Technical Documentation

for

Ice-Cream Monitoring

Version 1.0

written by

Anja Xhakani	Anja.Xhakani@stud.srh-campus-berlin.de
Chawin Sungwalngern	Chawin.Sungwalngern@stud.srh-campus-berlin.de
Luca-Maximilian Meier	Luca-Maximilian.Meier@stud.srh-campus-berlin.de
Siddhant Nitin Chavan	SiddhantNitin.Chavan@stud.srh-campus-berlin.de

Lecturer:	Franziska Shwarz
Course:	Software Engineering
Date:	23 June 2023

Table of contents

Document version history

Version	Date	Author(s)	Reason for change/remarks
0.1	*18/06/2023*	*Siddhant*	*Added introduction+quality requirements sections*
0.1	*19/06/2023*	*Anja*	*Added description of system functionality topic nr.2*
0.2	*20/06/20203*	*Siddhant*	*Added technology description*
0.2.11	*20/06/2023*	*Siddhant*	*Added architectural patterns*
0.2.2	*21/06/2023*	*Siddhant*	*Added more details to architectural patterns descriptions*
0.2	*20/06/2023*	*Anja*	*Added the UML Diagram and its content*
0.2.1	*21/06/2023*	*Anja*	*Added the Sequence Diagram in Communication topic nr.7 and explanation*
0.1	*22/06/2023*	*Chawin*	*Drew a component diagram (page: 9) and sequence diagram (pages:14, 15)*
0.2.1	*22/06/2023*	*Anja*	*Added the components on topic nr.5 and explanation*
0.2	*22/06/2023*	*Chawin*	*Added the interface description on topic nr.5.2 (pages:9, 10, 11)*
0.3	*23/06/2023*	*Anja*	*Added the rules page in topic nr. 6*
0.4	*23/06/2023*	*Siddhant*	*Added MVC implementation in section 5.2*
1.0	*23/06/2023*	*Siddhant*	*Added sources and finalize table of records*

Overview of roles/contribution:

Besides contribution from above table this also how we split our work:

Design: Luca + Siddhant

Development/Coding: Luca

Management of the project: Anja

Management of the report: Siddhant

Coordinating UMLs: Chawin + Anja

As a group we are confident that we divided the workload between each other fairly and everyone played an equally impactful role.

1. Introduction

Welcome to the comprehensive software engineering documentation for “Ice-cream station monitoring”. This documentation serves as a vital resource for understanding the inner workings, architecture, and functionality of our software solution, providing valuable insights for developers, stakeholders, and future contributors.

The project is derived from and built upon by a reference to “GUI Architectures” from Martin fowler. Described therein is a system referencing a GUI interface for monitoring ice cream health of the population of New England States. The GUI itself shows us a list of monitoring stations. When one is clicked the GUI displays the details for the same station including: “Station ID, Date, Target, Actual and Variance.”

Each of these fields contain relevant data. “Target” contains the desired ice cream particulates in the air, “Actual” contains the current value for the same and “Variance” contains the difference between the target and actual fields.

The UI enables the user to select one of the fields and edit its details barring the target and variance fields. If the actual value of the station is less than that of the target then the variance is highlighted in red. It's highlighted in green for the vice versa. The article describes design patterns and how it may be implemented.

The goal of our project is to replicate the same in our own way which is further described later while understanding and implementing different design patterns for our clients.

To do this we created a program using python and its libraries. Using Tkinter to create the GUI and JSON files to save data. We implemented the proposed concept.

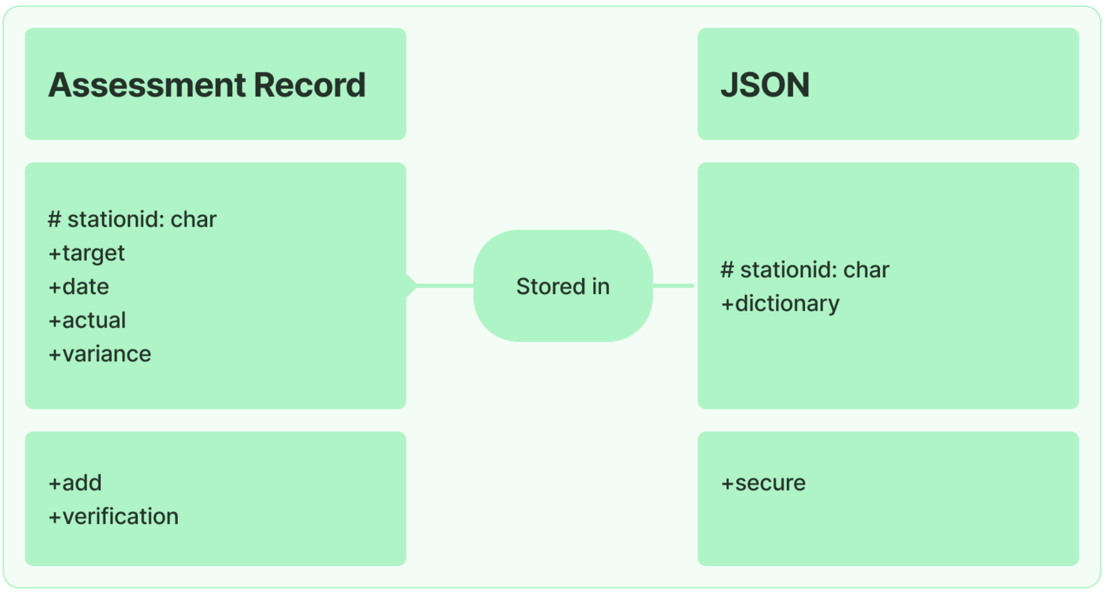
2. System functionality

To better give an understanding of the system functionality we are using a class UML diagram. The class diagram shown below provides an overview of the system's structure and organization. It allows not only the developers, but also the stakeholders to identify and represent the classes used in this system and their relationships, making it easier to explain the overall architecture of the system.

