UNIVERSITA' DEGLI STUDI DI NAPOLI
"PARTHENOPE"

FACOLTA' DI SCIENZE E TECNOLOGIE

CORSO DI LAUREA IN INFORMATICA

SMART CITY

DOCENTE
Angelo Ciaramella
Raffaele Montella

CANDIDATO
Plidher Luna
MATRICOLA
0124001709

ANNO ACCADEMICO 2020/2021

Sommario

Introduzione

Traccia - Monitoraggio ambientale

Si vuole sviluppare un sistema per la gestione di una *Smart City*.
Una *Smart City* comprende un insieme di strategie di pianificazione urbanistica al fine di migliorare la qualità della vita e soddisfare le esigenze di cittadini, imprese ed istituzioni.
Si suppone di gestire un sistema di monitoraggio ambientale centralizzato. In una città, per ogni strada, sono situate diverse centraline contenenti dei sensori di monitoraggio che permettono di registrare i livelli di tre parametri: inquinamento dell'aria, la temperatura e il numero di autoveicoli che transitano.
Per ogni parametro l'amministratore del sistema fissa una soglia di guardia.
Il sistema di monitoraggio si può trovare in questi tre stati:

- codice verde - Se tutti e tre i parametri sono sottosoglia
- codice giallo - Se i primi due parametri sono sopra soglia
- codice rosso - Se tutti i parametri sono sopra soglia

Nel caso si verifichi il codice rosso si può applicare una delle seguenti strategie:

- consentire il traffico solo a targhe alterne. Un dispositivo controlla automaticamente le vetture e procede con l'invio di una segnalazione alla polizia locale in caso di infrazione.
- il flusso del traffico viene inviato su un altro percorso.

Il sistema permette, inoltre, di inserire nuovi sensori alla rete e di fare il grafico dei parametri per un periodo fissato.

Requisiti

Il progetto è stato sviluppato secondo il rispetto dei principi della programmazione SOLID:

1. Single Responsibility Principle (SRP)
2. Open/Closed Principle (OCP)
3. Liskov Substitution Principle (LSP)
4. Interface Segregation Principle (ISP)
5. Dependency Inversion Principle (DIP)

È stato sviluppando usando il linguaggio Java, ed inserendo opportuni commenti per la comprensione del codice e per la generazione del Javadoc del progetto.

Si è utilizzato un database SQLite per la gestione dei dati persistenti.

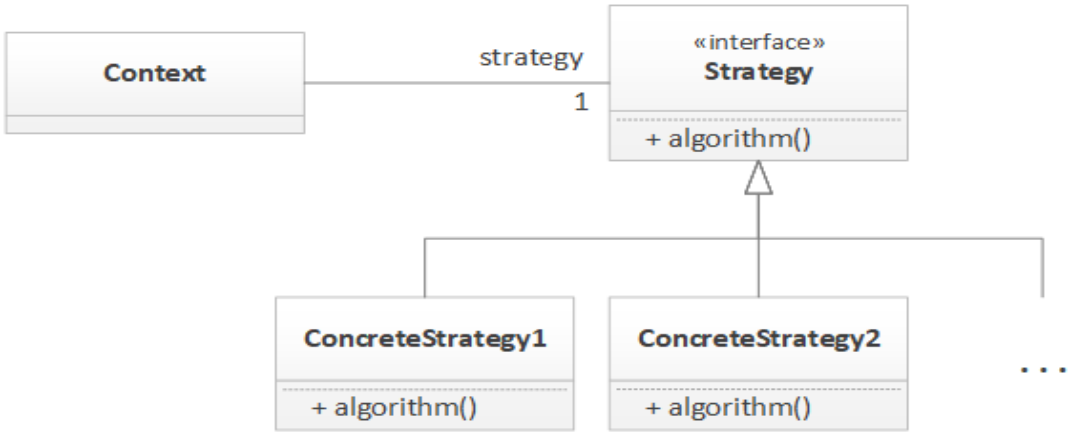
Pattern utilizzati

Sono stati utilizzati diversi Design Pattern nel progetto, essi sono:

- Strategy
- Observer
- Data Access Object (DAO)

