

Telemetry Tracker

Margarit Ane-Marie - 1217649

Written report Object Oriented Programming project

2023-2024

Index

1. Introduction.....	1
2. Model description.....	2
3. Polymorphism.....	3
4. Data persistence.....	4
5. Implemented features.....	5
6. Working hours reporting.....	6
7. Changes after project evaluation.....	7

1. Introduction

The Telemetry Tracker project consists of monitoring, collecting and managing telemetry sensor data. Its main goal is to develop an application that facilitates the visualization and management of sensor data in a user-friendly and efficient manner.

Telemetry refers to the process of collecting, transmitting, and receiving data from remote sources, typically sensors, and transmitting it for monitoring, analysis, and decision-making.

This project focuses on finding a solution that enables the user to interact with sensor data, providing features such as data visualization, data storage, adding, editing and deleting sensor data, moreover searching for a particular sensor through the sensor JSON file that a user can load and edit.

The telemetry focuses on monitoring and optimizing the performance and safety of vehicles. Among the key sensors chosen for this project are the tire pressure sensor, the brake temperature sensor and lastly the fuel flow sensor, each playing a critical role in enhancing the vehicle performance, efficiency and safety.

The tire pressure sensor, a vital component of modern automotive systems, monitors tire pressure levels. The remaining sensors, that is the brake temperature sensor and the fuel flow sensor monitor the brake temperature levels and the flow rate consumption of a vehicle.

The sensors can be created or loaded through a JSON file through the specific Open file button, and can be selected from a sensor panel where they are listed vertically. The user can select a specific sensor from the list and he can visualize it on the chart displayed on the right. The chart shows the numeric values of the sensors on the y axis in relation to the timestamp on the x axis.

The user can search for sensors through a search bar shown above the sensor panel, and the number of search results can be seen in the status bar; the matching sensors are updated dynamically whenever the user makes a research.

2. Model description

The logical model can be divided in two sections: the sensor management, which includes operations such as creating, reading, updating, deleting and searching as indicated for the requirements of this project; the second section represents the data visualization on the charts. The first section includes both the classes that represent the sensors and the classes that allow the user to manage JSON files and data. Qt libraries were used for these tasks, such as `QJsonObject`, `QJsonDocument`, `QJsonArray` and `QFile`.

The model starts from an abstract class `AbstractSensor` that represents the shared information for all sensors that is the sensor identifier, the timestamp and the sensor type.

Concrete classes deriving from the `AbstractSensor` class are `TirePressureSensor` which represents the tire pressure of a vehicle; `BrakeTemperatureSensor` which represents the brake temperature of a vehicle, and lastly the `FuelFlowSensor` that measures the rate of fuel consumption in the vehicle engine.

The sensor management operations were implemented in the user interface, they can be activated through the specific buttons that run them. The “Add sensor” button uses a dialogue window that allows the user to insert the data such as the identifier, the sensor type (that can be selected through a list menu in the dialogue window), the timestamp and the numeric value. According to the sensor type selected, the user can insert the specific numeric value for each sensor.

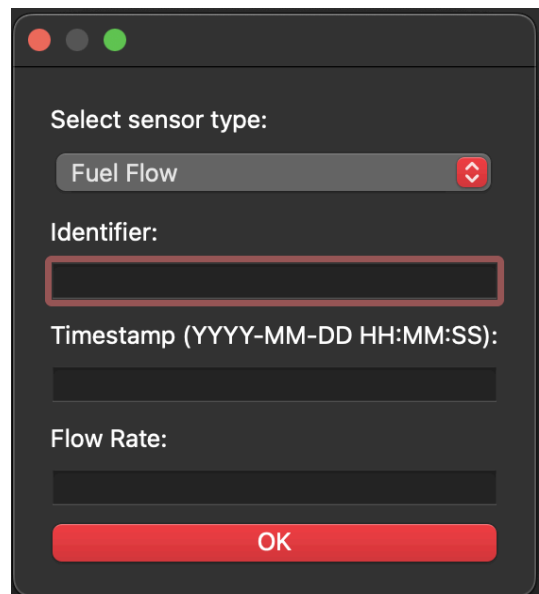
The user can load a JSON file through the “Open” button and visualize the sensor widgets created for each sensor inside the sensor panel, he can also save a JSON file through the “Save” button.