This class diagram supports abstraction, allowing the client to focus on the essential elements and hide unnecessary implementation details. This promotes clarity and simplifies the communication of the system's functionalities to stakeholders who may not have technical expertise.

Shown below is a clear representation of the system's functionalities which include the "Assessment Record" and "JSON" classes with their own representative attributes.

The first class, "Assessment Record" holds all the attributes that the user needs to complete a record such as the station ID, date etc. The main operation is the "add" function that helps the user to add a record in the data structure and the "verification" function that helps the user to know that the record was verified and saved in the system.
The second class, "JSON", holds the attributes that the system needs to save the records in the dictionary. The main function is to be secure as this data structure holds all the information that the user needs to access the system.



(Class UML diagram to understand the system functionality)

3. Quality requirements

A number of quality requirements have been highlighted here that are needed to be met during the development of the project.

- Use Case: Add station.

  A new station has been added to the system and indicated the same in the GUI. This new station can be acted upon by the user and its fields edited after which it's integrated into the system like any other.

- Usability: Variance.

  After a new value is added the variance i.e. the difference between the target and actual is available in a variance field which is highlighted.

- Usability: Variance II.

  The change in the field must be differentiable between a desired and non desired state

- Usability: New Station.

  When AddStation occurs the new system is visible to the user and ready for use.

- Modifiability.

  The GUI is flexible and can be modified and a new one can be implemented without changing the inherent logic.

- Modifiability II.

  A GUI can be implemented onto a new system within two days.

- Modifiability III.

  Development of a new GUI does not affect other currently operating ones

- Modifiability IV.

  A new functionality can be added given it's ready to implement.

- Availability.

  If the GUI fails it can be launched again within one day with existing data.

- Availability II.

  An operator/logic/software error causing system failures is available again.

- Performance.

  If a new station is added, it's visible to the user within two seconds.

- Performance II.

  If a value is edited by the user it's reflected within 0.5 seconds in the GUI.

- Testability.

  The developer wants to test a new GUI and can do so by creating tests for the same UI.

4. Architecture

We made use of the Model View Controller architecture pattern to design and implement our solution or parts of the solution. The Model-View-Controller (MVC) architectural pattern is a widely used design pattern for developing software applications with graphical user interfaces (GUIs). It separates the application into three interconnected components: the Model, the View, and the Controller. Here's an overview of each component and the strengths and weaknesses of the MVC pattern:

Model:

The Model represents the application's data and business logic. It encapsulates data structures, algorithms, and rules for manipulating and managing the data. The Model component is responsible for maintaining the integrity and consistency of the data, independent of the user interface. The Model separates the application's data and business logic from the user interface, enhancing modularity and maintainability.

Decoupling of the Model from the user interface allows for easier reuse of the data and logic across different views or controllers.

This also leads to an increase of complexity of the code as the program scales. The Model component does not directly interact with the user, which can make it challenging to handle certain user interactions or real-time updates without proper coordination.

View:

The View represents the presentation layer of the application and is responsible for displaying the data to the user. It provides a visual representation of the Model's data and communicates user actions to the Controller for processing. View is responsible for handling the visual representation and user interaction, keeping it decoupled from the underlying data and business logic.

The View can be tailored to specific user interface requirements, allowing for customization and adaptability. This allows for supporting multiple views that can present the same underlying data in different ways, enabling diverse user experiences.

Without careful design, the View component can accumulate presentation-specific logic, leading to a bloated and less maintainable codebase. Views are often closely tied to a specific user interface framework or technology, making it challenging to reuse them across different platforms or frameworks.

Controller:

The Controller acts as an intermediary between the View and the Model. It receives user inputs from the View, processes them, and updates the Model accordingly. It also manages the flow of data between the View and the Model.

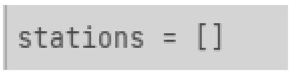
The Controller component manages the overall flow and coordination of the application, orchestrating interactions between the View and the Model.

In complex applications, the Controller component can become complex, making it challenging to maintain and understand the logic flow.

Controllers often have dependencies on the user interface framework or technology being used, reducing flexibility and portability.

As you can see the MVC has its strengths and weaknesses. Especially relating to an increased complexity. reduced flexibility and portability of the software. We solved these problems by our choice of technologies, elaborated further in the “Technologies Section”

We accomplished this by separating our data handling functions to be separate from that of the GUI/Tkinter functions. We created a list of dictionaries that's part of the controller layer.

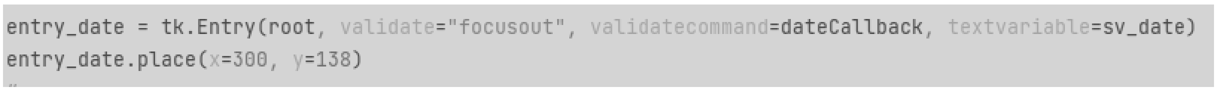


```
stations = []
```

This stores data during runtime as data is communicated between the view (TKinter GUI) and controller (our logic and functions). Then only writes data to the model (JSON file) when the user saves their current state and data. This decouples the model from the view and controller and it is only called upon when it's being read from or data is being stored within it.

Moreover the view (Tkinter components) only contain functional callbacks to the controller functions and not the entire logic further embracing the MVC pattern.

