Università degli Studi di Urbino Carlo Bo
Corsi di Laurea in Informatica Applicata
Programmazione e Modellazione a Oggetti, a.a. 2023/2024

Relazione per il Progetto del Corso di Programmazione e Modellazione a Oggetti

Borrelli Francesco
Matricola: 313658

Petreti Alessandro
Matricola: 316984

Pula Gioele
Matricola: 313875

Gestionale di un Autonoleggio

Indice:

Capitolo 1: Analisi

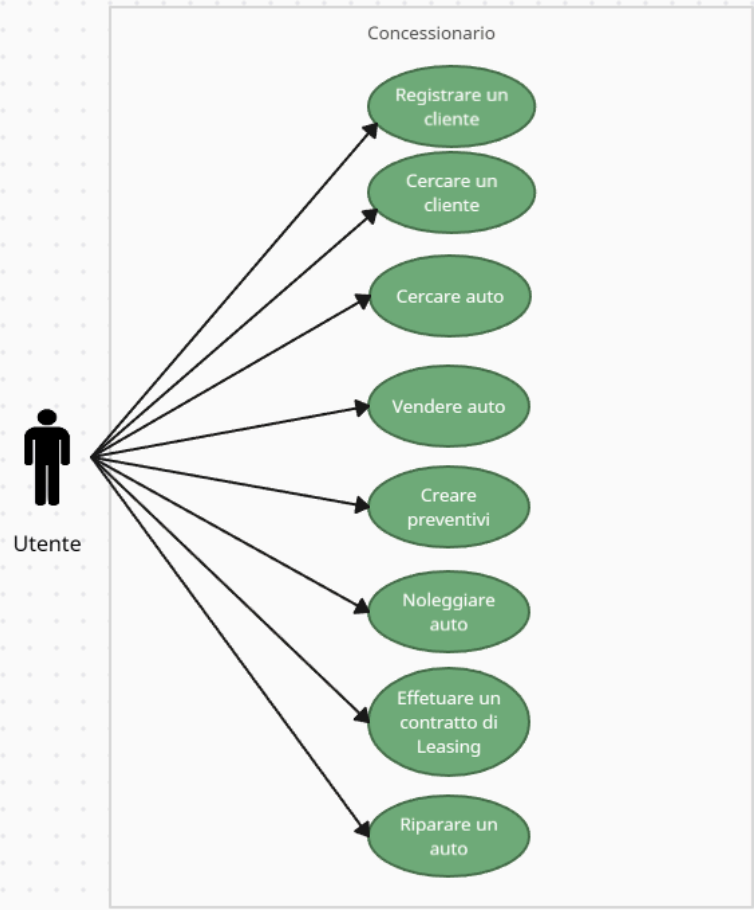
Il gruppo pone come obiettivo la gestione di un concessionario. Un concessionario è una grande organizzazione per la vendita e il noleggio di auto nuove e usate. Dispone di diversi reparti, come ad esempio l'autosalone per acquistare, noleggiare o effettuare il leasing di automobili in base a marca e modello. L' officina per la riparazione delle auto noleggiate.

1. **Vendita di Automobili**:
    - **Auto Nuove**: Vendita di veicoli nuovi dalla casa automobilistica.
    - **Auto Usate**: Vendita di veicoli di seconda mano che sono stati riparati e preparati per la rivendita.
2. **Servizi Finanziari**:
    - **Leasing**: Opzione di leasing a lungo termine di veicoli.
    - **Noleggio:** Opzione di noleggio a breve, medio e lungo termine di veicoli.
3. **Servizi di Manutenzione e Riparazione**:
    - Officina attrezzata per la manutenzione e la riparazione dei veicoli.
4. **Gestione del anagrafica dei Clienti:**
    - gestione dei dati di clienti registrati

1.1 Requisiti

Requisiti funzionali:

- L'applicazione fornirà la possibilità di registrare un cliente;
- L'utente avrà la possibilità di cercare tra i clienti registrati;
- L'utente deve avere la possibilità di filtrare le auto in base a specifici requisiti.
- L'applicazione fornirà la possibilità di vendere auto(nuove, usate);
- L'utente avrà la possibilità di creare un preventivo selezionando l'auto e il cliente interessato;
- L'applicazione fornirà la possibilità di scegliere il tipo di noleggio;
- L'utente potrà selezionare la durata noleggio e inserire una copertura assicurativa;
- L'applicazione fornirà la possibilità di effettuare un contratto di leasing per un'auto;
- Selezionare auto usate da far revisionare in officina;



Requisiti non funzionali:

- L'utente può vedere le auto vendute
- Può essere inserita una copertura assicurativa su 3 livelli (bassa. media o alta) per le auto che vengono noleggiate;
- Scegliendo di effettuare un contratto di leasing per un'auto, l'applicazione darà la possibilità di selezionare la durata del contratto stesso e un limite chilometrico annuale. Sulla base di questi valori viene calcolata la rata.

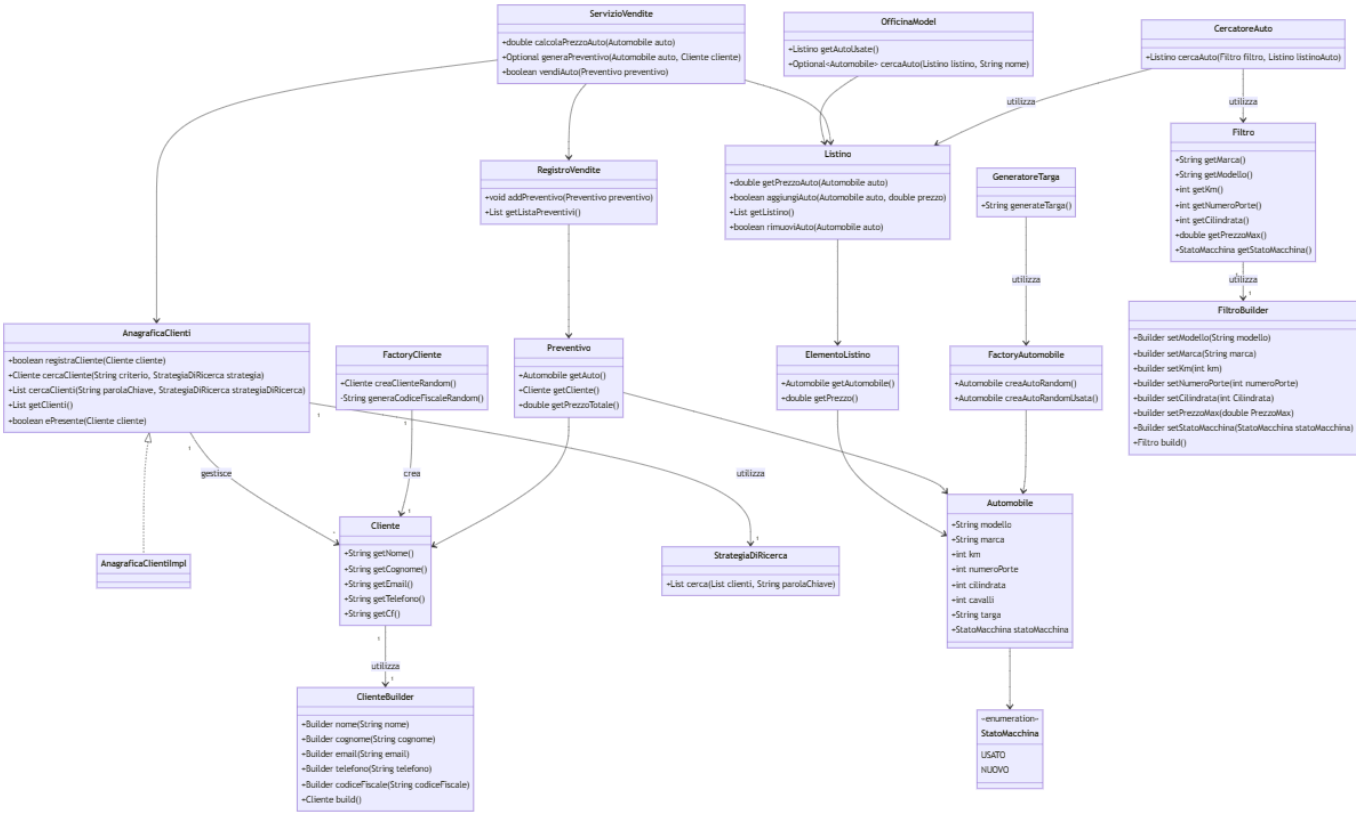
1.2 Analisi e modello del dominio

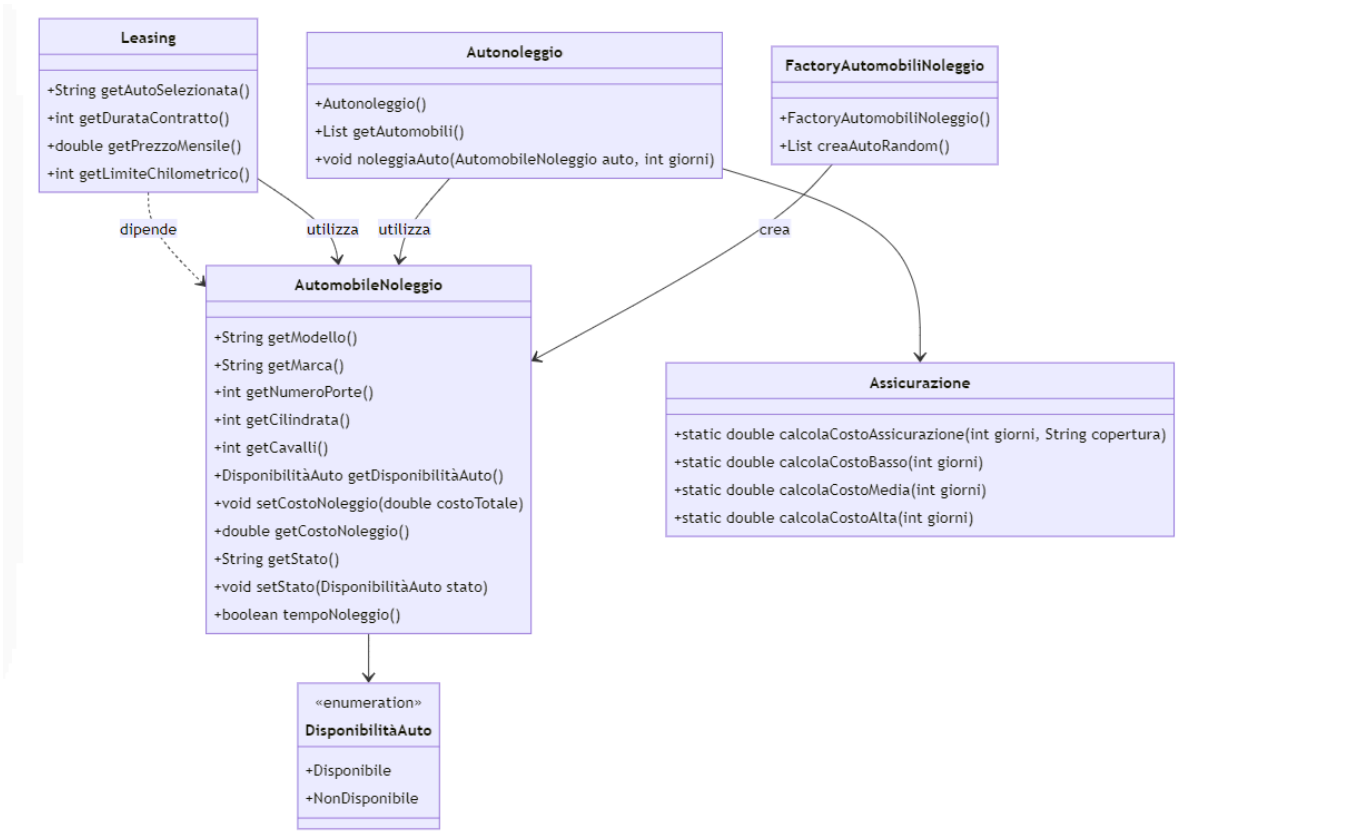
Il dominio applicativo riguarda la gestione di un concessionario di auto con all'interno un autonoleggio. Le attività principali includono la gestione dei clienti, la gestione di automobili, la gestione delle vendite, la creazione di preventivi e la gestione del noleggio.

Entità del dominio:

- Cliente, ogni cliente ha un' anagrafica che contiene nome, cognome, e-mail, numero di telefono e codice fiscale;

- Automobile, ogni automobile ha un modello, una marca, una sua cilindrata, un numero di porte, un numero di chilometri, inoltre le macchine sono caratterizzate da uno stato(usato, nuovo);
- Preventivo, che contiene informazioni sull' automobile, il prezzo e il cliente che ha effettuato l'acquisto;
- Listino, é un inventario al cui interno sono presenti le auto;
- Vendita, ogni vendita coinvolge un cliente, un'automobile e un prezzo finale;
- Automobile Noleggio, ogni automobile per il noleggio ha un modello, una marca, una sua cilindrata, un numero di porte, un numero di chilometri, inoltre le macchine sono caratterizzate da una disponibilità (disponibile, non disponibile);
- Autonoleggio, gestisce delle auto riservate al noleggio con possibilità di inserire una copertura assicurativa;
- Leasing, gestisce le operazioni di leasing, mettendo a disposizione le auto del noleggio, stipulando un contratto di una certa durata.