The design pattern Visitor was used for visiting the JSON objects representing sensor data, creating corresponding widgets for displaying the data in the user interface. The `JsonControllerVisitor` is tasked with visiting JSON objects, extracting the data and setting the data in the user interface. Sensor widgets are displayed in different colors according to the sensor type visited by the `Json` visitor.

The application provides a Delete sensor button that allows the user to remove a specific sensor by inserting a particular identifier that corresponds to the sensor that has to be deleted. The chosen identifier will therefore be removed both from the sensor panel within the user interface and from the JSON file that has previously been loaded. This way data will be consistent and changes within the user interface can be reflected throughout the JSON file.

The “search sensor” edit line allows the user to search for a particular sensor through the identifier or the sensor type. The sensors can be searched for even through a partial string inside the edit line. The sensor panel will be updated automatically as soon as the user writes a text inside the edit line.

As for the visualization of sensor data, the application creates a chart if a sensor widget has been selected and clicked in the sensor panel.

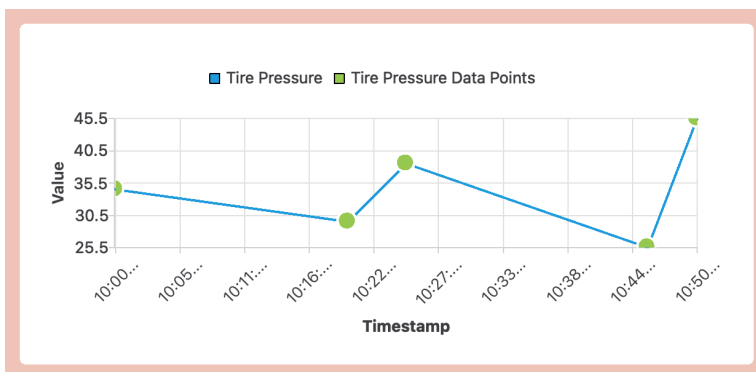


3. Polymorphism

Polymorphism was mainly implemented in the JSON Controller file. In the "jsonControllerVisitor" class the "visit(const QJsonObject&)" method was implemented, which is called when the visitor encounters a JSON object representing a single sensor. In this method the application creates a "SensorWidgetVisitor" object, which displays sensor data in a widget inside the sensor panel; we can see different widget backgrounds according to the sensor type displayed in the sensor panel, all of this because of polymorphism.

Another way of implementing polymorphism was through the methods of adding, deleting and updating sensor data. For example, when updating some sensor data, polymorphism is used to determine the sensor type and the associated value name, then the data will be updated in the corresponding JSON object. The update method is able to dynamically handle different types of sensor types of sensors and their respecting data without the need to explicitly know the sensor type.

Another way of implementing polymorphism was through the creation of a graphic chart used to visualize the sensor data according to the selected sensor within the sensor panel. When displaying sensor data on the chart the appropriate derived class implementation of the virtual functions are dynamically bound and executed based on the actual type of sensor data being processed. Sensor visitors were used for these implementations .



Timestamp: 2024-04-20 11:11:11
Brake Temperature: 111.11

Identifier: 555555
Type: Fuel Flow
Timestamp: 2024-04-19 15:55:55
Flow Rate: 15

Identifier: 123456
Type: Tire Pressure
Timestamp: 2024-03-10 10:00:00

Add Sensor

4. Data persistence

The data persistence aspect of the project involves managing JSON files to store and retrieve sensor data efficiently.

The project uses `QFile`, `QJsonDocument` and `QJsonObject` classes in Qt to handle reading from and writing to JSON files. This allows for seamless interaction with JSON files on the filesystem. The application can read existing JSON files containing sensor data and parse them into `QJsonDocument` objects for further processing. Conversely, it can also write sensor data into JSON files. JSON files serve as a structured format for storing sensor data.