An example of the same:



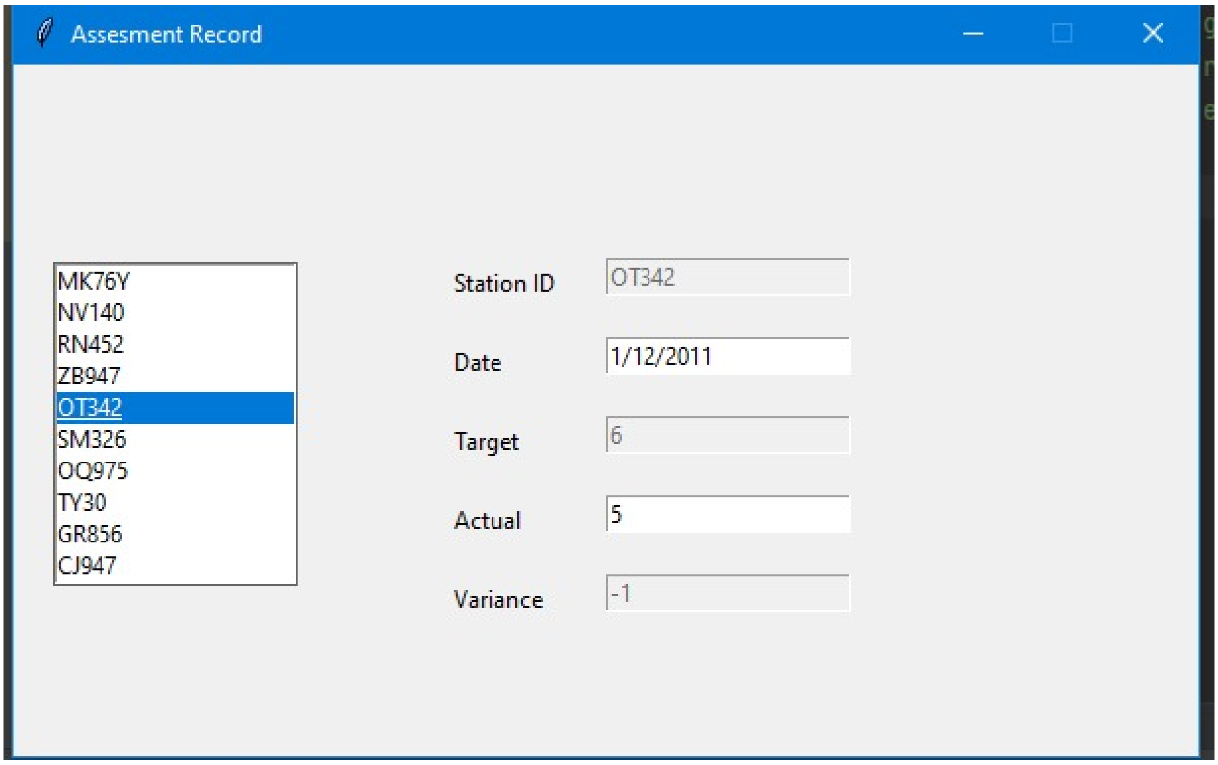
```
entry_date = tk.Entry(root, validate="focusout", validatecommand=dateCallback, textvariable=sv_date)
entry_date.place(x=300, y=138)
```

Here this highlights the date label on the GUI. As you can see it contains a call back to the controller layer in the form of “dateCallback”. This controller then fetches data from the model and updates the view.
This logic is true for every component on the GUI.

Overall, the MVC pattern provides clear separation of concerns, promotes reusability, and enables easier testing and maintenance. However, it can introduce complexity and may require careful design and coordination to prevent excessive coupling between components.

5. Components (Static View)

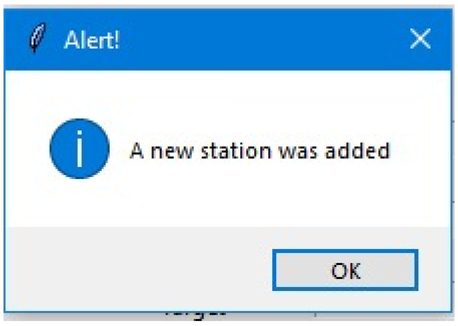
A brief explanation of the GUI used in this project is shown in the three images below. First, we have the Assessment Record UI where the user has the chance to fill out the form.
Second, we have the alert screen with the information letting the user know that the new station was added in the database.
Third, we have the pop up screen allowing the user to choose between saving the record or not.



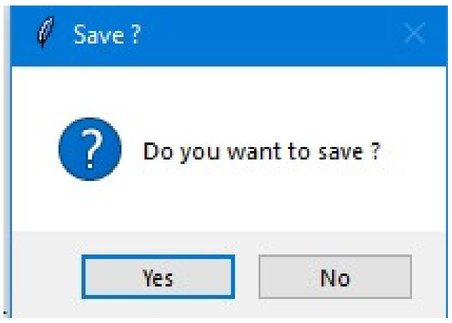
MK76Y
NV140
RN452
ZB947
OT342
SM326
OQ975
TY30
GR856
CJ947

Station ID	OT342
Date	1/12/2011
Target	6
Actual	5
Variance	-1

(Image 1: Assessment Record GUI)



