Entwurf

PhyPiGUI

Datenerfassung und -visualisierung mit einem Raspberry Pi

Version 1.0

Simon Essig *2115219*
Christian Hauser *1933900*
Fritz Hund *2115764*
Ahmad Jayossi *1950794*
Sandro Negri *2061545*

12. Juli 2020

Inhaltsverzeichnis

1 Architektur

1.1 Model-View Architektur

Die grundsätzliche Architektur des Programms baut auf dem Konzept des Model-View-Controller(MVC) auf. Diese beschreibt eine getrennte Steuerung(Model), Darstellung(View) und Benutzerinteraktion(Controller) innerhalb eines Programms.

Für die Benutzeroberfläche wurde das Pyhton GBO-Rahmenwerk PyQt5 gewählt. Dieses unterstützt aber nicht die MVC-Architektur direkt, sondern benutzt eine Abweichung davon, die sich nur Model-View(MV) nennt und den View- und Controller-Teil in eins zusammen führt.

Die Hauptüberlegung bei diesem Konzept ist, dass der Model-Teil nichts über den View weiß und das View sich nur über das Beobachter-Entwurfsmodell am Model anmeldet und so vom View über Änderungen benachrichtigt wird.

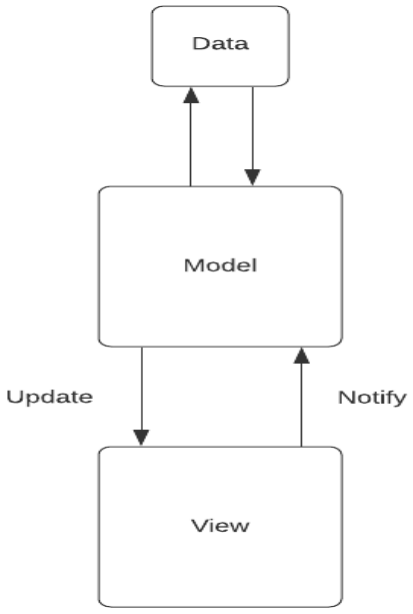

Abbildung 1: Die Modell-View-Architektur

Diese Trennung soll die Implementierung und das Testen eines Programms erleichtern und es einfach machen, die Benutzeroberfläche auszutauschen. Außerdem ermöglicht sie mehr Flexibilität in der Darstellung der Benutzeroberfläche.

Wir haben und aus folgenden Gründen für PyQt5 entschieden:

- PyQt5 ist eine Python-Übersetzung von Qt5, ein weit verbreitetes und gut dokumentiertes GBO-Rahmenwerk für C++. Sie funktioniert gleich wie Qt5 und besitzt somit auch alle Vorteile davon.

- Qt5 bietet eine sehr mächtige und gleichzeitig äußerst saubere und konsistente Schnittstelle, die sich schon lange bewährt hat.

- Die MV Architektur ist in PyQt5 einfach umzusetzen.

- Qt5 unterstützt sowohl Windows als auch viele Linux Distributionen inklusive Raspbian und ist somit einfach auch ohne Raspberry Pi zu testen.

- Weiter ist es auch für Einsteiger in die GBO Programmierung geeignet und hat eine vergleichsweise schnelle Einlesezeit, was uns voraussichtlich mehr Zeit für den andere wichtige Aspekte unseres Programms lässt.

1.1.1 Model

Grundlegende Konzepte:

In der Modell-/Viewarchitektur stellt das Modell eine Standardschnittstelle zur Verfügung, über die Ansichten auf Daten zugreifen können. In Qt wird die Standardschnittstelle durch die Klasse QAbstractItemModel definiert. Unabhängig davon, wie die Datenelemente in einer zugrunde liegenden Datenstruktur gespeichert sind, stellen alle Unterklassen von QAbstractItemModel die Daten als hierarchische Struktur mit Tabellen von Elementen dar. Views verwenden diese Konvention, um auf Datenelemente im Modell zuzugreifen, aber sie sind nicht in der Art und Weise eingeschränkt, wie sie diese Informationen dem Benutzer präsentieren. Modelle benachrichtigen auch alle angehängten Ansichten über Änderungen an Daten durch den Signal- und Zeitschlitzmechanismus.

Alle Itemmodelle basieren auf der Klasse QAbstractItemModel. Diese Klasse definiert eine Schnittstelle, die von Views für den Zugriff auf Daten verwendet wird. Die Daten selbst müssen nicht im Modell gespeichert werden. Stattdessen können sie in einer Datenstruktur oder einem Repository gespeichert werden, welche von einer separaten Klasse, einer Datei, einer Datenbank oder einer anderen Anwendungskomponente bereitgestellt werden.

Wenn die Standardmodelle den Anforderungen nicht entsprechen, können die Unterklassen QAbstractItemModel, QAbstractListModel oder QAbstractTableModel zur Erstellung eigener Modelle verwendet werden.

1.1.2 View

In der Modell-/Viewarchitektur bezieht die View Datenelemente aus dem Modell und präsentiert sie dem Benutzer. Die Art und Weise, wie die Daten präsentiert werden, muss nicht der Darstellung der vom Modell bereitgestellten Daten entsprechen und kann sich völlig von der zugrunde liegenden Datenstruktur unterscheiden, die zur Speicherung von Datenelementen verwendet wird.

Die Trennung von Inhalt und Präsentation wird durch die Verwendung einer Standardmodellschnittstelle von QAbstractItemModel, einer Standardansichtsschnittstelle von QAbstractItemView und die Verwendung von Modellindizes erreicht, die Datenelemente allgemein darstellen. Views verwalten typischerweise das Gesamtlayout der aus Modellen gewonnenen Daten.

1.2 Model

Das Model ist in vier grundlegende Module **Model_Item**, **Model_Config**, **Model_Manager** und **Model_Workspace** unterteilbar.
Ziel des Entwurfs war es, es dem Programmierer so einfach wie möglich zu machen neue Elemente hinzuzufügen ohne viel Code schreiben oder ändern zu müssen. Dafür sind insbesondere *Model_Item* und *Model_Config* verantwortlich. Während *Model_Item* dafür verantwortlich ist, den Aufbau aller Elemente im *Model* zu strukturieren und so zu abstrahieren, dass es dem Programm möglich ist, alles außer die Grundfunktionen eines Elements generisch erstellen und handhaben zu können, ist die *Model_Config* für einen wichtigen Teil der Elemente zuständig. Nämlich ihre einstellbaren Optionen.
Die *Model_Config* stellt eine Grundstruktur zur Verfügung, bestehend aus mehreren Optionsklassen, die alle einen eignen abstrakten Typ von Optionen darstellen, und einer Konfigurationsklasse zum verwalten aller Optionen eines Elements. Dadurch ist es dem Programmierer erlaubt, alle Optionen, die ein Element aufweist in dem Konstruktor seiner *Model-Klasse* zu definieren ohne in das *View* in irgendeiner Weise eingreifen zu müssen. Das *View* kann sich durch den Aufbau der *Model_Config* generisch sein Einstellungsfenster bauen. Es ist auch durchaus möglich weitere Optionsklassen zu implementieren, es muss dann nur die nötige Konfigurationsklasse und die Element-Klasse im *Model* angepasst werden. Außerdem muss im *View* definiert werden, wie die neue Optionsklasse graphisch darzustellen ist.
Das *Model_Workspace* ist dafür da, die Verbindungen zwischen Elementen im *Model* zu verwalten. So weißt er beispielsweise jedem Eingang und Ausgang eines jeden Elements auf der Arbeitsfläche eine feste ID zu und im *Model_Item* wird nur noch mit den IDs gearbeitet, um unnötige Referenzen zu vermeiden. Auch aus Sicherheitsgründen, um so gekapselt wie möglich arbeiten zu können.
Der *Model_Manager* ist das Herzstück des *Models*. Seine Aufgabe ist es Daten zu holen und Daten zur Verarbeitung weiterzugeben. Beim starten eines Testdurchlaufes erstellt er für jedes Element Funktionen, die mit Sensordaten die Werte an ihren Ausgängen berechnen können. Er fragt in einem festen Takt die Sensor-Elemente nach neuen Messdaten ab und puffert diese einen Takt lang. Dann startet er das Berechnen der Endwerte, die bei zum Beispiel Diagramm-Elementen ankommen und visualisiert werden sollen. Der *Model_Manager* ist außerdem der Hauptansprechpartner für das *View* in dem Sinne, dass das *View* hauptsächlich auf ihn zugreift und ihn startet und stoppt. Er benachrichtigt auch die Beobachter der *Model-Elemente* über, für das *View* wichtige, Änderungen.
Die Berechnung der Werte im Manager für jedes Element wird mit Lambda-Funktionen verwirklicht, wobei jedes Element seine Lambda-Funktionen, zur Berechnung der an den Ausgängen anliegenden Werten, rekursiv mithilfe den Funktionen der verbundenen Elemente. Jede Lambda-Funktion hat als Eingabewert eine Hash-Map von *SensorModel* und einer Liste von *float*, die die gelesenen Werte der Sensoren beschreibt und gibt einen Wert des Typs *float* zurück.

So ist das *Model* ganz und gar unabhängig von dem *View* implementiert und weiß von seiner Anwesenheit nichts. Dies ermöglicht es das *View* auszutauschen ohne das Model anzutasten.

1.2.1 Butler

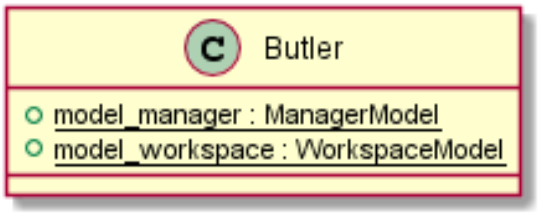

Abbildung 2: Die Klasse Butler

Butler ist eine Klasse, die als Ansprechpartner dient für, sowohl Komponenten aus dem *Model*, als auch aus dem *View*. Sie stellt zwei statische Attribute zur Verfügung, deren Funktionen von überall aus genutzt werden können.

model_manager : ManagerModel ist eine Referenz auf ManagerModel.

model_workspace : WorkspaceModel ist eine Referenz auf WorkspaceModel.

1.2.2 Elemente

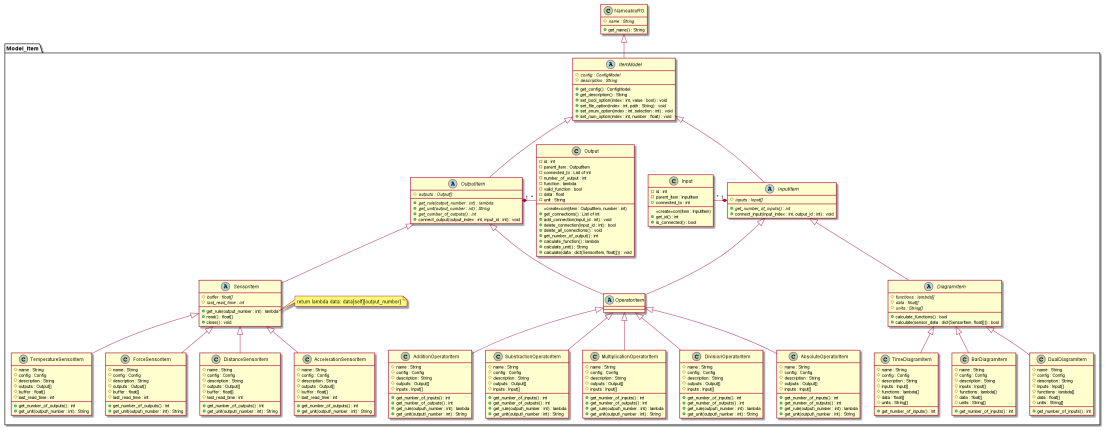

Abbildung 3: Das Paket Item

Das **Model_Item** ist in zwei Arten von Elementen abstrahiert. Einmal die Elemente die Eingänge haben und zum anderen, die Elemente die Ausgänge haben. Die drei Hauptklassen von Elementen (Sensor-Elemente, Operatoren, Diagramm-Elemente) erben dann jeweils von den zwei Artenklassen. So können Elemente als Unterklassen der jeweiligen Hauptklasse implementiert werden. Eine weitere Abstraktion sind die Klassen, die Eingänge und Ausgänge repräsentieren. Da in der Regel der Datenstrom, der ein Element verlässt, kontextabhängig von seinem Ausgang und nicht von seinem Eingang ist, werden die möglicherweise unterschiedlichen Lambda-Funktionen, Einheiten und andere wichtige Daten für den Datenstrom in den *Output* Klassen gespeichert. Nur Diagramm-Elemente verfahren etwas anders und daher sind in ihnen zugehörig zu den *Input* Klassen alle Daten für den Datenstrom gespeichert.(MK9)

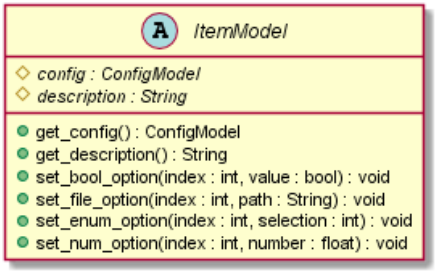

Abbildung 4: Die Klasse ItemModel

Die abstrakte Klasse **ItemModel** enthält alle Komponenten, die jedes Element im *Model* enthalten muss.

config : Config speichert die Konfiguration des Elements, in der alle möglichen Optionen und deren Werte für dieses gespeichert sind.

description : String speichert eine Beschreibung des Elements.

get_config() : Config gibt eine Kopie von **config** zurück.

get_destciption() : String gibt die Zeichenkette in **description** zurück.

set_bool_option(index : int, enabled : bool) : void greift über **config** auf die Liste *bool_options* zu und setzt den Wert der Option an Index *index* auf den Wert *enabled*.

set_file_option(index : int, path : String) : void greift über **config** auf die Liste *file_options* zu und setzt den Pfad der Option an Index *index* auf den Wert *path*.

set_enum_option(index : int, selection : int) : void greift über **config** auf die Liste *enum_options* zu und setzt das Attribut *selection* der Option an Index *index* auf den Wert des Parameters *selection*.

set_num_option(index : int, number : float) : void greift über **config** auf die Liste *num_options* zu und setzt den Wert der Option an Index *index* auf den Wert *number*.

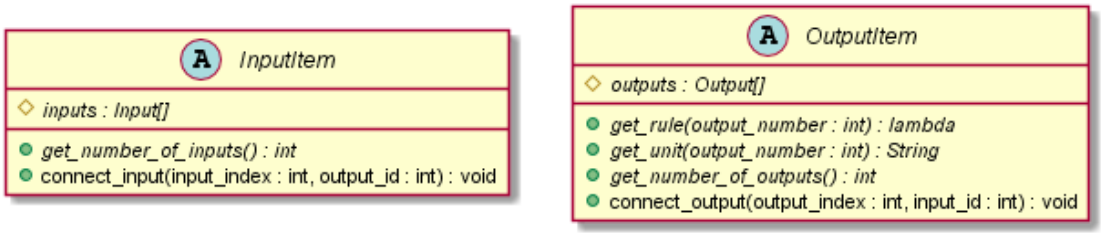

Abbildung 5: Die Klassen InputItem und OutputItem

Die Klasse **InputItem** ist der Repräsentant eines Elements mit Eingängen und **OutputItem** für ein Element mit Ausgängen. Jedes Element im *Model* muss von einer dieser Beiden Klassen erben. *OperatorItem* erbt von beiden Klassen, da diese Elemente sowohl Eingänge als auch Ausgänge hat.

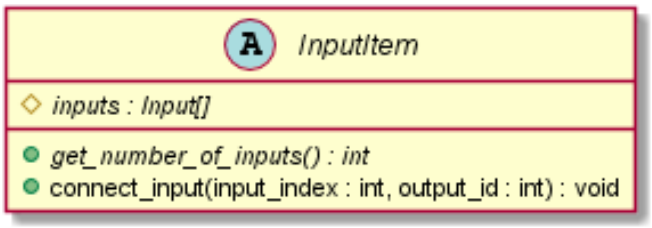

Abbildung 6: Die Klasse InputItem

inputs : Input[] ist ein Array von *Input* Objekten. Seine Größe ist in den Unterklassen von **InputItem** fest definiert, da sich diese nicht ändert. Die Größe wird im Konstruktor der Unterklasse festgelegt.

get_number_of_inputs() : int gibt die Größe des Arrays **inputs** zurück.

connect_input(input_index : int, output_id : int) : void ruft auf dem *Input* an Stelle *input_index* in **inputs** die Methode *get_id()* auf um die *input_id* zu bekommen. Dann wird *connect(input_id, output_id) : void* über *model_workspace* aus *Butler* aufgerufen.

