Design Documentation

Flutter smart sensing library
for medical and psychological study apps

edited by

Leonhard Alkewitz, Florian Gebhardt, Hermann Fröhlich,
Mukhtar Muse, Felix Schlegel

Contents

1 Introduction

This document lists all the architectures for the Flutter Smart Sensing Library. Each diagram includes a description of why it was designed in that way and why certain changes were necessary. Every change is listed in this document.

2 Software Architecture

This section covers the software architecture. It is split into two core components: the Smart Sensing Library and the Sensing Plugin. The reason for this is that we have one component to read and standardize sensor data in a platform-independent way, while the other component represents most of the smart features like storage and filtering of values.

2.1 Smart Sensing Library

The Smart Sensing Library contains most of the features requested by the customer. It serves as the access point to read sensor data. Its design is inspired by the pipe and filter architecture. That way the customer can chain different filters on one dataset. The library is split into three main components: BufferManager, FilterTools, and API. The API has all public methods that can be used by the customer to subscribe to sensors and retrieve filtered sensor data. The BufferManager is responsible for managing all sensor data streams and writing them into the database. It also contains a buffer cache for faster access times. The FilterTools are used to apply different filters to the sensor data.

2.1.1 Version 1

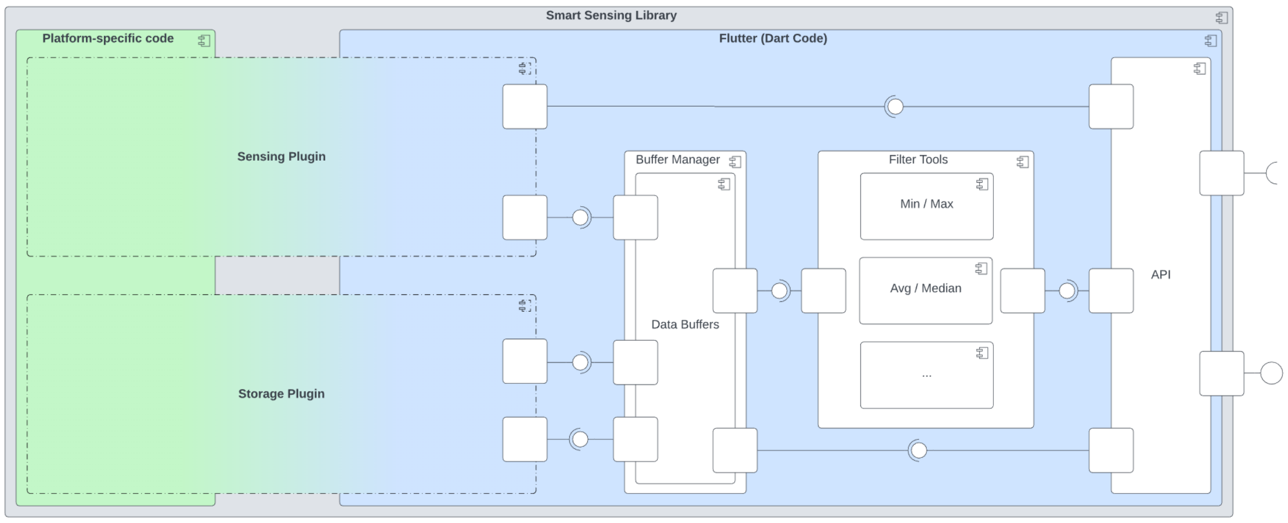


Figure 1: Version 1.0

2.1.2 Version 1.1

An architectural change was made where the BufferManager and FilterTools were incorporated into a new component called IOManager. The reason for this change is that the coordination between the API, FilterTools, and BufferManager was not considered while creating the first iteration. The IOManager is now the primary communication point between the API and the underlying services.