A new station was added

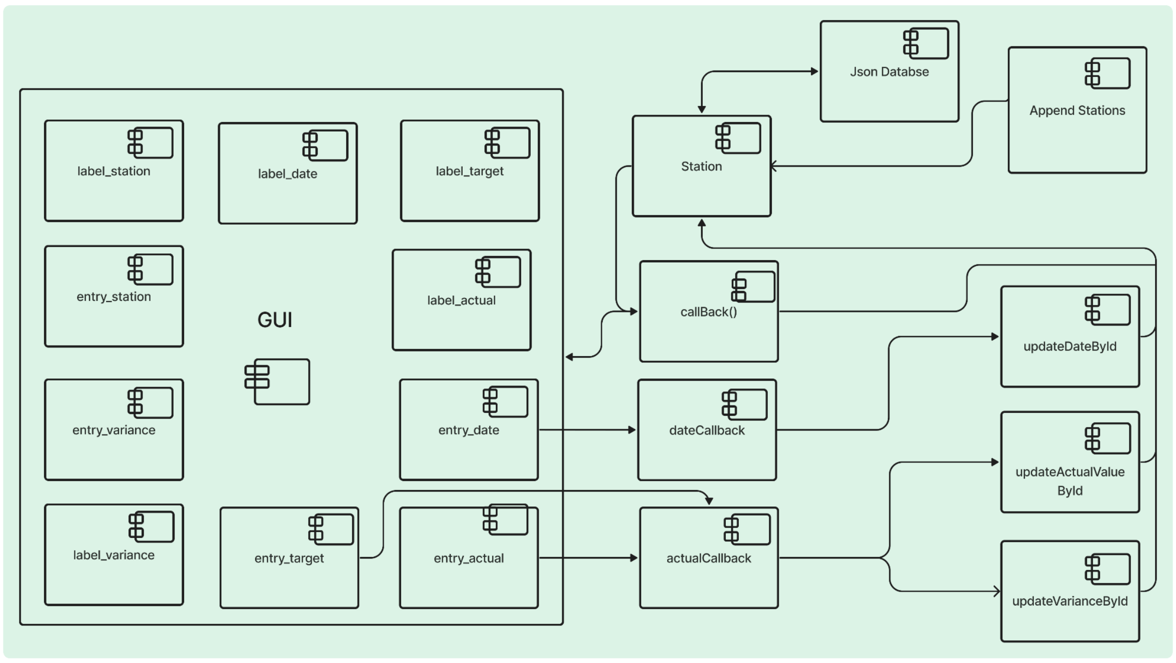


Do you want to save ?

(Image 2 and 3: Alert window and Save window)

5.1. Components

Using a multi-layered architecture diagram helps to illustrate the structural design of the Ice-Cream Station Monitoring system, emphasizing the separation of different concerns, the scalability, modularity, and maintainability of the project. It provides stakeholders with a clear understanding of the system's architecture.



(Component Diagram shown with multi-layered architecture)

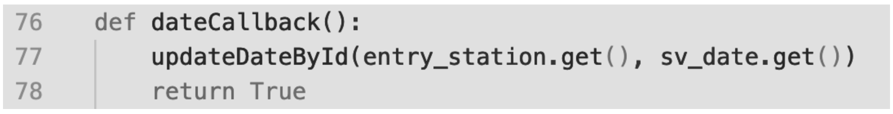
This component diagram explains how the user interface illustrates the logical components of the system and their relationships. We begin by identifying the major components of the system, such as the User Interface, Database as the JSON file and any other relevant components.
After representing each component on the table we determine the interfaces or communication channels between the components. Different components depend on other components to function such as the Target. Target needs to be connected to a Station to be shown in the GUI, which as an instance needs to get this information from the database.

The Station component needs to be connected to the Database component in order to retrieve the stored data. Every component connected to the Station requires an ID to function as well.

We can see the MVC implementation in our coding patterns here: The GUI (View) components never interact directly with the database (Model). The callback(), dateCallback(), actualCallback() among others act as our controller layer and update data on the GUI as they fetch data from the database. This allows us to have freedom when designing the GUI without having to work around a bloated interface that's difficult to change or edit, among other advantages mentioned in the architectural section.

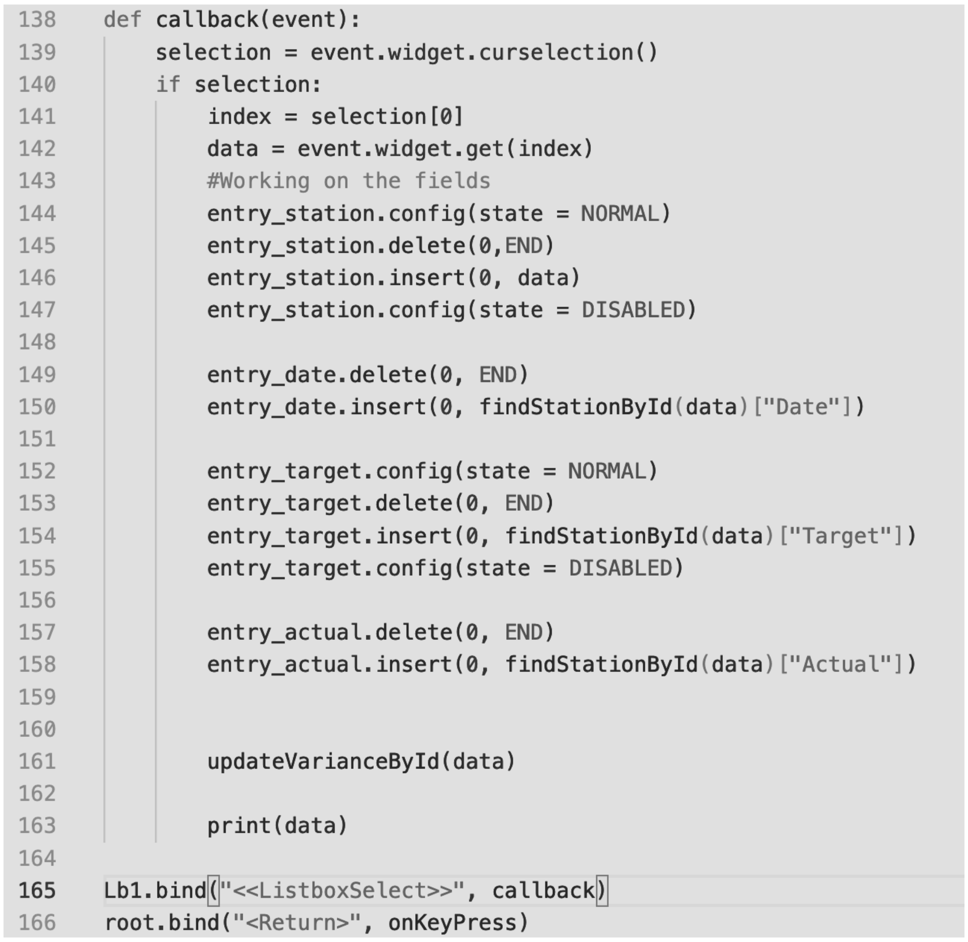
5.2. Interfaces

The interface functions are the event-driven functions that directly interact with the graphical user interface (GUI) elements. They allow the user to input data and trigger actions in response to events. Based on the code snippet provided below, there are 4 functions considered interface functions:



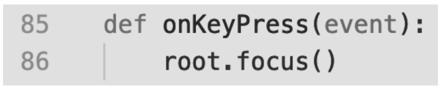
```
def dateCallback():
    updateDateById(entry_station.get(), sv_date.get())
    return True
```

- dateCallback() - This function is called when the focus is lost from the `entry_date` field. It updates the date value for a specific station ID based on the user input.



