

Entwurfsdokumentation

# **Visuelle Programmiersprache für den Physikunterricht zur Datenerfassung auf einem Raspberry Pi**

**Version 0.0.0**

David Gawron      Stefan Geretschläger      Leon Huck  
Jan Küblbeck      Linus Ruhnke

6. Juli 2019

# Inhaltsverzeichnis

<b>1</b>	<b>Ziel der Entwurfsdokumentation</b>	<b>3</b>
<b>2</b>	<b>Klassenbeschreibung</b>	<b>4</b>
2.1	Backend . . . . .	5
2.2	Model . . . . .	6
2.2.1	MRunReaction . . . . .	6
2.2.2	BuildingBlockDirectory . . . . .	6
2.2.3	Core . . . . .	6
2.2.4	Core.SensorLogic . . . . .	7
2.2.5	Core.TransformationLogic . . . . .	7
2.2.6	Transformation . . . . .	7
2.2.7	Function . . . . .	7
2.2.8	Core.RepresentationLogic . . . . .	7
2.2.9	Core.ChannelLogic . . . . .	7
2.2.10	BuildingBlockBuilder . . . . .	8
2.3	Controller . . . . .	11
2.3.1	CommandManager . . . . .	12
2.3.2	Command bzw. konkrete Befehle . . . . .	12
2.3.3	Verbindung zum View . . . . .	14
2.3.4	Verbindung zum Model . . . . .	14
2.4	View . . . . .	15
2.4.1	MainWindow . . . . .	15
2.4.2	Menues . . . . .	16
2.4.3	Configuration . . . . .	17
2.4.4	BuildingBlockProperties . . . . .	20
2.4.5	Button . . . . .	22
2.4.6	OptionAndHelp . . . . .	24
2.4.7	Exception . . . . .	25
2.4.8	FacadeModelView . . . . .	27
<b>3</b>	<b>Sequenzdiagramme</b>	<b>28</b>
<b>4</b>	<b>Änderungen am Pflichtenheft</b>	<b>29</b>
<b>5</b>	<b>Formale Spezifikationen von Kernkomponenten</b>	<b>30</b>
<b>6</b>	<b>Weitere UML Diagramme</b>	<b>31</b>
<b>7</b>	<b>Anhang</b>	<b>32</b>
7.1	Vollständiges Klassendiagramm . . . . .	33
<b>8</b>	<b>Glossar</b>	<b>34</b>

# 1 Ziel der Entwurfsdokumentation

Die Entwurfsdokumentation soll, aufbauend auf das Pflichtenheft, Entwurfsentscheidungen festhalten. Der Rahmen des Entwurfes wird durch einen *Model-View-Controller* (MVC) gebildet. Die Daten werden durch das Backend zu der Verfügung gestellt. Jedes dieser Pakete kommuniziert über eine Fassade. Dadurch werden die Pakete von einander abgekoppelt. Durch diesen grundlegenden Aufbau wird die Software in vier unabhängige Komponenten aufgeteilt, die unabhängig voneinander implementiert und später erweitert werden können.



Abbildung 1: Die grobe Struktur des Entwurfs

## **2 Klassenbeschreibung**

Im folgenden sollen alle Klassen mit ihren Funktion beschrieben werden. Der Aufbau orientiert sich dabei an der in 1 aufgeführten Struktur.

## 2.1 Backend

## 2.2 Model

### 2.2.1 MRunReaction

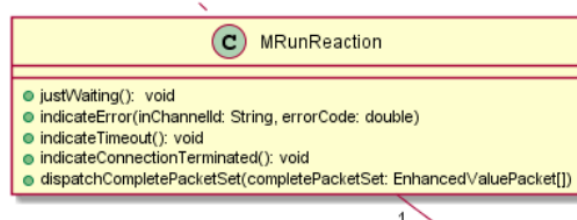


Abbildung 2: Darstellung der Klasse MRunReaction

Die Klasse MRunReaction ist in Abbildung 2 zu sehen. Sie implementiert das Interface MRunForward, welches im Cache-Modul zu finden ist. MRunReaction dient als Verbindung zwischen Cache und Modul. Durch ihre fünf Methoden wird der Datenfluss vom Cache zu den Sensorbausteinen im Model verwaltet. Die Methode `justWaiting` signalisiert dem Modul, dass eine Verbindung besteht, aber kein Datenfluss stattfindet. Durch die Methode `indicateError` dient dazu, dem Model das Auftreten eines Fehlers zu signalisieren. Dabei wird als Parameter ein Fehlercode und die ID des betroffenen Eingangschannels beigefügt. Durch die Methode `timeOut` wird eine außerplanmäßige Unterbrechung einer Verbindung signalisiert. Durch die Methode `connectionTerminated` wird hingegen das planmäßige Schließen einer Verbindung signalisiert. Die Methode `dispatchCompletePacketSet` übergibt dem Model ein Set aus Datenpaketen, so dass jeder Eingangskanal jedes Sensors in der Messkonfiguration ein Packet erhält. Ein Datenpaket besteht hier aus Wert, Zielchannel und Zeitstempel.