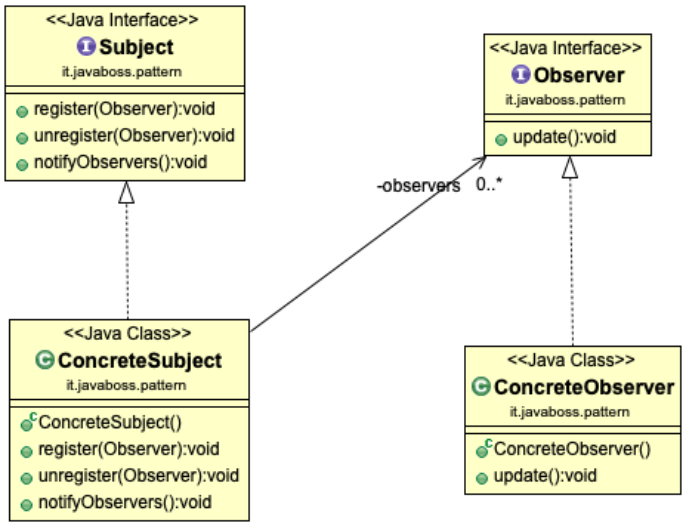
Strategy

Definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. L'algoritmo cambia indipendentemente dai client che lo usano.



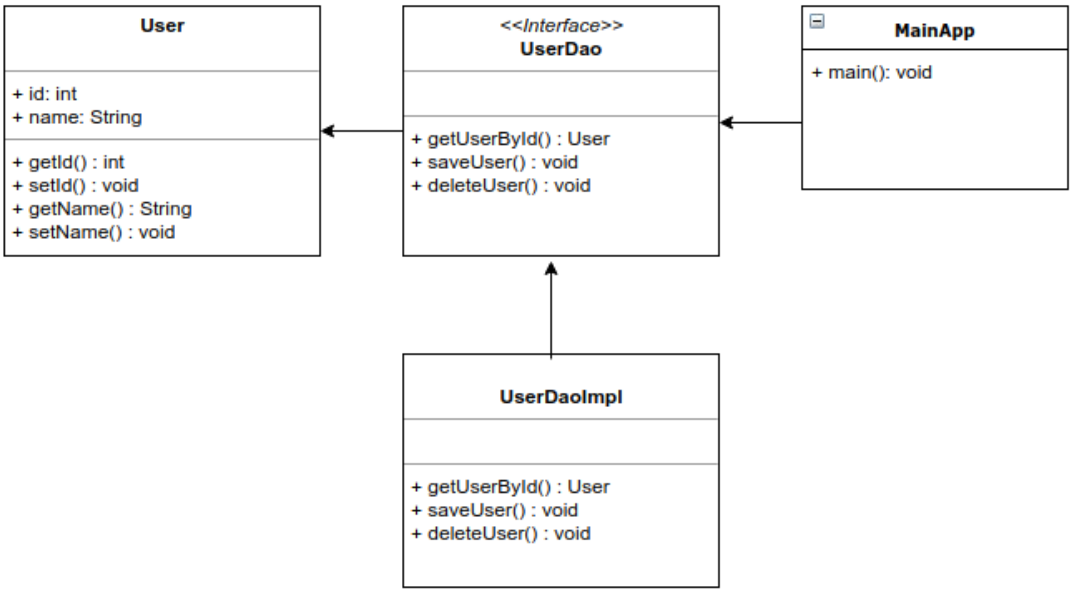
Observer

Definisce una dipendenza una a molti tra oggetti, tale che se un oggetto cambia stato, tutte le sue dipendenze sono notificate e aggiornate automaticamente.



Data Access Object (DAO)

Il ***DAO*** (*Data Access Object*) è un pattern architetturale per la gestione della persistenza: si tratta fondamentalmente di una classe con relativi metodi che rappresenta un'entità tabellare di un RDBMS, usata principalmente per stratificare e isolare l'accesso ad una tabella tramite query (poste all'interno dei metodi della classe) ovvero al data layer da parte della business logic creando un maggiore livello di astrazione ed una più facile manutenibilità. I metodi del DAO con le rispettive query dentro verranno così richiamati dalle classi della business logic.