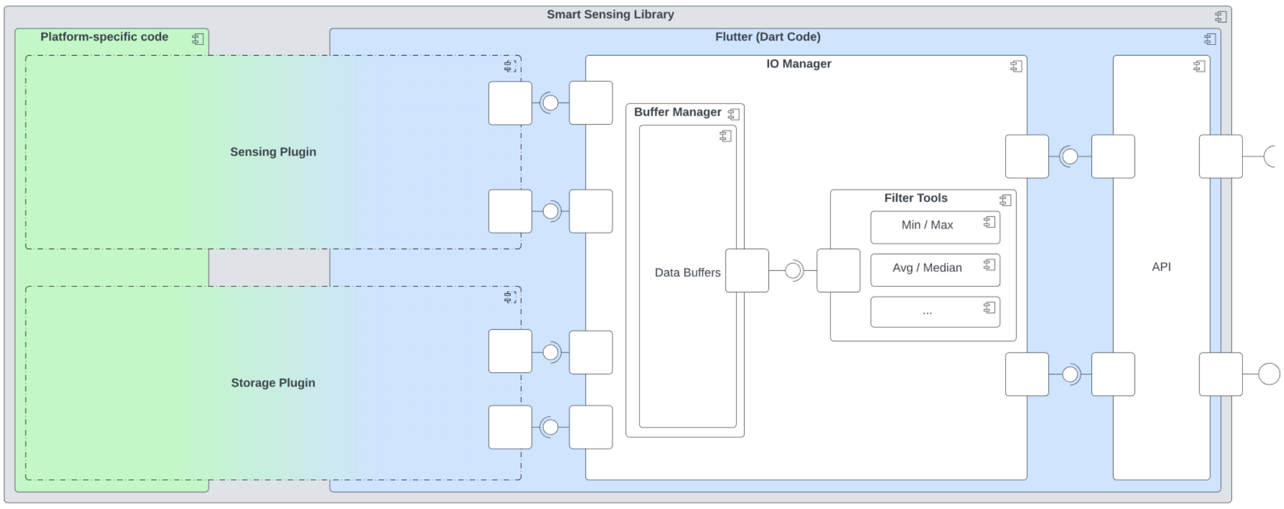


Figure 2: Version 1.1

2.2 Sensing Plugin

The Sensing Plugin is responsible for reading sensor data from iOS and Android devices and performing pre-processing on the data. This standardizes the data and allows the customer to apply various configurations. It's split into two main parts: the platform-specific code and the Dart code. The platform-specific code is responsible for reading sensor data from the device using factory methods provided by the corresponding operating system. These methods can then be called by the Dart code using method channels. In the Dart code, the data is unified (e.g. unit, decimal precision and axis orientation) so that the Smart Sensing Library can work with the data. The SensorManager has two main components. The Command Caller is responsible for delegating customer request like sensor subscription. The Sensor Event Subscriber handels the return of the subscriptions.

2.2.1 Version 1

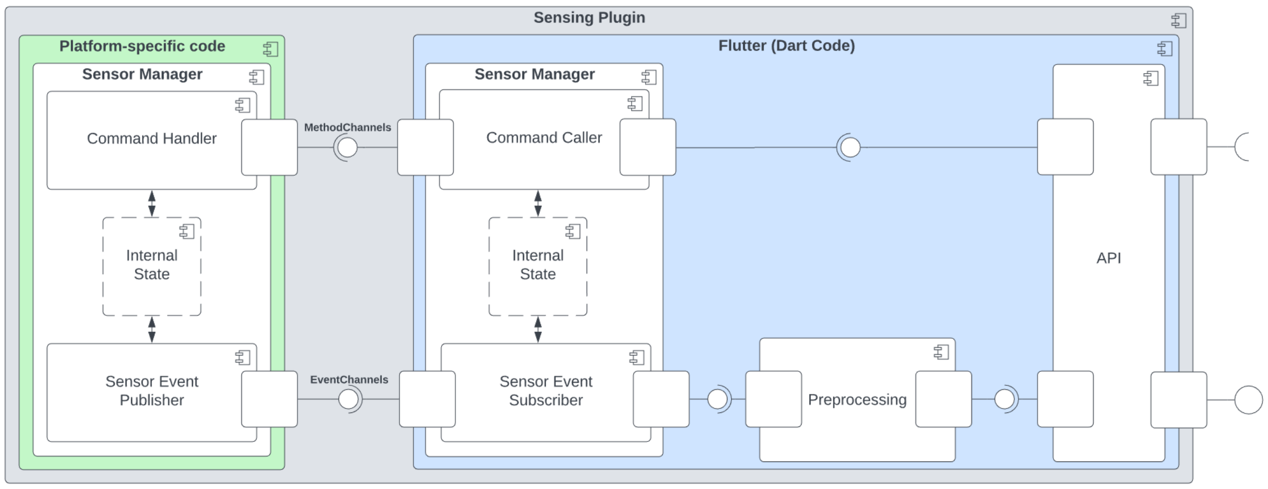


Figure 3: Version 1.0

2.2.2 Version 1.1

There was an architectural change made where the pre-processing is now part of the Sensor-Manager. The reason for this change is that the preprocessor should be a component for the SensorManager to use. In the first iteration, the preprocessor was seen as a single instance that would process all data, but it was later reconsidered that it would be more beneficial to use one preprocessor for each sensor. This allows for easier configuration of each sensor's output.

