1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

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

Corso di Programmazione e Modellazione a Oggetti
Progetto per la sessione estiva 2020-2021

Studente:
Filippo Giacchetti - Matricola: 282558

Indice:

1. Specifica del Problema:

Si tratta di un programma C# in Windows form che simula la gestione di un parcheggio.
Il pedaggio per l'utilizzo del parcheggio dipende dal tipo di veicolo (più è grande il mezzo e più il prezzo aumenta) e dal tempo in cui il veicolo rimane all'interno del parcheggio.
Arrivato all'entrata del parcheggio il cliente dovrà esporre i propri dati personali che, verificata la disponibilità di posti liberi, verranno salvati insieme ai dati del veicolo (tipo di veicolo, marca del veicolo e targa).
A questo punto il cliente potrà entrare all'interno del parcheggio e lasciare il proprio veicolo.
Il tempo è tenuto sotto controllo da un timer che si avvia nel momento in cui i dati del cliente vengono salvati.
Il programma potrà terminare la sua esecuzione in qualsiasi momento premendo il pulsante di arresto.

2. Studio del problema

- Input:
  gli input dell'applicazione sono i vari bottoni, combo box, ecc… che l'utente esegue, più i dati relativi ai clienti che usufruiscono del parcheggio.
- Output:
  l'output è formato dalle varie form e dalle messagebox mostrate a video dal programma.

Essendo un programma che gestisce un parcheggio si suppone rimanga in esecuzione per tutto il tempo in cui la struttura rimane aperta, ma durante l'utilizzo può capitare di chiuderlo erroneamente portando alla perdita di tutti i dati (riguardanti i clienti) che sono stati acquisiti. Scelta di progetto è stato quindi decidere di salvare i dati non solo in memoria ma anche su file. Quest'ultimo permette, combinandolo con una sequenza di passaggi, di ricaricare le informazioni in memoria dopo aver ri-eseguito il programma e quindi fare in modo che nulla vada perso.

Il file ".csv" è formattato nel seguente modo:
[Cognome] [Nome]; [Tipo documento] [Numero documento]; [Tipo veicolo] [Targa]; [Ora ingresso];

L'utilizzo di un file per la gestione dei dati ha portato a sua volta qualche complicazione, precisamente nel momento in cui bisogna andare ad eliminare la riga che contiene le informazioni relative ad un cliente che sta lasciando la struttura. Questo problema andrebbe affrontato aprendo contemporaneamente il file dati sia in modalità scrittura che lettura, ma è sconsigliato. Per questo motivo si è deciso di utilizzare due file di testo: il file "clienti" da aprire in modalità lettura e il file "clienti1" da aprire in modalità scrittura, in questo modo vengono lette tutte le righe del primo file e nel secondo verranno riscritte tutte le righe eccetto quella relativa al cliente che esce dal parcheggio. Alla fine delle operazioni il file non aggiornato viene eliminato, in modo che all'interno della cartella del programma risulti sempre esserci un unico file. A questo punto se un altro cliente dovesse abbandonare la struttura verranno eseguite le stesse operazioni citate precedentemente ma al contrario, ovvero verrà letto il file "clienti1" e verrà scritto sul file "clienti" (infine eliminato "clienti1"). Tutte le operazioni, all'interno del progetto, che lavorano sul file dati sono adattate in modo che possano usare indistintamente uno dei due file (in base a quello che c'è all'interno della cartella).

Sono stati utilizzati inoltre alcuni costrutti del linguaggio, quali:

- Gestione delle eccezioni;
- Gestione di strutture dati di alto livello offerte dalla libreria standard
  (sono state utilizzate delle List in cui sono stati memorizzati degli oggetti).

Altra scelta di progetto è stata quella di utilizzare eccezioni personalizzate, in particolare:

- ClientNotFoundException:
  l'eccezione verrà lanciata nel momento in cui il numero documento, relativo al cliente che vuole lasciare la struttura, non sarà trovato all'interno della memoria/file dati.
- TextBoxNotFilledException:
  l'eccezione verrà lanciata quando, all'atto dell'inserimento delle informazioni relative al cliente, almeno uno dei campi viene lasciato vuoto.