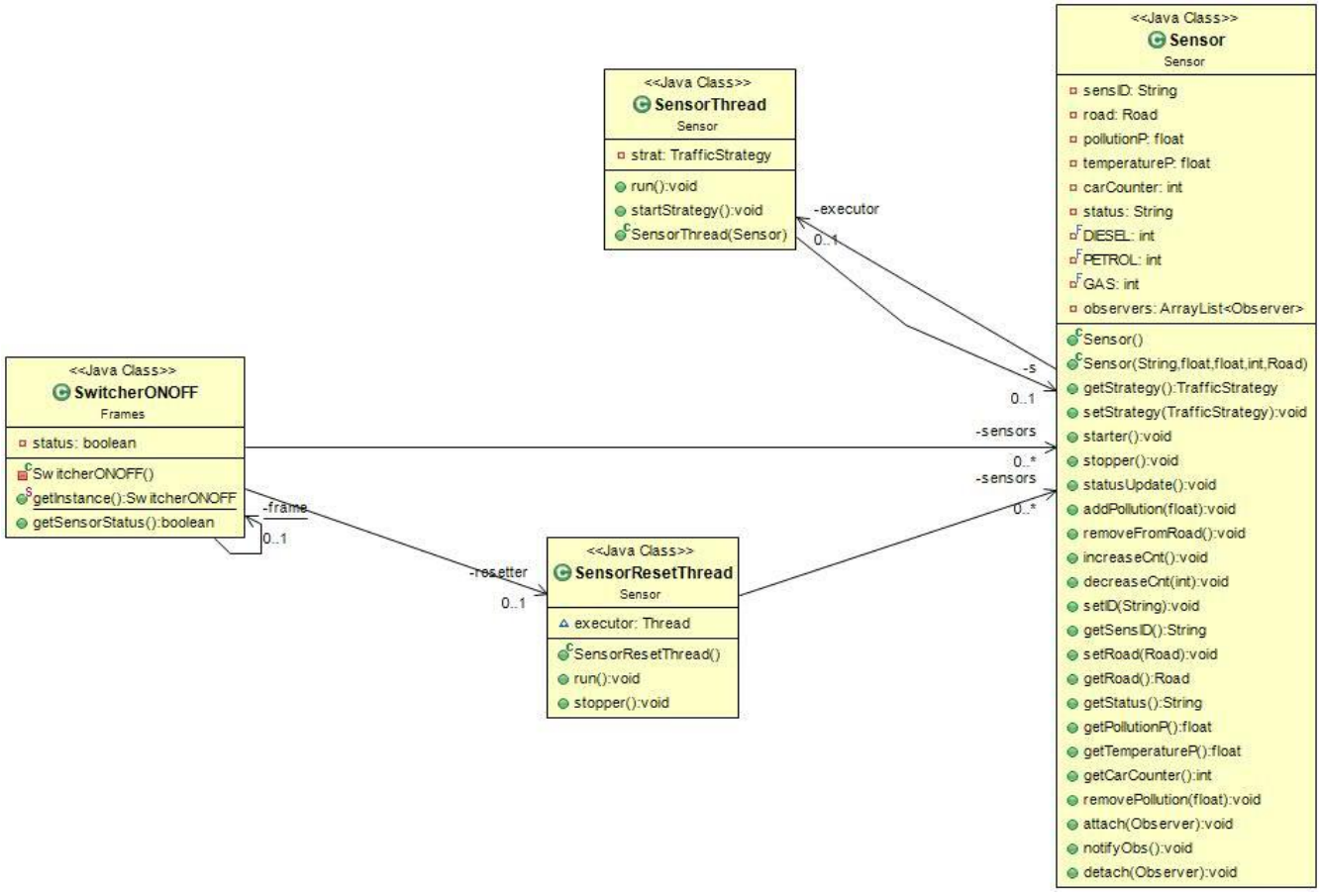
Diagrammi UML

Si è deciso di generare dei diagrammi delle classi in base alle funzionalità più importanti del progetto, esse sono:

- simulazione dei sensori
- monitoraggio dei sensori
- aggiornamento grafico delle strade
- aggiornamento grafico dei sensori
- applicazione di strategie
- accesso ai dati del DB

Diagramma delle classi per la simulazione dei sensori

Qui si mostra la relazione tra le classi per simulare il comportamento dei sensori.
Il loro comportamento è messo in atto da un Thread "*SensorThread*" che ogni sensore possiede.
Vengono attivati e/o disattivati da un pulsante di attivazione/disattivazione presente nel frame "*SwitcherONOFF*" insieme ad essi viene attivato un Thread "*SensorResetThread*" che ha la funzione di decrementare i parametri con il passare del tempo.



Classe del SensorThread che simula il comportamento
https://github.com/PlidherLuna/SmartCity/blob/main/src/Sensor/SensorThread.java

Diagramma delle classi per il monitoraggio dei sensori

Qui mostriamo la relazione tra le classi usando il Design Pattern "*Observer*" dove al cambiamento di stato di un sensore (*Observable*) verrà aggiornato se necessario i parametri generali della strada (*Observer*).

