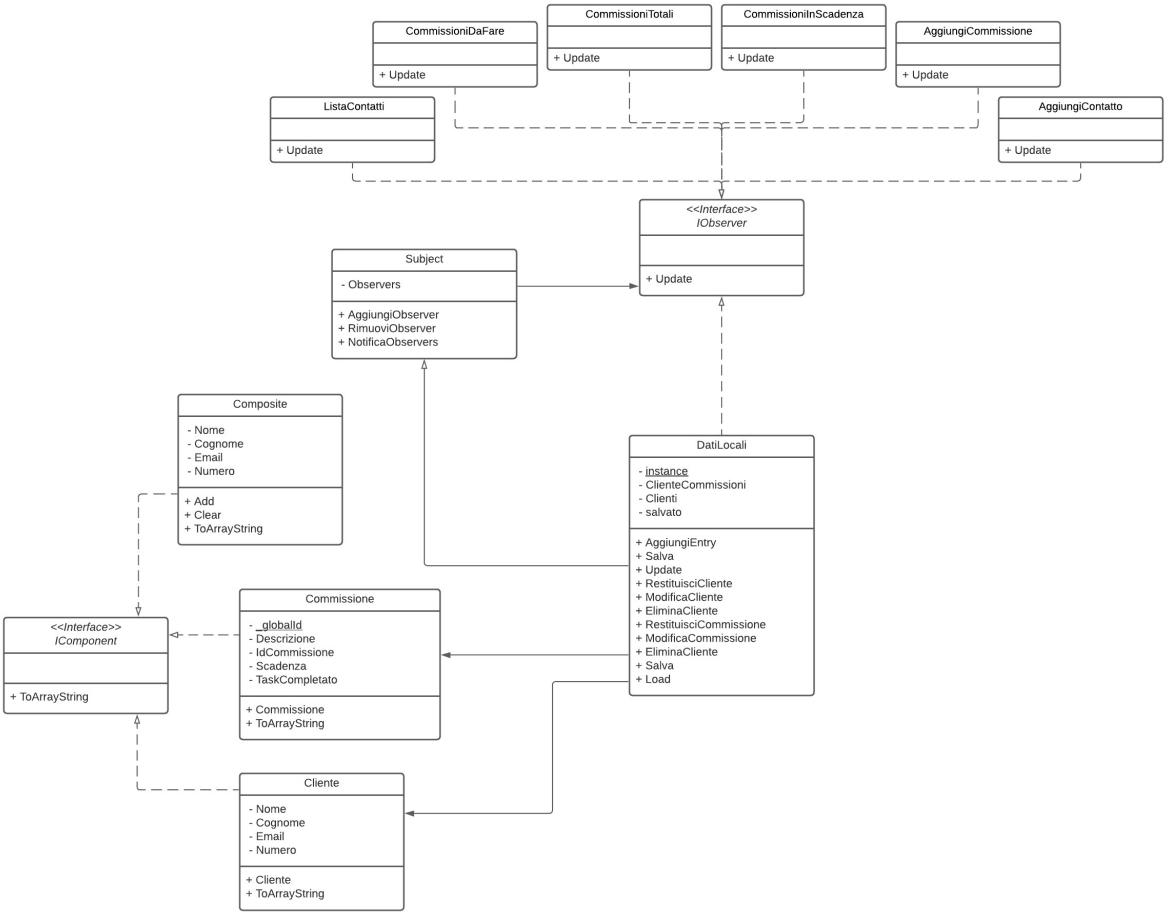
- **Observer**, pattern comportamentale che consente il sistema basato su eventi.
  Viene usato per notificare a tutte le classi dedite alla rappresentazione che son stati apportati cambiamenti in locale e quindi devono essere aggiornate con dati recenti. Oltre all'aspetto grafico, con questo pattern è stato possibile applicare un controllo in fase di uscita: usufruiamo della notifica per indicare che c'è stato un cambiamento e che quindi è presente la necessità di salvare.

  Per poter applicare il pattern abbiamo bisogno di quattro elementi:

  - **Subject**: oggetto che viene osservato ed è a conoscenza dei propri observer. Mette a disposizione operazioni di aggiunta e cancellazione di observer.
  - **observer (interfaccia)**: serve per la notifica di eventi agli oggetti interessati ad una classe **Subject**. Deve essere implementata da tutti gli observer.
  - **ConcreteSubject**: classe figlia di Subject, invoca le operazioni di notifica ereditate quando gli observer devono essere notificati. Nel nostro software si tratta della classe `DatiLocali`
  - **ConcreteObserver**: implementa l'interfaccia dell'observer definendo per ogni soggetto il comportamento.

  L'approccio usato è di tipo pull, ovvero la classe che notifica non invia dati ma il solo segnale, saranno poi le classi notificate ad occuparsi di recuperare i dati attraverso metodi esposti da parte della classe dedicata alla memoria.
  Il tipo di approccio pull permette una maggiore flessibilità, ogni observer decide cosa richiedere.
  In genere si usa questo approccio quando ci sono diversi tipi di observer.
  In virtù di questo pattern si è riusciti a separare dati e modelli.