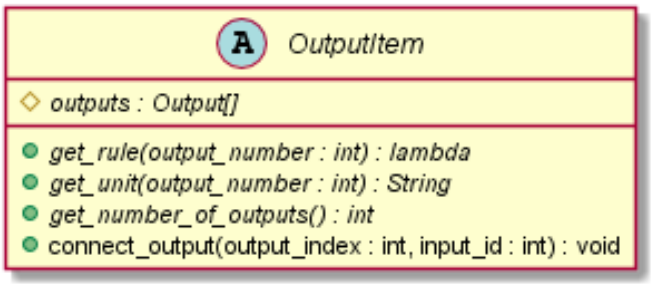

Abbildung 7: Die Klasse OutputItem

outputs : Output[] ist ein Array von *Output* Objekten. Seine Größe ist in den Unterklassen von **OutputItem** fest definiert, da sich diese nicht ändert. Die Größe wird im Konstruktor der Unterklasse festgelegt.

get_rule(output_number: int) : lambda wird in den Unterklassen angefasst auf die Funktionalität, die jeder *Output* eines Elements aufweist. So kann für jeden *Output* eine eigene Funktionalität implementiert werden. *output_number* bestimmt die zurückgegebene Funktion für den *Output* an Stelle *output_number*. Die Lambda-Funktionen für *OperatorItem* werden jeweils berechnet, durch einen Aufruf von *calculate_functions(output_id) : bool* auf *model_workspace* aus *Butler*. *output_id* wird sich jeweils von den, für den *Output* wichtigen, *Inputs* geholt und stellt die ID des *Outputs*, der mit dem *Input* verbunden ist dar. Die so geholten Funktionen werden dann verknüpft und zurückgegeben. Bei Sensoren wird eine Lambda-Funktion zurückgegeben, die aus ihrem Parameter das Daten-Array, für die die eigene Klasse der Key ist, holt und den Wert an Stelle *output_number* zurückgibt.

get_unit(output_number : int) : String wird von den Unterklassen überschrieben und dient zur Berechnung der Einheit eines jeden Ausgangs.

get_number_of_outputs() : int gibt die Größe des Arrays **outputs** zurück.

connect_output(output_index : int, input_id : int) : void ruft auf dem *Output* an Stelle *output_index* in **outputs** die Methode *get_id()* auf um die *output_id* zu bekommen. Dann wird *connect(input_id, output_id) : void* über *model_workspace* aus *Butler* aufgerufen.

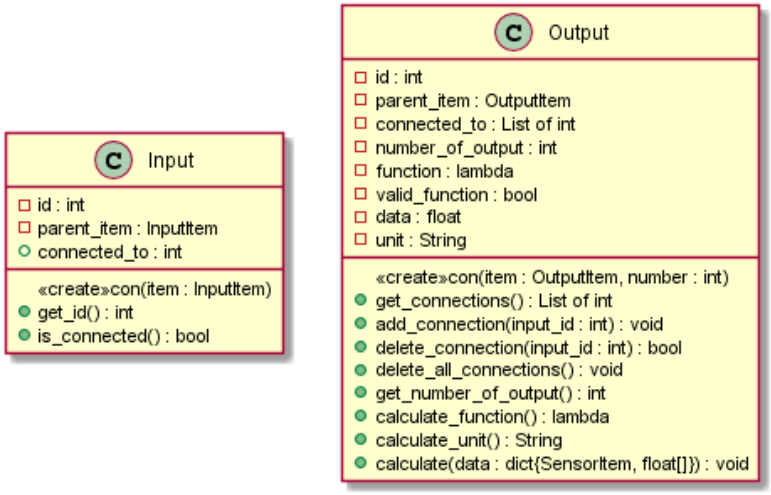

Abbildung 8: Die Klassen Input und Output

Input und **Output** repräsentieren jeweils immer einen Eingang eines Elements und einen Ausgang. In erster Linie ist **Input** sehr stark mit *InputItem* und **Output** mit *OutputItem* gekoppelt. Jeder **Input** und **Output** hat eine eindeutige ID und wird neben der zugehörigen *Item-Klasse* auch noch in *model_ workspace* aus *Butler* gespeichert.

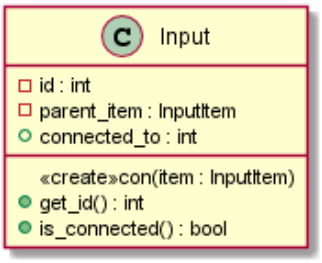

Abbildung 9: Die Klasse Input

id : int ist die ID des **Inputs**. Sie wird im Konstruktor gesetzt und danach nicht mehr geändert.

parent_item : InputItem speichert eine Referenz auf das *InputItem* zu dem dieser **Input** gehört.

connected_to : int speichert die ID des *Outputs* mit dem dieser **Input** verbunden ist. Der Wert von **connected_to** ist -1, wenn der **Input** mit keinem *Output* verbunden ist.

con(item : InputItem) initialisiert **parent_item** mit *item* und ruft über *model_workspace* aus *Butler* die Methode *add_input()* mit sich selbst als Parameter auf. Mit dem Rückgabewert von *add_input()* wird **id** initialisiert.

get_id() : int gibt den Wert in **id** zurück.

is_connected() : bool gibt *False* zurück, wenn **connected_to** -1 ist, ansonsten *True*.

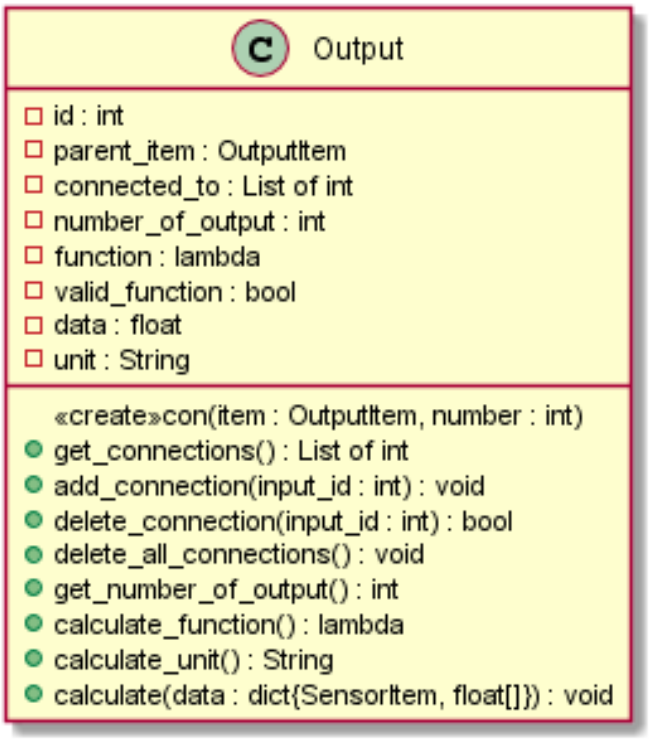

Abbildung 10: Die Klasse Output

id : int ist die ID des **Outputs**. Sie wird im Konstruktor gesetzt und danach nicht mehr geändert.

parent_item : OutputItem speichert eine Referenz auf das *OutputItem* zu dem dieser **Output** gehört.

connected_to : List of int speichert die IDs der *Inputs* mit dem dieser **Output** verbunden ist. **connected_to** ist eine leere Liste, wenn der **Output** mit keinem *Input* verbunden ist.

number_of_output : int speichert den Index des **Output**. Das Attribut wird von dem **parent_item**, zu dem er der **Output** gehört, über seinen Konstruktor gesetzt.

function : lambda speichert die Lambda-Funktion, die bis zu diesem **Output** aufgebaut wurde.

data : float speichert den zuletzt berechneten Wert des **Outputs** durch **calculate()**.

unit : String speichert die zuletzt berechnete Einheit durch die Methode **calculate_unit()**.

con(item : OutputItem, number : int) initialisiert **parent_item** mit *item* und **number_of_output** mit number. Dann wird über *model_workspace* aus *Butler* die Methode *add_output()* mit sich selbst als Parameter aufgerufen. Mit dem Rückgabewert von *add_output()* wird **id** initialisiert.

get_connections() : List of int gibt eine Kopie der Liste, die in **connected_to** gespeichert wird zurück.

add_connection(input_id : int) : void fügt *input_id* in die Liste **connected_to** ein.

delete_connection(input_id : int) : bool entfernt *input_id* aus der Liste **connected_to**.

delete_all_connections() : void ersetzt **connected_to** durch eine leere Liste von int. Die Methode sollte ausschließlich aus dem *model_workspace* aus *Butler* aufgerufen werden, da man sich auch noch um die *Inputs* kümmern muss, die auf diesen *Output* zeigen.

get_number_of_output() : int gibt den Wert in **number_of_output** zurück.

calculate_function() : lambda ruft *get_rule(**number_of_output**)* auf **parent_item** auf und die Rückgabe wird in **function** gespeichert. Anschließend wird **function** zurückgegeben.

calculate_unit() : String ruft auf **parent_item** die Methode *get_unit(**number_of_output**)* auf und speichert deren Wert in **unit**. Anschließend wird **unit** zurückgegeben.

calculate(data : dictSensorItem, float[]) : void setzt den Parameter *data* in die von ihm gespeicherte Funktion **function** als Parameter ein und speichert das Ergebnis in **data**.

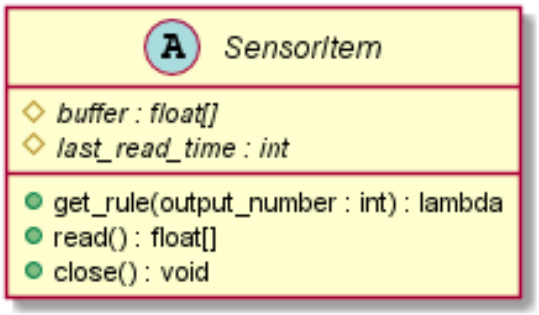

Abbildung 11: Die Klasse SensorItem

SensorItem ist in der Regel die Oberklasse für alle Elemente, die keinen Eingang haben und Daten, an ein mit dem Ausgang verbundenes Element, senden. Jedes Sensor-Element importiert seine zugehörige Konfigurationsklasse aus dem Rahmenwerk PhyPiDAQ, um die benötigte Konfiguration und Schnittstelle des physischen Sensors zu erhalten. **SensorItem** erbt nur von *OutputItem*.

buffer : float[] puffert die zuletzt ausgelesenen Messwerte und wird verwendet, falls die Abfragerate vom Manager höher ist, als die virtuell Ausleserate. Jeder Eintrag entspricht einem *Output* des *SensorItems*.

last_read_time : int speichert den Zeitstempel des letzten Aufrufs von **read()**.

get_rule(output_number : int) : lambda gibt eine Lambda-Funktion zurück, die in ihren Parameter das eigene Objekt von **SensorItem** selbst als Key einsetzt und das zugehörige Daten-Array erhält. Von diesem gibt sie den Eintrag an der Stelle *output_number* zurück.

read() : float[] vergleicht mit Hilfe von **last_read_time**, wie lange der letzte **read()** Aufruf her ist. Ist die Zeitdifferenz kleiner, als die in den Optionen eingestellte Ausleserate vom **SensorItem**, so gibt die Methode **buffer** zurück. Ansonsten ruft die Methode aus der Konfigurationsklasse des Sensors die Methode *acquireData* auf und übergibt eine Liste zur Speicherung der vom physischen Sensor eingelesenen Daten. Die Werte in der Liste schreibt die Methode in ein Array und speichert dieses in **buffer**. Die Reihenfolge der Werte im Array muss übereinstimmen mit der Nummerierung der *Outputs* eines Sensors. Dann speichert die Methode den aktuellen Zeitstempel in **last_read_time** und gibt den Wert in **buffer** zurück.

close() : void wird benötigt, um die Verbindung zum physischen Sensor zu trennen. Diese Methode wird beim löschen des Elements aufgerufen.

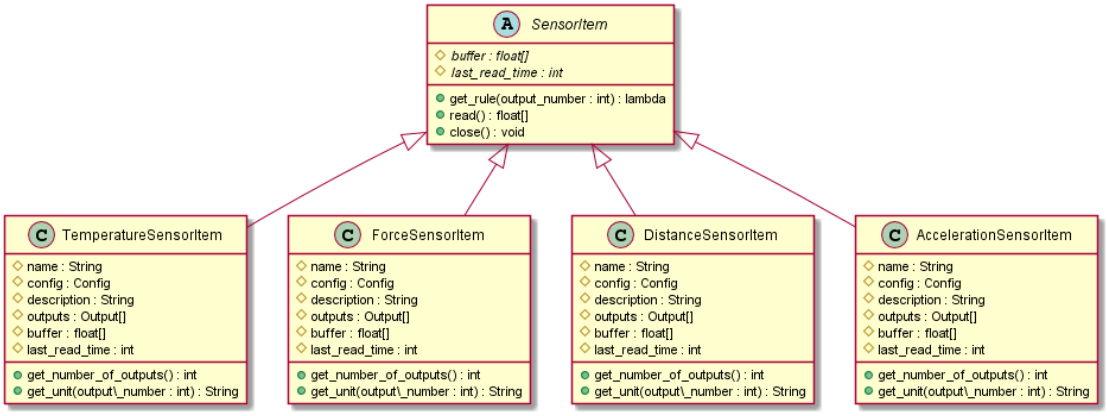

Abbildung 12: Die Unterklassen von SensorItem

Jede Unterklasse von *SensorItem* repräsentiert das *Model* eines Sensor-Elements und gehört damit zu genau einem Hardware-Sensor. Die Klassen implementieren die abstrakten Attribute aus ihren Oberklassen in ihrem Konstruktor. Um außerhalb der Klasse die Attribute lesen zu können, werden die *get...-Methoden* vererbt aus den Oberklassen. Im Konstruktor der Unterklassen werden die Attribute **name**, **config**, **description**, **outputs**, **buffer** und **last_read_time** gesetzt.

TemperatureSensorItem

name = Temperatursensor

description = Der Temperatursensor misst die Temperatur in Grad oder Kelvin.

last_read_time = 0

buffer wird mit einem leeren Array der Größe 1 initialisiert.

config: Für **config** wird ein neues Config-Objekt erstellt. Dann wird ein *EnumOption* und ein *NumOption* Objekt erstellt. Sie werden mit den Konstruktoren *con(Einheit, (Kelvin, Celsius) : Enum)* und *con(Ausleserate, 100)* erstellt. Über *add_enum_option()* und *add_num_option()* werden die Optionen zu **config** hinzugefügt.

outputs: Für **outputs** wird ein *Output* Objekt erstellt und in ein Array der Größe 1 gespeichert. Dieses Array wird **outputs** zugewiesen.

get_number_of_outputs() gibt die Zahlt *1* zurück.

get_unit(output_number) prüft für *output_number* = 0 den Wert des Arrays in *enum_options[0]* in **config** an der Stelle *selection* aus *enum_options*. Ist dieser

Wert gleich 'Kelvin', so wird K zurückgegeben oder wenn dieser 'Celsius' ist, wird $°C$ zurückgegeben.

ForceSensorItem

name = Kraftsensor

description = Der Kraftsensor misst die Kraft in Newton.

last_read_time = 0

buffer wird mit einem leeren Array der Größe 1 initialisiert.

config: Für **config** wird ein neues Config-Objekt erstellt. Dann wird ein *NumOption* Objekt erstellt mit dem Konstruktor *con(Ausleserate, 100)*. Über *add_num_option()* wird die Option zu **config** hinzugefügt.

outputs: Für **outputs** wird ein *Output* Objekt erstellt und in ein Array der Größe 1 gespeichert. Dieses Array wird **outputs** zugewiesen.

get_number_of_outputs() gibt die Zahlt *1* zurück.

get_unit(output_number) gibt für *output_number* = 0 den String *N*.

DistanceSensorItem

name = Distanzsensor

description = Der Distanzsensor misst die Distanz eines Objektes.

last_read_time = 0

buffer wird mit einem leeren Array der Größe 1 initialisiert.

config: Für **config** wird ein neues Config-Objekt erstellt. Dann wird ein *NumOption* Objekt erstellt mit dem Konstruktor *con(Ausleserate, 100)*. Über *add_num_option()* wird die Option zu **config** hinzugefügt.

outputs: Für **outputs** wird ein *Output* Objekt erstellt und in ein Array der Größe 1 gespeichert. Dieses Array wird **outputs** zugewiesen.

get_number_of_outputs() gibt die Zahlt *1* zurück.

get_unit(output_number) gibt für *output_number* = 0 den String *m*.