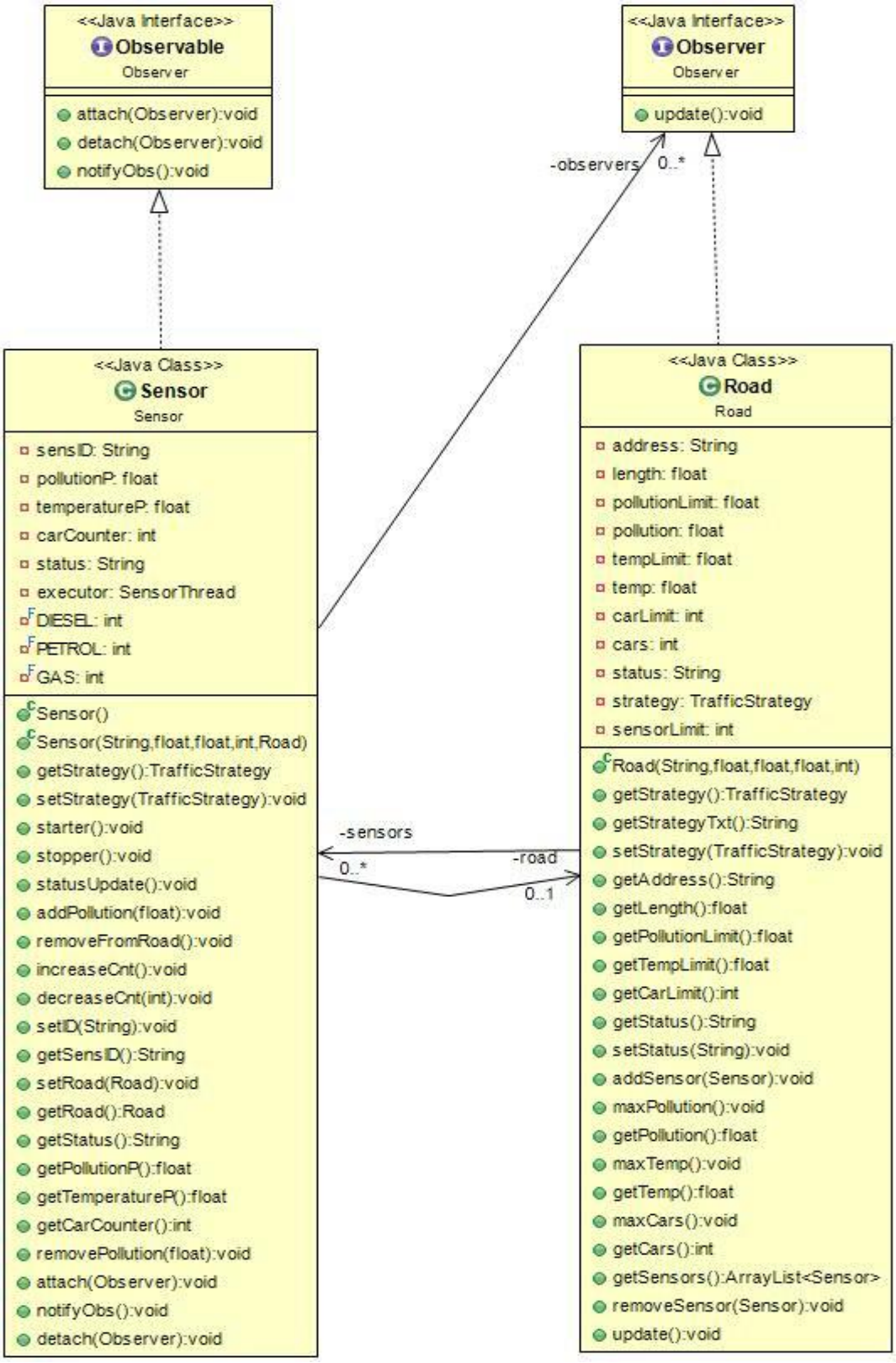