Each sensor is represented as a JSON object with key-value pairs containing information such as sensor identifier, type, timestamp and specific data values.

When new sensor data is generated, the application appends this data to existing JSON files. This ensures that historical sensor data is preserved and allows for the accumulation of data over time.

This project incorporates error handling mechanisms so that the user can deal with situations such as inability to open or write JSON files. These error messages provide feedback to users.

The application can retrieve sensor data from JSON files as needed for visualization purposes. This enables users to access historical sensor data.

Examples of the file structure are given through the JSON files presented along with the code, there are 3 different documents containing alternate combinations of sensors.

5. Implemented features

The following features enrich the project with ulterior details in addition to the former requirements given for the assignment.

Graphic implementations and logical implementations were developed.

Logical implementations:

- Conversion and storage of JSON files

Graphic implementations:

- Logo inside the logo frame
- Menu bar above the main window
- Status bar below the main window, showing the search results, whether a file was saved successfully or not, whether a sensor was deleted successfully or whether sensor data was updated correctly
- Each type of sensor has its own personalized chart for the sole purpose of managing the main window in an efficient and polymorph way
- Data points markers for each data value retrieved from the JSON file
- Each type of sensor has its own personalized widget inside the sensor panel, so the user can easily identify the different types of sensors displayed
- Sensor widgets can be selected and clicked within the sensor panel
- Focus and border added to a selected widget within the sensor panel

6. Working hours reporting

Tasks	Expected working hours	Actual working hours
Study and Design	10	10
Model Code Development	10	25
Qt Framework Study	10	15
GUI code development	10	20
Test and debug	5	10
Written report writing	5	5
Total	50	85

In spite of the fact that the initial expected working hours for this project were 50, given the fact that I've never done a project like this on my own before, I decided not to pressure myself knowing that my skills were not enough at the beginning.

I chose to study carefully and plan everything that needed to be done for this project so I calculated the expected working hours in a different way.

The study and design of the project was quite easy to organize since I already had some ideas in mind. I encountered some setbacks when I started to actually code what I had in mind with C++.

The most difficult part for me was developing the model code. I knew what my panels, buttons and charts had to do but implementing the right functions associated with them took me some time.

The JSON files management was one great setback, I had to study the right Qt documentation that allowed me to implement this feature, along with the JSON files management I had to connect all the panels, widgets and buttons to the areas that used the JSON data, so this task was not that easy to fulfill. I had to put up with some incongruences for data persistence in particular. I could see on the user interface how I was adding, editing and deleting data, but reflecting those changes through the JSON file that had to be saved was another task to complete.

I decided to keep things simple since I lacked the right knowledge to create a more complex project. Even though my project is quite linear and simple, during its development I had to spend many hours on testing and debugging for the implementation of features. For example I didn't know how to include header files and Qt libraries properly at the beginning, so initially there were some errors occurring regarding some undeclared identifiers and Qt types that I didn't know how to solve, sometimes I even got some build issues, so I needed to fix that as well.

The written report was rather fast to complete since I provided my code with the necessary comments that kept track of what I was doing, so it was quite easy to explain in the written report how my project was implemented.

In conclusion I would say that my project took some time because I lacked some knowledge in order to complete it in a faster way.

7. Changes after project evaluation

In order to implement polymorphism, I've integrated AbstractSensor objects with the JsonController.

Specifically I modified the following methods:

1. `Sensor::AbstractSensor *getSensor()`
2. `QVector<Sensor::AbstractSensor*>getSensorData()`
3. `addSensorData(Sensor::AbstractSensor * sensor)`
4. `updateSensorData(Sensor::AbstractSensor * sensor)`
5. `setSensorData(Sensor::AbstractSensor * sensor)`
6. `addSensorToPanel`
7. `collectSensorData`

I tried to make sure that I used AbstractSensor objects and its derivatives rather than QJsonObjects.

Furthermore I added a method called "toJson" to each class in order to convert AbstractSensor objects to QJsonObjects when the file needs to be saved as you suggested.

I realized that I wasn't using the pattern Visitor correctly in the SensorWidgetVisitor, so I just changed its definition in order to avoid confusion.