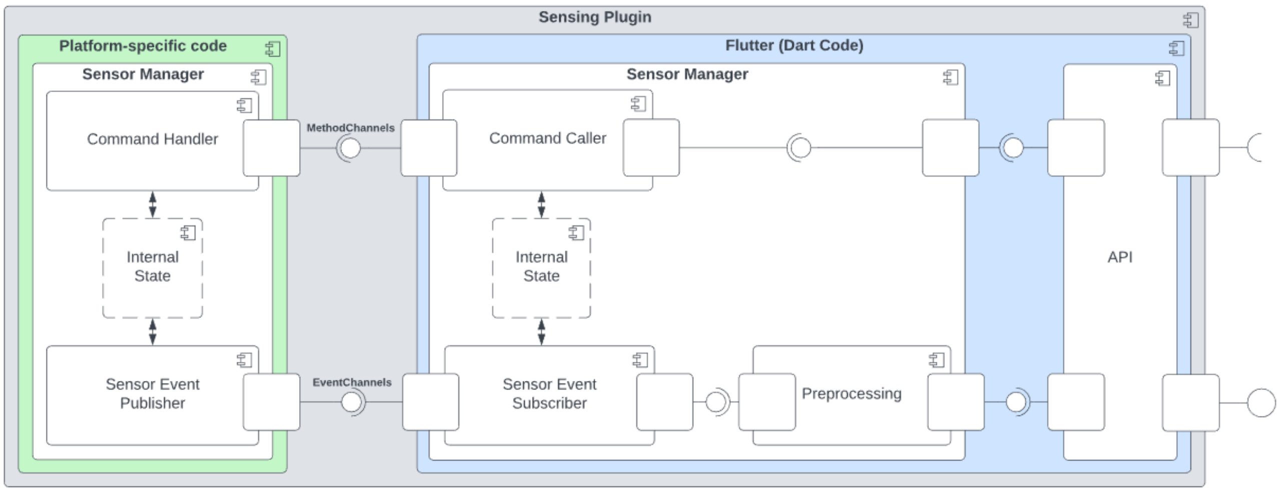


Figure 4: Version 1.1

3 Class Diagramms

This section covers the class diagrams for each component. Each diagram includes a description of the design choices made for that component.

3.1 IOManager

The IOManager component manages all functions related to data storage and transfer. It is the main component for the Smart Sensing Library.

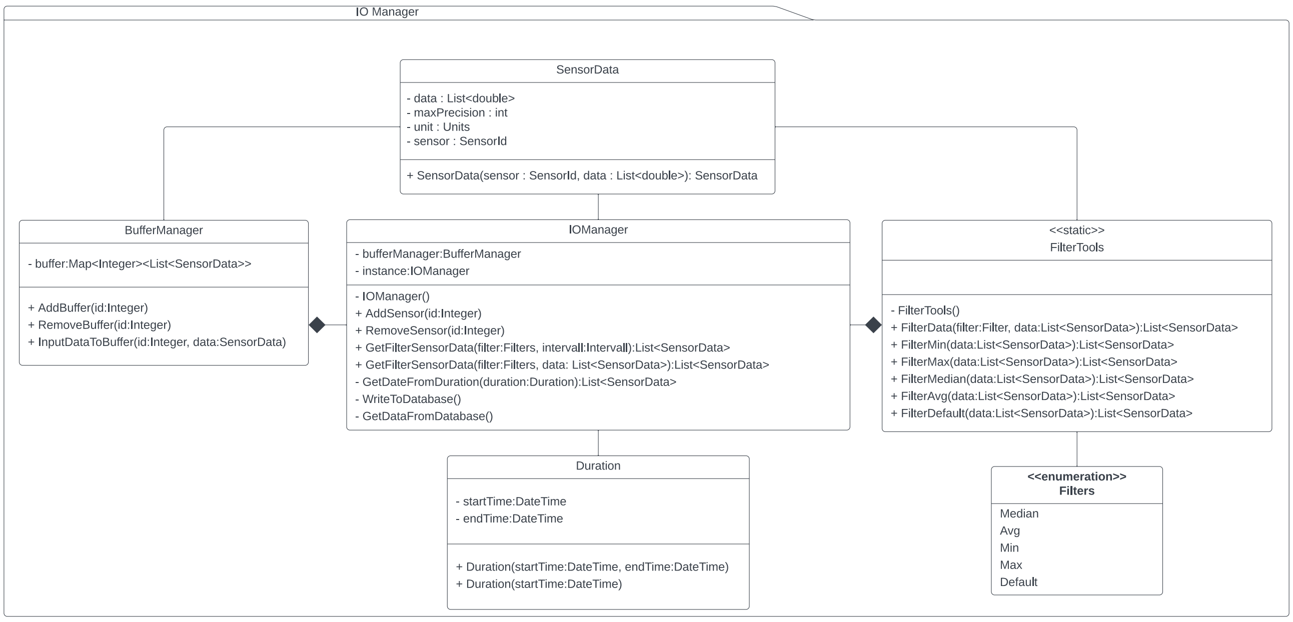

IO Manager
SensorData
- data : List<double>
- maxPrecision : int
- unit : Units
- sensor : SensorId
+ SensorData(sensor : SensorId, data : List<double>): SensorData
BufferManager
- buffer:Map<Integer><List<SensorData>>
+ AddBuffer(id:Integer)
+ RemoveBuffer(id:Integer)
+ InputDataToBuffer(id:Integer, data:SensorData)
IOManager
- bufferManager:BufferManager
- instance:IOManager
- IOManager()
+ AddSensor(id:Integer)
+ RemoveSensor(id:Integer)
+ GetFilterSensorData(filter:Filters, intervall:Intervall):List<SensorData>
+ GetFilterSensorData(filter:Filters, data: List<SensorData>):List<SensorData>
- GetDateFromDuration(duration:Duration):List<SensorData>
- WriteToDatabase()
- GetDataFromDatabase()
<<static>>
FilterTools
- FilterTools()
+ FilterData(filter:Filter, data:List<SensorData>):List<SensorData>
+ FilterMin(data:List<SensorData>):List<SensorData>
+ FilterMax(data:List<SensorData>):List<SensorData>
+ FilterMedian(data:List<SensorData>):List<SensorData>
+ FilterAvg(data:List<SensorData>):List<SensorData>
+ FilterDefault(data:List<SensorData>):List<SensorData>
Duration
- startTime:DateTime
- endTime:DateTime
+ Duration(startTime:DateTime, endTime:DateTime)
+ Duration(startTime:DateTime)
<<enumeration>>
Filters
Median
Avg
Min
Max
Default

Figure 5: IOManager Component

3.1.1 IOManager

The IOManager is implemented using the Singleton pattern. The reason for this is that sensor data should be controlled by only one component, not multiple, to prevent multiple access to data streams. There is always only one IOManager present in the Smart Sensing Library. It controls all data operations related to sensor data.