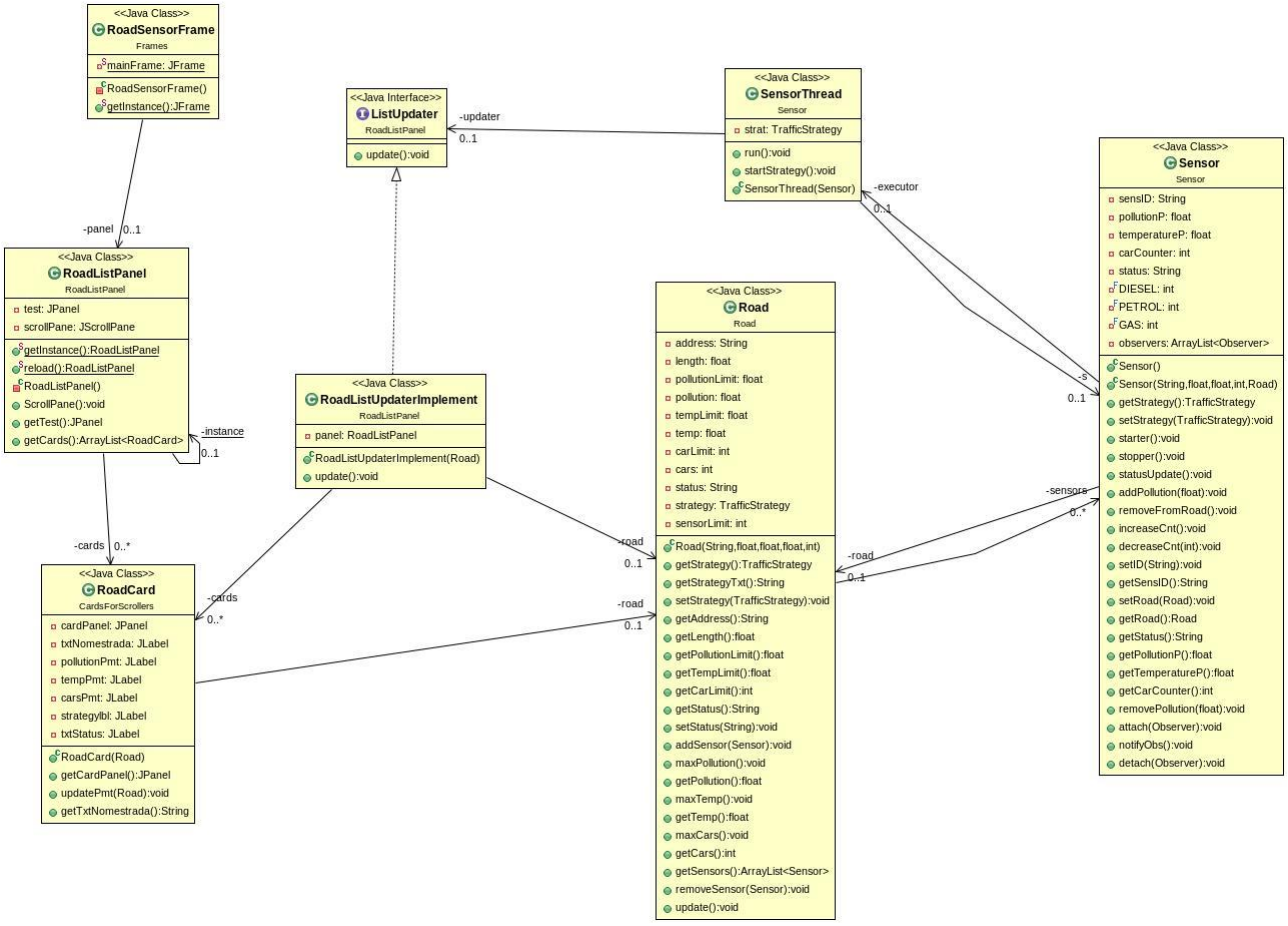