ServizioVendite
+double calcolaPrezzoAuto(Automobile auto)
+Optional generaPreventivo(Automobile auto, Cliente cliente)
+boolean vendiAuto(Preventivo preventivo)
OfficinaModel
+Listino getAutoUsate()
+Optional<Automobile> cercaAuto(Listino listino, String nome)
CercatoreAuto
+Listino cercaAuto(Filtro filtro, Listino listinoAuto)
utilizza
utilizza
RegistroVendite
+void addPreventivo(Preventivo preventivo)
+List getListaPreventivi()
Listino
+double getPrezzoAuto(Automobile auto)
+boolean aggiungiAuto(Automobile auto, double prezzo)
+List getListino()
+boolean rimuoviAuto(Automobile auto)
GeneratoreTarga
+String generateTarga()
Filtro
+String getMarca()
+String getModello()
+int getKm()
+int getNumeroPorte()
+int getCilindrata()
+double getPrezzoMax()
+StatoMacchina getStatoMacchina()
utilizza
utilizza
FiltroBuilder
+Builder setModello(String modello)
+builder setMarca(String marca)
+builder setKm(int km)
+builder setNumeroPorte(int numeroPorte)
+builder setCilindrata(int Cilindrata)
+builder setPrezzoMax(double PrezzoMax)
+Builder setStatoMacchina(StatoMacchina statoMacchina)
+Filtro build()
AnagraficaClienti
+boolean registraCliente(Cliente cliente)
+Cliente cercaCliente(String criterio, StrategiaDiRicerca strategia)
+List cercaClienti(String parolaChiave, StrategiaDiRicerca strategiaDiRicerca)
+List getClienti()
+boolean ePresente(Cliente cliente)
FactoryCliente
+Cliente creaClienteRandom()
-String generaCodiceFiscaleRandom()
Preventivo
+Automobile getAuto()
+Cliente getCliente()
+double getPrezzoTotale()
ElementoListino
+Automobile getAutomobile()
+double getPrezzo()
FactoryAutomobile
+Automobile creaAutoRandom()
+Automobile creaAutoRandomUsata()
utilizza
gestisce
crea
utilizza
AnagraficaClientiImpl
Cliente
+String getNome()
+String getCognome()
+String getEmail()
+String getTelefono()
+String getCf()
StrategiaDiRicerca
+List cerca(List clienti, String parolaChiave)
Automobile
+String modello
+String marca
+int km
+int numeroPorte
+int cilindrata
+int cavalli
+String targa
+StatoMacchina statoMacchina
utilizza
ClienteBuilder
+Builder nome(String nome)
+Builder cognome(String cognome)
+Builder email(String email)
+Builder telefono(String telefono)
+Builder codiceFiscale(String codiceFiscale)
+Cliente build()
-enumeration-
StatoMacchina
USATO
NUOVO


Leasing
+String getAutoSelezionata()
+int getDurataContratto()
+double getPrezzoMensile()
+int getLimiteChilometrico()
Autonoleggio
+Autonoleggio()
+List getAutomobili()
+void noleggiaAuto(AutomobileNoleggio auto, int giorni)
FactoryAutomobiliNoleggio
+FactoryAutomobiliNoleggio()
+List creaAutoRandom()
dipende
utilizza
utilizza
crea
AutomobileNoleggio
+String getModello()
+String getMarca()
+int getNumeroPorte()
+int getCilindrata()
+int getCavalli()
+DisponibilitàAuto getDisponibilitàAuto()
+void setCostoNoleggio(double costoTotale)
+double getCostoNoleggio()
+String getStato()
+void setStato(DisponibilitàAuto stato)
+boolean tempoNoleggio()
Assicurazione
+static double calcolaCostoAssicurazione(int giorni, String copertura)
+static double calcolaCostoBasso(int giorni)
+static double calcolaCostoMedia(int giorni)
+static double calcolaCostoAlta(int giorni)
«enumeration»
DisponibilitàAuto
+Disponibile
+NonDisponibile

Capitolo 2: Design

2.1 Architettura

Per lo sviluppo dell'architettura del concessionario si è deciso di seguire il pattern architetturale MVC (Model- View- Controller). Questo pattern separa le responsabilità del software in tre componenti principali: Model (modello), View (vista), e Controller (controllore). L 'obiettivo di questa separazione è migliorare la manutenibilità, la scalabilità e la modularità del software, consentendo di modificare una componente senza influenzare le altre.

2.1.1 Model

Il Modello si occupa di gestire le metodologie di accesso e modifica dei dati dell'applicazione e del dominio applicativo, determinando come le interazioni dell'utente influenzano lo stato corrente dell'applicazione. Lo scopo del modello sarà quindi fornire al Controller un accesso allo stato attuale dell'applicazione, inoltre è responsabile di includere al suo interno i principali componenti come cliente, automobile e vendita.

2.1.2 View

La View consiste nella schermata dell'applicazione che si occupa della User Experience e dell'interazione con l'utente. É compito delle varie View registrare ed informare il rispettivo Controller di interazioni con l'applicazione da parte dell'utente, in attesa di una sua risposta sul cambiamento dei dati. Si è deciso di utilizzare un' associazione 1:1 tra View e Controller. Ogni View ha associato a sé il relativo Controller, quest' ultimo è il responsabile di fornire e lavorare sui dati richiesti dal contesto di quella pagina.

2.1.3 Controller

Il Controller è quella componente a cui spetta di gestire in maniera consequenziale le interazioni da parte dell'utente nella View, comunicando quindi al Model il cambiamento avvenuto. Una volta che il Model avrà completato l'elaborazione della richiesta di cambiamento, il Controller avviserà la propria View, in modo tale che quest'ultima possa aggiornarsi in maniera coerente secondo le regole specificate dal Model.

2.2 Design dettagliato

2.2.1 Francesco Borrelli: Officina, Generatore targa

Implementazione del Generatore di Targa, Officina, Modifiche Varie, Sviluppo Collettivo delle Classi

Il progetto riguarda lo sviluppo di un generatore casuale di targhe automobilistiche basate su un formato comune introdotto dal Regolamento dell'Unione europea (CE) n. 2411/98. L'obiettivo principale è integrare le classi del progetto che fanno riferimento alle automobili (e alla loro creazione) in modo coeso. Un modulo chiave è quello dell'Officina, progettato per facilitare la gestione delle automobili, sia nuove che usate, includendo operazioni di manutenzione e controllo.
L'obiettivo principale è creare un'interfaccia grafica (GUI) che consenta di visualizzare il listino delle auto usate, selezionare l'auto da sottoporre a controllo e successivamente rimuoverla dalla lista delle auto disponibili. Il sistema segue un approccio orientato agli oggetti, con una netta separazione delle responsabilità tra le classi, rispettando il modello MVC. La FactoryAutomobile, seguendo il pattern Factory, gestisce la creazione delle automobili. Questo pattern centralizza la logica di istanziazione degli oggetti Automobile, rendendo il processo di creazione più organizzato e flessibile. Inoltre, il pattern Factory facilita l'integrazione di nuovi modelli di auto senza modificare il codice esistente, contribuendo alla manutenibilità e all'espansione futura del sistema.
Per garantire che l'interfaccia utente rifletta sempre accuratamente lo stato corrente del sistema, viene implementato anche il Pattern Observer. Questo pattern permette al modello (che contiene i dati delle automobili) di notificare automaticamente alla vista (GUI) qualsiasi cambiamento, come ad esempio l'aggiunta o la rimozione di un'auto dal listino. Quando l'utente seleziona un'auto per il controllo o la manutenzione, il modello invia una notifica agli osservatori registrati, come la GUI. La GUI, in risposta, aggiorna automaticamente l'interfaccia per riflettere il cambiamento, rimuovendo l'auto dalla lista delle auto disponibili. Questo meccanismo di aggiornamento automatico rende il sistema più reattivo e semplifica l'interazione tra i vari componenti.
Grazie all'adozione di questi design pattern, come il Factory e l'Observer, il sistema risulta non solo modulare, ma anche scalabile e facilmente estensibile. Questi pattern, applicati insieme al Model-View-Controller (MVC), contribuiscono a mantenere una chiara separazione tra la logica di business, l'interfaccia utente e il flusso di dati, facilitando sia la manutenzione che l'eventuale espansione del progetto in futuro.

Il modulo sviluppato si concentra su:

- **Gestione dell'Officina**: Consente di selezionare l'auto desiderata e di mandarla in officina per un controllo di routine;
- **Gestione delle targhe**: Consente di generare targhe secondo le normative UE;
- **Gestione del lavoro collettivo** : Suddivisione del lavoro tra i vari componenti dove fosse necessario.

Architettura e Design Pattern

Pattern MVC (Model-View-Controller)

Il sistema segue il Pattern MVC che separa le tre componenti principali:

- Model:OfficinaModel rappresenta la logica di business e la gestione dei dati relativi alle automobili usate. È responsabile del recupero e della manipolazione delle informazioni sulle automobili nel listino dell'officina.
- View:OfficinaView gestisce l'interfaccia utente e la visualizzazione delle informazioni sulle auto. La GUI è implementata utilizzando Swing e permette all'utente di selezionare un'auto dalla lista e avviare la riparazione.
- Controller:OfficinaController funge da mediatore tra il modello e la vista, gestendo la logica dell'applicazione, come il processo di riparazione delle automobili, e aggiornando la vista in base alle modifiche del modello.

Il Pattern MVC facilita l'estensione e la manutenzione del codice, poiché permette di aggiornare una singola componente senza influenzare direttamente le altre. Ad esempio, è possibile aggiungere nuove funzionalità al modello o modificare l'interfaccia utente senza alterare la logica di business.

Pattern Factory

Nel sistema è stato utilizzato anche il Pattern Factory per la creazione di oggetti complessi come le automobili. Questo pattern permette di centralizzare la logica di creazione degli oggetti, rendendo più semplice e modulare l'aggiunta di nuovi tipi di oggetti in futuro.

- FactoryAutomobiliUsate: Una classe factory dedicata viene utilizzata per la creazione delle auto usate da inserire nel listino. Questo pattern facilita l'aggiunta di nuovi modelli di auto o varianti specifiche senza modificare la logica principale.

Pattern Observer

E’ un meccanismo di progettazione che prevede la presenza di due ruoli principali: l'Observable (o soggetto osservato) e gli Observer (o osservatori). Il modello agisce come soggetto osservato, e ogni volta che viene modificato (ad esempio, con l’aggiunta o la rimozione di un’automobile), notifica automaticamente tutti gli osservatori registrati. Questo permette alle varie componenti del sistema che si sono registrate come osservatori (come la vista) di aggiornarsi in modo automatico e reattivo senza la necessità di controllare manualmente lo stato del modello.

Modulo Officina

Classe: OfficinaModel

La classe OfficinaModel gestisce le automobili disponibili nell'officina e le riparazioni delle auto usate. La lista delle auto usate è rappresentata da un oggetto Listino, che contiene gli oggetti ElementoListino, i quali a loro volta racchiudono le automobili.

- Attributi principali:
    - Listino autoUsate: Un elenco di auto usate presenti nell'officina.

- Metodi principali:
    - getAutoUsate(): Restituisce la lista delle auto usate disponibili.
    - cercaAuto(String nome): Cerca un'auto usata per marca nel listino.
    - rimuoviAuto(Automobile auto): Rimuove un'auto specifica dal listino delle auto usate una volta che è stata riparata o venduta.
    - aggiungiAuto(Automobile auto): Aggiunge una nuova auto al listino delle auto usate.
    - aggiornaStatoAuto(Automobile auto, String nuovoStato): Aggiorna lo stato dell'auto (ad esempio, da "In Riparazione" a "Disponibile").
    - getAutoById(String id): Ottiene un'auto specifica dal listino utilizzando un identificatore unico.

Classe: OfficinaView

La classe OfficinaView rappresenta l'interfaccia utente dell'officina e fornisce una GUI che mostra le auto disponibili per la riparazione e consente all'utente di avviare il processo di riparazione.

- Elementi principali della GUI:
    - JList<String> listaAuto: Visualizza l'elenco delle auto disponibili.
    - JLabel dettagliAuto: Mostra i dettagli dell'auto selezionata.
    - JButton bottoneRipara: Un pulsante per avviare la riparazione dell'auto selezionata.
- Metodi principali:
    - mostraAutoDisponibili(List<Automobile> automobili): Aggiorna l'elenco delle auto visualizzate nella GUI.
    - mostraDettagliAuto(Automobile auto): Mostra i dettagli dell'auto selezionata nella GUI.
    - mostraMessaggioRiparazioneAvviata(Automobile auto): Mostra un messaggio che notifica l'inizio della riparazione dell'auto selezionata.
    - aggiornaListaAuto(): Ricarica l'elenco delle auto disponibili dopo una modifica nel modello.
    - mostraErrore(String messaggio): Mostra messaggi di errore all'utente.
    - getSelezioneAuto(): Ottiene l'auto selezionata dall'utente.