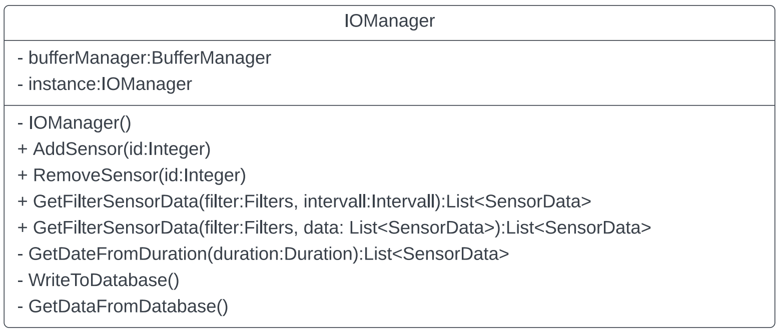

IOManager
- bufferManager:BufferManager
- instance:IOManager
- IOManager()
+ AddSensor(id:Integer)
+ RemoveSensor(id:Integer)
+ GetFilterSensorData(filter:Filters, intervall:Intervall):List<SensorData>
+ GetFilterSensorData(filter:Filters, data: List<SensorData>):List<SensorData>
- GetDateFromDuration(duration:Duration):List<SensorData>
- WriteToDatabase()
- GetDataFromDatabase()

Figure 6: IOManager

3.1.2 BufferManager

The BufferManager class functions as a cache for currently used data. All new data is first saved into its buffer. If it becomes full, the IOManager takes its current content and saves it to the database if necessary. The Buffer Manager only contains sensor data that hasn't been modified by filters, this way the data can be reused by different functions.

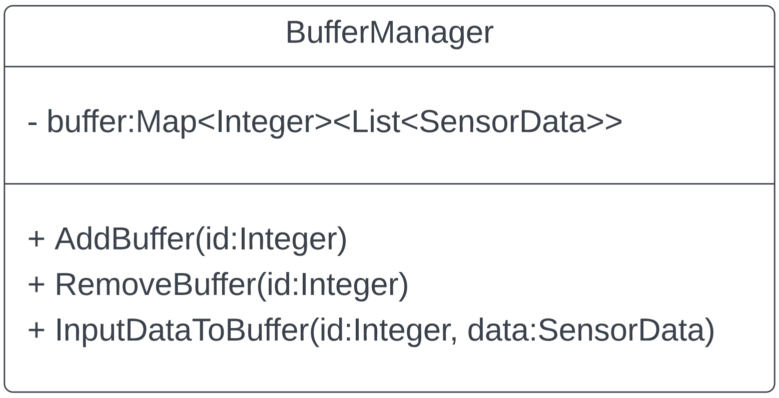

BufferManager
- buffer:Map<Integer><List<SensorData>>
+ AddBuffer(id:Integer)
+ RemoveBuffer(id:Integer)
+ InputDataToBuffer(id:Integer, data:SensorData)

Figure 7: BufferManager

3.1.3 FilterTools

The FilterTools are used to modify the data from the BufferManager by deep-copying the needed values. After the initial copy, filters can be chained to create more complex queries. The FilterTools provide the only means for accessing the data.

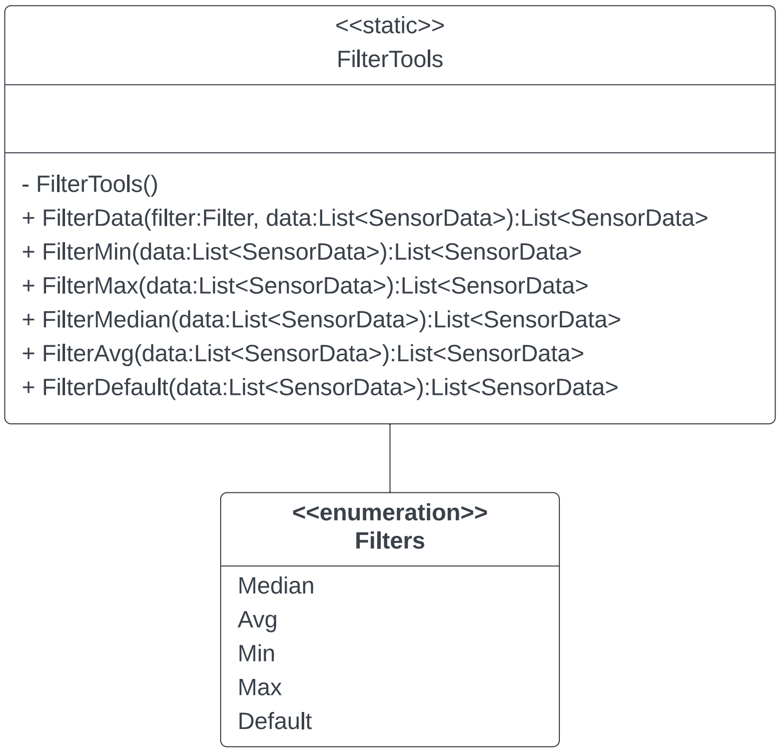

<<static>>
FilterTools
- FilterTools()
+ FilterData(filter:Filter, data:List<SensorData>):List<SensorData>
+ FilterMin(data:List<SensorData>):List<SensorData>
+ FilterMax(data:List<SensorData>):List<SensorData>
+ FilterMedian(data:List<SensorData>):List<SensorData>
+ FilterAvg(data:List<SensorData>):List<SensorData>
+ FilterDefault(data:List<SensorData>):List<SensorData>
<<enumeration>>
Filters
Median
Avg
Min
Max
Default