AccelerationSensorItem

name = Beschleunigungssensor

description = Der Beschleunigungssensor misst die Beschleunigung in 3 Richtung x,y,z.

last_read_time = 0

buffer wird mit einem leeren Array der Größe 3 initialisiert.

config: Für **config** wird ein neues Config-Objekt erstellt. Dann wird ein *NumOption* Objekt erstellt mit dem Konstruktor *con(Ausleserate, 100)*. Über *add_num_option()* wird die Option zu **config** hinzugefügt.

outputs: Für **outputs** werden drei *Output* Objekte erstellt und dieser Reihenfolge in ein Array der Größe 3 gespeichert. Dieses Array wird **outputs** zugewiesen.

get_number_of_outputs() gibt die Zahlt *1* zurück.

get_unit(output_number) gibt für *output_number* = 0 oder 1 oder 2 den String *m*s^2*.

Abbildung 13: Die Klasse OperatorItem

OperatorItem stellt die gemeinsame Oberklasse für alle Operatoren dar oder allgemein für alle Elemente, die sowohl Eingänge als auch Ausgänge haben und einen Datenstrom verarbeiten. Deswegen erbt **OperatorItem** sowohl von *InputItem*, als auch von *OutputItem*.

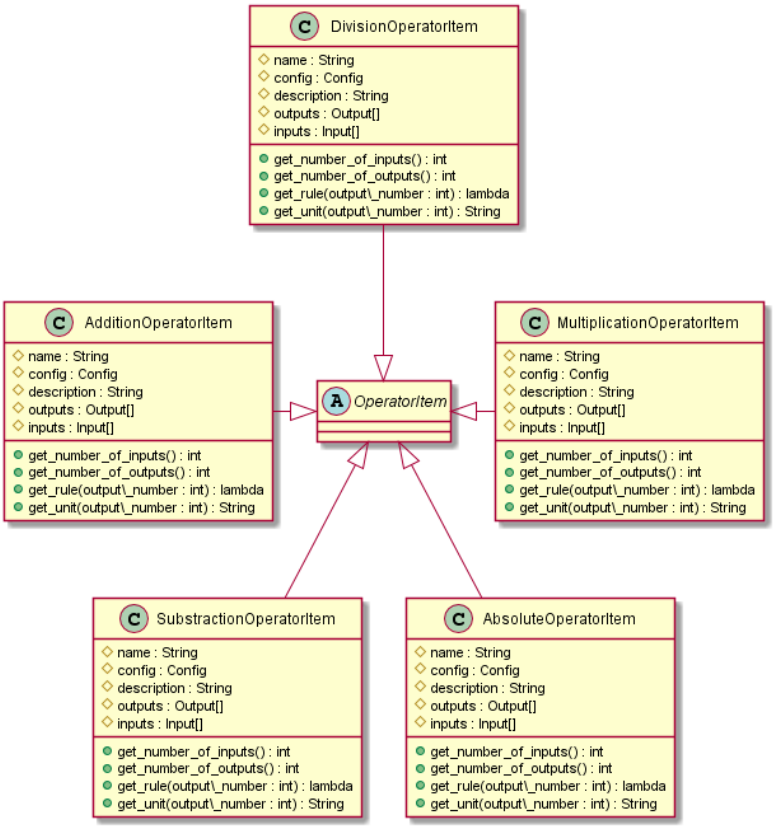

Abbildung 14: Die Unterklassen von OperatorItem

Jede Unterklasse von *OperatorItem* repräsentiert das *Model* eines Operator-Elements und übernimmt die Verarbeitung von Daten. Die Klassen implementieren die abstrakten Attribute aus ihren Oberklassen in ihrem Konstruktor. Um außerhalb der Klasse die Attribute lesen zu können, werden die *get...-Methoden* vererbt aus den Oberklassen. Im Konstruktor der Unterklassen werden die Attribute **name**, **config**, **description**, **outputs** und **inputs** gesetzt.

AdditionOperatorItem

name = Plusoperator

description = Dieser Operator addiert zwei Werte.

config: Für **config** wird ein neues Config-Objekt erstellt.

outputs: Für **outputs** wird ein *Output* Objekt erstellt und in ein Array der Größe 1 gespeichert. Dieses Array wird **outputs** zugewiesen.

outputs: Für **inputs** werden zwei *Input* Objekter erstellt und in dieser Reihenfolge ein Array der Größe 2 gespeichert. Dieses Array wird **inputs** zugewiesen.

get_number_of_outputs() gibt die Zahlt *1* zurück.

get_number_of_inputs() gibt die Zahlt *2* zurück.

get_rule() holt sich für jeden Eintrag in **inputs** die ID *connected_to* und ruft mit dieser über *model_workspace* in *Butler* die Methode *calculate_function()* auf. Die zurückgegebenen Lambda-Funktionen werden mit einem „+“ verknüpft und in einer neuen Lambda-Funktion gespeichert. Diese Funktion setzt ihren Parameter in die beiden verknüpften Funktionen ein. Diese Funktion wird nun zurückgegeben.

get_unit(output_number) holt sich für jeden Eintrag in **inputs** die ID *connected_to* und ruft mit dieser über *model_workspace* in *Butler* die Methode *calculate_unit()* auf. Die zurückgegebenen Werte r1 und r2 werden wie folgt verknüpft und zurückgegeben. „(r1) + (r2)“

Die anderen Operatoren werden Analog zu diesem erstellt.

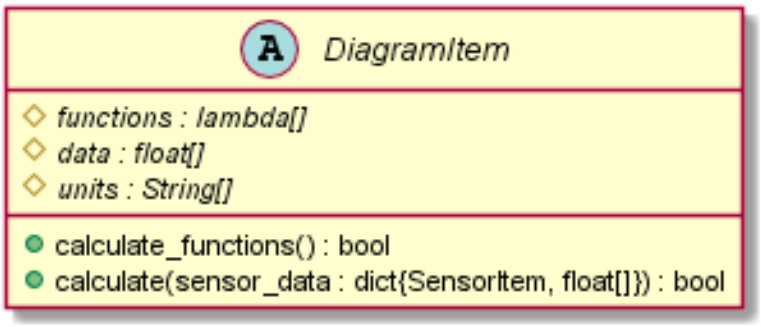

Abbildung 15: Die Klasse DiagramItem

DiagramItem stellt eine Oberklasse dar für alle Elemente, die nur Eingänge haben und als Endpunkt für einen Datenstrom gesehen werden. Dadurch haben die Unterklassen dieser Klasse besondere Attribute, die nicht jedes *InputItem* benötigt. **DiagramItem** erbt nur von *InputItem*.

functions : lambda[] speichert ein Array fester Größe von Lambda-Funktionen. Jede Funktion darin entspricht genau einem *Input* des **DiagramItems** und steht für den ankommenden Datenstrom. Die Reihenfolge der Funktionen im Array müssen mit der Nummerierung der *Inputs* übereinstimmen.

data : float[] puffert zu jedem *Input* das Datum, dort an ihm ankommen würde. In Wirklichkeit wird das Datum über die Lambda-Funktion, die zu dem *Input* gehört berechnet. Die Reihenfolge der Werte im Array müssen mit der Nummerierung der *Inputs* übereinstimmen.

units : String[] speichert zu jedem Wert in **data** die zugehörige Einheit als String und genau wie in **data** muss die Reihenfolge der *Strings* im Array mit der Nummerierung der *Inputs* übereinstimmen.

calculate_functions() : bool geht alle seine *Inputs* in *inputs* durch und holt sich für jeden die Output-ID *connected_to*. Mit dieser als Parameter wird über *model_workspace* von *Butler calculate_function()* und *calculate_unit()* aufgerufen. Deren Rückgabewerte werden jeweils an der gleichen Stelle in **functions** und **units** gespeichert. Ging etwas schief bei der Berechnung, so wird *False* zurückgegeben.

calculate(sensor_data : dictSensorItem, float[]) : bool setzt in jede Funktion in **functions** *sensor_data* als Parameter ein und speichert die jeweiligen Werte in **data**. Die Werte in **data** werden an der gleichen Position, wie die der Lambda-Funktion in **functions** mit der der Wert berechnet wurde, gespeichert.

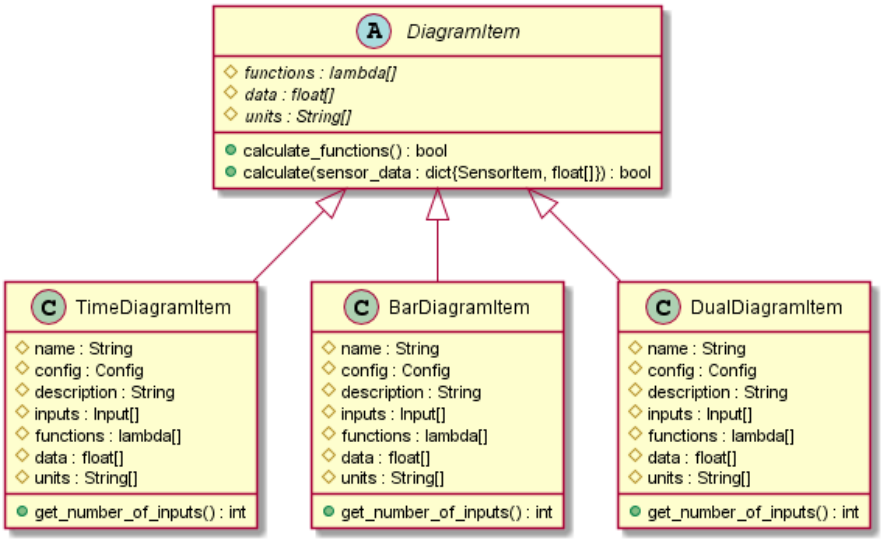

Abbildung 16: Die Unterklassen von DiagramItem

Jede Unterklasse von *DiagramItem* repräsentiert das *Model* eines Diagramm-Elements und übernimmt das Auswerten von Datenströmen. Die Klassen implementieren die abstrakten Attribute aus ihren Oberklassen in ihrem Konstruktor. Um außerhalb der Klasse die Attribute lesen zu können, werden die *get...-Methoden* vererbt aus den Oberklassen. Im Konstruktor der Unterklassen werden die Attribute **name**, **config**, **description** und **inputs** gesetzt.

TimeDiagramItem

name = Zeitdiagramm

description =

functions wird mit einem leeren Array der Größe 1 initialisiert.

data wird mit einem leeren Array der Größe 1 initialisiert.

unit wird mit einem leeren Array der Größe 1 initialisiert.

config: Für **config** wird ein neues Objekt von *Config* erstellt und ihm zugewiesen

inputs: Für **inputs** wird ein *Input* Objekt erstellt und in ein Array der Größe 1 gespeichert. Dieses Array wird **inputs** zugewiesen.

get_number_of_inputs() gibt die Zahlt *1* zurück.

BarDiagramItem

name = Balkendiagramm

description =

functions wird mit einem leeren Array der Größe 3 initialisiert.

data wird mit einem leeren Array der Größe 3 initialisiert.

unit wird mit einem leeren Array der Größe 3 initialisiert.

config: Für **config** wird ein neues Objekt von *Config* erstellt und ihm zugewiesen

inputs: Für **inputs** werden drei *Input* Objekte erstellt und in dieser Reihenfolge in ein Array der Größe 3 gespeichert. Dieses Array wird **inputs** zugewiesen.

get_number_of_inputs() gibt die Zahlt *3* zurück.

DualDiagramItem

name = 2D-Diagramm(MK10)

description =

functions wird mit einem leeren Array der Größe 2 initialisiert.

data wird mit einem leeren Array der Größe 2 initialisiert.

unit wird mit einem leeren Array der Größe 2 initialisiert.

config: Für **config** wird ein neues Objekt von *Config* erstellt und ihm zugewiesen

inputs: Für **inputs** werden drei *Input* Objekte erstellt und in dieser Reihenfolge in ein Array der Größe 2 gespeichert. Dieses Array wird **inputs** zugewiesen.

get_number_of_inputs() gibt die Zahlt *2* zurück.

1.2.3 Elementkonfigurationen

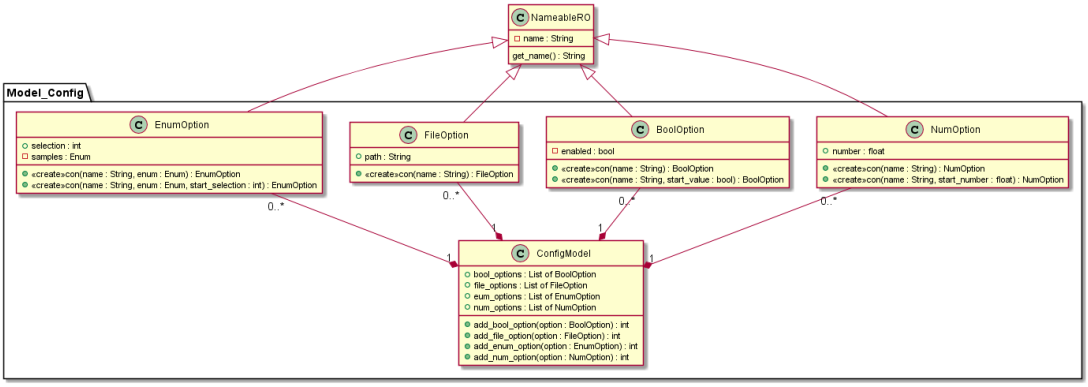

Abbildung 17: Das Paket Config

Um das Projekt möglichst modular zu halten, wurde dieser Aufbau gewählt. Alle einstellbaren Werte/Eigenschaften eines Elements werden in eigenen Klassen zu jedem Datentyp gespeichert. Jede der vier Klassen repräsentiert einen Optionstyp. Mit diesem Aufbau soll es dem *View* ermöglicht werden, das Einstellungsfenster eines Elements generisch zu erstellen. So muss, wenn ein Element hinzugefügt werden soll, lediglich es selbst und eine Konfiguration von ihm in seiner Klasse definiert werden. Dadurch muss kein neues Einstellungsfenster für jedes Element programmiert werden. Der Programmierer kann so unabhängig von dem GUI-Framework Elemente mit beliebig einstellbaren Optionen programmieren. Es ist so auch kein großer Aufwand das fertige Projekt nachträglich um weitere Elemente zu erweitern. Die vier vordefinierten Optionstypen dienen hierbei dazu, alle gängigen einstellbaren Optionen darzustellen. Es können aber dank dem Aufbau auch beliebig weitere Optionstypen hinzugefügt werden. Nur muss hierbei beachtet werden, dass für jede neue Optionstyp-Klasse einzeln im *View* programmiert werden muss, wie die Referenzen dieser Option im Einstellungsfenster graphisch umgesetzt werden.