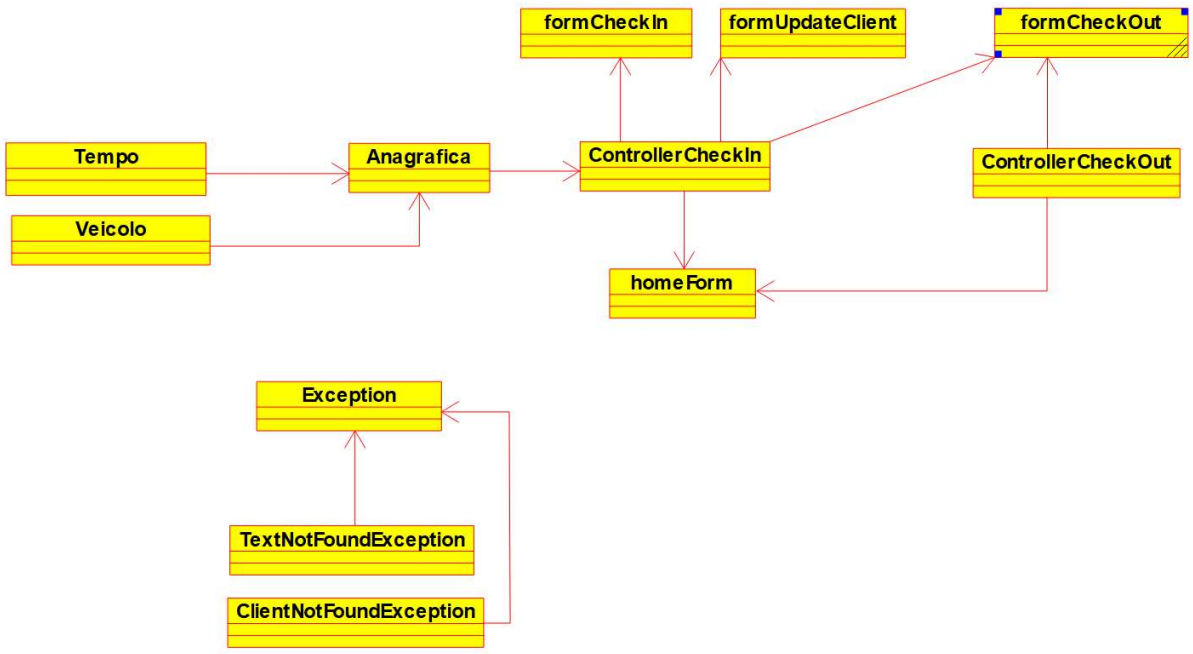
3. Scelte architetturali

Per la realizzazione del progetto è stato sfruttato il pattern di sviluppo MVC (Model-View Controller). Il pattern consente di dividere i compiti tra i componenti software:

- Model:
  fornisce le classi con i dati utili al programma ed i metodi per accedervi;
- View:
  visualizza i dati dei vari Model e si occupa dell'interazione tra l'utente e la struttura sottostante;
- Controller:
  riceve i comandi dall'utente tramite il View, esegue operazioni che possono interessare il Model e solitamente cambia lo stato del View. Questo schema implica la separazione fra la logica applicativa (a carico del Controller e del Model) e l'interfaccia utente a carico del View. Per l'interfaccia utente sono disponibili delle Form, messe a disposizione dal framework di Microsoft .NET.

3.1 Diagramma delle classi

Il diagramma delle classi costituisce lo strumento principale per rappresentare graficamente le classi utilizzate, ed illustrare la struttura del programma.



3.2 Modelli

3.2.1 Classe Anagrafica

Classe che rappresenta il cliente che entra all'interno del parcheggio.

3.2.2 Classe Veicolo

Classe che rappresenta il veicolo con cui il cliente entra all'interno del parcheggio.