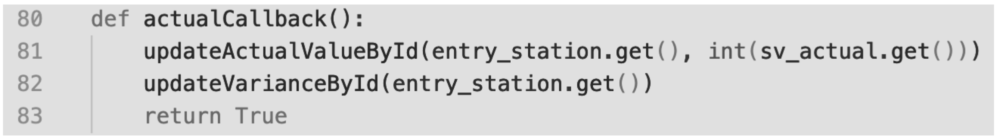
```python
def callback(event):
    selection = event.widget.curselection()
    if selection:
        index = selection[0]
        data = event.widget.get(index)
        #Working on the fields
        entry_station.config(state = NORMAL)
        entry_station.delete(0,END)
        entry_station.insert(0, data)
        entry_station.config(state = DISABLED)

        entry_date.delete(0, END)
        entry_date.insert(0, findStationById(data)["Date"])

        entry_target.config(state = NORMAL)
        entry_target.delete(0, END)
        entry_target.insert(0, findStationById(data)["Target"])
        entry_target.config(state = DISABLED)

        entry_actual.delete(0, END)
        entry_actual.insert(0, findStationById(data)["Actual"])


        updateVarianceById(data)

        print(data)

Lb1.bind("<<ListboxSelect>>", callback)
root.bind("<Return>", onKeyPress)
```

- callback(event) - This function is triggered when an item is selected in the `Lb1` listbox. It updates various entry fields based on the selected station ID, allowing the user to view and modify the station data.
    - corresponding parameters: event



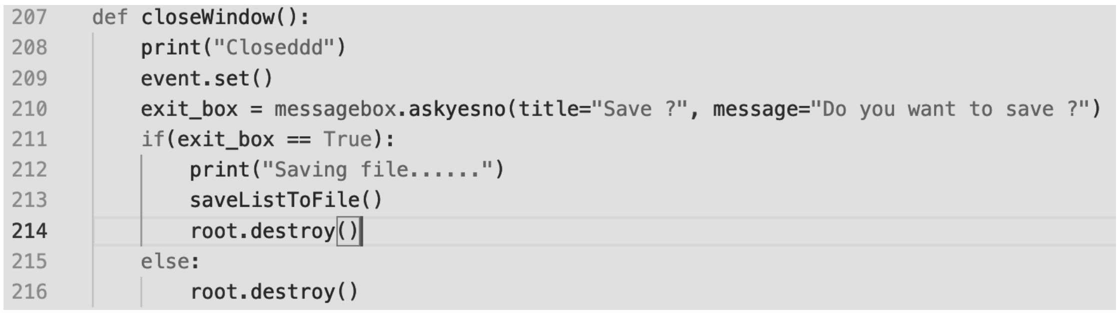
```python
def onKeyPress(event):
    root.focus()
```

- onKeyPress(event) - This function is called when the Return/Enter key is pressed. It sets the focus to the root window, ensuring that key events are captured.
    - corresponding parameters: event



```python
def actualCallback():
    updateActualValueById(entry_station.get(), int(sv_actual.get()))
    updateVarianceById(entry_station.get())
    return True
```

- actualCallback() - This function is called when the focus is lost from the `entry_actual` field. It updates the actual value for a specific station ID based on the user input and updates the corresponding variance.



```
def closeWindow():
    print("Closeddd")
    event.set()
    exit_box = messagebox.askyesno(title="Save ?", message="Do you want to save ?")
    if(exit_box == True):
        print("Saving file......")
        saveListToFile()
        root.destroy()
    else:
        root.destroy()
```

- closeWindow() - function is to the window close event. Inside the closeWindow() function, the following actions are performed:
    - event.set() - Sets the event object to signal the appendRandomStation() thread to stop generating random stations. A message box is displayed using messagebox.askyesno() to prompt the user with a yes/no question, asking if they want to save the data before closing.
        - When the user chooses to save by clicking "Yes", the saveListToFile() function is called to save the stations list to a JSON file and exit the program.