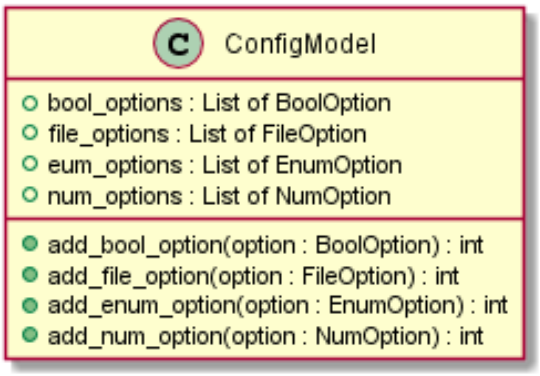

Abbildung 18: Die Klasse ConfigModel

Die Config-Klasse verwaltet die verschiedenen Optionstypen. Jedes Element hat eine eigenes Config-Objekt. Dieses stellt eine Konfiguration von Optionen für das Element dar. Jede Option, die der Benutzer im Einstellungsfenster für das Element einstellen kann, wird in seinem Config-Objekt hinterlegt. Config enthält eine Liste zu jedem Optionstyp. Die Optionen des Elements werden nach ihrem Typ passend in die Listen, beim erstellen des Elements, hinzugefügt. Config stellt zu jeder dieser Listen Methoden zur Verfügung, um diese zu verwalten. Die **add-Methoden** werden verwendet um die Listen der Optionstypen aufzubauen. Sie werden aus dem Konstruktor eines Elements aufgerufen. Die **set-Methoden** dienen dazu die Werte von den einzelnen Optionen zu setzen. Sie werden meist aus dem *View* aufgerufen, wenn der Benutzer über das Einstellungsfenster eine Einstellung vornimmt.

bool_options : List of BoolOption ist eine Liste von Objekten der BoolOption-Klasse. Jedes Objekt in dieser Liste steht für eine Option von dem Optionstyps BoolOption.

file_options : List of FileOption ist eine Liste von Objekten der FileOption-Klasse. Jedes Objekt in dieser Liste steht für eine Option von dem Optionstyps FileOption.

eum_options : List of EnumOption ist eine Liste von Objekten der EnumOption-Klasse. Jedes Objekt in dieser Liste steht für eine Option von dem Optionstyps EnumOption.

num_options : List of NumOption ist eine Liste von Objekten der NumOption-Klasse. Jedes Objekt in dieser Liste steht für eine Option von dem Optionstyps NumOption.

add_bool_option(option : BoolOption) : int fügt *option* an das Ende der Liste **bool_options** hinzu. War dies erfolgreich wird *true* zurückgegeben. Die Methode wird nach dem Erstellen einer Konfiguration dazu benutzt Optionen der Konfiguration

hinzuzufügen. Die Methode gibt den Index, an dem die Option in die Liste eingefügt wurde, zurück.

add_file_option(option : FileOption) : int fügt *option* an das Ende der Liste **file_options** hinzu. War dies erfolgreich wird *true* zurückgegeben. Die Methode wird nach dem Erstellen einer Konfiguration dazu benutzt Optionen der Konfiguration hinzuzufügen. Die Methode gibt den Index, an dem die Option in die Liste eingefügt wurde, zurück.

add_enum_option(option : EnumOption) : int fügt *option* an das Ende der Liste **enum_options** hinzu. War dies erfolgreich wird *true* zurückgegeben. Die Methode wird nach dem Erstellen einer Konfiguration dazu benutzt Optionen der Konfiguration hinzuzufügen. Die Methode gibt den Index, an dem die Option in die Liste eingefügt wurde, zurück.

add_num_option(option : NumOption) : int fügt *option* an das Ende der Liste **num_options** hinzu. War dies erfolgreich wird *true* zurückgegeben. Die Methode wird nach dem Erstellen einer Konfiguration dazu benutzt Optionen der Konfiguration hinzuzufügen. Die Methode gibt den Index, an dem die Option in die Liste eingefügt wurde, zurück.

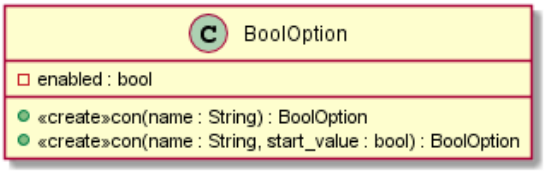

Abbildung 19: Die Klasse BoolOption

BoolOption repräsentiert einen Optionstyp, der nur zwei Zustände annehmen kann und k im Einstellungsfenster zum Beispiel durch eine Checkbox dargestellt wird. Diese Option kann nur zwei Werte annehmen.

enabled : bool speichert den Zustand, der vom Benutzer eingestellt wurde.

con(name : String) ruft den Konstruktor **con(name : String, start_value : bool) : BoolOption** mit den Parametern *name* und *false* auf.

con(name : String, start_value : bool) initialisiert die Attribute **name** mit *name* und **enabled** mit *start_value*.

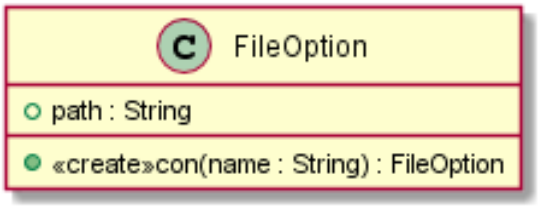

Abbildung 20: Die Klasse FileOption

FileOption repräsentiert einen Optionstyp, bei dem ein Dateipfad gesetzt oder als Ziel gewählt wird.

path : String speichert den Dateipfad als String, der im Einstellungsfenster des Elements gesetzt wurde.

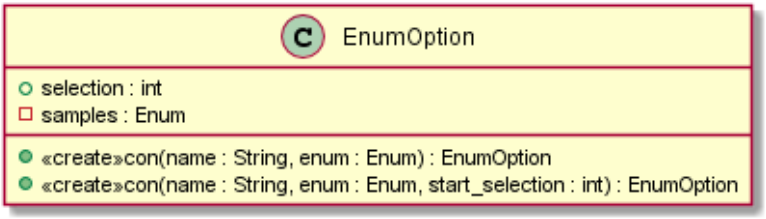

Abbildung 21: Die Klasse EnumOption

EnumOption repräsentiert einen Optionstyp, der dem Benutzer eine Auswahlmöglichkeit, von zum Beispiel Werten oder Modi, zur Verfügung stellt aus denen er wählen kann.

selection: int speichert den Index, an dem sich der, vom Benutzer, ausgewählte Wert oder Modi im Enum **samples** befindet.

samples : Enum speichert eine Auswahlmöglichkeit von verschiedenen Werten oder Modi, die eingestellt werden können. Es gibt nur Setter für dieses Attribut, da es nach dem Initialisieren des Objekts nicht mehr verändert werden soll

con(name : String, enum : Enum) ruft den Konstruktor **con(name : String, enum : Enum, start_selection : int) : EnumOption** mit den Parametern *name*, *enum* und 0 auf.

con(name : String, enum : Enum, start_selection : int) initialisiert die Attribute **name** mit *name*, **enum** mit *enum* und **selection** mit *start_selection*.

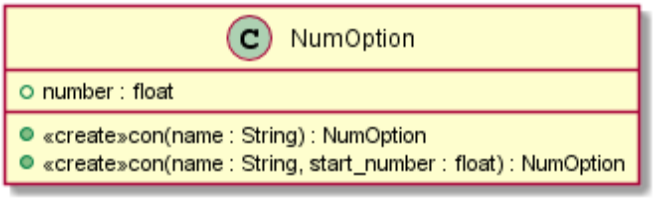

Abbildung 22: Die Klasse NumOption

NumOption repräsentiert einen Optionstyp, bei der eine Zahl als Wert eingestellt werden kann (z.B. eine Konstante eines *Konstanten-Elements*, welches einen einstellbaren konstanten Wert ausgeben soll).

number : float ist der Wert der Option, den der Benutzer im Einstellungsfenster eingestellt hat.

con(name : String) ruft den Konstruktor **con(name : String, start_number : float) : NumOption** mit den Parametern *name* und 0 auf.

con(name : String, start_number : float) initialisiert die Attribute **name** mit *name* und **number** mit *start_number*.

1.2.4 Manager

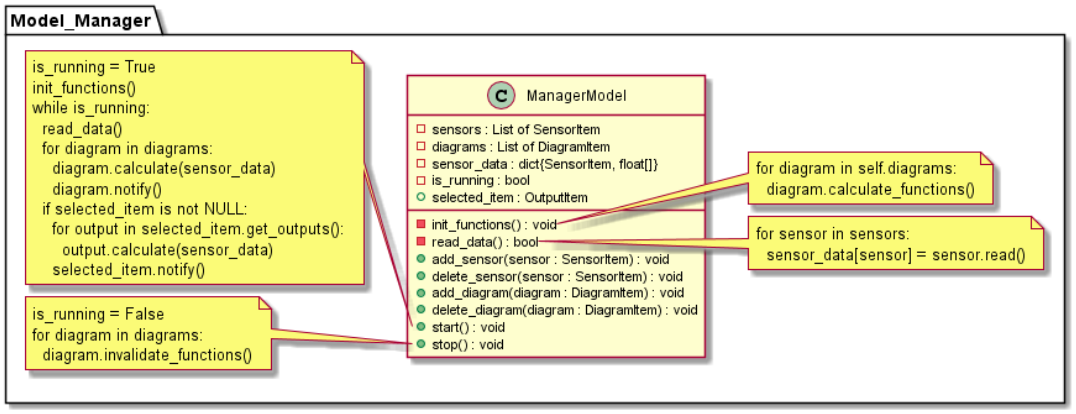

Abbildung 23: Das Paket Manager

Das **ManagerModel** repräsentiert eine Klasse, die die Hauptverwaltung der Sensordaten übernimmt. Die Aufgabe des Manager ist es sich in einem festen Takt Daten von den *SenorItems* zu holen und für jedes *DiagramItem* im Model seine aktuellen Werte zu berechnen und den jeweiligen Beobachter über *notify()* zu benachrichtigen. Die Ausleserate, die in dem Programm eingestellt werden kann, wird in den *SensorItem* individuell gehandhabt und nicht im Manager. Außerdem hat der Manager auch ein Attribut, um das vom Benutzer ausgewählte Element zu speichern. Damit bleibt er komplett unabhängig vom *View* und berechnet außerdem effizient nur für das ausgewählte Element seine Werte, die im *View* gebraucht werden.

sensors : List of SensorItem ist eine Liste von *SensorItem*, die alle *SensorItem* speichert, deren zugehöriges *SensorItemView* sich aktuell in der Arbeitsfläche befinden. Es werden nur Daten von Sensoren eingelesen, deren *SensorItem* sich in dieser Liste befindet.

diagrams : List of DiagramItem ist eine Liste von *DiagramItem*, die alle *DiagramItem* speichert, deren zugehöriges *DiagramItemView* sich aktuell in der Arbeitsfläche befinden. Es wird nur die Berechnungsfunktion *calculate()* von einem *DiagramItem*, das sich in der Liste befindet, aufgerufen.

sensor_data : dict{SensorItem, float[]} ist eine Key-Value-Hashmap. Sie ist lediglich ein Puffer zum Speichern aller Messdaten einmal pro Messzyklus. Ihre Keys sind die *SensorItem* aus der Liste **sensors** und ihre Values sind Arrays von Messdaten. In der Regel hat jedes Array nur einen Eintrag. Es kann aber Sensoren, wie den Lage-Sensor geben, der drei Werte jeden Messzyklis liefert und diese werden in einer festen Reihenfolge in dem Array gespeichert.

is_running : bool ist ein privates Attribut, dass nur solange *True* ist, wie der Manager jeden Takt Daten aus den Sensoren auslesen soll. Es wird zum Beispiel *False*, wenn der Stopp-Knopf gedrückt wird.

selected_item : OutputItem speichert das *SensorItem* oder *OperatorItem*, zu dem das vom Benutzer auf der Arbeitsfläche ausgewählte *WorkspaceItemView* gehört. Wird vom Benutzer ein *WireView* oder *DiagramItemView* ausgewählt, so wird **selected_item** auf einen ungültigen Wert gesetzt.

init_functions() : void erstellt durch den Aufruf von *calculate_functions()* auf jedem *DiagramItem* rekursiv alle Lambda-Funktionen für jeden Ausgang jedes Elements.

read_data() : bool ruft für jedes *SensorItem* in **sensors** die Methode *read()* darauf auf, die ein Array an Messdaten zurück gibt. So wird **sensor_data** gesetzt.

add_sensor(sensor : SensorItem) : void fügt eine *SensorItem* zu der Liste **sensors** hinzu.

delete_sensor(senor : SensorItem) : void löscht ein *SensorItem* aus der Liste **sensors**.

add_diagram(diagram : DiagramItem) : void fügt eine *DiagramItem* zu der Liste **diagrams** hinzu.

delete_diagram(diagram : DiagramItem) : void löscht ein *DiagramItem* aus der Liste **diagrams**.

start() : void setzt **is_running** auf *True*, ruft **init_functions()** auf und beginnt, solange **is_running** *True* ist, in einem festen Takt alle Sensormessdaten zu lesen durch Aufruf von **read_data()**. Danach die Werte aller Diagramme und die, die an den Ausgängen von **selected_item** anliegen berechnet, sofern ein gültiges Element ausgewählt wurde. Dann wird noch der Beobachter von dem **selected_item** benachrichtigt. Das Berechnen der Werte und die Benachrichtigung der View-Elemente kann parallelisiert werden.

stop() : void setzt **is_running** auf *False* und ruft *invalidate_functions()* auf, um die Funktion in allen *DiagramItems* und rekursiv alle Funktionen zu invalidieren. Die Methode wird üblicherweise vom Start/Stopp-Knopf aufgerufen, oder wenn etwas aus der Arbeitsfläche gelöscht wird.

1.2.5 Arbeitsfläche

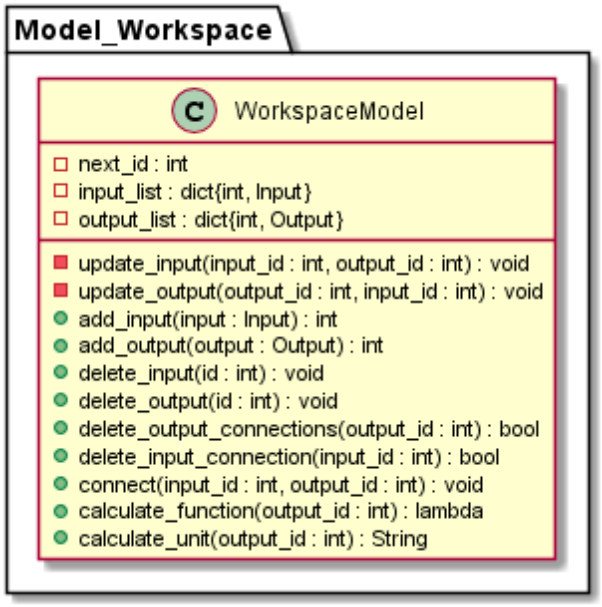

Abbildung 24: Das Paket Item

WorkspaceModel verwaltet auf *Model-Ebene* alle Verbindungen zwischen den *Item-Models* der Elemente. **WorkspaceModel** enthält Listen aller *Inputs* und *Outputs* und weißt jedem eine eindeutige ID zu. Über diese Klasse werden alle rekursiven Aufrufe abgehandelt, da nur diese Klasse zugriff auf die Vorgänger und Nachfolger eines *Inputs* oder *Outputs* hat. Alle anderen Klassen, insbesondere *Input* und *Output* selbst, haben lediglich die feste ID des Vorgängers oder Nachfolgers.

next_id : int speichert die nächste freie ID, die dem nächsten *Input* oder *Output* zugewiesen wird. (Wird mit null initialisiert)

input_list : dict{int, Input} ist eine Key-Value-Hashmap, deren Keys die IDs aller *Inputs* und deren Values die zugehörigen *Inputs* zu den Keys sind.

output_list : dict{int, Output} ist eine Key-Value-Hashmap, deren Keys die IDs aller *Outputs* und deren Values die zugehörigen *Outputs* zu den Keys sind.

update_input(input_id : int, output_id : int) : void holt sich *Input* zu dem Key *input_id* aus **input_list** und setzt sein Attribut *connected_to* auf den Wert des Parameters *output_id*.

update_output(output_id : int, input_id : int) : void holt sich *Output* zu dem

Key *output_id* aus **output_list**. Ist *input_id* gleich -1, so wird *delete_all_connections()* auf *Output* aufgerufen. Andernfalls wird auf *Output add_connection(input_id)* aufgerufen.

add_input(input : Input) : int fügt *input* in **input_list** mit dem Key **next_id** hinzu. Dann wird **next_id** um eins erhöht und der Key von *input* zurückgegeben.

add_output(output : Output) : int fügt *output* in **output_list** mit dem Key **next_id** hinzu. Dann wird **next_id** um eins erhöht und der Key von *output* zurückgegeben.

delete_input(id : int) : void ruft **delete_input_connection(*id*)** auf und entfernt *id* aus **input_list**.

delete_output(id : int) : void ruft **delete_output_connections(*id*)** auf und entfernt *id* aus **output_list**.

delete_output_connections(output_id : int) : bool holt sich aus **output_list** den *Output* an Stelle *output_id*. Aus dem *Output* wird sich die Liste *connected_to* geholt durch *get_connections()*. Für jede ID aus *connected_to* wird **update_input(ID, -1)** aufgerufen. Dann wird **update_output(*output_id*, -1)** aufgerufen. Geht irgendetwas schief, so wird *False* zurückgegeben.

delete_input_connection(input_id : int) : bool holt sich aus **input_list** den *Input* an Stelle *input_id*. Aus dem *Input* wird sich *output_id* geholt und aus **output_list** den zugehörigen *Output* geholt. Auf dem *Output* wird *delete_connection(input_id)* aufgerufen. Dann wird **update_input(*input_id*, -1)** aufgerufen. Geht irgendetwas schief, so wird *False* zurückgegeben.

connect(input_id : int, output_id : int) : void ruft **update_input(input_id, output_id)** und **update_output(output_id, input_id)** auf.

calculate_function(output_id : int) : lambda holt sich den *Output* zu *output_id* aus **output_list** und ruft auf ihm *calculate_function()* auf. Der Rückgabewert davon wird zurückgegeben.

calculate_unit(output_id : int) : lambda holt sich den *Output* zu *output_id* aus **output_list** und ruft auf ihm *calculate_unit()* auf. Der Rückgabewert davon wird zurückgegeben.