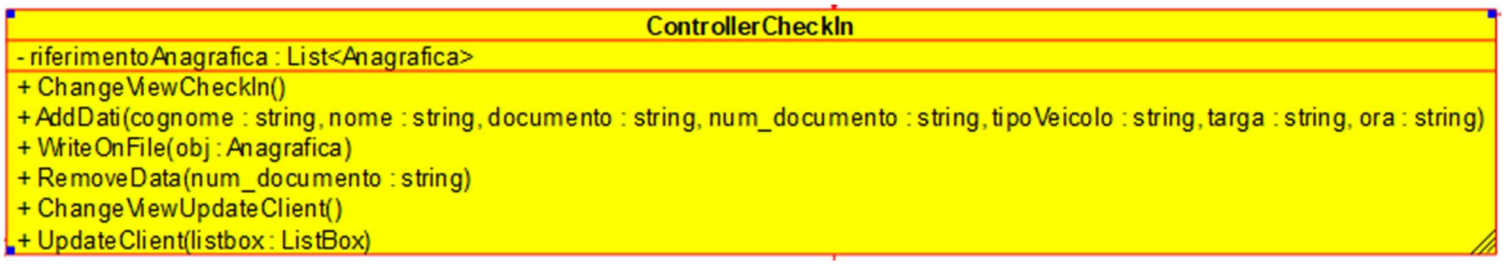
3.2.3 Classe Tempo

Classe che rappresenta l'ora di ingresso del cliente all'interno del parcheggio.

3.3 Controller

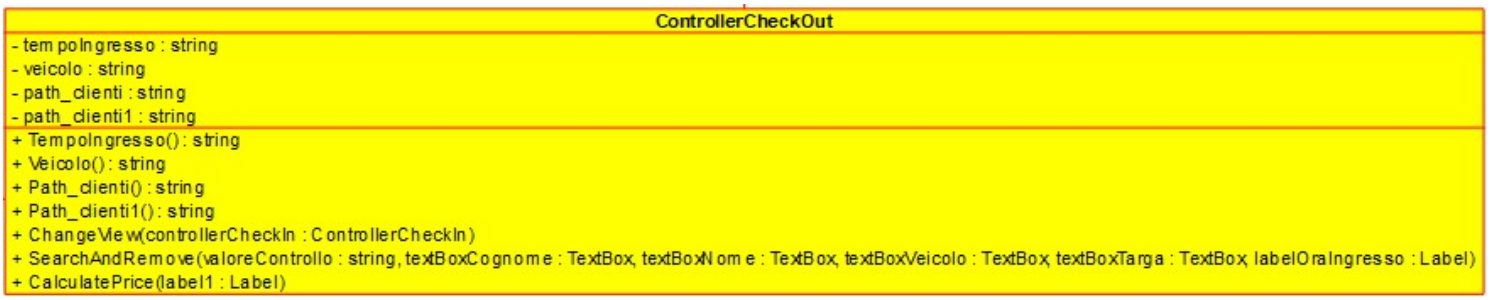
3.3.1 Classe ControllerCheckIn

È il controller che gestisce il check-in del cliente. (inserimento delle informazioni del cliente con successivo salvataggio in memoria/file).
Contiene attributi e metodi usati per interagire con i modelli.



3.3.2 Classe ControllerCheckOut

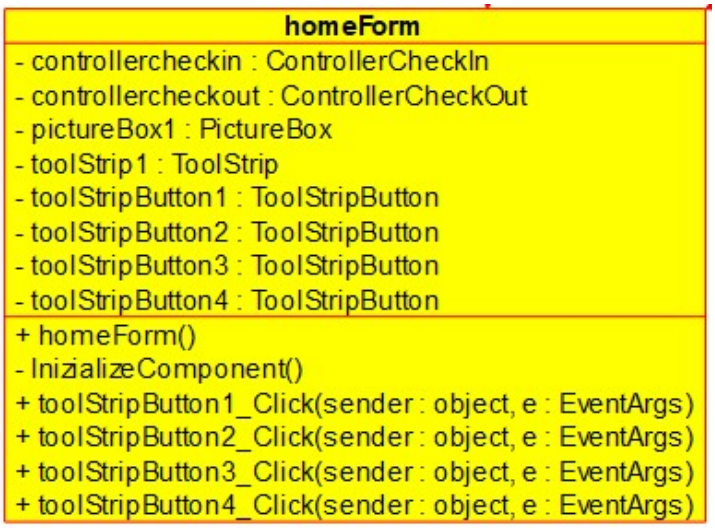
È il controller che gestisce il check-out del cliente (ricerca/eliminazione cliente da memoria/file, calcolo e comunicazione del prezzo).
Contiene attributi e metodi usati per interagire con i modelli.



3.4 View

3.4.1 Classe homeForm

Classe che estende Form, è la vista che permette di scegliere tra: check-in, check-out e caricamento dei clienti in memoria da file, in relazione all'operazione che l'utente deve svolgere.