Figure 8: FilterTools

3.2 SensorManager

The SensorManager component serves as the interface between Dart code and platform-specific code. It offers the ability to subscribe and unsubscribe to sensor data streams, which implements the pub/sub pattern. It also preprocesses the data to ensure that Android and iOS return consistent target units. The preprocessing can be configured as needed.

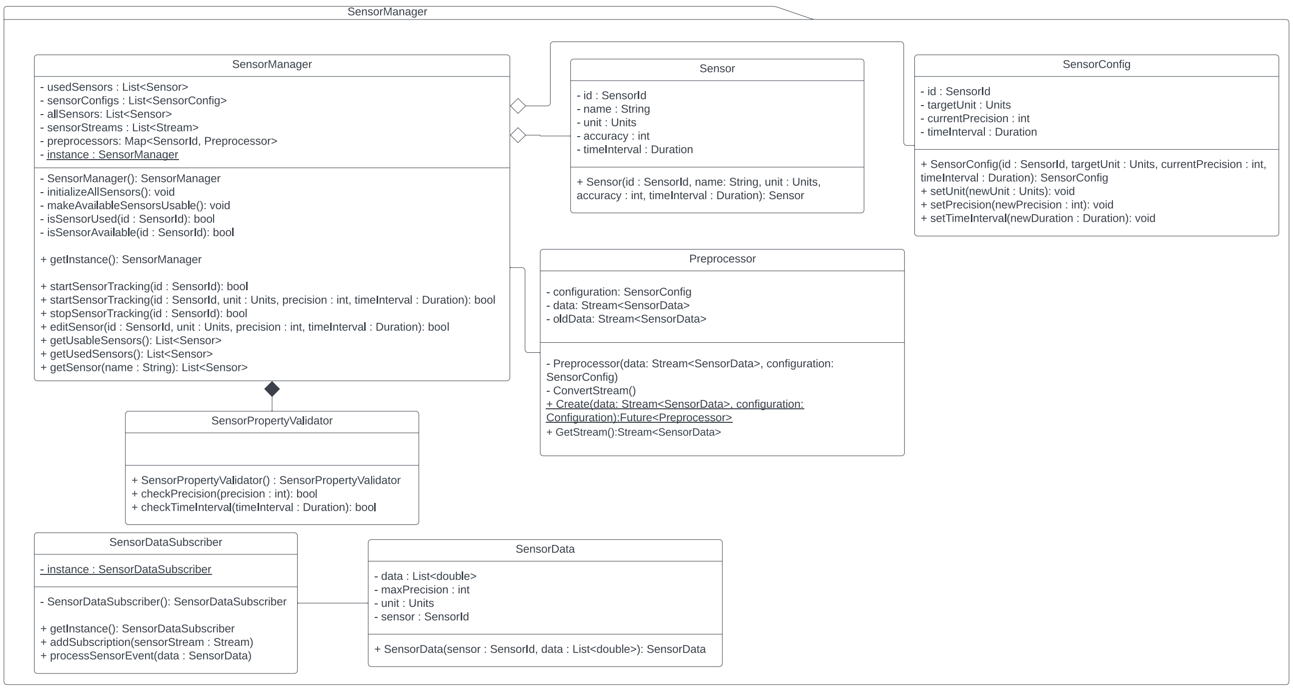

SensorManager
SensorManager
- usedSensors : List<Sensor>
- sensorConfigs : List<SensorConfig>
- allSensors: List<Sensor>
- sensorStreams : List<Stream>
- preprocessors: Map<SensorId, Preprocessor>
- instance : SensorManager
- SensorManager(): SensorManager
- initializeAllSensors(): void
- makeAvailableSensorsUsable(): void
- isSensorUsed(id : SensorId): bool
- isSensorAvailable(id : SensorId): bool
+ getInstance(): SensorManager
+ startSensorTracking(id : SensorId): bool
+ startSensorTracking(id : SensorId, unit : Units, precision : int, timeInterval : Duration): bool
+ stopSensorTracking(id : SensorId): bool
+ editSensor(id : SensorId, unit : Units, precision : int, timeInterval : Duration): bool
+ getUsableSensors(): List<Sensor>
+ getUsedSensors(): List<Sensor>
+ getSensor(name : String): List<Sensor>
Sensor
- id : SensorId
- name : String
- unit : Units
- accuracy : int
- timeInterval : Duration
+ Sensor(id : SensorId, name: String, unit : Units, accuracy : int, timeInterval : Duration): Sensor
SensorConfig
- id : SensorId
- targetUnit : Units
- currentPrecision : int
- timeInterval : Duration
+ SensorConfig(id : SensorId, targetUnit : Units, currentPrecision : int, timeInterval : Duration): SensorConfig
+ setUnit(newUnit : Units): void
+ setPrecision(newPrecision : int): void
+ setTimeInterval(newDuration : Duration): void
Preprocessor
- configuration: SensorConfig
- data: Stream<SensorData>
- oldData: Stream<SensorData>
- Preprocessor(data: Stream<SensorData>, configuration: SensorConfig)
- ConvertStream()
+ Create(data: Stream<SensorData>, configuration: Configuration):Future<Preprocessor>
+ GetStream():Stream<SensorData>
SensorPropertyValidator
+ SensorPropertyValidator() : SensorPropertyValidator
+ checkPrecision(precision : int): bool
+ checkTimeInterval(timeInterval : Duration): bool
SensorDataSubscriber
- instance : SensorDataSubscriber
- SensorDataSubscriber(): SensorDataSubscriber
+ getInstance(): SensorDataSubscriber
+ addSubscription(sensorStream : Stream)
+ processSensorEvent(data : SensorData)
SensorData
- data : List<double>
- maxPrecision : int
- unit : Units
- sensor : SensorId
+ SensorData(sensor : SensorId, data : List<double>): SensorData