1.3 View

1.3.1 Hauptfenster

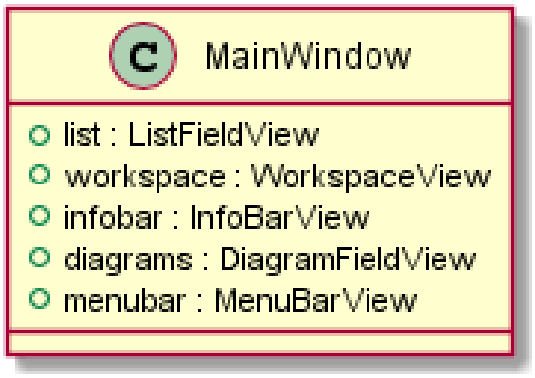

Abbildung 25: Die Klasse MainWindow

MainWindow stellt das Hauptfenster der GUI dar. Es besteht aus fünf Teilen, die alle eine eigene Aufgabe im Fenster haben. **list** repräsentiert die Elementliste auf der linken Seite des Fensters, in der, in drei Tabs gegliedert, alle Elemente aufgelistet sind. **workspace** ist die Arbeitsfläche, in der Elemente angeordnet und verbunden werden können. **infobar** repräsentiert die Informationsleiste am unteren Rand des Fensters, in der Informationen über das ausgewählte Element angezeigt werden. **diagrams** ist die Fläche auf der rechten Seite des Fensters, in der alle Diagramme angezeigt werden. **menubar** repräsentiert die Menüleiste am oberen Rand des Fensters, indem in verschiedenen Untermenüs globale Aktionen ausgeführt werden können.

list : ListFieldView speichert den Verweis auf ein Objekt der Klasse *ListFieldView*.

workspace : WorkspaceView speichert den Verweis auf ein Objekt der Klasse *WorkspaceView*.

infobar : InfoBarView speichert den Verweis auf ein Objekt der Klasse *InfoBarView*.

diagrams : DiagramFieldView speichert den Verweis auf ein Objekt der Klasse *DiagramFieldView*.

menubar : MenuBarView speichert den Verweis auf ein Objekt der Klasse *MenuBarView*.

1.3.2 Informationsfeld

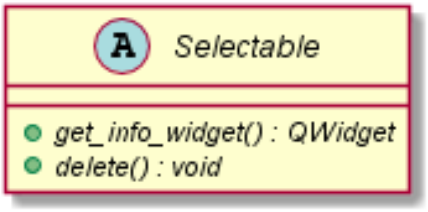

Abbildung 26: Die Klasse Selectable

Selectable alle Klassen, die von dieser Klasse erben, gelten als Auswählbar in der Arbeitsfläche im *View*. Von ihr erben in diesem Entwurf nur zwei Klassen, *WorkspaceItemView* und *WireView*.

get_info_widget() : QWidget wird in *WorkspaceItemView* und *WireView* überschrieben. In *WorhspaceItemView* holt sich die Methode ihre nötigen Daten über das Attribut **model**, setzt sie in den vordefinierten Rahmen **info_widget** ein und gibt diesen zurück.(MK4)
In *WireView* wird ein QWidget mit dem Ein- und Ausgang zurückgegeben.

delete() : void wird vom Löschknopf aufgerufen. Mit der Methode soll das ausgewählte Objekt gelöscht werden.

1.3.3 Elemente

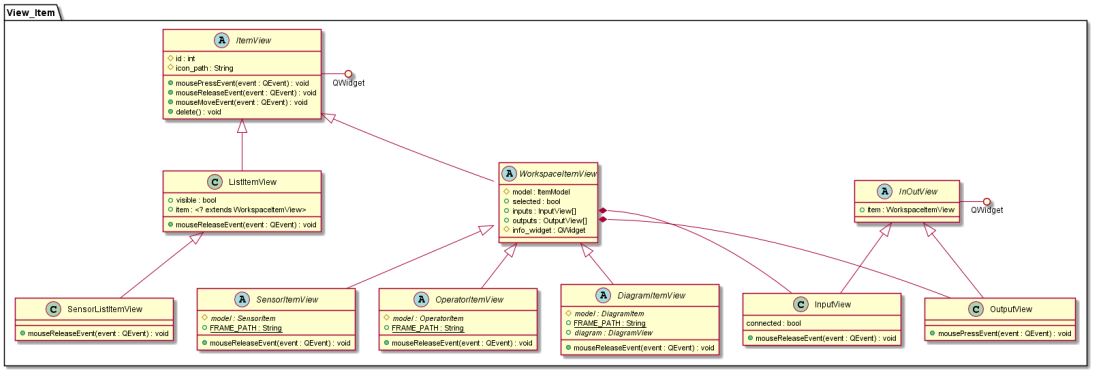

Abbildung 27: Das Paket Item

Das Paket **View_Item** beinhaltet alle Klassen die notwendig sind, um dem Benutzer alle Elemente die ihm zur Verfügung stehen anzeigen zu können. Dabei wurde Wert darauf gelegt, dass hierbei ebenfalls das Prinzip der Modularität eingehalten wird, sodass eine Erweiterung der Elemente wie Sensoren,Operatoren und Diagramm einfach gehalten wird. So muss der Programmierer nur die entsprechende Klasse erstellen.Somit ist ein abgestimmtes Zusammenspiel mit der Modularität im Model gewährleistet.
Im wesentlichen ist das *View_Item* in zwei Teile unterteilbar. Einmal das *ListItemView*, welches die Darstellung der Elemente in der Elementliste umsetzt und zum Anderen das *WorkspaceItemView*, das für die Darstellung des Elements in der Arbeitsfläche und für seine Verknüpfung mit dem Model verantwortlich ist.

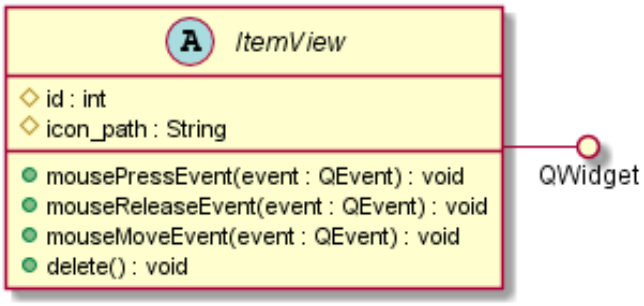

Abbildung 28: Die Klasse ItemView

Die abstrakte Klasse **ItemView** besitzt alle Attribute, die jedes Element in seinem *View* enthalten muss. Sowohl ein *ListItemView*, als auch ein *WorkspaceItemView*. Sie erbt von der *PyQt5-Klasse QWidget* und *NameableRO*.

id : int speichert eine eindeutige ID für den *View-Teil* des Elements. Zugehörige *WorkspaceItemViews* zu *ListItemViews* hiben die selbe **id**.

icon_path : String speichert den Pfad zu einem Icon, das auf Element angezeigt wird.

mousePressEvent(event : QEvent) wird von *QWidget* geerbt und überschrieben. Die Methode kümmert sich um das „Aufheben“ des Elements beim Linksklicken.

mouseReleaseEvent(event : QEvent) wird von *QWidget* geerbt und überschrieben. Die Methode kümmert sich um das „Ablegen“ des Elements beim Loslassen der linken Maustaste.

mouseMoveEvent(event : QEvent) wird von *QWidget* geerbt und überschrieben. Die Methode kümmert sich um das „Verschieben“ des Elements beim Bewegen der Maus bei gedrückter linken Maustaste.

delete() : void ist die überschriebene Methode von *Selectable*. Sie ruft *delete_item()* in *WorkspaceView* auf.

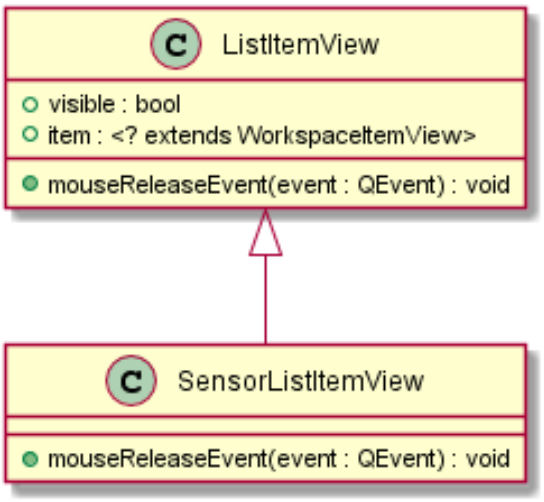

Abbildung 29: Die Klasse ListItemView

Die Klasse **ListItemView** beschreibt ein Element im Listenfeld und erzeugt eine Instanz einer **WorkspaceItemView**-Unterklasse auf der Arbeitsfläche, wenn es darüber losgelassen wird. In dem Konstruktor der **ListItemView** wird die **id** fest gesetzt. Alle Elemente in den Listen in *ListFieldView* sind vom Typ **ListItemView**, außer die Sensor-Elemente, da diese in der Liste nach dem Rausziehen aus der Liste dort nicht mehr angezeigt werden sollen. Sie sind vom Dyp **SensorListItemView**.

visible : bool ist *False*, wenn es in der Liste in *ListFieldView* nicht mehr angezeigt werden soll.

item : <? extends WorkspaceItemView speichert die Klasse, von der es beim Ablegen des Elements auf der Arbeitsfläche eine Instanz erstellen soll.

mouseReleaseEvent(event : QEvent) überschreibt die Methode der Oberklasse *ItemView* und kümmert sich zusätzlich um das Erstellen eine **WorkspaceItemView**-Unterklasseninstanz mit gleichem Namen und ID auf der Arbeitsfläche, wenn es darüber losgelassen wird.
SensorListItemViews überschreiben diese Methode auch. Die Methode macht genau das selbe, wie in **ListItemView**, mit der Ausnahme, dass das Attribut **visible** auf *False* gesetzt wird, wenn das Element erfolgreich in die Arbeitsfläche gezogen wurde.

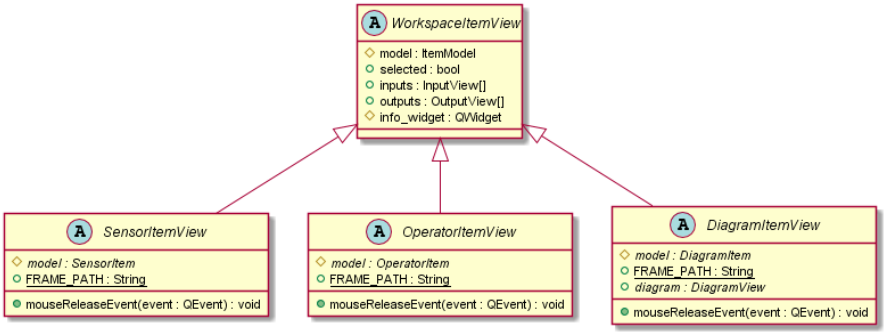

Abbildung 30: Klassendiagramm zu WorkspaceItemView

Die abstrakte Klasse **WorkspaceItemView** beschreibt ein Element auf der Arbeitsfläche. Sie besitzt drei Unterklassen für jeweils eins der drei Typen Sensoren, Operatoren und Diagramme. Unterklassen von **DiagramItemView** rufen *update_ diagram()* auf ihrem zugehörigen *diagram* auf, wenn ein Beobachter sie über eine Änderung benachrichtigt.

model : ItemModel speichert eine Referenz zu dem zugehören *ItemModel* im Model. **model** wird in den Unterklassen spezifiziert, um den aktuellen Wert später im *ManagerModel* berechnen zu können.

selected : bool speichert, ob das Element gerade ausgewählt ist.

inputs : InputView[] speichert die zum Element gehörenden Eingänge.

outputs : OutputView[] speichert die zum Element gehörenden Ausgänge.

info_widget : QWidget speichert ein *QWidget*, das im Informationsfeld angezeigt wird, wenn das Element ausgewählt ist.

FRAME_PATH : String speichert den Pfad zu einem Bild, dass die Form des Elemententyps darstellt.

diagram : Diagram ist eine Referenz auf zum Diagrammelement zugehörigen Diagramms. Besitzt das Diagrammelement kein Diagramm steht hier *None*. **diagram** wird im Konstruktor der Klasse erstellt und definiert genau, um welche Art von Diagramm es sich handelt.

mouseReleaseEvent(event : QEvent) : void ruft für **SensorItemView** und **OperatorItemView** setzt das statisch Attribut *selected_ item* über *ManagerModel* auf den Wert des jeweiligen **model** und ruft *refresh_ infobar()* über *InfoBarView* auf. **DiagramItemView** setzt das Attribut auf *NULL*.

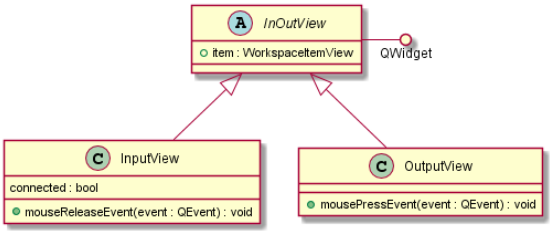

Abbildung 31: KlassenDiagramm zu InOutView

Die Klasse **InOutView** ist die Oberklasse für alle Ein- und Ausgänge. Dies wird Benötigt, da die Eingänge und Ausgänge auch visuell an den Elementen dargestellt werden sollen. Sie erbt von der *PyQt5-Klasse QWidget*. Die Klasse vererbt den Klassen **InputView** und **OutputView** ein Element mit einer Referenz auf *WorkspaceItemView*.

connected : bool wird auf *True* gesetzt falls eine Verbindung besteht, sonst *False*.

item : WorkspaceItemView ist eine Referenz auf das Element in der Arbeitsfläche, für das es ein Ein- bzw. Ausgang ist.

mousePressEvent(event : QEvent) : void wird nur in **OutputView** implementiert und erstellt ein *WireView* mit diesem *OutputView* im Konstruktor.

mouseReleaseEvent(event : QEvent) : void wird nur in **InputView** überschrieben und setzt, wenn *wire_in_hand* in *WorkspaceView* nicht *NULL*, das Attribut *input* von *wire_in_hand* auf diesen **InputView**. Außerdem wird dann *add_wire()* mit dem *WireView* auf dem *WorkspaceView* aufgerufen.