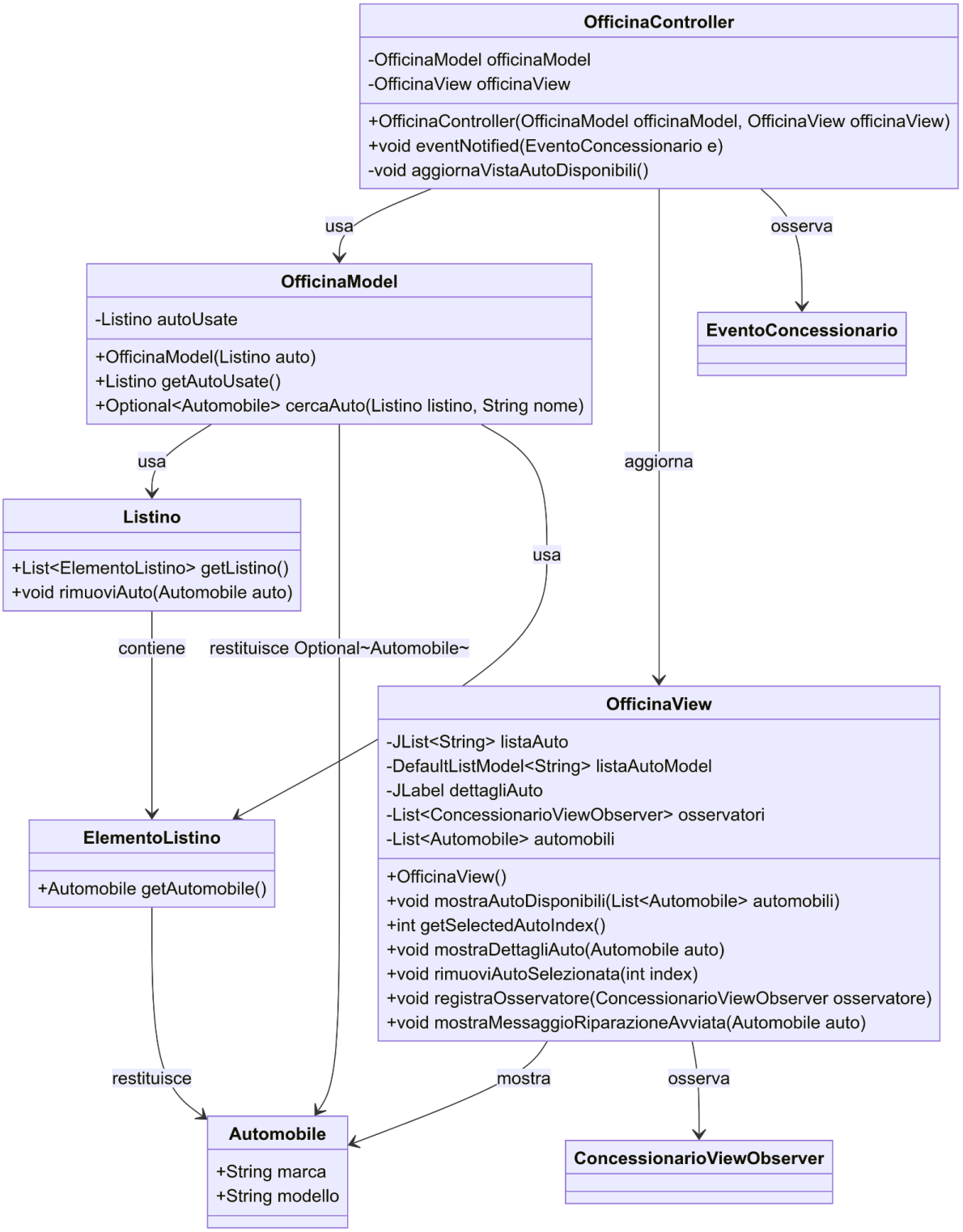
Classe: OfficinaController

La classe OfficinaController gestisce la logica di business tra il modello e la vista e coordina il processo di riparazione delle auto.

- Attributi principali:
    - OfficinaModel officinaModel: Un riferimento al modello dell'officina.
    - OfficinaView officinaView: Un riferimento alla vista dell'officina.
- Metodi principali:
    - eventNotified(EventoConcessionario e): Gestisce gli eventi provenienti dalla vista, come la richiesta di riparazione di un'auto.
    - aggiornaVistaAutoDisponibili(): Aggiorna l'elenco delle auto visualizzate nella vista in base allo stato attuale del modello.

- gestisciSelezioneAuto(): Gestisce la selezione di un'auto da parte dell'utente e avvia il processo di riparazione o visualizzazione dei dettagli.
- eseguiRiparazione(Automobile auto): Avvia il processo di riparazione per l'auto selezionata.
- gestisciAggiornamentiAuto(): Gestisce aggiornamenti o cambiamenti nell'auto e aggiorna la vista di conseguenza.

la scelta dei design pattern ed una buona implementazione hanno contribuito a creare un sistema efficace, scalabile e manutenibile, pronto per ulteriori sviluppi e miglioramenti futuri.


OfficinaController
-OfficinaModel officinaModel
-OfficinaView officinaView
+OfficinaController(OfficinaModel officinaModel, OfficinaView officinaView)
+void eventNotified(EventoConcessionario e)
-void aggiornaVistaAutoDisponibili()
usa
osserva
OfficinaModel
-Listino autoUsate
+OfficinaModel(Listino auto)
+Listino getAutoUsate()
+Optional<Automobile> cercaAuto(Listino listino, String nome)
EventoConcessionario
usa
aggiorna
Listino
+List<ElementoListino> getListino()
+void rimuoviAuto(Automobile auto)
usa
contiene
restituisce Optional~Automobile~
OfficinaView
-JList<String> listaAuto
-DefaultListModel<String> listaAutoModel
-JLabel dettagliAuto
-List<ConcessionarioViewObserver> osservatori
-List<Automobile> automobili
+OfficinaView()
+void mostraAutoDisponibili(List<Automobile> automobili)
+int getSelectedAutoIndex()
+void mostraDettagliAuto(Automobile auto)
+void rimuoviAutoSelezionata(int index)
+void registraOsservatore(ConcessionarioViewObserver osservatore)
+void mostraMessaggioRiparazioneAvviata(Automobile auto)
ElementoListino
+Automobile getAutomobile()
restituisce
mostra
osserva
Automobile
+String marca
+String modello
ConcessionarioViewObserver

2.2.2 Gioele Pula: Modulo Noleggio auto, Leasing auto e assicurazione

Implementazione di Noleggio auto, leasing auto e assicurazione.
Il sistema segue un approccio orientato agli oggetti con una chiara separazione delle responsabilità tra le classi. L'architettura MVC permette una facile manutenzione e scalabilità del codice. L'uso di factory (FactoryAutomobiliNoleggio) per creare oggetti e l'utilizzo di thread per gestire lo stato delle automobili (es. disponibilità dopo il noleggio) mostra un'attenzione per quella che è la gestione dello stato in tempo reale. L'idea di noleggio e leasing è stata volutamente implementata come a se stante, in maniera tale da rispecchiare quella che è la realtà, in quanto la concessionaria permette si di acquistare e vendere auto ma, anche di effettuare il noleggio e il leasing.

Il modulo sviluppato si concentra su:

- **Gestione del Leasing**: Consente di selezionare un'auto, definire la durata del contratto di leasing, stabilire un limite chilometrico, e calcolare il prezzo mensile del leasing;
- **Gestione del Noleggio**: Consente di selezionare un'auto, definire la durata del noleggio, inserire una copertura assicurativa e definire il prezzo del contratto.

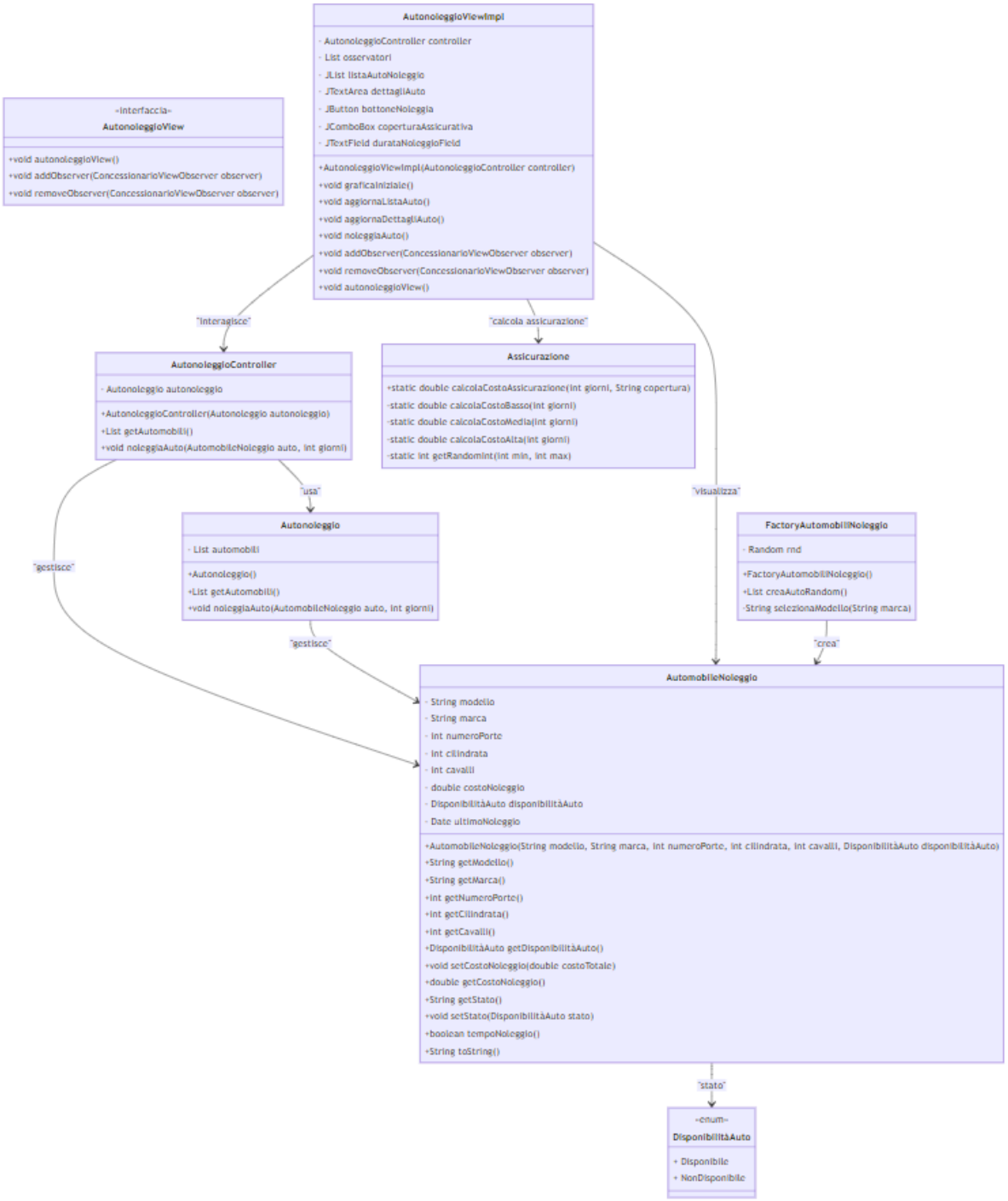
Architettura e Design Pattern:

Pattern MVC (Model-View-Controller)

Il modulo utilizza il Pattern MVC, che separa la logica dell'applicazione (Controller), la gestione dei dati (Model), e la presentazione grafica (View). Questo pattern facilita l'estensione e la manutenzione del codice.

Il modulo di **noleggio** è progettato per essere modulare e scalabile, con una chiara separazione tra la logica di business e l'interfaccia utente. Le automobili possono essere facilmente gestite e monitorate attraverso il controller, mentre la view fornisce una GUI intuitiva per l'utente finale. La gestione delle assicurazioni e la disponibilità delle auto sono trattate con metodi dedicati, garantendo una logica di business robusta e flessibile.

Modulo Autonoleggio



Modello

Classe: AutomobileNoleggio

- **Attributi**:
    - Dati dell'auto come String modello, String marca, int numeroPorte, int cilindrata, int cavalli.
    - double costoNoleggio: Costo del noleggio.
    - DisponibilitàAuto disponibilitàAuto: Stato di disponibilità dell'auto (Disponibile o NonDisponibile).
    - Data ultimoNoleggio: Data dell'ultimo noleggio.
- **Metodi**:
    - Getter e setter per i vari attributi.
    - boolean tempoNoleggio(): Controlla se è trascorso il tempo necessario per rendere nuovamente disponibile l'auto.
    - String getStato(): Restituisce lo stato attuale di disponibilità dell'auto.
    - void setStato(DisponibilitàAuto stato): Modifica lo stato dell'auto e aggiorna la data dell'ultimo noleggio se l'auto diventa non disponibile.
    - String toString(): Restituisce una stringa con i dettagli dell'auto.

Classe: DisponibilitàAuto

Una semplice classe enum che ci permette di rappresentare lo stato di disponibilità di un'auto per il noleggio.

L'enum DisponibilitàAuto ha due valori costanti:

1. Disponibile;
2. NonDisponibile;

Questi valori rappresentano rispettivamente i due stati possibili di un'auto in termini di disponibilità per il noleggio.

Classe: FactoryAutomobiliNoleggio

- **Attributi**:
    - Random rnd: Per generare valori casuali per le caratteristiche delle auto.
- **Metodi**:

- List<AutomobileNoleggio> creaAutoRandom(): Crea e restituisce una lista di auto con attributi generati casualmente.
- String selezionaModello(String marca): Restituisce un modello casuale in base alla marca selezionata.

Classe Assicurazione

Classe fondamentale per l'implementazione del progetto, permette di andare a scegliere una copertura assicurativa per le auto che vengono noleggiate, basandosi su tre diversi livelli di copertura assicurativa in base alle proprie esigenze.

- **Metodi**:
  - double calcolaCostoAssicurazione(int giorni, String copertura): Calcola il costo dell'assicurazione in base alla copertura selezionata e alla durata del noleggio.
  - Metodi privati come calcolaCostoBasso(int giorni), calcolaCostoMedia(int giorni), calcolaCostoAlta(int giorni), che calcolano il costo assicurativo per diversi livelli di copertura.

Classe Autonoleggio

- **Attributi**:
  - List<AutomobileNoleggio> automobili: Lista delle automobili disponibili per il noleggio.
- **Metodi**:
  - List<AutomobileNoleggio> getAutomobili(): Restituisce la lista delle automobili.
  - Void noleggiaAuto(AutomobileNoleggio auto, int giorni): Gestisce il processo di noleggio, calcola il costo totale del noleggio e avvia un thread per rendere disponibile l'auto dopo un periodo.

Interfaccia: AutonoleggioView

AutonoleggioViewImpl implementa l'interfaccia AutonoleggioView e interagisce con il controller AutonoleggioController per aggiornare la GUI in base alle operazioni effettuate dall'utente. Questa classe visualizza le informazioni delle automobili (oggetti AutomobileNoleggio) e permette agli utenti di interagire con il sistema, ad esempio per noleggiare un'auto.

- **Metodi**:
    - void addObserver(ConcessionarioViewObserver observer): Permette di aggiungere un osservatore al modello della vista.
    - void removeObserver(ConcessionarioViewObserver observer): Permette di rimuovere un osservatore.