Figure 9: SensorManager Component

3.2.1 SensorManager

The SensorManager class is the central control unit for the component and follows the singleton design pattern. It manages all aspects related to subscribing and unsubscribing to sensors and delegates these requests to the platform-specific code, which then returns the corresponding responses.

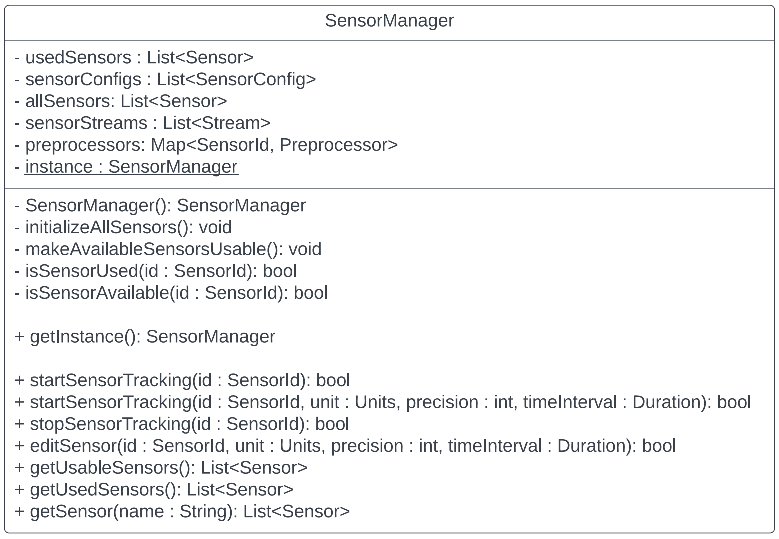

SensorManager
- usedSensors : List<Sensor>
- sensorConfigs : List<SensorConfig>
- allSensors: List<Sensor>
- sensorStreams : List<Stream>
- preprocessors: Map<SensorId, Preprocessor>
- instance : SensorManager
- SensorManager(): SensorManager
- initializeAllSensors(): void
- makeAvailableSensorsUsable(): void
- isSensorUsed(id : SensorId): bool
- isSensorAvailable(id : SensorId): bool
+ getInstance(): SensorManager
+ startSensorTracking(id : SensorId): bool
+ startSensorTracking(id : SensorId, unit : Units, precision : int, timeInterval : Duration): bool
+ stopSensorTracking(id : SensorId): bool
+ editSensor(id : SensorId, unit : Units, precision : int, timeInterval : Duration): bool
+ getUsableSensors(): List<Sensor>
+ getUsedSensors(): List<Sensor>
+ getSensor(name : String): List<Sensor>

Figure 10: SensorManager

3.2.2 Sensor

This class is representing a sensor.

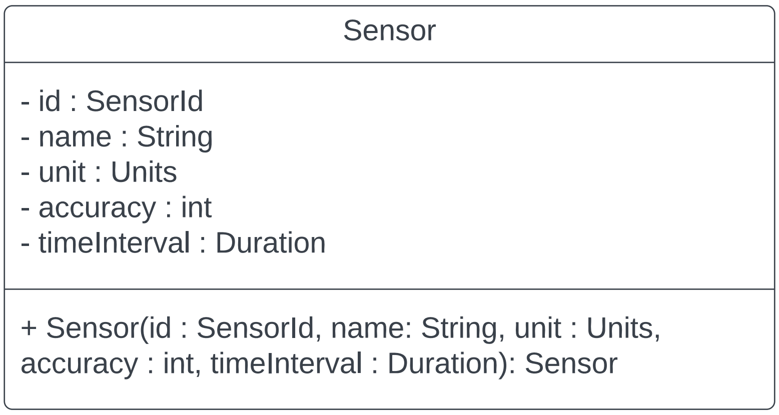

Sensor
- id : SensorId
- name : String
- unit : Units
- accuracy : int
- timeInterval : Duration
+ Sensor(id : SensorId, name: String, unit : Units, accuracy : int, timeInterval : Duration): Sensor

Figure 11: Sensor

3.2.3 SensorPropertyValidator

The SensorPropertyValidator class ensures that the specified precision and time interval are maintained.