1.3.4 Listenfläche

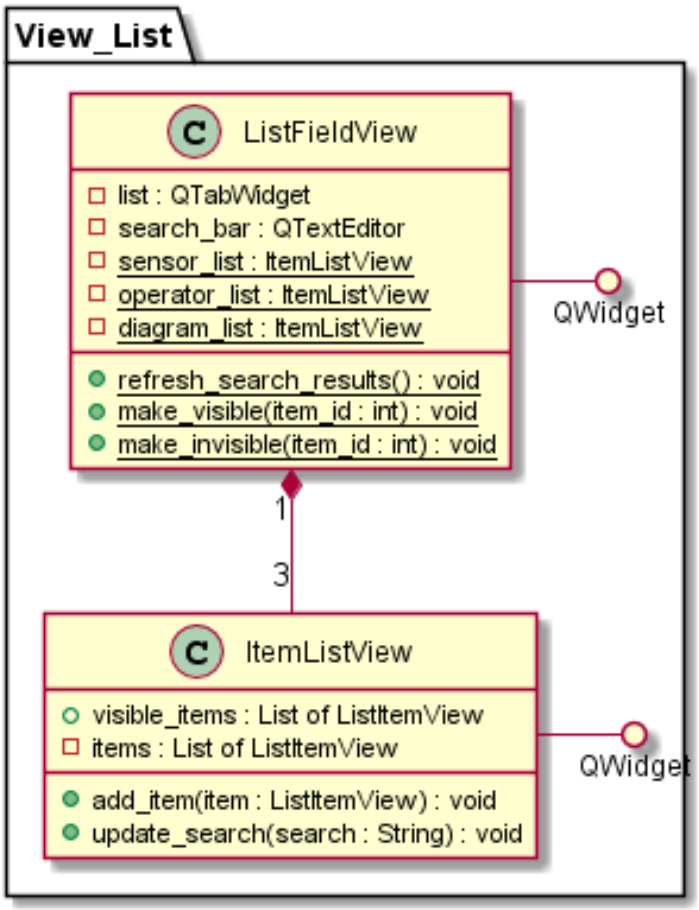

Abbildung 32: Das Paket List

Das Paket **View_List** enthält eine Klasse *ListFiledView* welche die drei Tabs für Sensoren, Operatoren und Diagramme repräsentiert. Jede dieser Listen wird durch eine Klasse *ItemListView* dargestellt. Sie stellt Methoden zur Verfügung, um Elemente in der Liste auszublenden, was zum Beispiel bei Sucheingaben geschehen muss.(MK1)

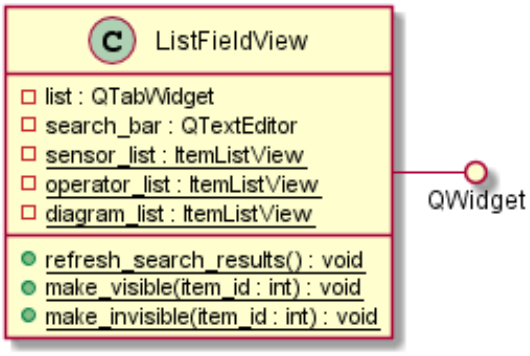

Abbildung 33: Die Klasse ListFieldView

Im Konstruktor von **ListFieldView** werden alle *ListItemView* mit der Methode *add_item()* aus *ItemViewList* zu den drei Listen hinzugefügt. Jedes Element des Programms muss also hier erstellt werden und zu einer der Listen hinzugefügt werden um im Programm genutzt werden zu können.

list : QTabWidget ist ein von Qt5 zur Verfügung gestelltes Konstrukt zum Visualisierung von mehreren Tabs. In ihm werden *ItemListViews* aus den drei Listen **sensor_list**, **operator_list** und **diagram_list** in jeweils drei Tabs dargestellt. **list** visualisiert jeweils nur die *ListItemView* aus*visible_items*.

search_bar : QTextEditor ist eine Suchleiste und ermöglicht das Suchen einzelner Elemente, die in **list** visualisiert werden.(WK4)

sensor_list : ItemListView ist eine Liste die alle Sensor-Elemente enthält, die vom Benutzer ausgewählt werden kann.

operator_list : ItemListView enthält alle Operator-Elemente, die vom Benutzer ausgewählt werden kann.

diagram_list : ItemListView enthält alle Diagramm-Elemente, die vom Benutzer ausgewählt werden kann.

refresh_search_results() : void wird aufgerufen, sobald der Benutzer etwas in die Suchleiste eingibt und aktualisiert die angezeigten Elemente. Die Methode ruft auf jeder der drei Listen *update_search()* mit dem Text als Parameter, der in der Suchleiste steht. Dann wird **list** aktualisiert.

make_visible(item_id : int) : void setzt das Attribut *visible* in dem zugehörigen *ListItemView* aus einer der Listen auf *True* und ruft auf der zugehörigen Liste *update_search()* auf mit einem leeren String als Parameter.

make_invisible(item_id : int) : void setzt das Attribut *visible* in dem zugehörigen *ListItemView* aus einer der Listen auf *False*.

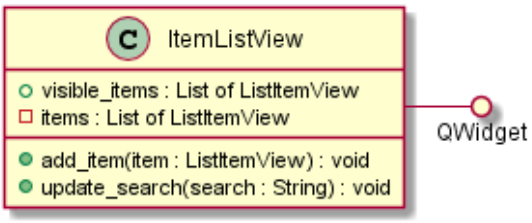

Abbildung 34: Die Klasse ItemListView

ItemListView repräsentiert eine Liste von *ListItemViews* und stellt alle nötigen Methoden zur Verwaltung dieser zur Verfügung.

visible_items : List of ListItemView speichert eine Liste von sichtbaren *ListItemView*.

items : List of ListItem speichert eine Liste von allen *ListItemView*, die zu dieser Klasse gehören.

add_item(item : ListItemView) : void fügt *item* in die Liste **items** und **visible_items** ein.

update_search(search : String) : void setzt **visible_items** auf die leere Liste und fügt alle *ListItemView* aus **items** hinzu, für die *search* ein Präfix von dem in ihrem Attribut *name* gespeicherten String ist und *visible True* ist.

1.3.5 Arbeitsfläche

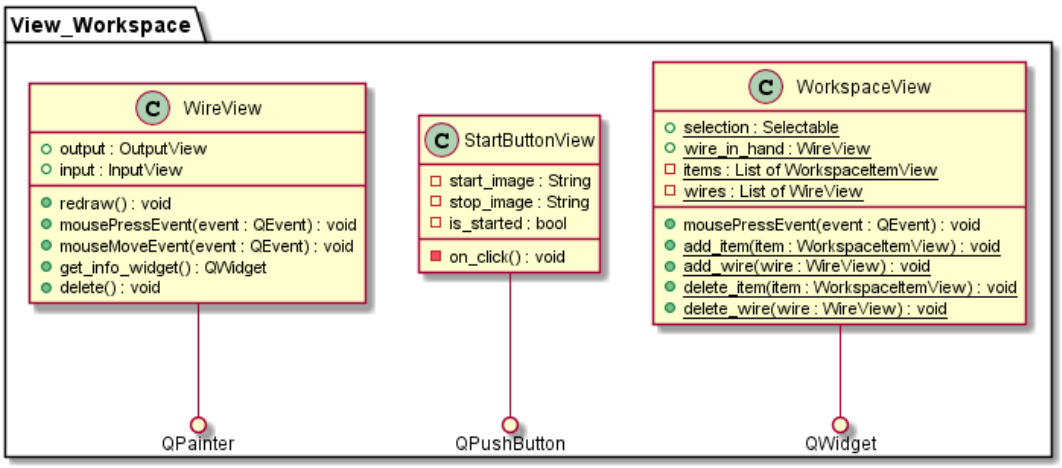

Abbildung 35: Das Paket Workspace

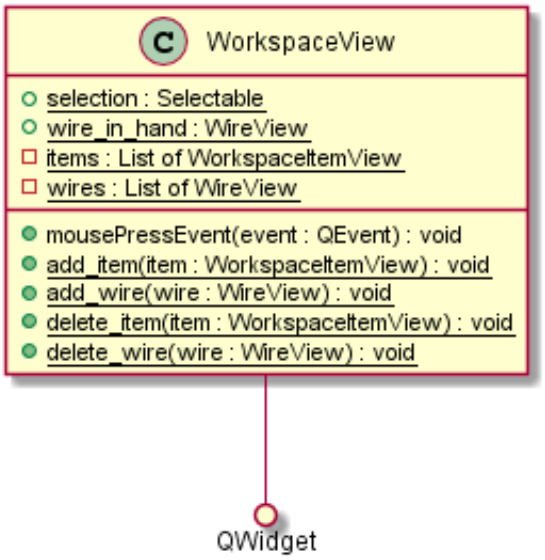

Abbildung 36: Die Klasse WorkspaceView

selection : Selectable speichert das ausgewählte *WorkspaceItemView* oder *WireView*.

wire_in_hand : WireView speichert das erstellte *WireView*, wenn der Benutzer den Ausgang eines *WorkspaceItemView* angeklickt hat und eine Verbindung zu einem Eingang eines anderen Elements ziehen möchte. Ansonsten *NULL*.

items : List of WorkspaceItemView speichert eine Liste aller *WorkspaceItemView*, die sich in der Arbeitsfläche befinden.

wires : List of WireView speichert eine Liste aller *WireView*, die sich in der Arbeitsfläche befinden.

mousePressEvent(event : QEvent) : void schreibt in **selection** den Wert *NULL*.

add_item(item : WorkspaceItemView) : void fügt *item* in die Liste **items** ein.

add_wire(wire : WireView) : void fügt *wire* in die Liste **wires** ein.

delete_item(item : WorkspaceItemView) : void entfernt *item* aus der Liste **items**, ruft *make_visible()* mit seiner *id* auf *ListFieldView* auf und ruft **delete_wire()** auf allen *WireView* auf, die mit *item* verbunden sind.

delete_wire(wire : WireView) : void entfernt *wire* aus der Liste **wires** und aktualisiert alle *WorkspaceItemView*, die mit der Verbindung verbunden waren.

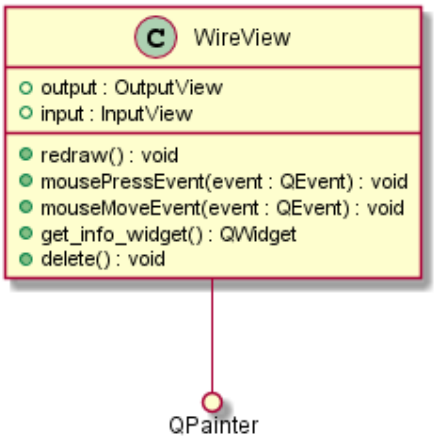

Abbildung 37: Die Klasse WireView

WireView repräsentiert die Visualisierung eine Verbindung zwischen zwei Elementen im Arbeitsbereich.(MK2)

output : OutputView speichert den Ausgang, von dem die Verbindung ausgeht.

input : InputView speichert den Eingang, zu dem die Verbindung geht.

redraw() : void wird aufgerufen, um die Verbindung neu zu zeichnen, wenn ein Element, an dem die Verbindung hängt, bewegt wird.

mousePressEvent(event : QEvent) : void setzt *selection* in *WorkspaceView* auf diese Verbindung und ruft *refresh_infobar()* über *InfoBarView* auf.

mouseMoveEvent(event : QEvent) : void ist **output** noch *NULL*, so folgt die Spitze der Verbindung dem Mauszeiger, ansonsten tut sich garnichts.

get_info_widget() : QWidget gibt ein QWidget, in dem die Namen der beiden Elemente aufgelistet sind, zwischen denen die Verbindung existiert.

delete() : void ist die überschriebene Methode von *Selectable*. Sie ruft *delete_wire()* in *WorkspaceView* auf.

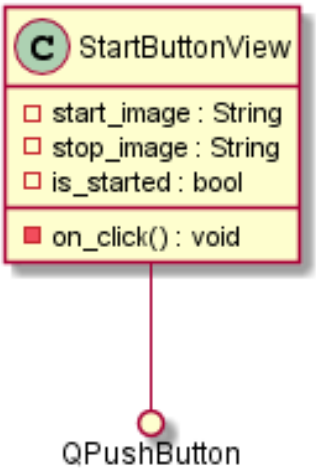

Abbildung 38: Die Klasse StartButtonView

Die Klasse **StartButtonView** repräsentiert den Start-und Stoppknopf, der betätigt werden muss, um den Messvorgang zu starten und die Daten anschließend im Diagramm angezeigt zu bekommen.(MK6)

start_image : String speichert den Dateipfad zu dem Bild, dass der Knopf anzeigt, solange er auf *stopp* steht. Das Attribut wird im Konstruktor gesetzt.

stop_image : String speichert den Dateipfad zu dem Bild, dass der Knopf anzeigt, solange das Programm Messdaten ausliest. Das Attribut wird im Konstruktor gesetzt.

is_started : bool ist *True* falls Messvorgang gestartet wurde sonst *False*.

on_click() : void wir der Start-/Stoppknopf vom Benutzer betätigt wird der Messvorgang gestartet, durch einen Aufruf von *start()* auf *ManagerModel* und *is_started* auf *True* gesetzt. Das Bild des Knopfes wird auf das an **stop_image** gesetzt. Durch erneute Betätigung wird *is_started* auf *False* gesetzt und der Messvorgang gestoppt, durch einen Aufruf von *stopp* auf *ManagerModel*. Das Bild des Knopfes wird auf das an **start_image** gesetzt

1.3.6 Informationsfeld

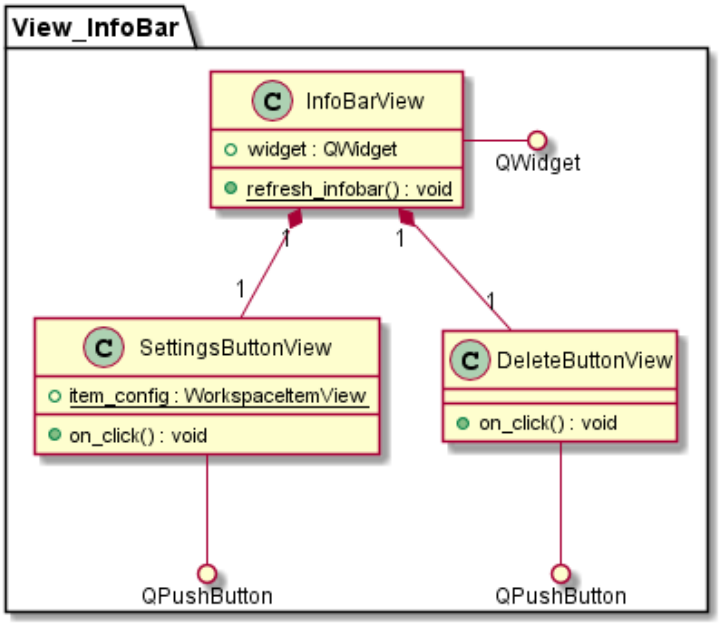

Abbildung 39: Das Paket InfoBar

Das Paket **InfoBar** repräsentiert die Informationsleiste und ihre Funktionen, sowie alle Knöpfe, die sich in ihr befinden.

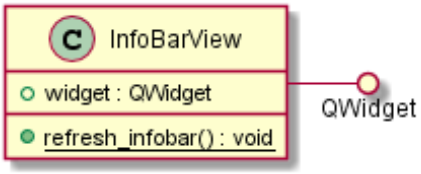

Abbildung 40: Die Klasse InfoBarView

Die Klasse **InfoBar** dient der Darstellung der Informationsleiste. Sie erbt von der *PyQt5-Klasse QWidget*. Jedes mal, wenn sich Daten des ausgewählten Elements ändern wird die Informationsleiste über das *ManagerModel* benachrichtigt durch einen Beobachter. Dann wird **refresh_infobar()** aufgerufen.

widget : QWidget speichert den Rahmen für die Knöpfe und und das Informationsfeld des ausgekählten Elements oder Verbindung.

refresh_infobar() : void schaut, ob das Attribut *selection* in *WorkspaceView NULL* ist und wenn nicht, wird *get_info_widget()* auf ihm aufgerufen und die Rückgabe in die *InfoBarView* eingebunden.

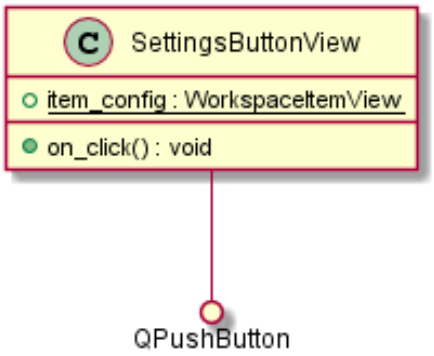

Abbildung 41: Die Klasse SettingsButtonView

Die Klasse **SettingsButtonView** repräsentiert den Einstellungsknopf in der Informationsleiste zur Einstellungsänderung der jeweiligen Elemente auf der Arbeitsfläche.(MK5)

item_config : WorkspaceItemView ...

on_click() : void ...

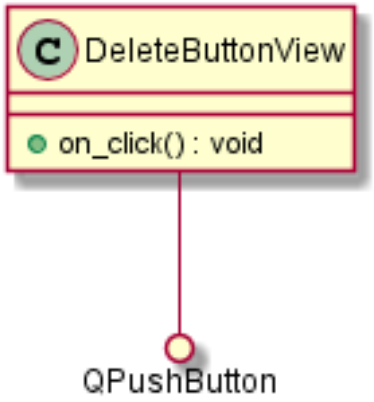

Abbildung 42: Die Klasse DeleteButtonView

Die Klasse **DeleteButtonView** repräsentiert den Löschknopf, um ein ausgewähltes Element oder einen Verbindungspfeil auf der Arbeitsfläche zu entfernen.

on_click() : void ruft auf *selection* in *WorkspaceView* die Methode *delete()* auf.

1.3.7 Diagrammfeld

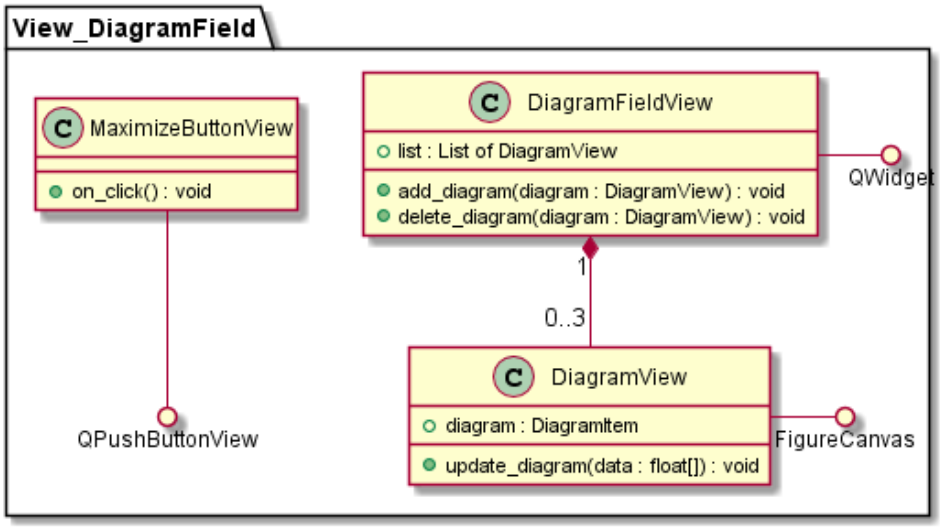

Abbildung 43: Das Paket DiagramField

View_DiagramField ist für die Visualisierung der Diagramme verantwortlich und setzt sich aus einem Maximierungsknopf und einem Feld für die Diagramme zusammen.