Classe: AutonoleggioViewImpl (Implementazione di AutonoleggioView)

- **Attributi**:
    - AutonoleggioController controller: Riferimento al controller per invocare la logica di business.
    - List<ConcessionarioViewObserver> osservatori: Lista di osservatori per il pattern Observer.
    - Elementi GUI come JList<AutomobileNoleggio> listaAutoNoleggio, JTextArea dettagliAuto, JButton bottoneNoleggia, JComboBox<String> coperturaAssicurativa, JTextField durataNoleggioField, che rappresentano i componenti grafici per l'interazione utente.
- **Metodi**:
    - void graficaIniziale(): Inizializza la GUI, configura i vari pannelli, i layout e imposta gli ascoltatori per le azioni dell'utente.
    - void aggiornaListaAuto(): Aggiorna la lista delle automobili visualizzata nell'interfaccia.
    - void aggiornaDettagliAuto(): Aggiorna i dettagli dell'auto selezionata, mostrando informazioni come lo stato di disponibilità.
    - void noleggiaAuto(): Gestisce il processo di noleggio quando l'utente seleziona un auto e inserisce la durata del noleggio, calcolando il costo totale incluso l'assicurazione.

Controller AutonoleggioController

AutonoleggioController interagisce con Autonoleggio per gestire le operazioni di noleggio e recuperare la lista delle automobili disponibili. Funziona come mediatore tra il modello (Autonoleggio, AutomobileNoleggio) e la view (AutonoleggioViewImpl), facilitando la comunicazione tra questi componenti.

Classe: AutonoleggioController

- **Attributi**:
    - Autonoleggio autonoleggio: Un riferimento alla classe Autonoleggio, che rappresenta il modello di business del noleggio di automobili.
- **Metodi**:
    - List<AutomobileNoleggio> getAutomobili(): Restituisce la lista delle automobili disponibili nel modello.

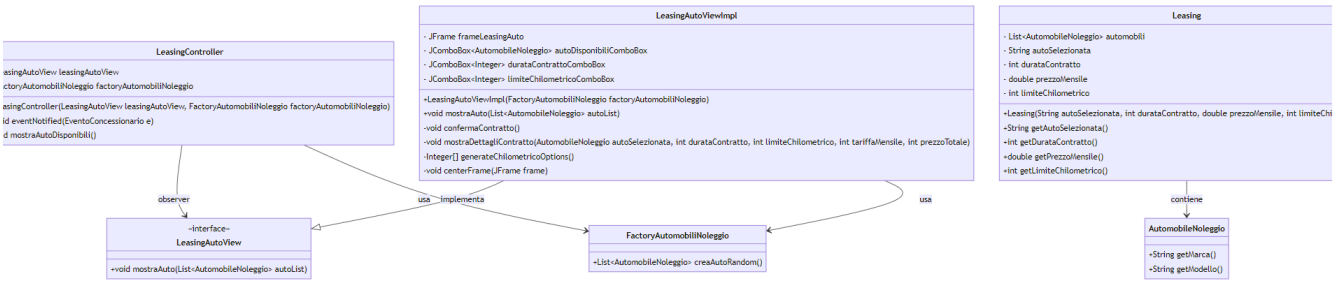
- void noleggiaAuto(AutomobileNoleggio auto, int giorni): Avvia il processo di noleggio di un'auto per un numero specifico di giorni, delegando al modello la gestione della disponibilità e del costo del noleggio.

Modulo Leasing

Pattern MVC (Model-View-Controller) per il leasing delle auto.

Il modulo utilizza il Pattern MVC, che separa la logica dell'applicazione (Controller), la gestione dei dati (Model), e la presentazione grafica (View). Questo pattern facilita l'estensione e la manutenzione del codice.

- Model: Include le classi che rappresentano i dati e la logica di business, come Leasing, che contiene informazioni sul contratto di leasing.
- View: Le classi LeasingAutoView e LeasingAutoViewImpl si occupano della presentazione delle informazioni all'utente e dell'interazione con esso. Le GUI sono implementate utilizzando Swing, con componenti come JComboBox, JLabel, e JButton per costruire l'interfaccia utente.
- Controller: La classe LeasingController gestisce l'interazione tra il Model e la View. Riceve eventi dalla View (come la selezione di un'auto per il leasing) e aggiorna il Modello di conseguenza. Successivamente, notifica la View affinché aggiorni l'interfaccia utente.



Modello:

La classe Leasing

Rappresenta un contratto di leasing per un'automobile e contiene gli attributi necessari per gestire i dettagli del contratto:

Attributi:

- autoSelezionata: Una stringa che identifica l'auto scelta per il leasing. Questo attributo è fondamentale per associare un contratto a un veicolo specifico.

- durataContratto: Un valore di tipo intero che indica la durata del contratto di leasing in anni. Questo valore influenza direttamente il costo totale del leasing.
- prezzoMensile: Un double che rappresenta il costo mensile del leasing. Viene calcolato in base alla durata del contratto e al tipo di auto selezionata.
- limiteChilometrico: Un valore di tipo intero che stabilisce il limite di chilometraggio annuale incluso nel contratto di leasing.
- automobili: Una lista di AutomobileNoleggio (auto noleggiabili) disponibile per il leasing. Questo permette al sistema di presentare le opzioni disponibili all'utente.

Metodi:

- I metodi getter come getAutoSelezionata(), getDurataContratto(), getPrezzoMensile(), e getLimiteChilometrico() permettono di accedere agli attributi privati della classe, garantendo l'incapsulamento dei dati.

View:

Classe LeasingAutoView:

LeasingAutoView è un'interfaccia che definisce un contratto per le classi che implementano la vista (view) dell'interfaccia utente per la gestione del leasing di automobili. Essa fornisce un metodo per visualizzare una lista di automobili disponibili.

Metodi:

- void mostraAuto(List<AutomobileNoleggio> autoList);

  La sua funzione è quella di ricevere una lista di oggetti AutomobileNoleggio e di farli visualizzare all'utente.

Classe LeasingAutoViewImpl implementazione della classe LeasingAutoView

La View (LeasingAutoViewImpl) fornisce un'interfaccia grafica che permette all'utente di selezionare un'auto, impostare i parametri del contratto di leasing e confermare l'operazione. Il LeasingController gestisce la logica della presentazione delle auto disponibili e la conferma del contratto di leasing.

Questa classe implementa l'interfaccia LeasingAutoView e gestisce l'interfaccia utente per la selezione e conferma del contratto di leasing.

- **Attributi**:
  - frameLeasingAuto: Un oggetto JFrame che rappresenta la finestra principale dell'interfaccia leasing.
  - autoDisponibiliComboBox: Un JComboBox che mostra le auto disponibili per il leasing. Le auto sono selezionabili dall'utente.

    - durataContrattoComboBox: Un JComboBox che permette all'utente di selezionare la durata del contratto in anni.
    - limiteChilometricoComboBox: Un JComboBox che offre opzioni di chilometraggio annuale.
    - factoryAutomobiliNoleggio: Un riferimento a FactoryAutomobiliNoleggio, usato per generare le auto da visualizzare.

- **Metodi**:
    - mostraAuto(List<AutomobileNoleggio> autoList): Mostra una lista di auto disponibili nel JComboBox. Questo metodo è chiamato dal controller per aggiornare la GUI quando l'utente richiede di visualizzare le auto.
    - confermaContratto(): Questo metodo raccoglie i dati selezionati dall'utente (auto, durata del contratto, limite chilometrico), calcola la tariffa mensile e il prezzo totale, e visualizza i dettagli del contratto in una nuova finestra.
    - mostraDettagliContratto(...): Crea una nuova finestra (JFrame) che mostra i dettagli del contratto di leasing, come l'auto selezionata, la durata del contratto, il limite chilometrico, la tariffa mensile e il prezzo totale.

Controller:

Classe LeasingController:

Il LeasingController implementa l'interfaccia ConcessionarioViewObserver e gestisce la logica di controllo tra il modello (i dati del leasing) e la view (interfaccia utente).

- **Attributi**:
    - leasingAutoView: Riferimento alla view (LeasingAutoView) che mostra e gestisce la selezione delle auto per il leasing.
    - factoryAutomobiliNoleggio: Utilizzata per creare e ottenere le auto disponibili per il leasing.
- **Metodi**:
    - eventNotified(EventoConcessionario e): Metodo sovrascritto che gestisce diversi eventi di interazione nella vista principale del concessionario. In particolare, quando viene notificato l'evento LEASING_AUTO, questo metodo richiama mostraAutoDisponibili() per aggiornare la lista delle auto disponibili nel modulo di leasing.
    - mostraAutoDisponibili(): Chiama il metodo creaAutoRandom() della factory per ottenere una lista di auto noleggiabili e la passa alla view tramite mostraAuto().

- **Randomizzazione dei Dati**: Durante l'inizializzazione, le automobili e i clienti vengono creati con dati casuali. Questo approccio è utile per testare il sistema con vari scenari e garantire la robustezza dell'applicazione.

- **Separazione delle Responsabilità**: Le classi sono progettate per avere una singola responsabilità, il che facilita la manutenzione e l'estensione del codice. Ad esempio, la classe Leasing si occupa esclusivamente della logica di leasing, mentre la LeasingAutoViewImpl si occupa della presentazione.
- **Estensibilità**: L'uso del pattern Factory rende semplice aggiungere nuovi tipi di automobili o clienti senza modificare il codice esistente.

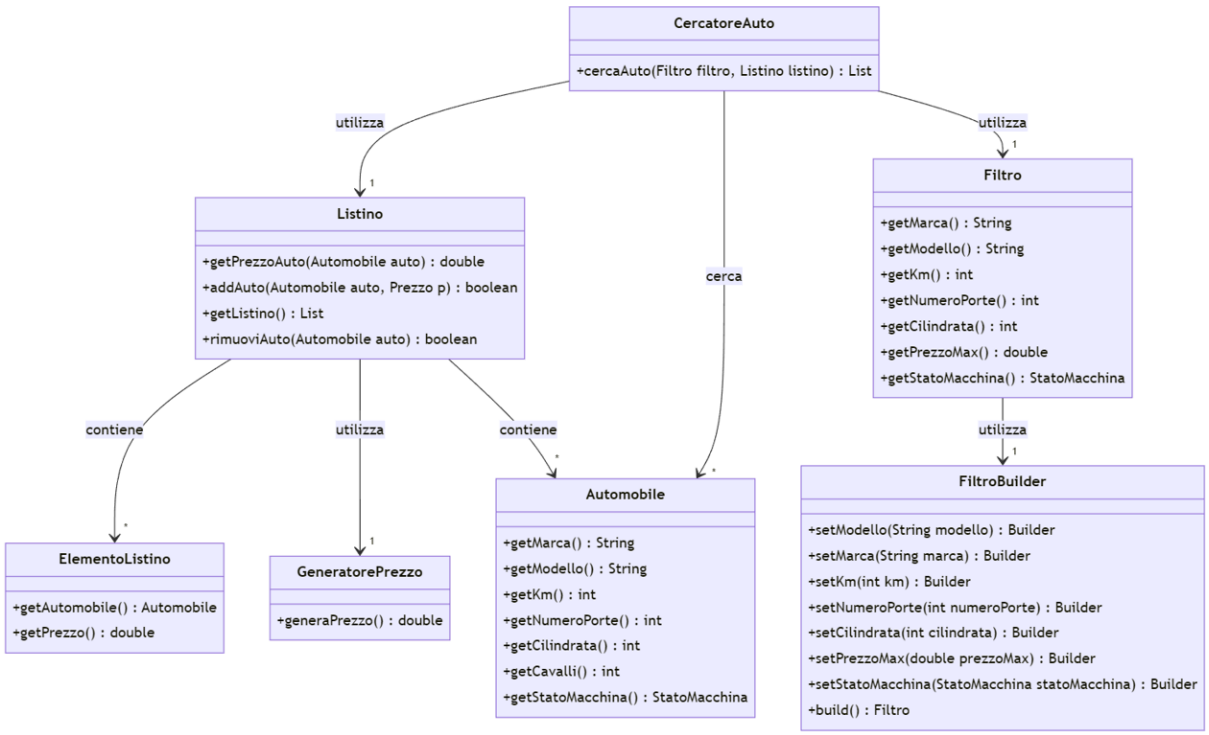
2.2.3 Alessandro Petreti: Gestione Clienti, Auto e Vendite

Ricerca delle auto

Il modulo di ricerca delle auto è progettato per filtrare un elenco di automobili in base a criteri specifici definiti dall'utente. I criteri di ricerca includono marca, modello, chilometraggio, numero di porte, cilindrata, stato della macchina e prezzo massimo. L'obiettivo è fornire un sistema flessibile e dinamico che permetta agli utenti di trovare le auto che soddisfano esattamente le loro preferenze.

Per realizzare questa funzionalità, è stata adottata una soluzione basata su design pattern ben noti. In particolare, la classe `Filtro` utilizza il **Builder Pattern**. Questo pattern è stato scelto per permettere una costruzione fluida degli oggetti filtro, che possono essere configurati con vari parametri opzionali. Il Builder Pattern facilita la creazione di istanze di `Filtro` consentendo di aggiungere o modificare i parametri senza influire sugli altri.

La classe `CercatoreAuto` è responsabile dell'applicazione dei criteri definiti nella classe `Filtro` per selezionare le automobili dal `Listino`. Utilizza i criteri di ricerca specificati per filtrare le auto e garantire che solo quelle che soddisfano tutti i criteri vengano incluse nei risultati. Questo approccio offre una chiara separazione tra la definizione dei criteri di ricerca e la loro applicazione.



Ricerca Clienti

Il modulo di gestione clienti all'interno della concessionaria è stato progettato per la registrazione e la ricerca dei clienti. Questo modulo consente di gestire operazioni essenziali come l'aggiunta di nuovi clienti, la ricerca per codice fiscale o per nome/cognome, e la gestione dell'anagrafica complessiva. Per risolvere le esigenze del sistema e favorire una struttura chiara e facilmente modificabile, sono stati impiegati diversi **design pattern** noti, tra cui **Strategy**, **Factory** e **Builder**

La necessità di gestire un anagrafica clienti in modo efficiente richiede un sistema che sia in grado di eseguire diverse operazioni su una lista di clienti, come l'aggiunta, la ricerca e il controllo della loro presenza. Tuttavia, la ricerca deve supportare criteri differenti, come il codice fiscale o il nome/cognome.