6. Rules

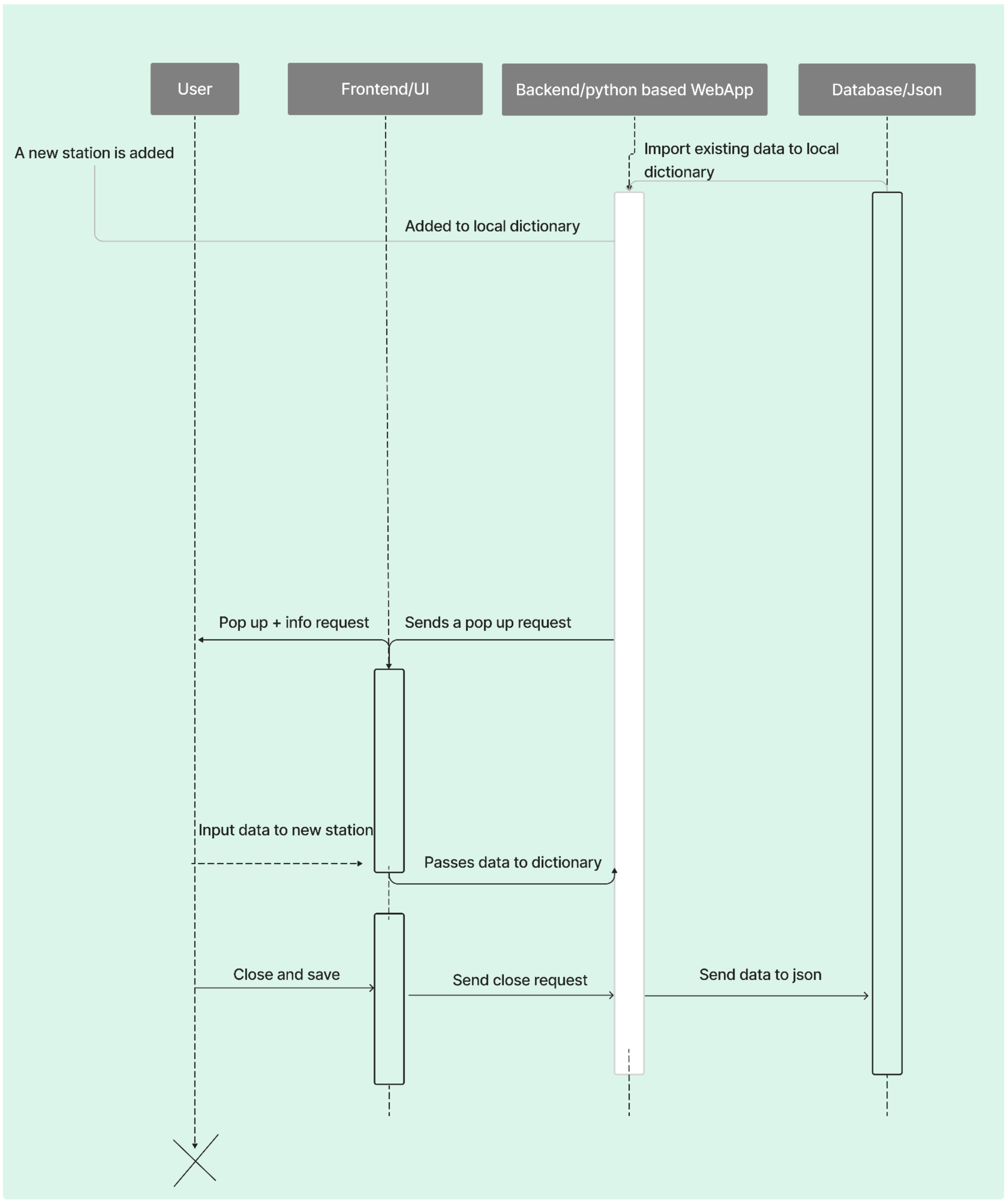
Here are some of the key rules along with their justifications relating to quality requirements:

- Real-time communication between GUI and the Database. It ensures prompt updates, enabling timely actions and decision-making, aligning with responsiveness requirements for effective monitoring and control.
- Secure communication ensures confidentiality, integrity, and compliance with security and privacy requirements, implementing encryption, authentication, and authorization mechanisms.
- Prioritize data integrity in communication components to ensure accurate transmission and reception, preventing errors in sensor readings and status updates, and ensuring trustworthy information quality.

7. Communication (Dynamic View)

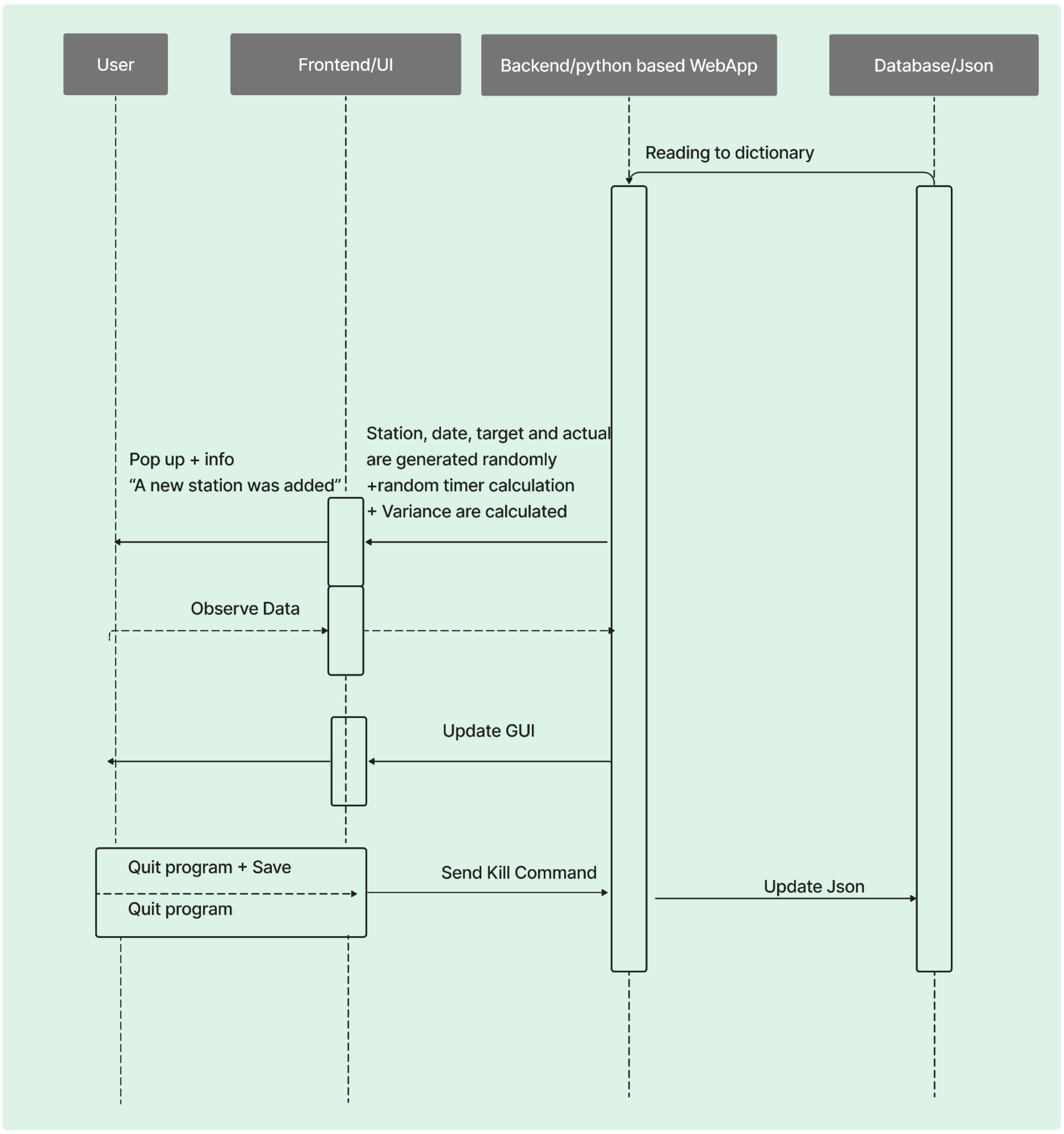
We are using a sequence diagram to explain the Ice-Cream Station Monitoring system functionality because it provides a clear and visual representation of the interactions and flow of messages between different components of the system.
The sequence diagram allows me to illustrate the chronological order of events, showing how the various objects in the system collaborate and communicate with each other.
By using a sequence diagram, we can depict the steps involved in our case, updating the status of a station, and highlighting any dependencies or interactions between different components.