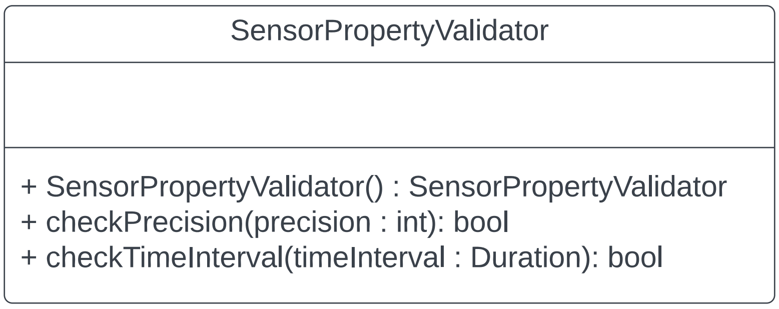

SensorPropertyValidator
+ SensorPropertyValidator() : SensorPropertyValidator
+ checkPrecision(precision : int): bool
+ checkTimeInterval(timeInterval : Duration): bool

Figure 12: SensorPropertyValidator

3.2.4 Preprocessor

The Preprocessor class ensures that Android and iOS return consistent value types. It can be configured with the SensorConfig.

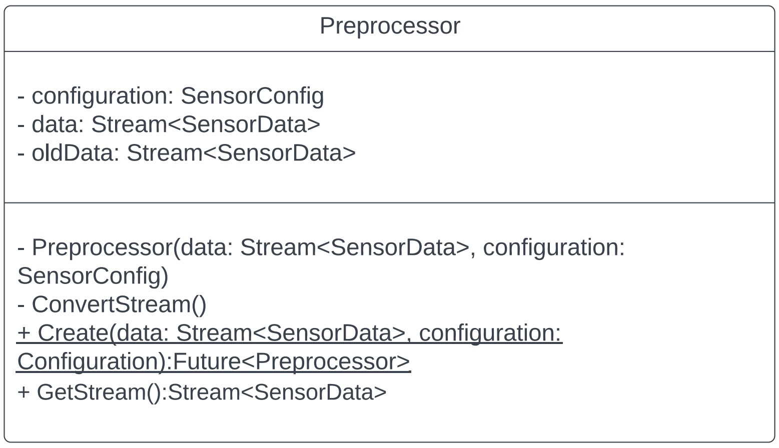
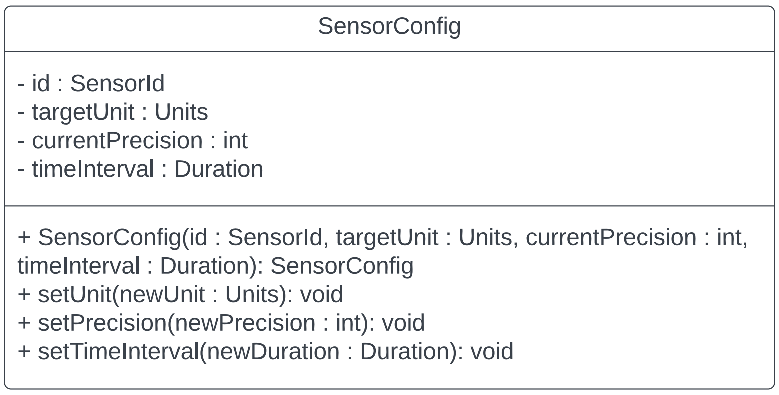

Preprocessor
- configuration: SensorConfig
- data: Stream<SensorData>
- oldData: Stream<SensorData>
- Preprocessor(data: Stream<SensorData>, configuration: SensorConfig)
- ConvertStream()
+ Create(data: Stream<SensorData>, configuration: Configuration):Future<Preprocessor>
+ GetStream():Stream<SensorData>

Figure 13: Preprocessor

3.2.5 SensorConfig

The SensorConfig class holds various configurations for the preprocessor, including the time interval, precision, and target unit.


SensorConfig

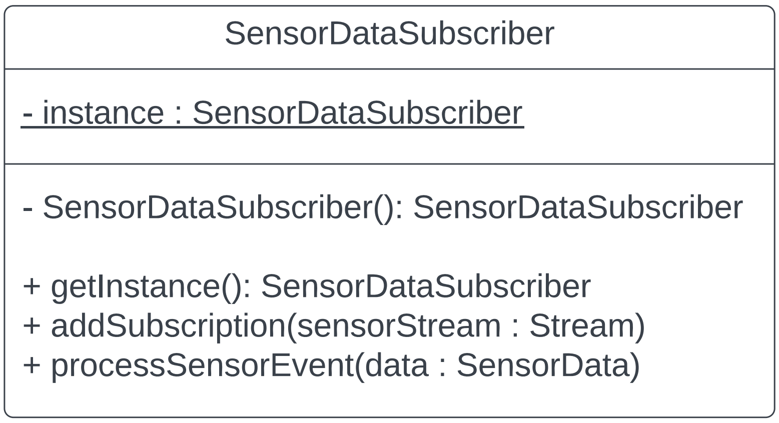
- id : SensorId
- targetUnit : Units
- currentPrecision : int
- timeInterval : Duration

+ SensorConfig(id : SensorId, targetUnit : Units, currentPrecision : int, timeInterval : Duration): SensorConfig
+ setUnit(newUnit : Units): void
+ setPrecision(newPrecision : int): void
+ setTimeInterval(newDuration : Duration): void

Figure 14: SensorConfig

3.2.6 SensorDataSubscriber

The SensorDataSubscriber class manages multiple subscriptions and is utilized by the Sensor-Manager class.