Il design adottato suddivide le responsabilità tra più classi, ognuna delle quali incapsula una specifica parte del processo di gestione dei clienti.

AnagraficaClienti è un'interfaccia che espone le operazioni principali per gestire l'anagrafica dei clienti. La sua implementazione concreta, `AnagraficaClientiImpl`, contiene la logica per aggiungere clienti e cercarli in base a diversi criteri, inoltre usa come struttura dati per memorizzare i clienti una lista.

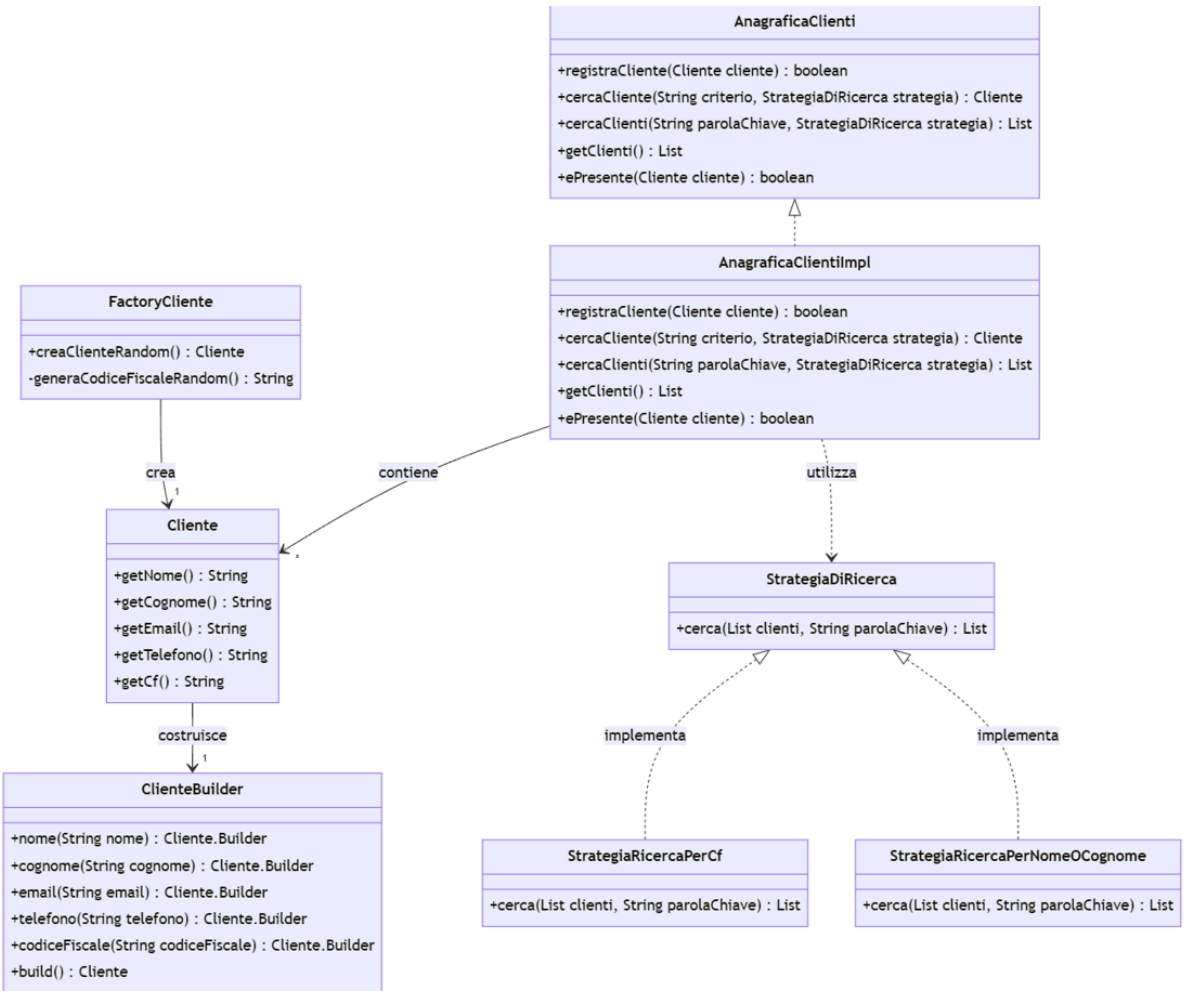
Strategy Pattern: Per risolvere il problema della ricerca flessibile, il sistema utilizza il pattern Strategy. La classe `AnagraficaClientiImpl` non contiene logica di ricerca specifica, ma delega questa responsabilità a una strategia. Le strategie di ricerca sono rappresentate dall'interfaccia `StrategiaDiRicerca`, che viene implementata da diverse classi, come `StrategiaRicercaPerCf` e `StrategiaRicercaPerNomeOCognome`. Ogni classe di strategia incapsula un diverso algoritmo di ricerca:

- `StrategiaRicercaPerCf` consente di cercare un cliente in base al codice fiscale;
- `StrategiaRicercaPerNomeOCognome` permette di cercare clienti in base a nome o cognome.

Grazie a questo pattern, è possibile aggiungere nuovi tipi di ricerca senza modificare il codice esistente di `AnagraficaClientiImpl`, facilitando l'estensione del sistema.

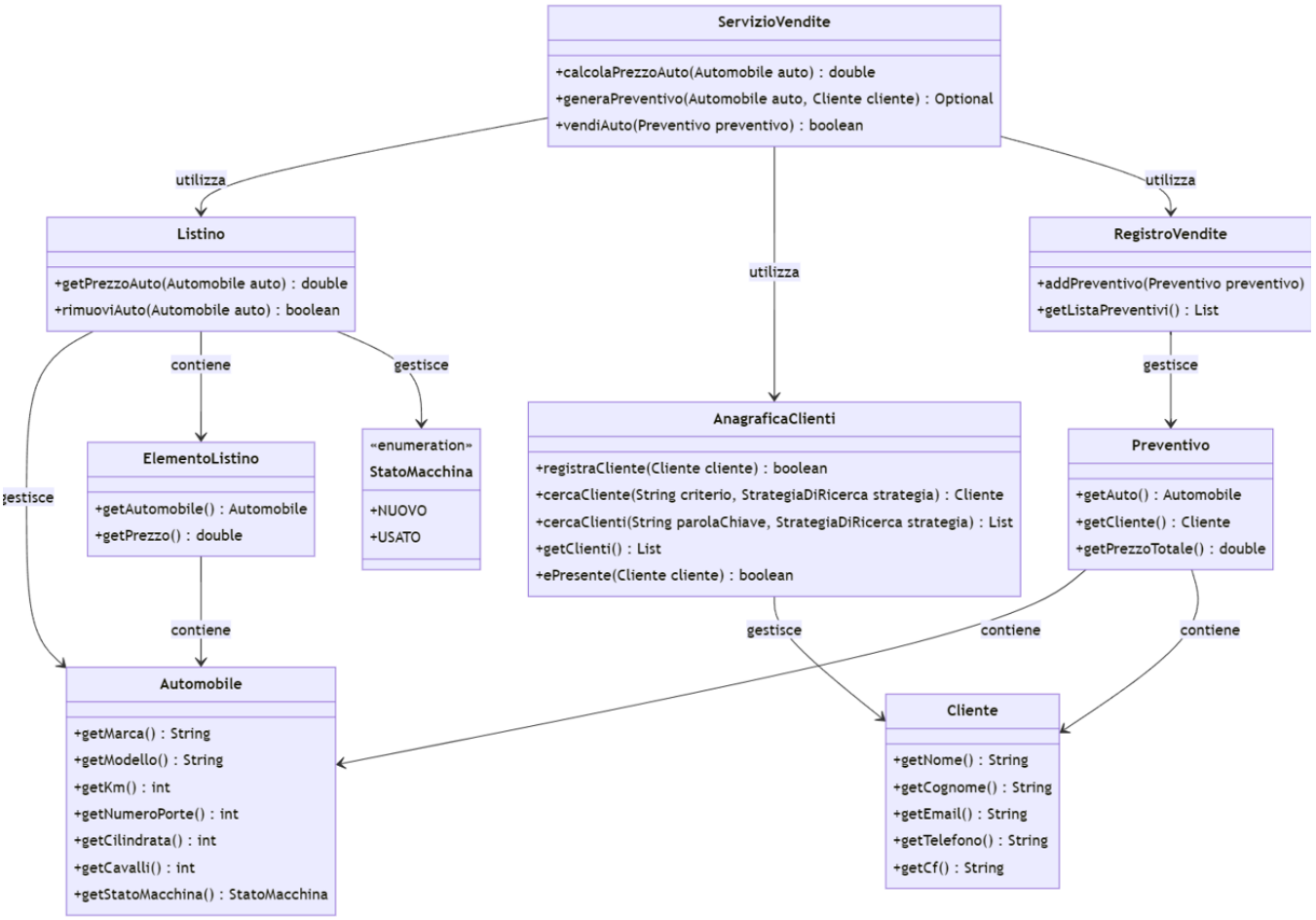
Factory Pattern: La creazione dei clienti è centralizzata nella classe `FactoryCliente`, che utilizza il Factory Pattern per generare oggetti `Cliente`. Questo approccio centralizzato evita di dover stanziare clienti direttamente all'interno del codice, separando la logica di creazione dal resto del sistema. In questo caso, la factory viene utilizzata per creare clienti con dati casuali utili alla generazione di un base di clienti iniziale.

Builder Pattern: La classe `Cliente` utilizza il Builder Pattern per la creazione di oggetti. Invece di dover gestire un costruttore con molti parametri, il Builder permette di impostare solo i campi necessari in modo fluido. Questo pattern è particolarmente utile quando si lavora con oggetti complessi o con molti attributi, migliorando la leggibilità e la manutenibilità del codice.



Reparto Vendite

Nel contesto di un concessionario automobilistico, il reparto vendite ha la necessità di gestire diverse operazioni: come la creazione di preventivi, il calcolo del prezzo delle automobili sia nuove che usate e la gestione delle vendite. Il sistema deve interfacciarsi con altre componenti essenziali, come l'anagrafica clienti, il listino delle auto nuove e usate, e il registro delle vendite. La sfida principale è fornire una soluzione che permetta di gestire tutte queste operazioni mantenendo separata la logica di ciascun modulo (gestione clienti, gestione delle automobili e delle vendite).



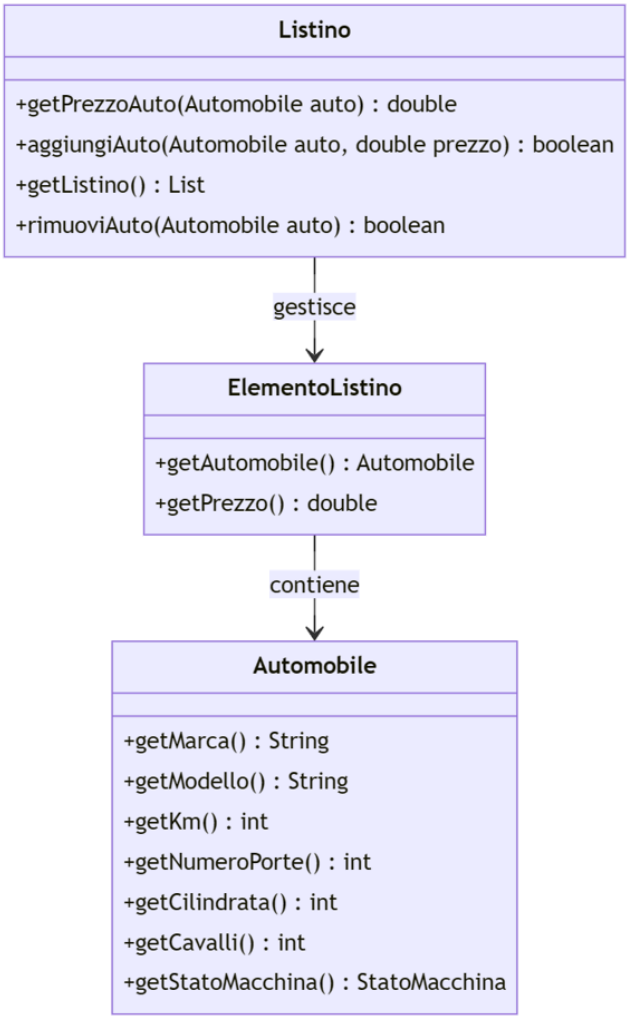
La classe **ServizioVendite** funge da gestore centrale, utilizzando altre classi per eseguire le operazioni necessarie. Tra queste troviamo:

- **Listino** e **ListinoUsato**, che servono per ottenere il prezzo delle automobili in base al loro stato (nuovo o usato).
- **AnagraficaClienti**, che consente di verificare la presenza di un cliente prima di generare un preventivo.
- **Preventivo**, rappresenta l'associazione tra cliente (presente in anagrafica) e l'auto selezionata con il prezzo totale calcolato.
- **RegistroVendite**, che memorizza i preventivi completati e rappresenta lo storico delle vendite effettuate.

Inoltre, un potenziale miglioramento del design potrebbe essere l'adozione del **pattern Strategy** per gestire in modo più flessibile le diverse modalità di calcolo del prezzo. Attualmente, il prezzo viene calcolato in base allo stato dell'auto (nuovo o usato), ma integrando il pattern Strategy, sarebbe possibile gestire più strategie, come ad esempio sconti stagionali, promozioni per determinati modelli, o tariffe speciali per clienti fedeli.

Listino Auto

Nel contesto di un concessionario automobilistico, il listino è responsabile della gestione dei prezzi delle automobili, sia nuove che usate. Deve permettere l'aggiunta, la rimozione e la consultazione dei prezzi delle automobili.