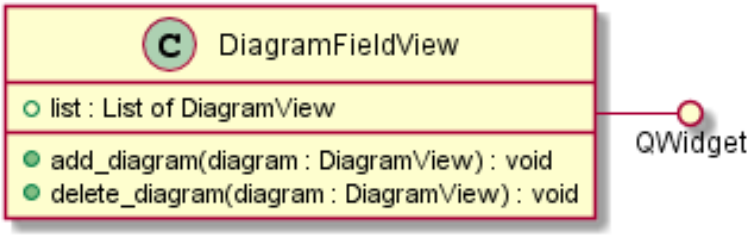

Abbildung 44: Die Klasse DiagramFieldView

Die Klasse **DiagramFieldView** repräsentiert das Feld für die Darstellung der Diagramme. Sie erbt von der *PyQt5-Klasse QWidget*

list : List of DiagramView ist eine Liste von bis zu drei Diagrammen(MK3), die auf der Diagrammfläche abgebildet werden sollen.

add_diagram(diagram : DiagramView) : void fügt ein Diagramm zu **list** hinzu und fügt *diagram* in das *Widget* ein, das die Klasse darstellt.

delete_diagram(diagram : DiagramView) : void entfernt ein Diagramm aus **list** und aus dem *Widget*.

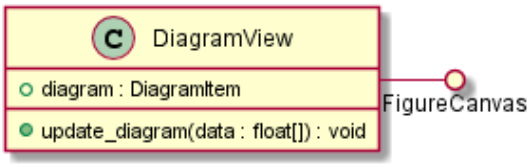

Abbildung 45: Die Klasse DiagramView

DiagramView erbt von der Klasse *FigureCanvas* und wird verwendet, um ein Diagramm darzustellen

diagram : DiagramItem speichert das Diagram-Element, zu dem das Diagramm gehört.

update_diagram(data : float[]) : void fügt dem Diagramm die Werte hinzu und zeichnet es neu.

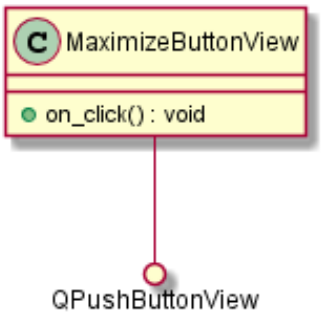

Abbildung 46: Die Klasse MaximizeButtonView

Die Klasse **MaximizeButtonView** ist für die Maximierung der Anzeigefläche zuständig. Sie erbt von der *PyQt5-Klasse QPushButtonView*

on_click : void betätigt der Benutzer den Maximierungsknopf, so wir die Anzeigefläche auf Hauptfenstergröße maximiert.

1.3.8 Konfigurationsfenster

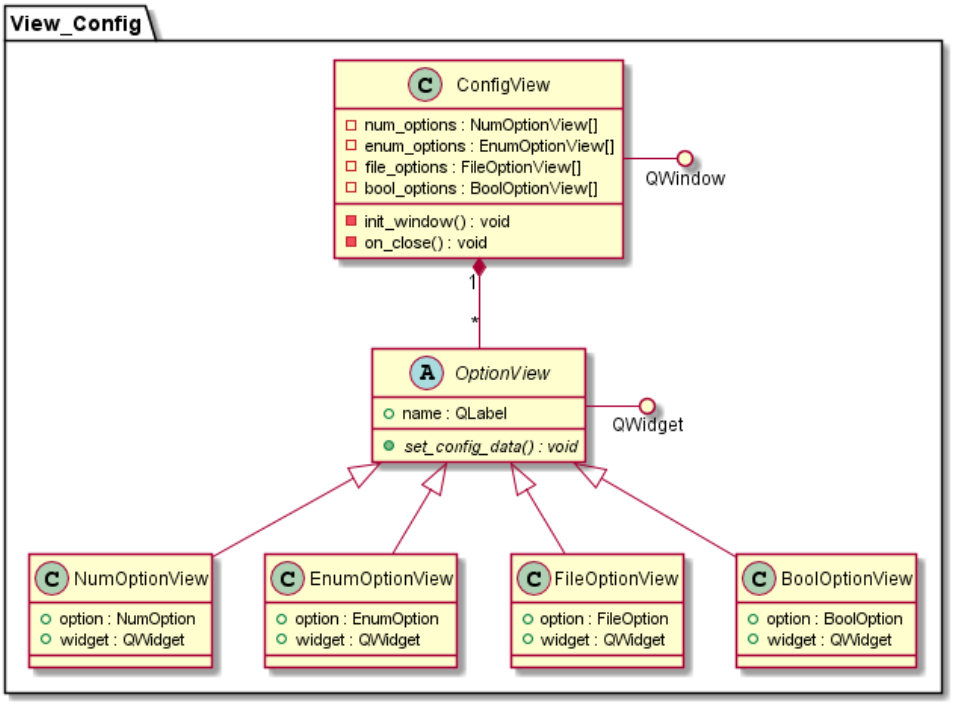

Abbildung 47: Das Paket ConfigView

Das Paket **ConfigView** repräsentiert das Öffnen und die Darstellung des Konfigurationsfensters der Arbeitsflächenelemente. Es hält die modulare Gestaltung der *Config* im Model bei und baut sein Fenster selbst modular anhand des Model auf.

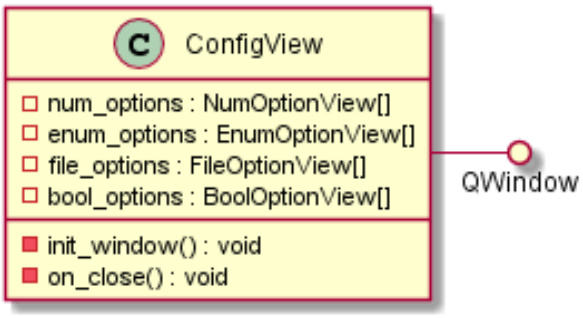

Abbildung 48: Die Klasse ConfigView

Die Klasse **ConfigView** ist für das Erstellen des Konfigurationsfensters verantwortlich und erbt von der *PyQt5-Klasse QWindow*.

num_options : List of NumOptionView speichert eine Liste von *NumOptionView*, die in Konfigurationsfenster angezeigt werden.

enum_options : List of EnumOptionView speichert eine Liste von *EnumOptionView*, die in Konfigurationsfenster angezeigt werden.

file_options : List of FileOptionView speichert eine Liste von *FileOptionView*, die in Konfigurationsfenster angezeigt werden.

bool_options : List of BoolOptionView speichert eine Liste von *BoolOptionView*, die in Konfigurationsfenster angezeigt werden.

init_window() : void wird beim Erstellen des Fensters aufgerufen und initialisiert es mit den Optionen des ausgewählten Elements. Dabei holt sie sich für jeden Typ von Option die zugehörige Liste aus der *Config* und fügt dem Inhalt des Fensters eine passende Instanz einer der Unterklassen von *OptionView* hinzu.

on_close() : void wird bei Schließen des Fensters aufgerufen und setzt die *Config* des ausgewählten Elements auf die geänderten Werte der *OptionViews*.

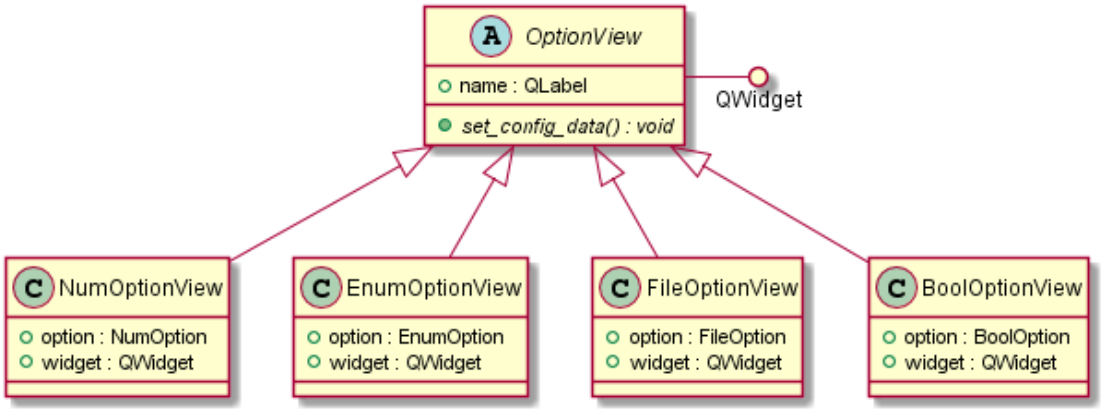

Abbildung 49: Klassendiagramm zu OptionView

Die Klasse **OptionView** beschreibt eine Option im Konfigurationsfensters und erbt von der *PyQt5-Klasse QWidget*. Die besitzt vier Unterklassen, die für die Darstellung je einer spezifischen Option zuständig sind.

name : QLabel speichert den Namen der Option in einem *QLabel* ab.

set_config_data() : void setzt den vom Benutzer ausgewählten Werts in die *Config* im Model.

option : Option speichert die zugehörige Option im Model.

widget : QWidget speichert das QWidget, dass hinter dem Namen angezeigt wird und ist für jeden Typ Option unterschiedlich:

- **NumOptionView** beinhaltet ein QLineEdit und lässt nur Zahlen zu
- **EnumOptionView** beinhaltet eine QComboBox
- **FileOptionView** beinhaltet einen QPushButton zum Öffnen eines QFileDialog
- **BoolOptionView** beinhaltet eine QCheckBox

1.3.9 Menüleiste

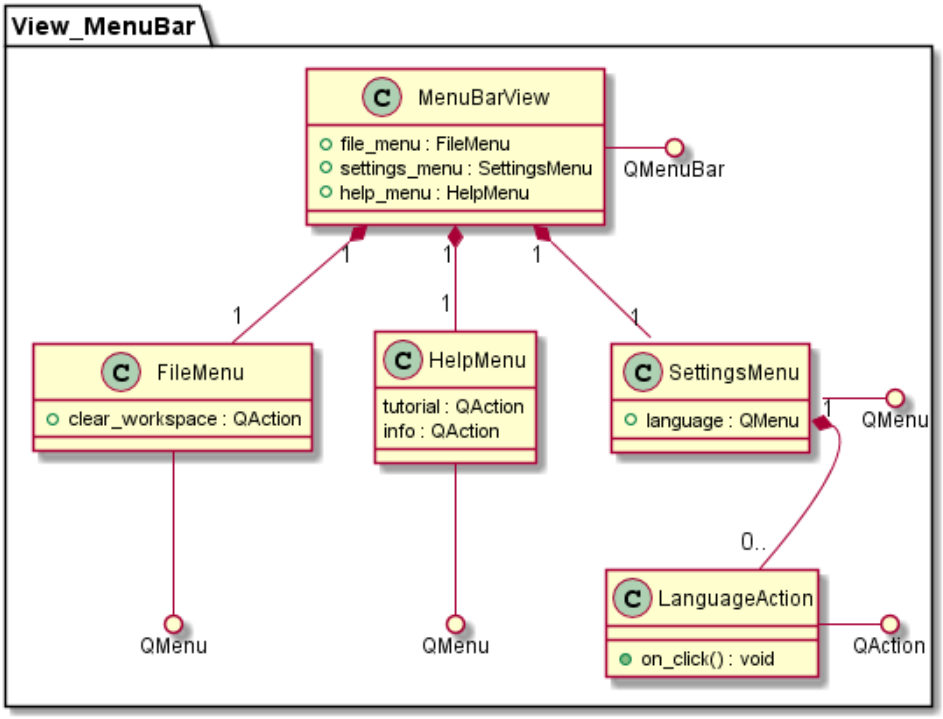

Abbildung 50: Das Paket MenuBar

Das Paket **MenuBar** repräsentiert die Menüleiste der GUI. Sie enthält alle Klassen zur Darstellung der Menüleiste und den weiteren Untermenüs.

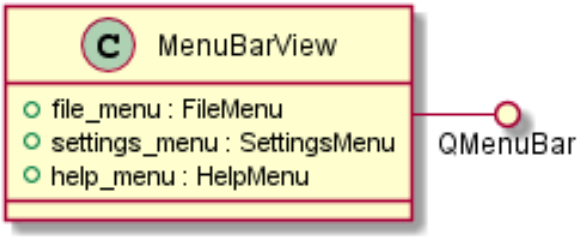

Abbildung 51: Die Klasse MenuBarView

Die Klasse **MenuBarView** stellt die Menüleiste dar. Sie erbt von der *PyQt5-Klasse QMenuBar*

file_menu : FileMenu ist eine Referenz auf die Klasse **FileMenu** und stellt das erste Menü da, das sich öffnet, sobald darauf gedrückt wird.

settings_menu : SettingsMenu ist eine Referenz auf die Klasse **SettingsMenu** und stellt das zweite Menü da, das sich öffnet, sobald darauf gedrückt wird.

help_menu : HelpMenu ist eine Referenz auf die Klasse **HelpMenu** und stellt das dritte Menü da, das sich öffnet, sobald darauf gedrückt wird.

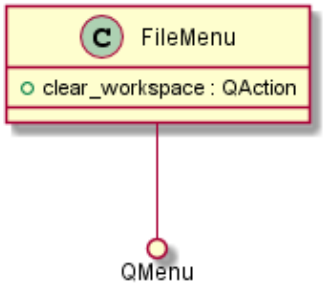

Abbildung 52: Die Klasse FileMenu

Die Klasse **FileMenu** repräsentiert das Dateimenü der Menübar. Sie erbt von der *PyQt5-Klasse QMenu.*

clear_workspace : QAction ist ein Untermenüpunkt von **FileMenu** und löscht beim betätigen die bestehende Konfiguration auf der Arbeitsfläche.

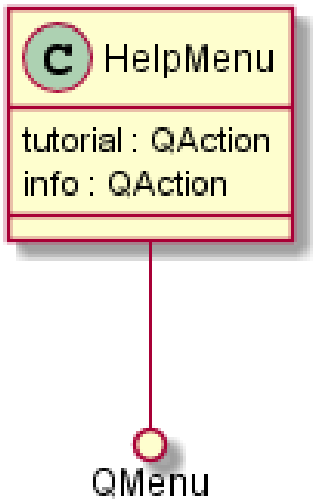

Abbildung 53: Die Klasse HelpMenu

Die Klasse **HelpMenu** repräsentiert das Untermenü *Hilfe*. Es kann ein Tutorial oder eine Information ausgewählt werden. Die Klasse erbt von der *PyQt5-Klasse QMenu*

tutorial : QAction es wird beim Betätigen ein Tutorial gestartet, das dem Benutzer die Funktionsweise des Programms erläutert.(WK3)

info : QAction es öffnet sich ein Fenster, indem der Benutzer die Version und die Lizenz des Programms angezeigt bekommt.

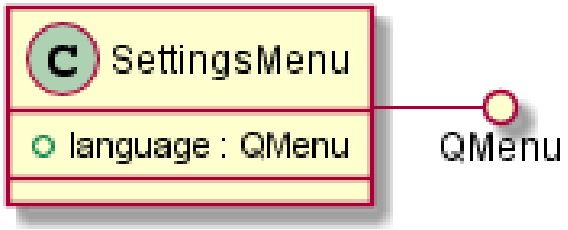

Abbildung 54: Die Klasse SettingsMenu

Die Klasse **SettingsMenu** repräsentiert das Einstellungsmenü. Sie erbt von der *PyQt5-Klasse QMenu*.

language : QMenu Dem Benutzer steht eine Auswahl an Sprachen zur Verfügung.(MK12)

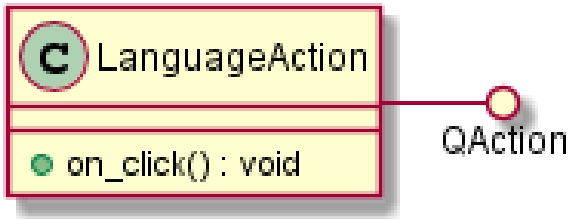

Abbildung 55: Die Klasse LanguageAction

Die Klasse **LanguageAction** ist für die Umsetzung der vom Benutzer ausgewählten Sprache zuständig. Sie erbt von der *PyQt5-Klasse QAction*.

on_click() : void Beim auswählen der jeweiligen Sprache wird das Programm in diese übersetzt.