Diagramma delle classi per l'aggiornamento grafico delle strade

Relazioni tra le classi quando avviene un cambiamento di status di una strada (inquinamento, temperatura, contatore auto o status codice) dovuto al massimo valore di ogni parametro tra i sensori.
I sensori avviano il metodo "*update()*" dell'interfaccia "*ListUpdater*" istanziata come "*RoadListUpdaterImplement*" che avvia i metodi di aggiornamento dei Panel di ciascuna strada (*RoadCard*), aggiornando così il "*RoadListPanel*".

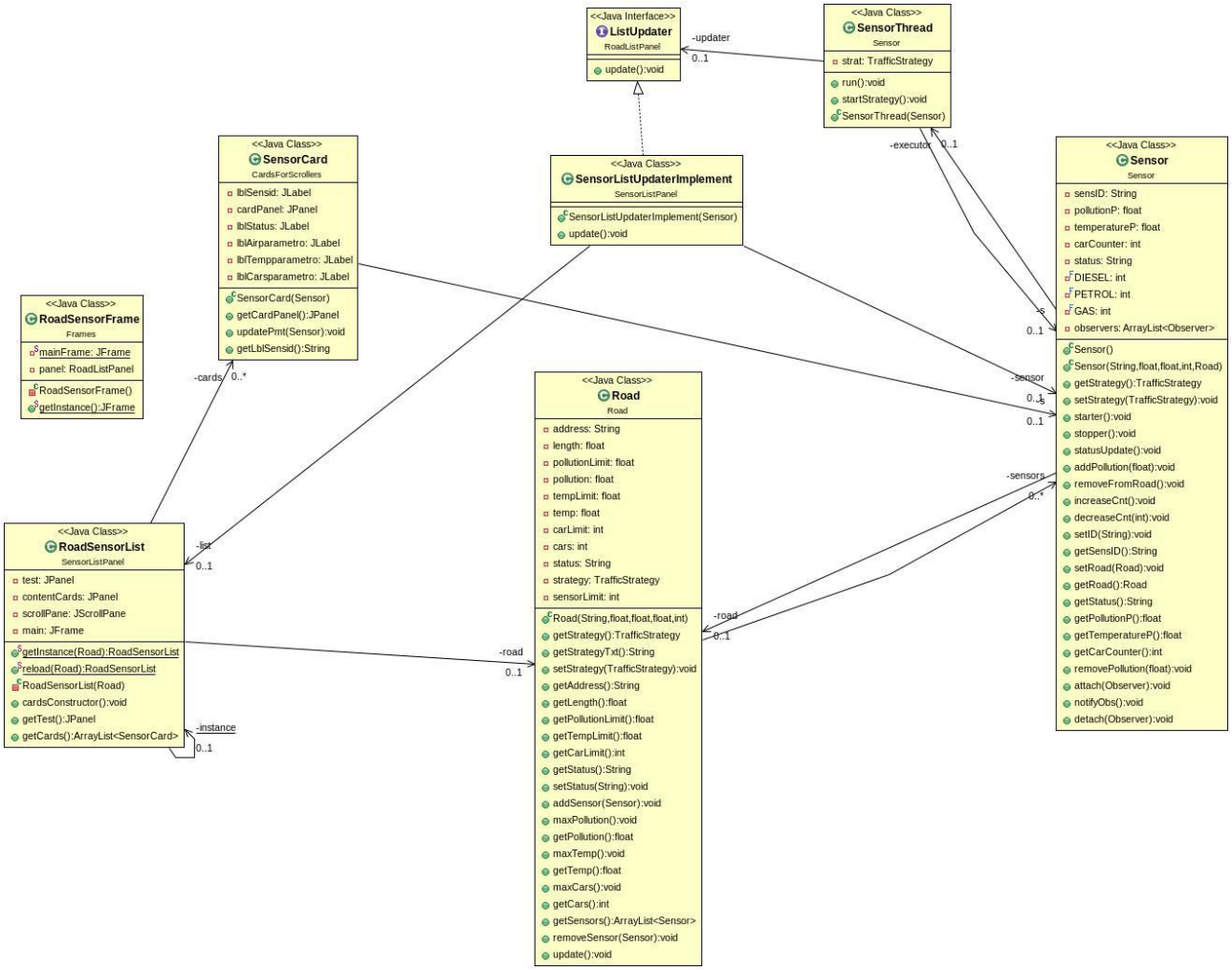


RoadListPanel con updater e metodi di aggiornamento
https://github.com/PlidherLuna/SmartCity/tree/main/src/RoadListPanel

Classe roadCard che aggiorna se stessa
https://github.com/PlidherLuna/SmartCity/blob/main/src/CardsForScrollers/RoadCard.java

Diagramma delle classi per l'aggiornamento grafico dei sensori

Come spiegato anche nel diagramma di aggiornamento delle strade qui il cambiamento non è dovuto dal cambiare dei massimi valori tra i sensori ma al cambiamento generale.
Quando un sensore cambia parametro verrà chiamato il metodo "*update()*" dell'interfaccia "*ListUpdater*" istanziata come "*SensorListUpdater*" che chiamerà i metodi di aggiornamento dei Panel di ogni sensore (*SensorCard*), aggiornando così il "*RoadSensorList*".