3.4.2 Classe formCheckIn

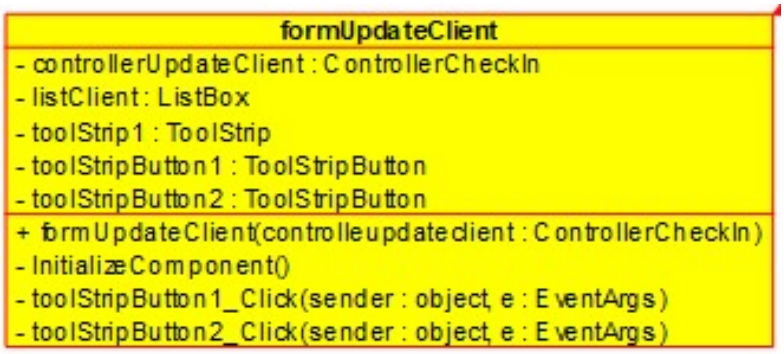
Classe che estende Form, è la vista che permette di eseguire il check-in di un cliente.

3.4.3 Classe formCheckOut

Classe che estende Form, è la vista che permette di eseguire il check-out di un cliente.

3.4.4 Classe formUpdateClient

Classe che estende Form, è la vista che permette di ricaricare in memoria le informazioni dei clienti salvate su file nel caso in cui il programma venga erroneamente chiuso



4. Use Cases

Caso d'uso: avvio dell'applicazione
ID: 1
Attori: utente/sorvegliante parcheggio
Pre-condizione: N/A
Evento base: l'utente avvia il programma utilizzando il file eseguibile
Post-condizione: viene avviato il programma
Percorsi Alternativi: N/A

Caso d'uso: check-in cliente
ID: 2
Attori: utente/sorvegliante parcheggio
Pre-condizione: aver avviato il programma e aver premuto sul pulsante "Check-in", che si trova in basso sulla sinistra della finestra principale del programma
Evento base: apertura finestra del programma che permette di eseguire il check-in di un cliente che vuole usufruire del parcheggio
Post-condizione: viene eseguito il check-in del cliente
Percorsi Alternativi: check-out, caricamento clienti in memoria, chiusura programma

Caso d'uso: check-out cliente
ID: 3
Attori: utente/sorvegliante parcheggio
Pre-condizione: aver premuto sul pulsante "Check-out", che si trova in basso sulla sinistra della finestra principale del programma
Evento base: apertura finestra del programma che permette di eseguire il check-out di un cliente che vuole uscire del parcheggio
Post-condizione: viene eseguito il check-out del cliente
Percorsi Alternativi: check-in, caricamento clienti in memoria, chiusura programma

Caso d'uso: update-client
ID: 4
Attori: utente/sorvegliante parcheggio
Pre-condizione: aver premuto sul pulsante "UpdateClient", che si trova in basso sulla sinistra della finestra principale del programma
Evento base: apertura finestra del programma che permette di ricaricare in memoria le informazioni dei clienti che si trovano all'interno del parcheggio, salvate su file, nel caso di erronea chiusura del programma
Post-condizione: vengono ricaricate in memoria le informazioni dei clienti
Percorsi Alternativi: check-in, check-out, chiusura programma

Caso d'uso: chiusura applicazione
ID: 5
Attori: utente/sorvegliante parcheggio
Pre-condizione: programma in esecuzione
Evento base: l'utente chiude la finestra del programma
Post-condizione: il programma viene chiuso
Percorsi Alternativi: N/A

5. Compilazione ed esecuzione

Gli strumenti utilizzati per la compilazione del programma sono i seguenti:

- Ambiente di sviluppo: Visual Studio Community 2019
- Versione: 16.8.2
- Framework: .NET Framework 4.7.2

Per eseguire l'applicazione fare doppio click sull'eseguibile nel percorso "Progetto_parcheggio\Parcheggio\Parcheggio\bin\Debug\Parcheggio.exe"

I requisiti minimi per l'esecuzione del programma sono i seguenti:

- Sistema operativo: Windows 10 Pro
- Architettura: 64 Bit
- Framework: .NET Framework 4.7.2

Il software è stato testato su un computer avente le seguenti caratteristiche:

- CPU: Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz 4.20 GHz
- RAM: 64GB
- GPU NVIDIA GeForce 1080Ti
- S.O: Windows 10 Pro, versione 20H2
- Architettura: 64 Bit