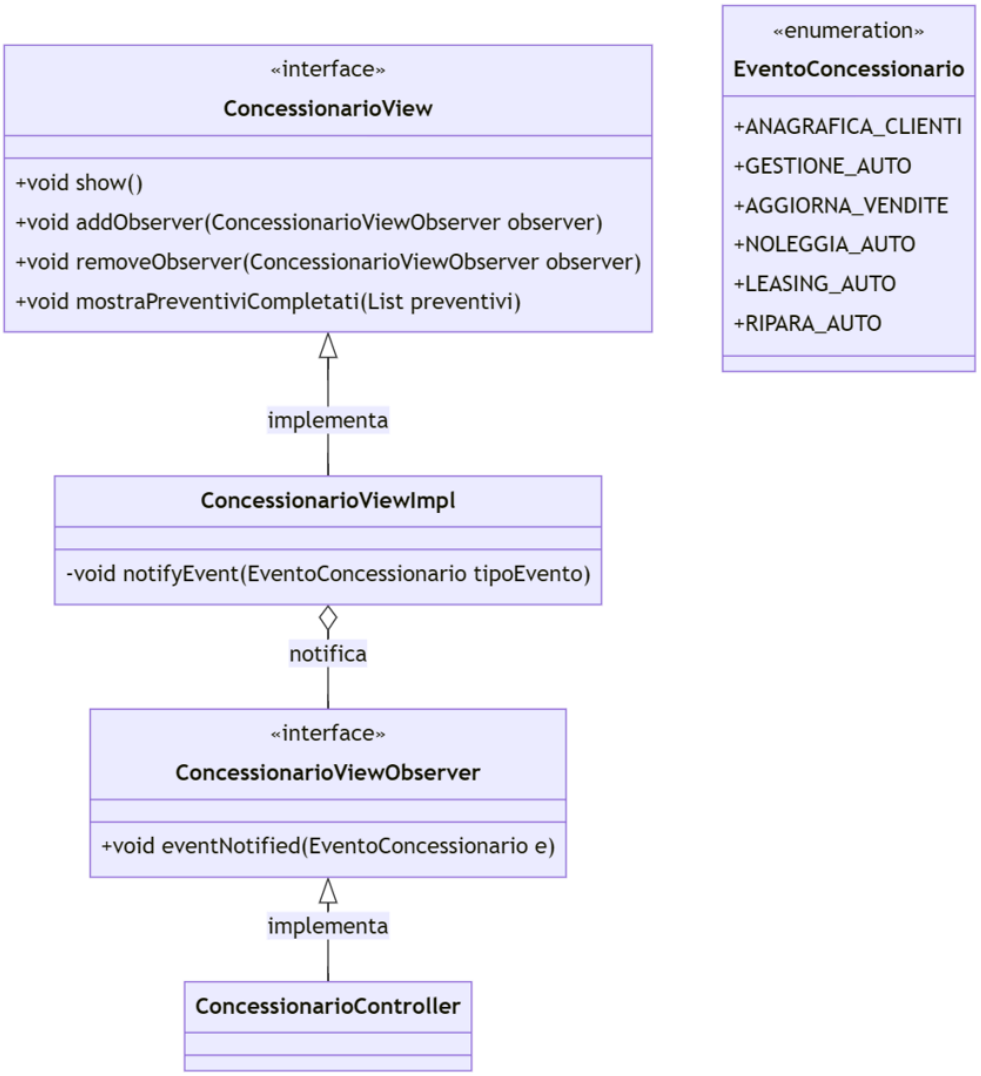
Il design del listino è basato su una struttura modulare che utilizza una lista collegata per archiviare gli elementi del listino. Ogni elemento del listino è rappresentato da una coppia `Automobile` e `Prezzo`, incapsulata nella classe `ElementoListino`. Questo approccio consente di mantenere i dati di prezzo associati alle automobili separati dall'automobile stessa. In generale un'automobile è descritta con le sue caratteristiche principali, mentre prezzo del nuovo o usato sono proprietà di un'auto specifica.

- **ElementoListino**: rappresenta un'associazione tra un'auto e il suo prezzo. Contiene due attributi principali: `automobile` e `prezzo`, e fornisce i metodi per accedere a questi attributi.
- **Listino**: gestisce una collezione di `ElementoListino`. Utilizza una `Lista` per memorizzare gli elementi, facilitando operazioni di inserimento e rimozione efficienti. La classe fornisce metodi per aggiungere (`aggiungiAuto`), rimuovere (`rimuoviAuto`), e ottenere il prezzo di un'auto (`getPrezzoAuto`).

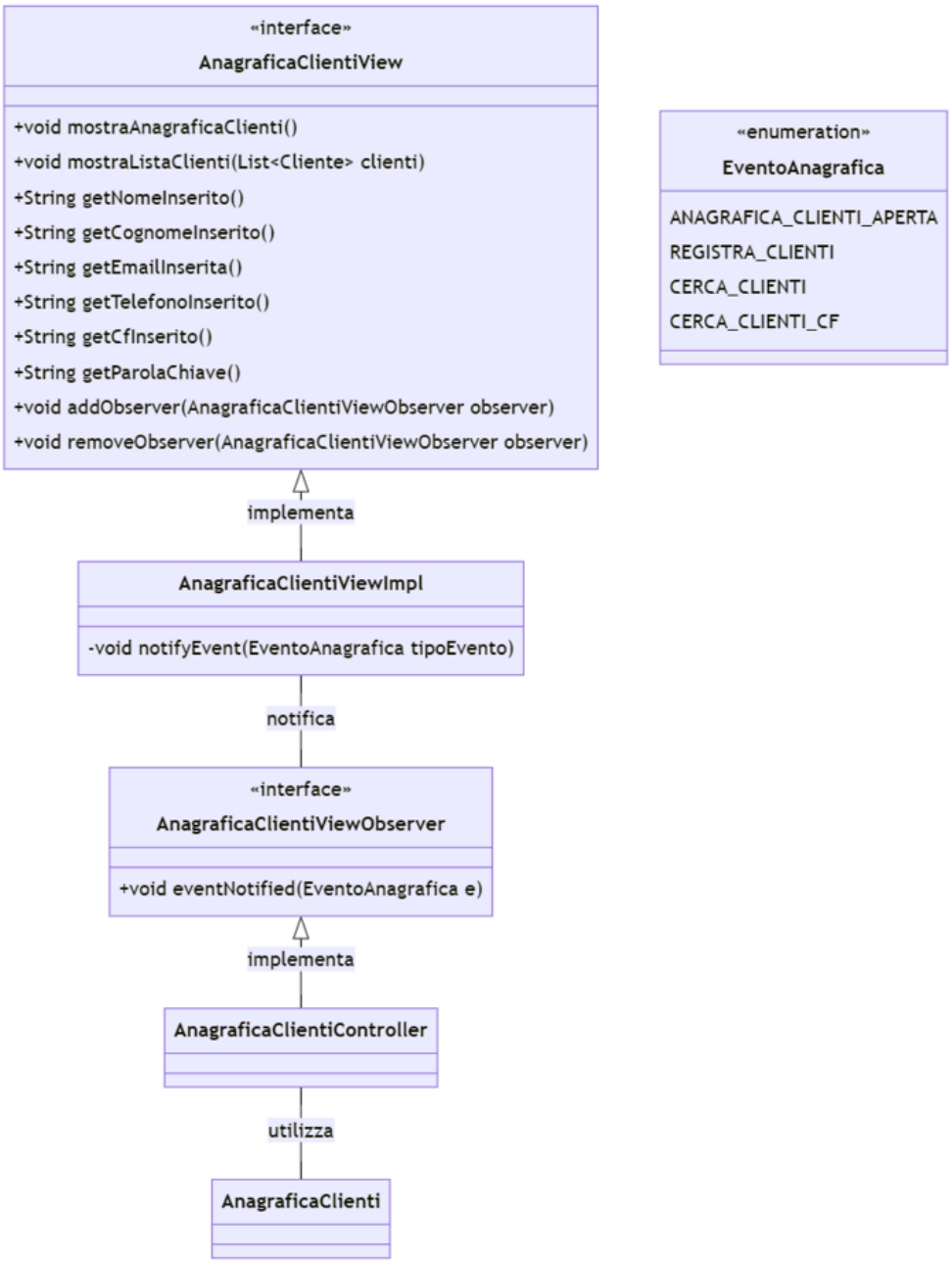
Le auto presenti all’interno del listino sono da intendersi come auto nuove e con disponibilità infinite, mentre il listino usato ha una logica simile a quella di un magazzino, quindi con disponibilità limitata.

View e Controller

In generale per la comunicazione tra view e controller è stato utilizzato il **pattern Observer**, che permette alla vista di notificare gli eventi agli osservatori registrati. Quando un evento viene generato dalla vista, viene notificato a tutti gli osservatori registrati tramite il metodo `notifyEvent`. In particolare, i controller sono gli osservatori della vista e reagiscono di conseguenza, ad esempio aggiornando il modello e controllando la View attraverso la sua interfaccia.

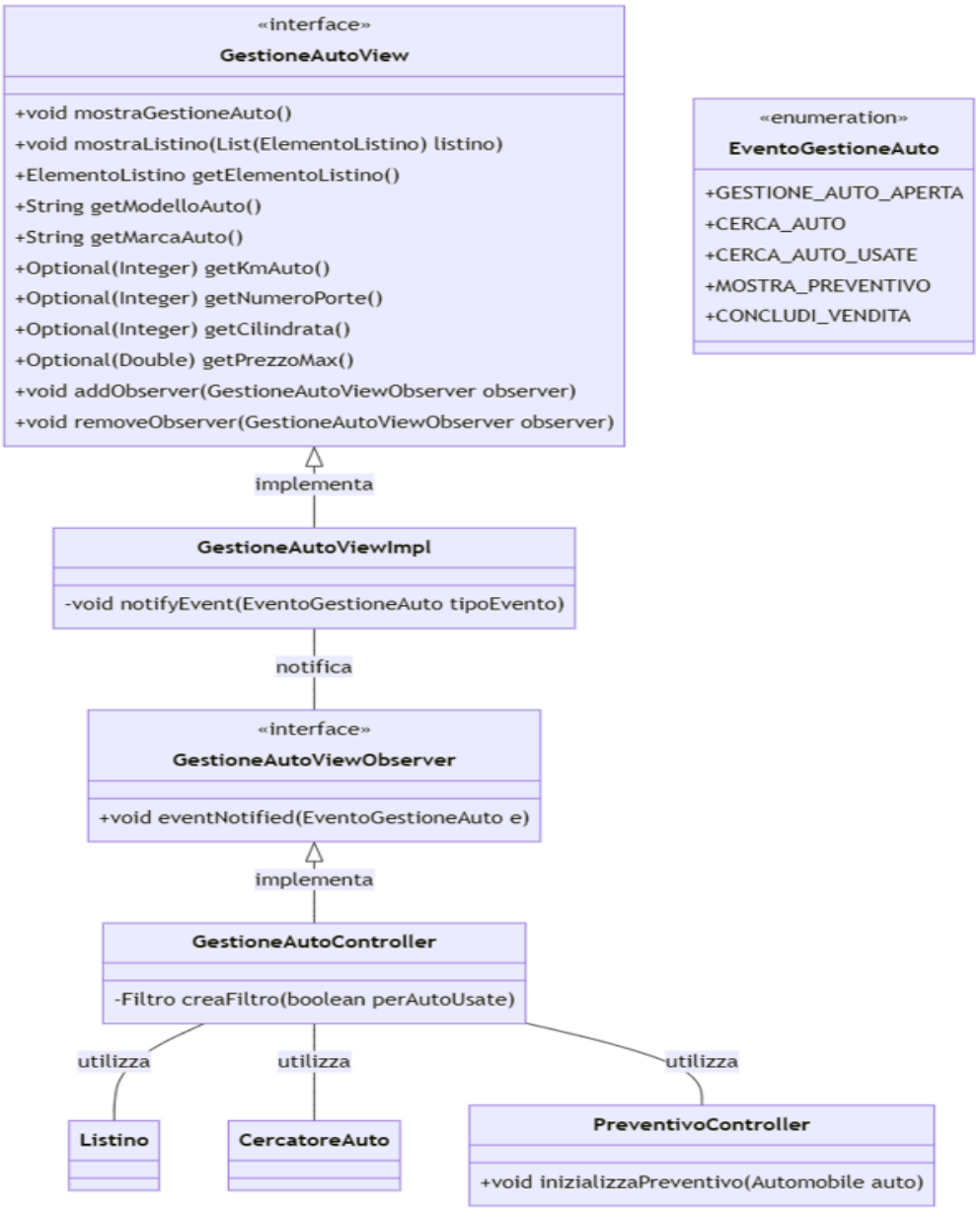


L'interfaccia `ConcessionarioView` rappresenta la vista utente principale del sistema che da cui è possibile scegliere navigare verso l'anagrafica clienti, la gestione delle auto(noleggio, leasing, riparazione, vendita). Inoltre consente di visualizzare il registro delle vendite. La navigazione avviene attraverso il pattern observer in particolare la view è in grado di lanciare eventi di tipo `EventoConcessionario`. Il controller, invece, agisce da osservatore e reagisce aprendo le diverse view del sistema.



Per quanto riguarda la gestione dell'anagrafica clienti, è stato utilizzato lo stesso approccio generale di comunicazione tra view e controller basato sul **pattern Observer.** L'interfaccia `AnagraficaClientiView` consente all'utente di gestire i clienti del concessionario, mostrando una lista di clienti, permettendo la registrazione di nuovi clienti e fornendo funzionalità di ricerca. La vista genera eventi come `EventoAnagrafica` per azioni specifiche, come l'apertura dell'anagrafica, la registrazione di un nuovo cliente o la ricerca di clienti per nome o codice fiscale.