The following diagram explains how the “Add Station” functions. As we can see, we have 4 objects, that include the User, the User Interface (UI), the Developing object which includes the backend/python web based application and the JSON file where our data will be stored.
“A new station is added” is the function that will start the process. First step is activating the local dictionary with the new entry which will cause the existing data to be imported to the local dictionary.
This will activate the application to send a pop up request to the user to request the information. After the user inputs the data, it will be passed to the dictionary so it can be stored. This will cause the user to close the request by saving it and all this information will be sent to the JSON file.


User
Frontend/UI
Backend/python based WebApp
Database/Json
A new station is added
Import existing data to local dictionary
Added to local dictionary
Pop up + info request
Sends a pop up request
Input data to new station
Passes data to dictionary
Close and save
Send close request
Send data to json

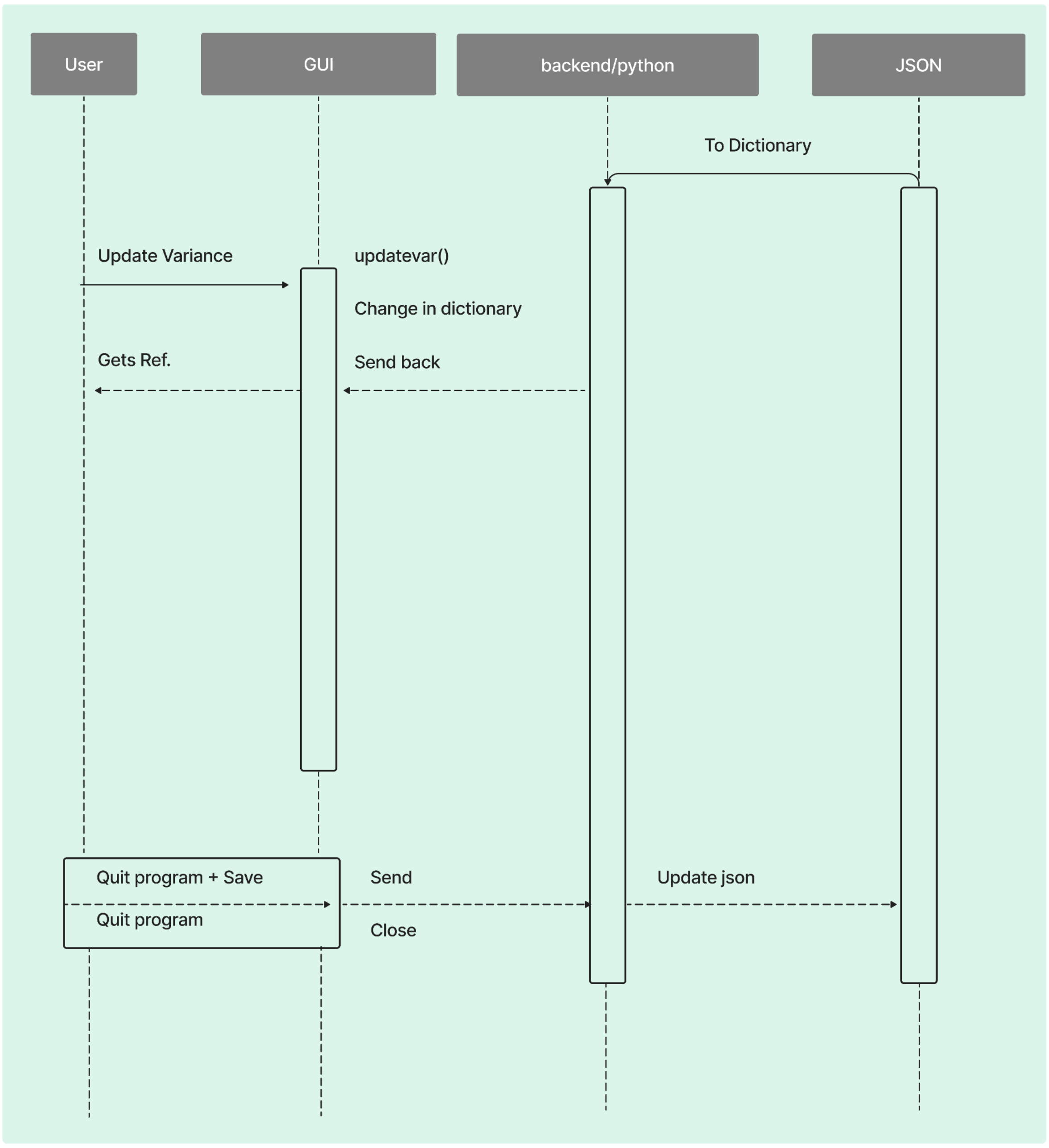
(Sequence Diagram 1: Add Station functionality)

The following diagram will explain the Appending process of a Random Station. It involves the user being informed with a pop up screen that the new station was added in the database. This will trigger the database to generate random information on all attributes, including a random timer calculation. This request can be observed from the user and after the action is finished the user can quit the program safely.