Di seguito è riportato il diagramma **UML** rappresentante i pattern Composite e Observer.


CommissioniDaFare
+ Update
CommissioniTotali
+ Update
CommissioniInScadenza
+ Update
AggiungiCommissione
+ Update
ListaContatti
+ Update
AggiungiContatto
+ Update
<<Interface>>
IObserver
+ Update
Subject
- Observers
+ AggiungiObserver
+ RimuoviObserver
+ NotificaObservers
Composite
- Nome
- Cognome
- Email
- Numero
+ Add
+ Clear
+ ToArrayString
DatiLocali
- instance
- ClienteCommissioni
- Clienti
- salvato
+ AggiungiEntry
+ Salva
+ Update
+ RestituisciCliente
+ ModificaCliente
+ EliminaCliente
+ RestituisciCommissione
+ ModificaCommissione
+ EliminaCliente
+ Salva
+ Load
<<Interface>>
IComponent
+ ToArrayString
Commissione
- _globalId
- Descrizione
- IdCommissione
- Scadenza
- TaskCompletato
+ Commissione
+ ToArrayString
Cliente
- Nome
- Cognome
- Email
- Numero
+ Cliente
+ ToArrayString

Nell'architettura del progetto è stato fatto uso di quelle tecniche che contraddistinguono un linguaggio ad oggetti come `C#`: Incapsulamento, ereditarietà, polimorfismo, eccezioni.
Sono state inoltre aggiunte alcune classi statiche, ad esempio quelle per il controllo degli input, al fine di non ripetere codice inutilmente.

3.1.1 Cliente

La classe `Cliente` è composta da diverse proprietà (membri che forniscono un meccanismo flessibile per leggere, scrivere o calcolare il valore di un campo privato) che definiscono il cliente. Troviamo Nome, Cognome, Email e Numero.
La classe implementa l'interfaccia IComponent e quindi realizza il metodo `ToArrayString()` così da poter applicare il pattern composite.

3.1.2 Commissione

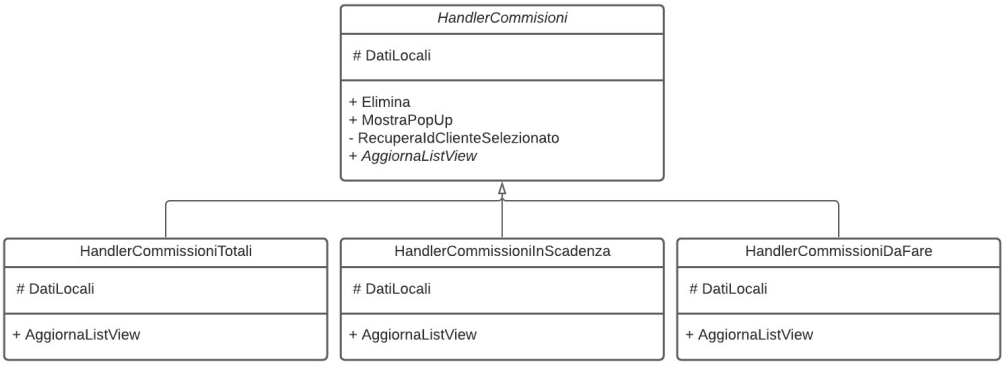
`Commissione` è descritta da IdCommissione, Descrizione, TaskCompletato e da una Scadenza. È presente anche un attributo statico che permette identificare in modo univoco ciascuna commissione. Usufruendo dell'overload poi, sono presenti due costruttori: uno utile nell'inserimento di una nuova commissione e l'altro necessario quando carichiamo le commissioni da file, in questo modo abbiamo più controllo sulla variabile globale statica. Come per la classe `Cliente`, anche qui viene implementata l'interfaccia IComponent.

3.1.3 DatiLocali

La gestione dei dati in locale è delegata alla classe `DatiLocali`, dove troviamo applicati i due pattern, **Singleton** e **Observer**. Quest'ultimo è applicato implementando l'interfaccia `IObserver` e estendendo la classe `Subject`.
Per il progetto era necessario che si creasse al più un'istanza della classe, così da non generarne altre con dati inconsistenti, e che questa potesse comunque estendere una classe o implementare un'interfaccia. In questo caso, come sottolineato, ci viene incontro il Singleton, che appunto rende possibili entrambe le richieste blindando il costruttore, rendendolo richiamabile una sola volta attraverso il metodo `GetInstance()`. Costruttore che al suo interno contiene l'iscrizione all'evento, così che, quando si verificano modifiche, l'attributo `Salvato` assume il valore false.
Questa classe è dotata anche del meccanismo di overload. Infatti, il metodo `AggiungiEntry()` viene sovraccaricato, accettando sia il solo parametro **Cliente** che i parametri **Cliente** e **Commissione** insieme.