Il controller, `AnagraficaClientiController`, è registrato come osservatore della view e quando un evento viene notificato (ad esempio, l'evento `CERCA_CLIENTI`), reagisce eseguendo l'operazione richiesta. Il controller si occupa quindi di interagire con il modello (`AnagraficaClienti`) per recuperare, aggiungere o cercare clienti, e infine aggiorna la vista con i risultati ottenuti.

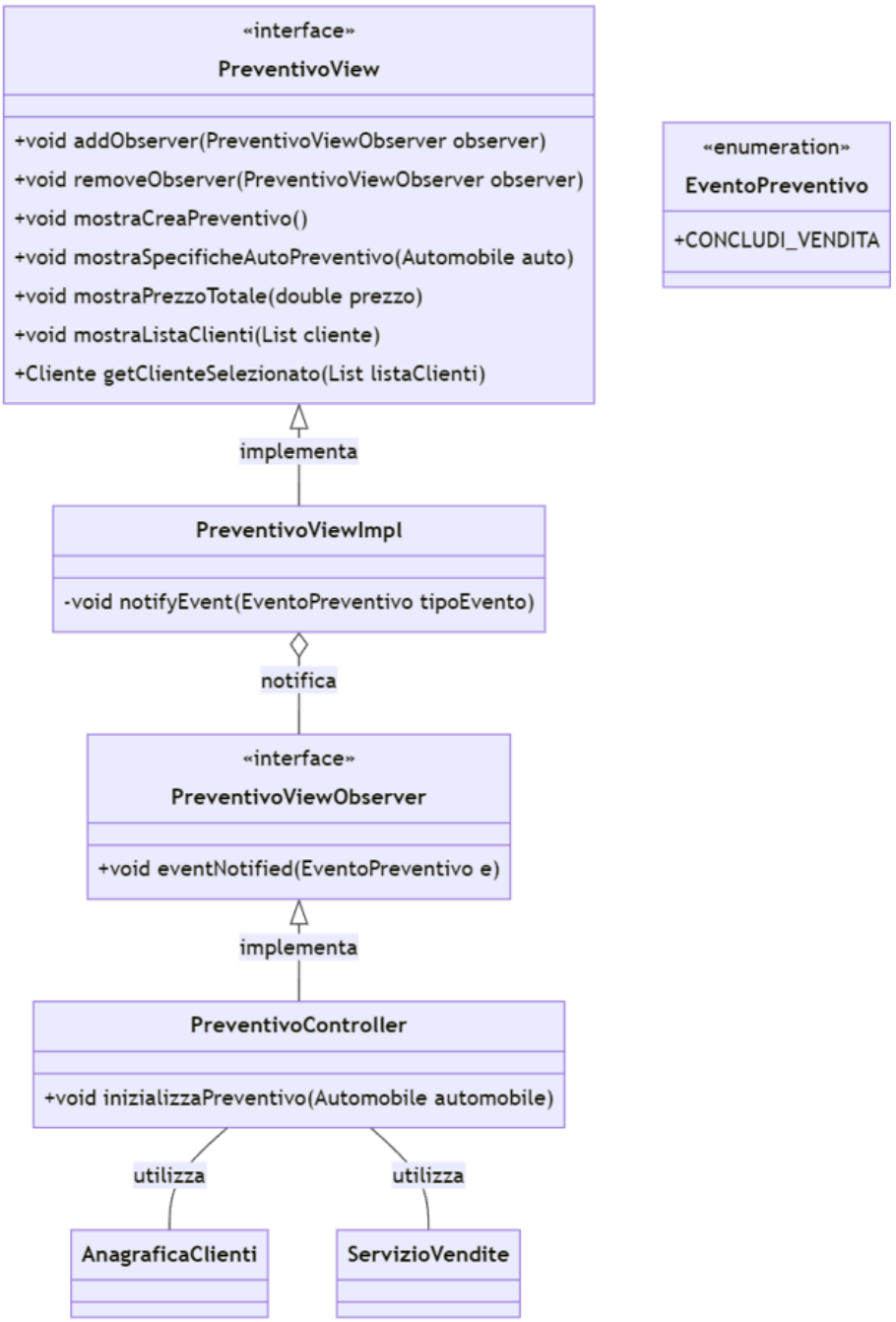


Per la gestione delle automobili, il sistema adotta anche qui lo schema basato sul **pattern Observer**. La view della gestione auto (`GestioneAutoView`) interagisce con il controller tramite eventi definiti nell'enum `EventoGestioneAuto`, che rappresentano azioni come la ricerca di auto, la richiesta di preventivi o la conclusione di una vendita.

La vista mostra un'interfaccia con un elenco di automobili disponibili per la gestione (sia nuove che usate), e offre campi di ricerca per filtrare le auto in base a vari parametri come marca, modello, chilometraggio e prezzo massimo. Quando l'utente interagisce con l'interfaccia, ad esempio cercando un'auto o visualizzando le specifiche di una vettura, la vista notifica l'evento corrispondente al controller, che a sua volta esegue l'azione richiesta.

Il controller (`GestioneAutoController`) è registrato come osservatore della view, e reagisce a questi eventi. Ad esempio, quando l'evento `CERCA_AUTO` viene notificato, il

controller costruisce un filtro basato sui parametri di ricerca inseriti dall'utente e utilizza un oggetto di dominio `CercatoreAuto` per trovare le auto che corrispondono ai criteri. Al termine della ricerca, il controller comunica alla vista la lista di auto da visualizzare. Nel caso in cui l'evento richiesto riguarda invece la visualizzazione di un preventivo (`MOSTRA_PREVENTIVO`), il controller delega la gestione del preventivo a un altro controller dedicato, `PreventivoController`.



Per la gestione dei preventivi, il sistema utilizza anche qua come in tutte le altre view il pattern Observer per garantire disaccopiamento tra la view e il controller. La view del preventivo, rappresentata da `PreventivoView`, funge da interfaccia utente attraverso la quale gli utenti possono visualizzare e gestire i preventivi. In particolare, la view consente di visualizzare i dettagli dell'auto, come marca, modello e numero di porte, ecc. Inoltre, mostra il prezzo totale dell'auto. Gli utenti possono anche selezionare un cliente da una lista dei clienti al quale vendere l'auto.

Il controller, `PreventivoController`, prepara la view con le informazioni necessarie, come le specifiche dell'auto, la lista dei clienti e il prezzo totale. Inoltre, gestisce la logica e risponde agli eventi generati dalla view. Quando l'utente interagisce con la view, come nel caso del clic su "*Concludi Vendita*", il controller delega al modello la creazione di un preventivo e la registrazione della vendita nel registro.

Capitolo 3: Sviluppo

3.1 Testing automatizzato

3.1.1 Francesco Borrelli

GeneratoreTargaTest

La classe GeneratoreTarga è responsabile della generazione di targhe automobilistiche secondo un formato standard. I test implementati per questa classe verificano l'accuratezza e l'unicità delle targhe generate:

- **testGenerateTargaLength():** Questo test verifica che la targa generata abbia una lunghezza di 7 caratteri. Un formato di lunghezza sbagliata indicherebbe un problema nella logica di generazione delle targhe.
- **testGenerateTargaFormat():** Assicura che la targa generata segua il formato corretto (LLNNNLL), dove "L" rappresenta una lettera e "N" un numero. Questo test controlla che le lettere e i numeri siano posizionati correttamente nella targa.
- **testGenerateTargaUniqueness():** Verifica che due targhe generate successivamente siano diverse. Anche se raro, questo test può fallire per coincidenza, ma è cruciale per garantire che le targhe non si ripetano.
- **testGenerateMultipleTargheUniqueness():** Testa la generazione di un numero elevato di targhe (10.000 nel test) per assicurarsi che tutte le targhe siano uniche. Utilizza un HashSet per verificare l'assenza di duplicati, garantendo che il generatore produca targhe uniche in un ampio set di dati.

OfficinaModelTest

La classe OfficinaModel gestisce le operazioni relative alle auto usate nell'officina, inclusa la loro aggiunta e rimozione dal listino. I test per questa classe verificano le funzionalità principali del modello dell'officina:

- **testAggiungiAutoUsata():** Questo test verifica che un'auto usata, creata tramite FactoryAutomobile, venga correttamente aggiunta al listino delle auto usate;Assicura che la dimensione del listino aumenti e che l'auto sia effettivamente inclusa;

- **testRimuoviAutoUsata():** Controlla che un'auto usata possa essere rimossa dal listino. Il test prevede l'aggiunta di due auto e la successiva rimozione di una di esse, assicurandosi che la rimozione avvenga correttamente e che l'auto rimanente sia quella prevista.

Questa sezione si concentra sui test per il generatore di targhe e per il modello dell'officina, evidenziando le verifiche critiche implementate per garantire la correttezza e l'affidabilità del sistema

3.1.2 Gioele Pula

AutomobileNoleggioTest

La classe AutomobileNoleggio rappresenta un'automobile nel contesto del noleggio. Ho implementato i seguenti test:

- **testGetModello()**: Verifica che il metodo getModello() restituisca correttamente il modello dell'auto. Ad esempio, se il modello è "Kuga", il test controllerà che venga restituito "Kuga".
- **testGetMarca()**: Controlla che il metodo getMarca() restituisca la marca corretta dell'auto, come "Ford".
- **testGetNumeroPorte()**: Assicura che il metodo getNumeroPorte() restituisca il numero corretto di porte, per esempio 5.
- **testGetCilindrata()**: Verifica che il metodo getCilindrata() restituisca la cilindrata corretta del motore, ad esempio 1600 cc.
- **testGetCavalli()**: Controlla che il metodo getCavalli() restituisca la potenza del motore espressa in cavalli, come 120.
- **testGetDisponibilitàAuto()**: Assicura che il metodo getDisponibilitàAuto() restituisca lo stato di disponibilità dell'auto, che dovrebbe essere "Disponibile".
- **testSetCostoNoleggio()**: Verifica che il metodo setCostoNoleggio() imposti correttamente il costo di noleggio e che il metodo getCostoNoleggio() restituisca il valore impostato, ad esempio 50.0.
- **testGetStato()**: Controlla che il metodo getStato() restituisca il valore corretto per lo stato dell'auto, che dovrebbe essere "Disponibile" se l'auto è disponibile.

FactoryAutomobiliNoleggioTest

La classe FactoryAutomobiliNoleggio è responsabile della creazione di oggetti AutomobileNoleggio. I test implementati sono i seguenti:

- **testCreaAutoRandom()**: Testa il metodo creaAutoRandom() che crea una lista di 10 automobili. Verifica che ogni automobile nella lista abbia attributi validi, come marca, modello, numero di porte, cilindrata e cavalli nel range specificato, e che tutte le auto siano disponibili.
- **Metodi di supporto**:
    - **isMarcaValida(String marca)**: Verifica se una marca è valida confrontandola con un elenco predefinito di marche.
    - **isModelloValido(String modello, String marca)**: Verifica se un modello è valido per una determinata marca confrontandolo con un elenco predefinito di modelli per ciascuna marca.

AssicurazioneTest

La classe Assicurazione calcola i costi delle assicurazioni. I test implementati includono:

- **testCalcolaCostoAssicurazioneCoperturaBassa()**: Verifica che il costo calcolato per una copertura bassa sia corretto e compreso tra i valori minimi e massimi attesi per una data durata del noleggio.
- **testCalcolaCostoAssicurazioneCoperturaMedia()**: Assicura che il costo per una copertura media sia calcolato correttamente e che rientri nei range previsti per la durata del noleggio specificata.
- **testCalcolaCostoAssicurazioneCoperturaAlta()**: Controlla che il costo per una copertura alta sia calcolato correttamente e che sia compreso tra i valori minimi e massimi previsti per la durata del noleggio.
- **testCalcolaCostoAssicurazioneCoperturaInesistente()**: Verifica che il costo per una copertura inesistente sia pari a zero, garantendo che l'input non valido non produca un costo.

AutonoleggioTest

La classe Autonoleggio gestisce il noleggio delle automobili. I test inclusi sono:

- **testNoleggiaAuto()**: Verifica che, dopo il noleggio di un'auto, lo stato dell'auto venga aggiornato correttamente a "NonDisponibile". Questo test include un'attesa breve per gestire la sincronizzazione dei thread, se necessario.
- **testGetAutomobili()**: Controlla che la lista delle automobili sia vuota inizialmente quando viene creata un'istanza di Autonoleggio.
- **testAggiungiAuto()**: Verifica che un'auto aggiunta alla lista di automobili di Autonoleggio venga effettivamente inclusa nella lista e che possa essere recuperata successivamente.

LeasingTest

La classe Leasing gestisce i dettagli del leasing delle auto. I test includono:

- **testGetAutoSelezionata()**: Verifica che il metodo getAutoSelezionata() restituisca correttamente il nome dell'auto selezionata per il leasing, come "Fiat Panda".
- **testGetDurataContratto()**: Controlla che il metodo getDurataContratto() restituisca la durata del contratto in mesi, ad esempio 36 mesi.
- **testGetPrezzoMensile()**: Verifica che il metodo getPrezzoMensile() restituisca il prezzo mensile del leasing, per esempio 199.99.
- **testGetLimiteChilometrico()**: Assicura che il metodo getLimiteChilometrico() restituisca il limite chilometrico del leasing, ad esempio 15000 km.