SensorDataSubscriber

- instance : SensorDataSubscriber

- SensorDataSubscriber(): SensorDataSubscriber

+ getInstance(): SensorDataSubscriber
+ addSubscription(sensorStream : Stream)
+ processSensorEvent(data : SensorData)

Figure 15: SensorDataSubscriber

3.2.7 SensorData

The SensorData class represents the data received from a sensor and serves as the data model for the document-based database.

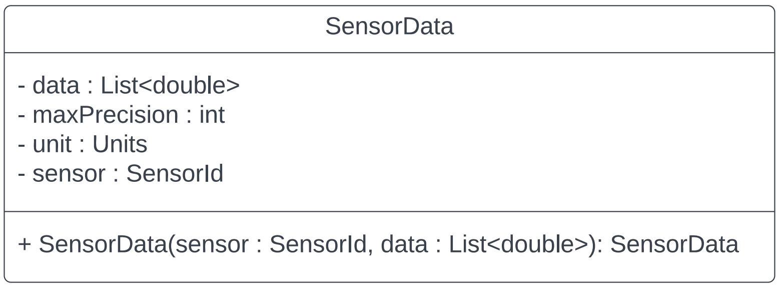


Figure 16: SensorData

4 Processes

4.1 Sequence Diagrams

This section contains sequence diagrams for retrieving and transforming data.

4.1.1 Starting a Sensor

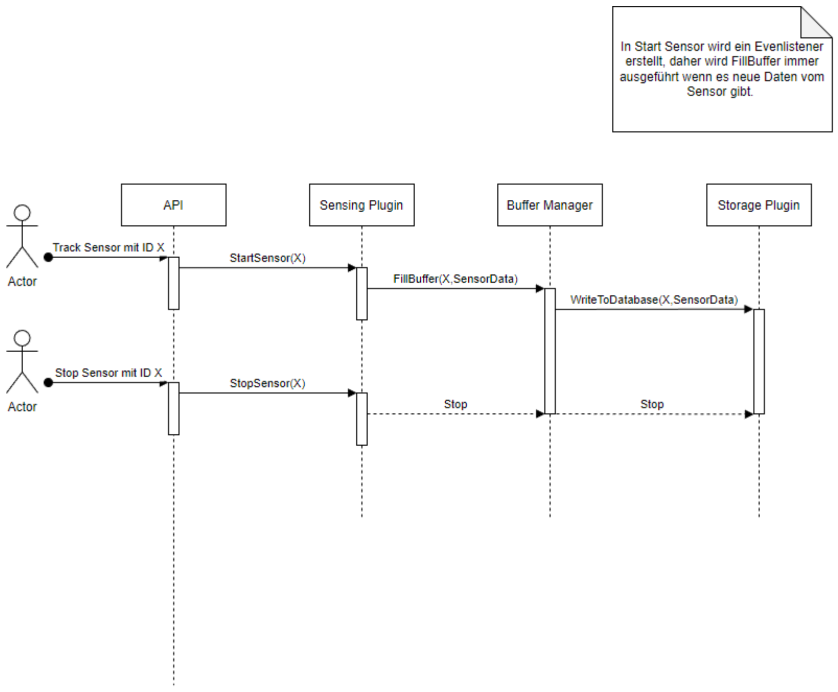


Figure 17: Starting sequence diagram

4.1.2 Getting Data via a filter

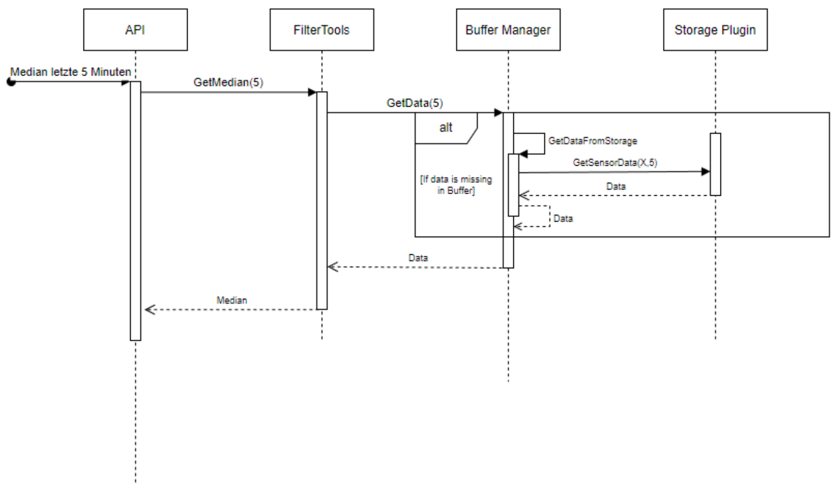


Figure 18: Filtering sequence diagram