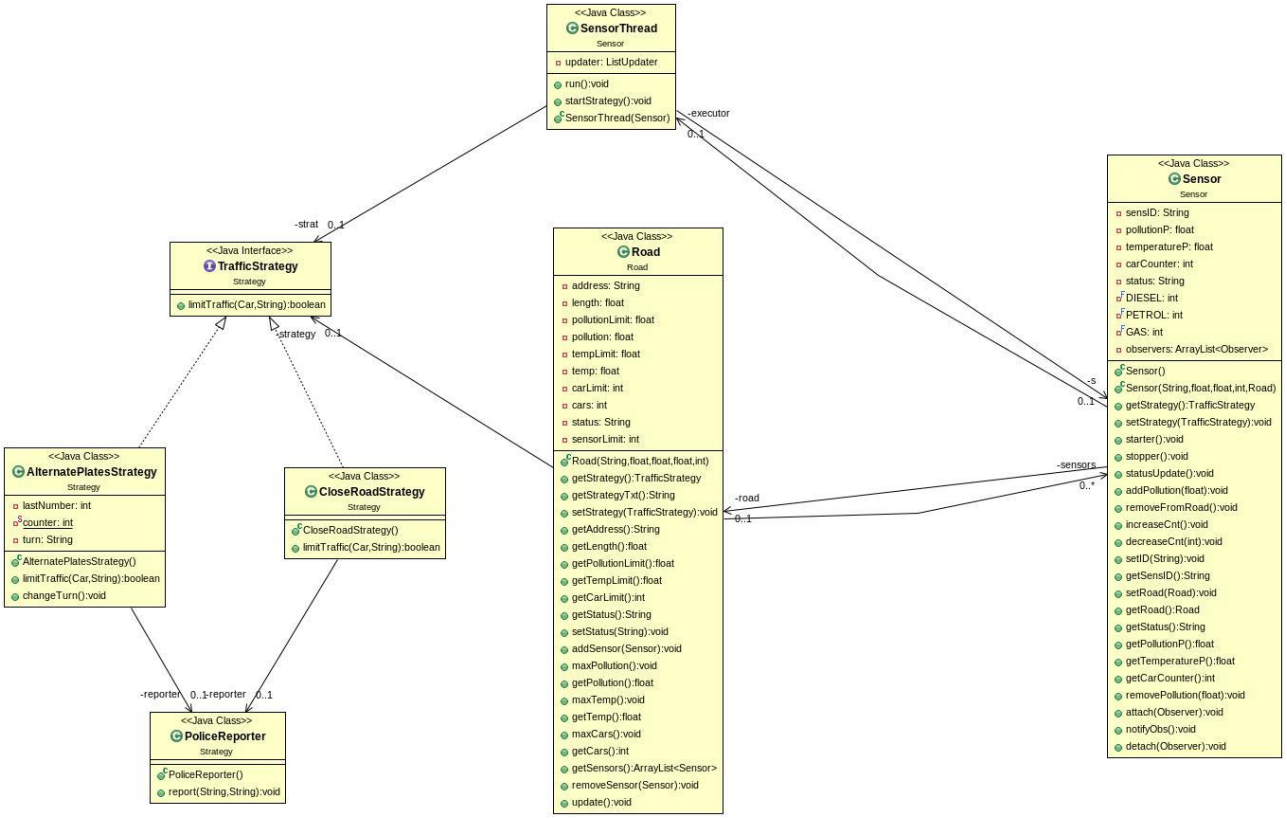


Pacchetto con panel dei sensori e relativo updater
https://github.com/PlidherLuna/SmartCity/tree/main/src/SensorListPanel

Classe SensorCard che aggiorna se stessa
https://github.com/PlidherLuna/SmartCity/blob/main/src/CardsForScrollers/SensorCard.java