3.1.3 Alessandro Petreti

Nello sviluppo del software, una parte cruciale è rappresentata dai test, che garantiscono la robustezza del sistema. Di seguito sono riportati i test effettuati per il modello, che coprono vari aspetti e funzionalità. La suddivisione dei test segue la struttura definita nella fase di design di dettaglio. I test sono stati eseguiti per confermare che le componenti interagiscono correttamente.Di seguito sono riportati i test svolti utilizzando JUnit

- **AutomobileTest** verifica che tutte le proprietà dell'auto, come marca, modello, numero di porte, cilindrata, cavalli, e stato della macchina, siano gestite e restituite correttamente.
- **FactoryAutomobileTest** i test assicurano che la factory generi correttamente automobili con le proprietà desiderate e gestisca eventuali errori nella creazione
- **AnagraficaTest** i test controllano che i clienti possano essere aggiunti, rimossi e recuperati correttamente dal sistema.
- **ClienteTest** assicurandosi che le operazioni sui dati del cliente, come la modifica e il recupero delle informazioni, avvengano senza problemi. Include la verifica delle proprietà e dei metodi della classe `Cliente`.
- **Factory Cliente Test** verifica che la factory generi correttamente oggetti `Cliente` con le caratteristiche appropriate e gestisca eventuali situazioni particolari.
- **GeneratorePrezzoTest** i test assicurano che i prezzi siano calcolati correttamente in base ai criteri specificati, come marca, modello, e condizioni dell'auto
- **PreventivoTest** Verifica che le informazioni del preventivo, come auto selezionata, cliente e prezzo, siano gestite correttamente e che i metodi associati funzionino come previsto.
- **RegistroVenditeTest** i test assicurano che le vendite possano essere registrate, visualizzate e recuperate correttamente dal sistema**.**
- **ServizioVenditeTest** i test verificano che il servizio possa generare preventivi, completare vendite e gestire le operazioni relative in modo accurato.
- **CercatoreAutoTest** i test assicurano che il cercatore possa trovare auto in base ai criteri di ricerca specificati e restituire i risultati corretti.
- **FiltroTest** i test verificano che il filtro funzioni correttamente per restringere l'elenco delle auto in base ai parametri scelti, come prezzo, chilometraggio e altre caratteristiche.

3.2 Metodologia di lavoro

Sviluppando il progetto si è cercato di mantenere più indipendenza possibile dal punto di vista del lavoro svolto, pur sempre confrontandoci quotidianamente per aggiornare l'un l'altro sui progressi fatti riguardo le rispettive sezioni di lavoro. La fase iniziale è stata di analisi, durante la quale abbiamo lavorato tutti e tre a stretto contatto per definire attraverso grafici UML ed idee quella che doveva essere la nostra architettura. Ci siamo soffermati per molte ore su questa fase. Nella definizione del progetto collaborazione e confronto sono stati fondamentali. Proseguendo con lo sviluppo ogni componente del gruppo ha lavorato con indipendenza su branch diversi alla propria parte modificando il progetto e parti di altri all'occorrenza. La progettazione è stata basata sullo sviluppo di pattern, quali : MVC (model, view, controller), Observer, Factory. In fase di sviluppo abbiamo pensato e architettato uno sviluppo il più lineare e modulare possibile.
Per questo motivo abbiamo deciso di utilizzare i pattern sopra elencati i quali si sono rivelati di fondamentale importanza per lo sviluppo del progetto.
Una volta definita la base architetturale del progetto, abbiamo sviluppato l'intero progetto sulla base dei Pattern scelti.

3.2.1 Francesco Borrelli

Sviluppo del modulo Officina e funzioni ausiliarie alla classe automobile.
Focus su interfaccia grafica (java Swing)e delle sezioni collegate seguendo il pattern MVC.
L'idea di lavoro è quella di integrare nuove parti integrandole alla struttura del progetto stesso.

3.2.2 Gioele Pula

Sviluppo delle sezioni di noleggio e leasing con classi annesse, in particolare l'implementazione si basa sullo sviluppo dei metodi che implementano il noleggio delle auto e il leasing. Sviluppo della classe assicurazione e relative funzionalità. Focus sull'interfaccia grafica (java swing) delle due sezioni seguendo il pattern MVC, con annessa implementazione dei relativi controller.

3.2.3 Alessandro Petreti

Sviluppo dei moduli cliente, automobile, listino, servizio vendite e ricerca auto con le relative classi. Realizzazione delle view preventivo, anagrafica, concessionario, auto e dei rispettivi contoller preventivo controller, Concessionario controller, GestioneAuto controller, anagrafica Clienti contoller.

3.3 Note di sviluppo

3.3.1 Francesco Borrelli

Optional

- Gestione dei Valori Nulli: Gli Optional sono stati utilizzati per gestire i potenziali valori nulli nelle operazioni di ricerca e manipolazione delle automobili nel listino dell'officina. Questo approccio ha migliorato la robustezza del codice, evitando eccezioni dovute a valori nulli e fornendo un modo chiaro e sicuro per gestire l'assenza di dati. Ad esempio, nel OfficinaModel, Optional è impiegato per gestire i casi in cui una ricerca per marca o modello potrebbe non restituire risultati, evitando così la gestione di null pointer exceptions.

Lambdas e Stream

- Elaborazione delle Collezioni: Le espressioni lambda e le API Stream sono state utilizzate per elaborare le collezioni di dati, come l'elenco delle automobili disponibili e le operazioni sul listino. Per esempio, nel OfficinaView, le lambdas sono state usate per gestire le azioni dell'utente e per applicare filtri sui dati mostrati nella GUI.

JUnit:

- Test Unitari: La libreria JUnit è stata fondamentale per assicurare la qualità del codice attraverso test unitari. Sono stati creati test specifici per le classi OfficinaModel, e GeneratoreTarga per verificare la correttezza delle operazioni di aggiunta, rimozione e visualizzazione delle automobili.

3.3.2 Gioele Pula

Optional: Largamente usati per gestire potenziali valori nulli nelle operazioni di ricerca, gestione dei veicoli e dei contratti di leasing e autonoleggio. Gli optional hanno migliorato la robustezza del codice, evitando eccezioni dovute a valori nulli e offrendo un modo chiaro per gestire l'assenza di dati.
Lambdas e Stream: Utilizzate per elaborare le collezioni di dati, come l'elenco dei veicoli disponibili o la lista dei contratti di leasing attivi. Questi strumenti hanno reso possibile implementare operazioni di filtro, mappatura e aggregazione in modo conciso e leggibile.
Junit: Librerie di testing utilizzate per assicurare la qualità del codice. Junit è stato essenziale per i test unitari.

3.3.3 Alessandro Petreti

Optional: Utilizzo di optional per gestire in modo sicuro valori potenzialmente nulli, specialmente durante la ricerca di auto basate su filtri opzionali (ad esempio numero di porte, prezzo massimo).

Stream: utilizzo degli Stream per operazioni di ricerca e filtraggio su liste di oggetti (ad esempio per cercare auto nel listino in base a criteri specifici).

Capitolo 4: Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Francesco Borrelli:

Lavorare in gruppo si è rivelato più complicato del previsto, non avendo mai lavorato in un gruppo di più di due persone e non avendo mai avuto esperienza di programmazione, eccetto quelle accademiche. Il lavoro con i miei colleghi a parere mio è stato di qualità.
Il tempo impiegato nel costruire il progetto mi ha aiutato a comprendere e a fissare i concetti sostenuti a lezione. Ha migliorato la mia capacità di problem solving e di lavorare in gruppo inseguendo un obiettivo comune. Il più grande vantaggio che ho ottenuto è stato quello di poter collaborare con i miei colleghi e, conseguentemente, osservare il loro lavoro e apprendere nuove tecniche di programmazione e di ragionamento. Il progetto mi ha dato modo di poter capire quali sono i miei limiti e le mie lacune da programmatore.
Una volta riconosciuto che le difficoltà costituiscono un elemento fondamentale del processo di apprendimento, ho sviluppato una maggiore fiducia in me stesso, il che mi ha reso più coinvolto nelle dinamiche del gruppo, contribuendo così a creare un team coeso e collaborativo. Sono contento della presenza mostrata dai miei colleghi nei momenti di difficoltà e ora che ho acquisito una solida base di programmazione, il mio obiettivo è lavorare su più fronti, al fine di migliorare ulteriormente sia le mie competenze tecniche che personali. Questo percorso mi fornirà, inoltre, preziosi spunti di riflessione su come relazionarsi al meglio con i membri del mio attuale e futuro team. Ritengo che questo progetto mi abbia permesso di crescere non solo a livello personale, ma soprattutto come programmatore.

4.1.2 Gioele Pula:

Questo lavoro di gruppo è stata una buona opportunità per mettere in pratica le conoscenze apprese durante il corso, non è stato però un progetto semplice dal punto di vista implementativo. Sin da subito ho lavorato con estrema cura e dedizione allo sviluppo delle varie classi in maniera tale che nulla fosse tralasciato, la procedura non è stata però dal mio punto di vista semplice in quanto non avevo basi pregresse di programmazione in java, ne tantomeno su lavori di gruppo di questo tipo. Sono molto soddisfatto del lavoro svolto da me e dal mio team, però sicuramente a mente lucida e a progetto finito posso dire che è stato un progetto molto complesso, sia per la gestione delle tempistiche sia per la suddivisione dei vari ruoli.
Sin da subito mi sono occupato di tutta la parte che riguarda autonoleggio e leasing in maniera tale da poter dare una buona mano ai miei compagni, cercando sempre di produrre codice chiaro e correttamente strutturato all'interno dei package per riuscire così una volta finito a unire tutto nel progetto finale in maniera più semplice.
L'utilizzo di GitHub è stato fondamentale per la suddivisione, condivisione dei vari codici e del lavoro da svolgere, una volta finito siamo stati in grado infatti attraverso le pull request di effettuare il merge del codice nel Master, senza questo tool sarebbe stato molto più difficile riuscire a scambiarsi codice, unire e modificare le varie parti.

Questa attività è stata sicuramente importante per la mia crescita e per farmi imparare abilità nella programmazione e nel come si lavora all'interno di un gruppo gettando le basi per quelli che potranno essere progetti futuri ancora più complessi e meglio implementati. Ringrazio i miei compagni che sono stati sempre presenti per avermi dato tutti gli aiuti necessari e ad avermi fornito la possibilità di poter lavorare in un team ben affiatato e senza particolari problemi.

4.1.3 Alessandro Petreti:

Questo progetto di gruppo ha rappresentato una preziosa opportunità per applicare le conoscenze acquisite durante il corso.Questa esperienza mi ha aiutato a migliorare le mie competenze di programmazione e a comprendere meglio il lavoro di squadra. È stata una preparazione importante per affrontare progetti futuri, che saranno sicuramente più complessi e meglio strutturati.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Francesco Borrelli:

Relazionarsi con due colleghi si è rivelato impegnativo, principalmente a causa di problematiche di natura personale, come l'astrazione dei concetti di programmazione e la comprensione di aspetti avanzati del linguaggio; la difficoltà maggiore è stata l'utilizzo di Git. Presupponendo eventuali imprecisioni nella configurazione del software, ho riscontrato notevoli difficoltà, ad esempio nella gestione dei branch, i push e la sincronizzazione delle repository, il che ha comportato disagi sia per me che per il mio gruppo. Questo ha obbligato i membri del team ad adattarsi alle mie esigenze, cui sono grato per la loro pazienza, disponibilità e impegno nel tentativo di risolvere le mie problematiche, sebbene talvolta senza successo.
Il corso mi ha dato una grande mano a capire come si compone lo sviluppo software e di come sia un mondo così meticoloso e in continuo sviluppo. L'insegnamento risulta lineare e strutturato con adeguate risorse all'apprendimento e supporto allo studente. Il docente inoltre mette a disposizione del materiale per eventuali approfondimenti.

4.2.1 Gioele Pula:

Il corso di Programmazione e Modellazione a Oggetti si è rivelato senza dubbio uno dei corsi più stimolanti e formativi che abbia seguito finora. Tuttavia, devo ammettere che è stato

anche uno dei più impegnativi, soprattutto a causa delle mie limitate conoscenze di programmazione in Java all'inizio del percorso. Fin dai primi incontri, ho cercato di seguire attentamente le spiegazioni fornite e di completare gli esercizi proposti, anche se in alcune occasioni ho incontrato delle difficoltà nel restare al passo con il ritmo delle lezioni. Consapevole di queste difficoltà, ho deciso di dedicare una quantità significativa di tempo allo studio individuale, approfondendo ogni concetto in modo meticoloso. Questo approccio mi ha permesso di colmare alcune lacune e di acquisire una comprensione più solida dei principi della programmazione orientata agli oggetti, senza trascurare nessun aspetto fondamentale.
Le lezioni in laboratorio si sono dimostrate particolarmente preziose. Attraverso queste sessioni pratiche, ho avuto la possibilità di applicare in maniera concreta ciò che veniva spiegato in aula. Il lavoro in laboratorio mi ha permesso di sviluppare le competenze necessarie per affrontare con sicurezza il nostro progetto finale, facilitando il passaggio dalla teoria alla pratica.
Uno degli aspetti più interessanti del corso è stato l'introduzione e l'utilizzo dei Design Pattern. Questi modelli progettuali, oltre a migliorare la struttura del mio codice, hanno reso il processo di sviluppo più efficiente. L'adozione dei Pattern ha contribuito in modo decisivo a produrre un codice non solo pulito e leggibile, ma anche facilmente modificabile e scalabile. Questo aspetto è stato particolarmente importante per gestire la complessità crescente del progetto, garantendo al contempo una maggiore flessibilità in caso di necessarie modifiche future.
In conclusione, nonostante le sfide iniziali, ritengo che il corso mi abbia fornito gli strumenti indispensabili per affrontare problemi di programmazione complessi con un approccio più strutturato e professionale. L'impegno individuale, unito al supporto delle lezioni in laboratorio e all'apprendimento dei Pattern, mi ha permesso di acquisire una maggiore consapevolezza delle mie capacità di programmatore, oltre a una comprensione più profonda dell'importanza di un codice ben progettato e mantenibile.

4.2.3 Alessandro Petreti:

La principale difficoltà è stata gestire l'integrazione tra il lavoro individuale e quello di gruppo. La mancanza di esperienza nella programmazione collaborativa ha rappresentato una sfida significativa, ma anche un'opportunità per mettere in gioco e affinare le mie competenze di programmatore.

Un aspetto particolarmente interessante del corso è stato scoprire il potenziale della programmazione orientata agli oggetti. Ho avuto l'opportunità di esplorare le sue applicazioni e di comprendere come questa metodologia possa facilitare la progettazione e la gestione di sistemi complessi.