User
Frontend/UI
Backend/python based WebApp
Database/Json
Reading to dictionary
Station, date, target and actual are generated randomly +random timer calculation + Variance are calculated
Pop up + info "A new station was added"
Observe Data
Update GUI
Quit program + Save
Quit program
Send Kill Command
Update Json

(Sequence Diagram 2: Append a Random Station functionality)

The last diagram involves the change of a variance. This diagram starts with the user updating the Variance functionality by using the updatevar() command. This action will be saved in the dictionary which will trigger the backend to send back to the GUI the updated reference. After the user has finished the action, they have the chance to quit the program and save the updated variance, where the backend will take this action to close the program while updating the JSON file at the same time, in order to save the new information.



(Sequence Diagram 3: Generating a Random Variance)

8. Technology

The program makes use of three main technologies and their implementations. A short description and the reason for their use is listed as follows:

Python:

Python is a versatile and high-level programming language known for its simplicity and readability. It has gained immense popularity due to its wide range of applications, ease of use, and vast community support. Python provides a robust standard library and supports multiple programming paradigms, including procedural, object-oriented, and functional programming. It is widely used in various domains such as web development, data analysis, machine learning, and scripting.

Advantages of Python:

- Large Standard Library: Python offers a comprehensive set of libraries and modules that facilitate development across multiple domains.

- Cross-Platform Compatibility: Python runs on different operating systems, including Windows, macOS, and Linux, allowing developers to write code once and run it anywhere.

- Integration Capabilities: Python easily integrates with other languages, enabling seamless collaboration and integration in software systems.

- Community and Ecosystem: Python has a vibrant and supportive community, providing a wealth of resources, frameworks, and libraries to leverage in development projects.

It was due to these reasons we best used python for this project. With cross-platform compatibility we were able to develop the program across windows and linux. With integration possibilities with other languages it leaves the software open to further expansion as required by the client at a future date. Moreover the large assortment of community frameworks and standard libraries this possibility is only enhanced.

It mainly acted as our controller for the program

Tkinter:

Tkinter is a Python library used for creating graphical user interfaces (GUIs). It provides a set of widgets and tools to design and build desktop applications with a visual interface. Tkinter is included as a standard library with Python, making it easily accessible and widely adopted.

Advantages of Tkinter:

- Simplicity and Ease of Use: Tkinter has a straightforward and intuitive API, allowing developers to quickly create GUI applications with minimal code.

- Cross-Platform Compatibility: Tkinter-based applications can run on different platforms, including Windows, macOS, and Linux, without requiring significant modifications.

- Customization and Theming: Tkinter provides various customizable widgets and the ability to create visually appealing interfaces through theming options.

- Integration with Python: Being a Python library, Tkinter seamlessly integrates with other Python libraries and frameworks, enabling developers to combine GUI functionality with other functionalities.

We used Tkinter to create the GUI and view aspect of the program. From creating the pop up alerts to saving windows as described in the earlier section of this documentation

JSON:

For our data storage solution we used JSON files. JSON (JavaScript Object Notation) is a lightweight and widely used data interchange format. It provides a simple and human-readable syntax for representing structured data as key-value pairs. JSON is language-independent, making it easy to exchange data between different programming languages and platforms..

Advantages of JSON:

- Readability and Simplicity: JSON's syntax is easy to read and understand, making it accessible for both humans and machines.

- Lightweight and Efficient: JSON has a compact representation, resulting in smaller data sizes, efficient transmission over networks, and faster parsing by software applications.

- Language Independence: JSON is widely supported across programming languages, allowing seamless integration and data exchange between different systems.

- Data Structure Flexibility: JSON supports various data types, including strings, numbers, booleans, arrays, and nested objects, enabling the representation of complex data structure.

  .

- Web-Friendly: JSON is a natural fit for web applications as it can be easily parsed by JavaScript, making it ideal for data transfer between web browsers and servers.

We used JSON files due to these listed reasons. With it being lightweight it would make transferring data from one system to another very simple. Moreover it is language independent. Should the customer intend to expand this application into a web app the same file JSON object can be used making it future proof. It can also be converted and stored into SQL and noSQL databases easily making it easy for the application to be scaled at a later stage.

9. Sources

Martin Fowler, GUI Architectures, July 2006:

https://martinfowler.com/eaaDev/uiArchs.html

Fitzpatrick, Martin. “PyQt vs. Tkinter: Which Should You Choose for Your next Python GUI?” Python GUIs, 11 Apr. 2023, www.pythonguis.com/faq/pyqt-vs-tkinter/.

Anon. “MVC Framework Tutorial.” Online Courses and eBooks Library, www.tutorialspoint.com/mvc_framework/index.htm.

Prasanna. “Advantages and Disadvantages of Python: Python Language Advantages, Disadvantages and Its Applications.” A Plus Topper, 7 Jan. 2022, www.aplustopper.com/advantages-and-disadvantages-of-python/. Accessed 23 June 2023.

Sahana. “9 Advantages and Disadvantages of JSON to Pay Heed To.” Tech Quintal, 5. Jan. 2022, www.techquintal.com/advantages-and-disadvantages-of-json/.