Diagramma delle classi per l’applicazione di strategie

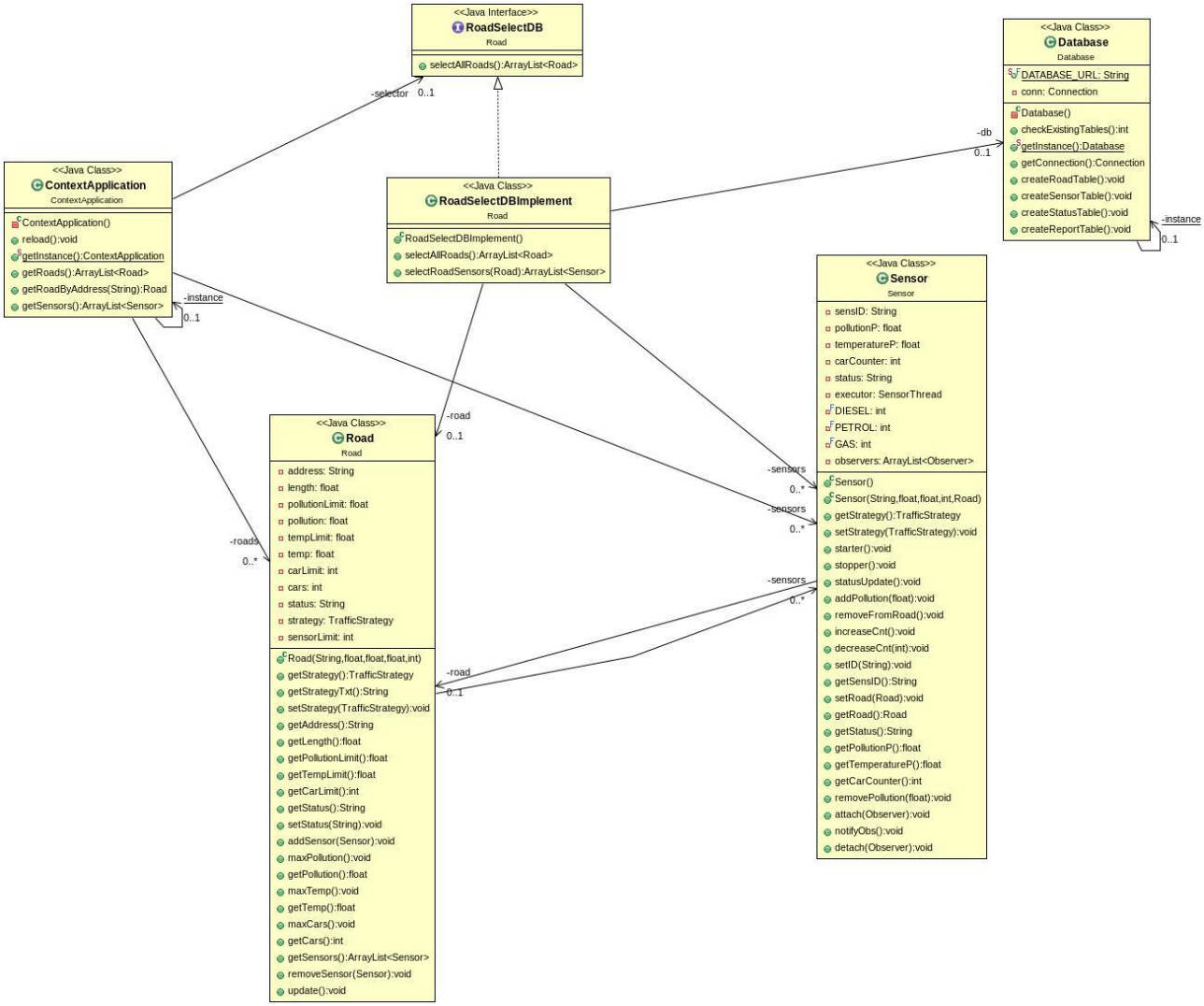
Al cambiamento dei parametri di un sensore se si superano i limiti stabiliti dalla strada si arriverà al codice ROSSO per il quale verrà avviata una strategia. Si è stabilito una probabilità di attivazione del 50% per ogni strategia. Il Thread del sensore che ha superato il limite avvia la strategia di tipo “*TrafficStrategy*” istanziata come “*AlternatePlateStrategy*” o “*CloseRoadStrategy*”, la prima limita il transito alle auto ponendo dei turni per alternare il transito (numero di targa pari o dispari), la seconda invece, limita il transito a tutte le auto. Se il divieto di transito non viene rispettato (probabilità del 10%) l’auto viene segnalata dal “*PoliceReporter*” che si occuperà di inserire la segnalazione nel DB.



Pacchetto con le strategie e metodi
https://github.com/PlidherLuna/SmartCity/tree/main/src/Strategy

Diagramma delle classi per l'accesso ai dati

Di seguito viene mostrato un singolo esempio di come viene effettuato l'accesso ai dati persistenti, si è utilizzato un Design Pattern Architetturale "*DataAccessObject*" (DAO) non facente parte della "*GangOfFour*" (GoF). Poiché ci sono molte classi ed interfacce che vi accedono si è deciso di porre un singolo esempio generale. Nessuna classe ha accesso diretto al Database così da poter utilizzarne altri se necessario. Ogni classe "*-DBImplement*" che accede alla classe Database ha le proprie istruzioni per aggiornare/inserire/eliminare contenuti dal DB. La classe Database fornisce la connessione.



Esempio del ContextApplication
https://github.com/PlidherLuna/SmartCity/blob/main/src/ContextApplication/ContextApplication.java

Classe Database
https://github.com/PlidherLuna/SmartCity/blob/main/src/Database/Database.java

Classe RoadSelectDBImplement con le query necessarie
https://github.com/PlidherLuna/SmartCity/blob/main/src/Road/RoadSelectDBImplement.java

Diagramma delle tabelle presenti nel DB

ROAD	
PK	Address: VARCHAR
	RoadLength: REAL
	PollutionLimit: REAL
	TemperatureLimit: REAL
	CarLimit: INTEGER
	CarLimit: INTEGER

SENSOR	
PK	SensID: VARCHAR
FK	RoadAddress: VARCHAR
	Pollution: REAL
	Temperature: REAL
	CarCounter: INTEGER

STATUS	
PK	DateUpdate: TEXT
PK,FK	SensID: VARCHAR
	Pollution: REAL
	Temperature: REAL
	Temperature: REAL

REPORT	
PK	ReportDate: TEXT
PK	CarPlate: VARCHAR
FK	RoadAddress: VARCHAR

Classe Database con la creazione delle tabelle se
https://github.com/PlidherLuna/SmartCity/blob/main/src/Database/Database.java