2 Abläufe

2.1 Programm starten

In der Abbildung Programmstart wird der Ablauf des Programmstarts beschrieben. Erst wird das Hauptfenster initialisiert. Darauf folgen die View-Elemente für das Menü, die Listen, Arbeitsfläche, Infobar und Diagrammansicht.

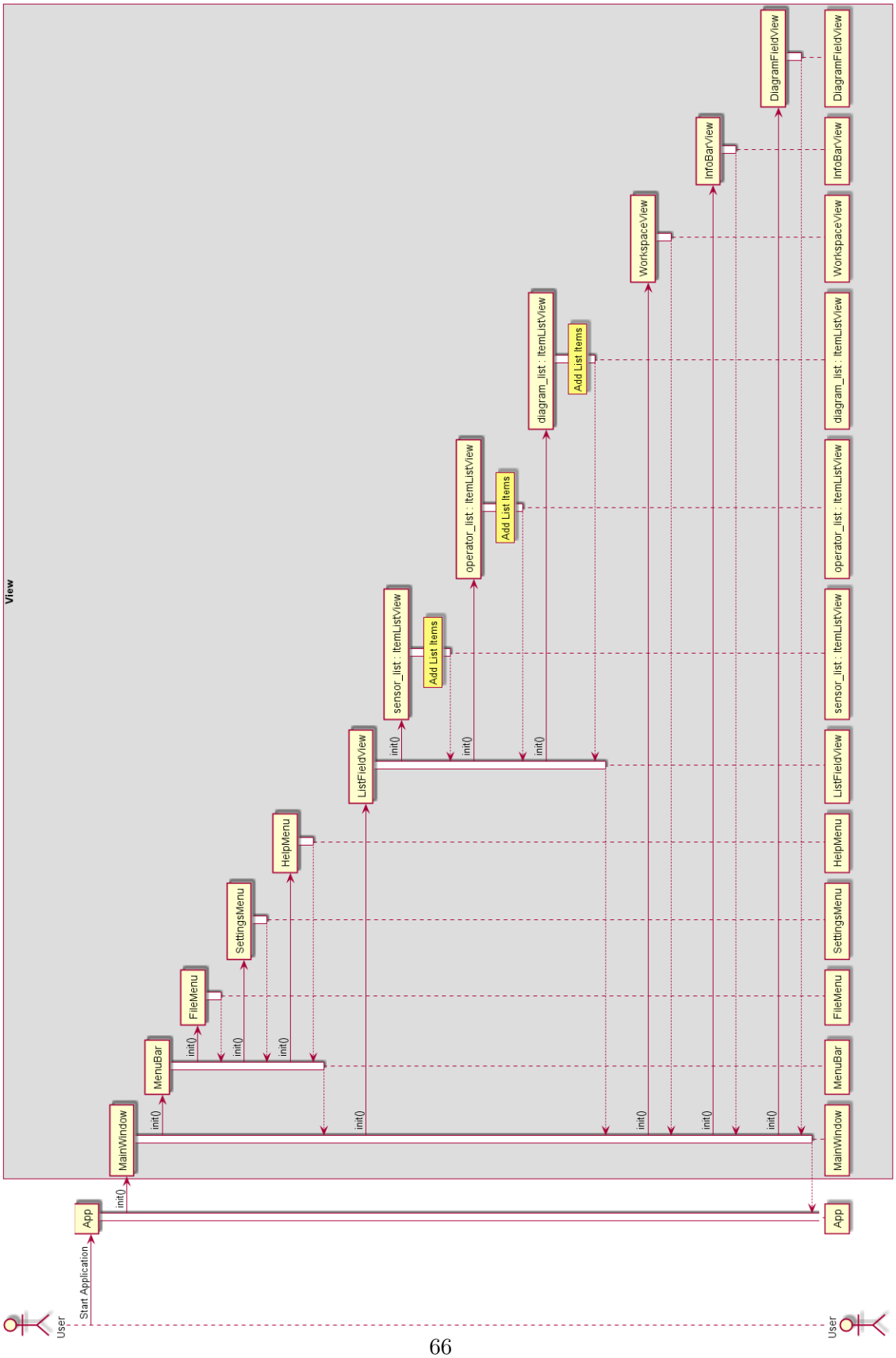

Abbildung 56: Sequenzdiagramm: Programmstart

2.2 Item in der Arbeitsfläche ablegen

In der Abbildung Item in der Arbeitsfläche ablegen wird beschrieben was passiert, wenn man ein Item in der Arbeitsfläche ablegt. Erst wird geprüft, ob das Item in der Arbeitsfläche abgelegt wurde. Dann wird das Itemmodel zu dem Item im View inklusive Ein- und Ausgängen initialisiert. Dann initialisiert das Itemview sein entsprechendes Itemmodel mit Ein- und Ausgängen.

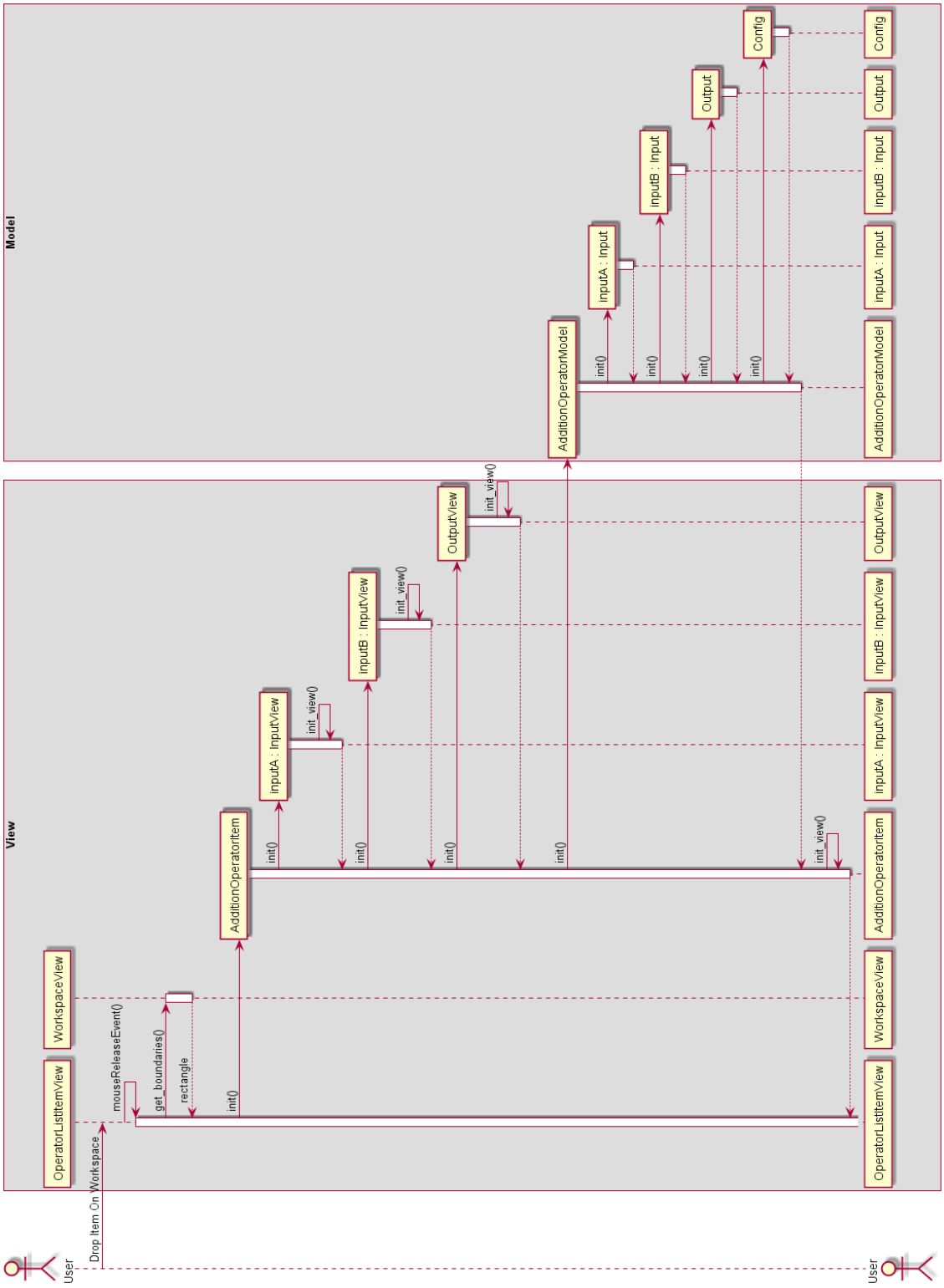

Abbildung 57: Sequenzdiagramm: Item in der Arbeitsfläche ablegen

2.3 Startknopf betätigen

Wird der Startknopf gedrückt, so stößt der Manager alle Diagramme an für ihre Ausgänge Funktionen und Einheiten zu berechnen. Dies wird rekursiv über den Workspace erreicht, der eine Liste aller Outputs, nach ihrer Id sortiert, hat. Dann wird eine Schleife betreten. Hier werden im Takt alle Sensoren angestoßen ihren Wert auszulesen. Dann berechnen die Diagramme ihren Wert. Und dann wird der View benachrichtigt, er soll die neuen Werte einzeichnen.

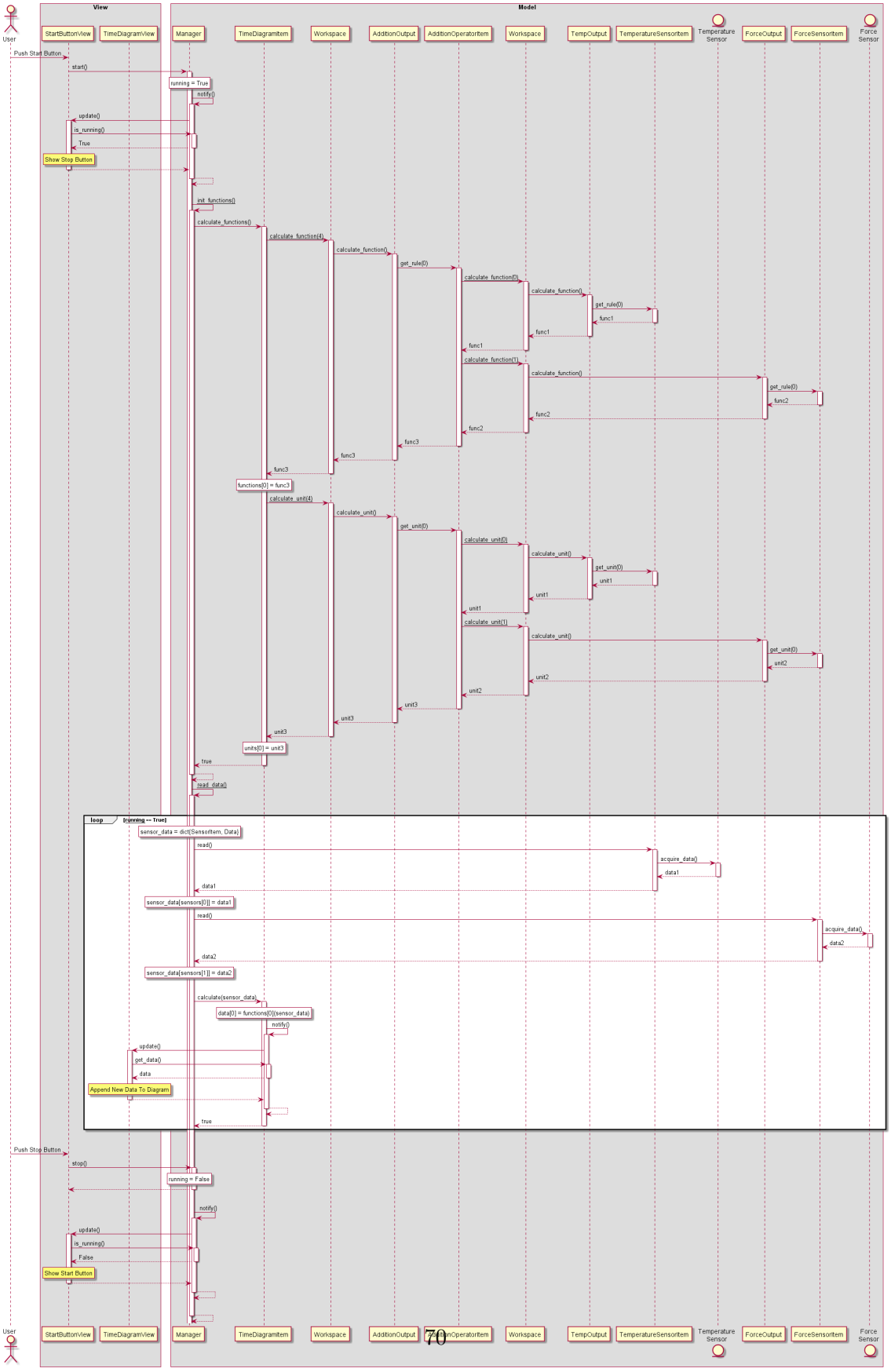

Abbildung 58: Sequenzdiagramm: Startknopf betätigen

3 Nicht-Umsetzung des Pflichtenheftes

Folgende Wunschkriterien konnten wir nicht umsetzen, da unsere Priorität zuerst auf alle Musskriterien des Pflichtenhefts lag und die unten stehenden Punkte den Zeitrahmen gesprengt hätten.

- WK1
- WK5
- WK6

4 Gantt-Diagramm

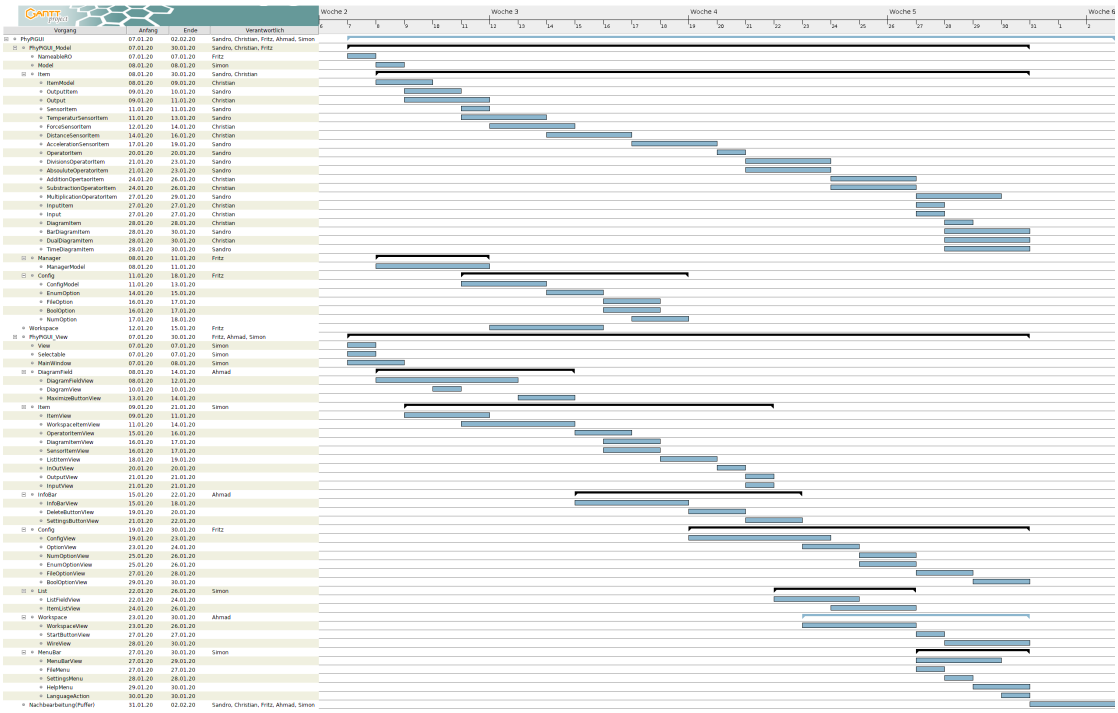

Abbildung 59: Gantt-Diagramm