3.1.4 Handler

Al fine ultimo di non sporcare le classi e mantenere ordine all'interno del codice, le logiche vengono delegate ad altre classi handler. Si è poi fatto uso dell'ereditarietà tra i vari handler, poichè alcune classi differivano solo di alcuni metodi: la classe astratta `HandlerCommissioni`, che funge da classe base, contiene il metodo astratto `AggiornaListView(ListView lst)` che deve essere implementato dalle classi figlie `HandlerCommissioniTotali`, `HandlerCommissioniInScadenza` e `HandlerCommissioniDaFare`. Richiamando lo stesso metodo `AggiornaListView` su oggetti diversi stiamo usando il polimorfismo.
Di seguito viene riportato il diagramma UML.



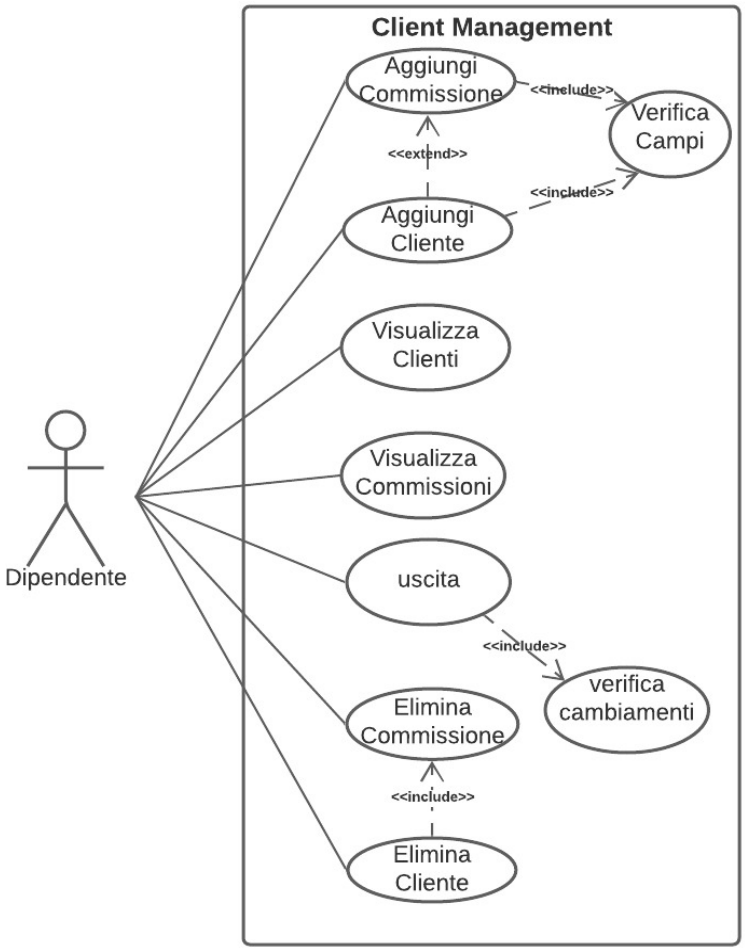
3.1.5 Eccezioni

Sono presenti poi svariati `try-catch` all'interno del codice con vari "lanci" di eccezioni, in particolare la classe statica `Controllo`, dedicata alla validazione degli input e alla generazione di eccezioni.

4 Use Cases

4.1 Diagramma use cases

Viene riportato il diagramma con le azioni principali che possono essere svolte all'interno del programma.



4.2 Descrizione use cases

Caso d'uso	Aggiungi Commissione
Id	1
attori	Dipendente
Pre-condizioni	L'utente deve aver cliccato sull'apposito pulsante di aggiunta commissione ed aver compilato in maniera corretta i vari campi presenti
Eventi base	L'utente clicca sul pulsante apposito
Post-condizioni	Viene aggiunta la commissione
Percorsi alternativi	N/A

Caso d'uso	Aggiungi Cliente
Id	2
attori	Dipendente
Pre-condizioni	L'utente deve aver cliccato sull'apposito pulsante ed aver compilato in maniera corretta i vari campi presenti
Eventi base	L'utente clicca sull'apposito pulsante
Post-condizioni	Viene aggiunto un cliente
Percorsi alternativi	Aggiungere una commissione di un cliente non presente all'interno della rubrica

Caso d'uso	Uscita
Id	3
attori	Dipendente
Pre-condizioni	N/A
Eventi base	L'utente clicca il pulsante di chiusura del software
Post-condizioni	Avviene un controllo per verificare se sono stati apportati cambiamenti poi l'utente decide se uscire o meno
Percorsi alternativi	N/A

Caso d'uso	Elimina commissione
Id	4
attori	Dipendente
Pre-condizioni	Devono essere presenti commissioni
Eventi base	L'utente seleziona la commissione e clicca il tasto *ELIMINA*
Post-condizioni	Viene eliminato un item Commissione dalla raccolta
Percorsi alternativi	L'utente seleziona un cliente e lo elimina, così facendo elimina tutte le commissioni ad esso associate

Caso d'uso	Eliminazione Cliente
Id	5
attori	Dipendente
Pre-condizioni	Devono essere presenti clienti in rubrica
Eventi base	L'utente seleziona il cliente e clicca il tasto *ELIMINA*
Post-condizioni	Viene eliminato il cliente e tutte le commissioni ad esso associate
Percorsi alternativi	N/A