### 2.2.2 BuildingBlockDirectory

### 2.2.3 Core

#### MeasurementConfiguration

#### MeasurementRun

#### BuildingBlock

**YamlRepresentation**

**PrototypCollection**

**HelpMessage**

**2.2.4 Core.SensorLogic**

**Sensor**

**VirtualSensor**

**PhysicalSensor**

**2.2.5 Core.TransformationLogic**

**2.2.6 Transformation**

**2.2.7 Function**

**2.2.8 Core.RepresentationLogic**

**Representation**

**TableRepresentation**

**XYRepresentation**

**2.2.9 Core.ChannelLogic**

**Channel**

**InChannel**

**OutChannel**

**ChannelState**

**Connected**

**UnConnected**

**ValueReady**

**MRunInfo**

## **2.2.10 BuildingBlockBuilder**

**Director**

**Builder**

**SensorBuilder**

**VirtualBuilder**

**PhysicalBuilder**

**TransformationBuilder**

**RepresentaationBuilder**



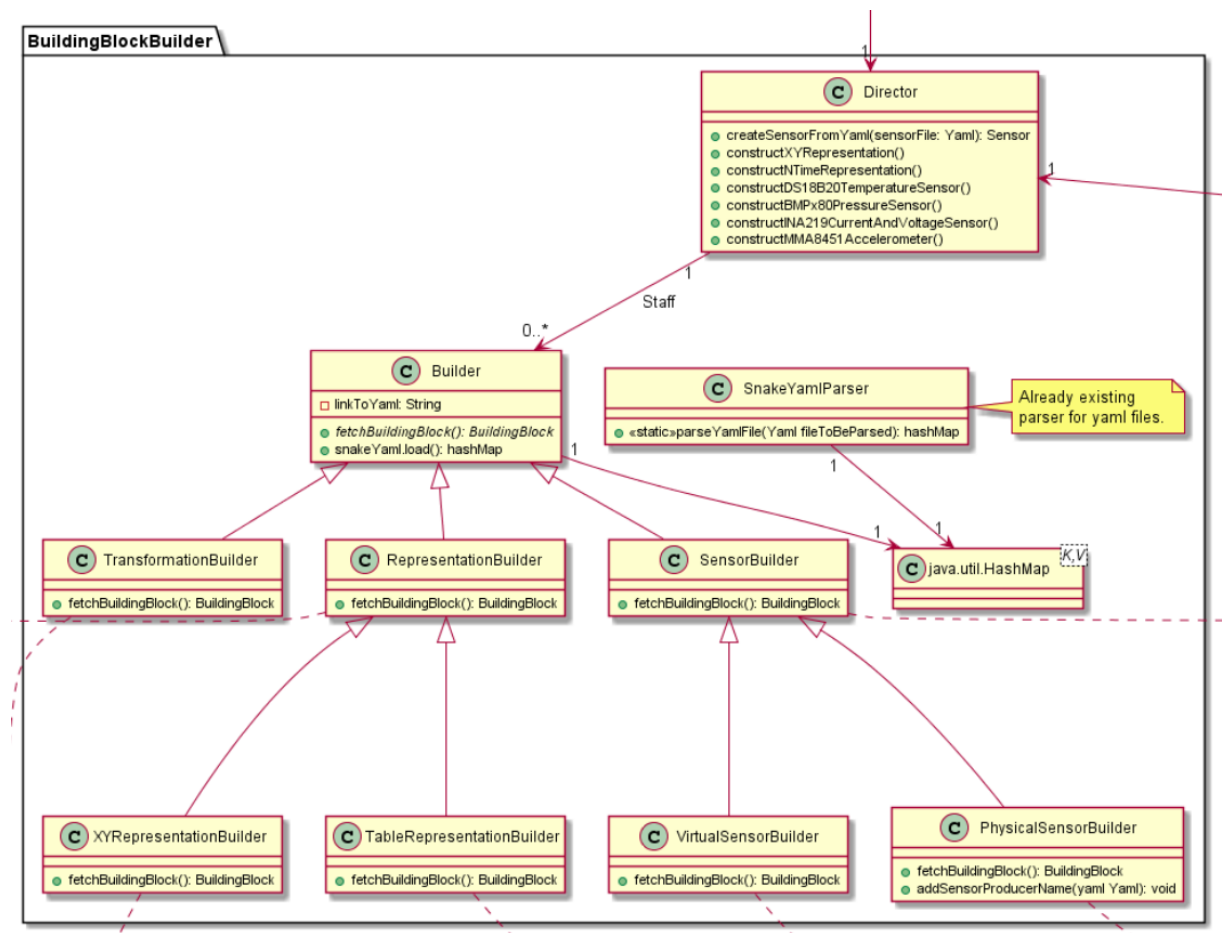


Abbildung 3: Darstellung des BuildingBlockBuilder

**TableRepresentstationBuilder**

**XYRepresentstationBuilder**

**java.util.HashMap** $\langle K, V \rangle$

**SnakeYamlParser**

## 2.3 Controller

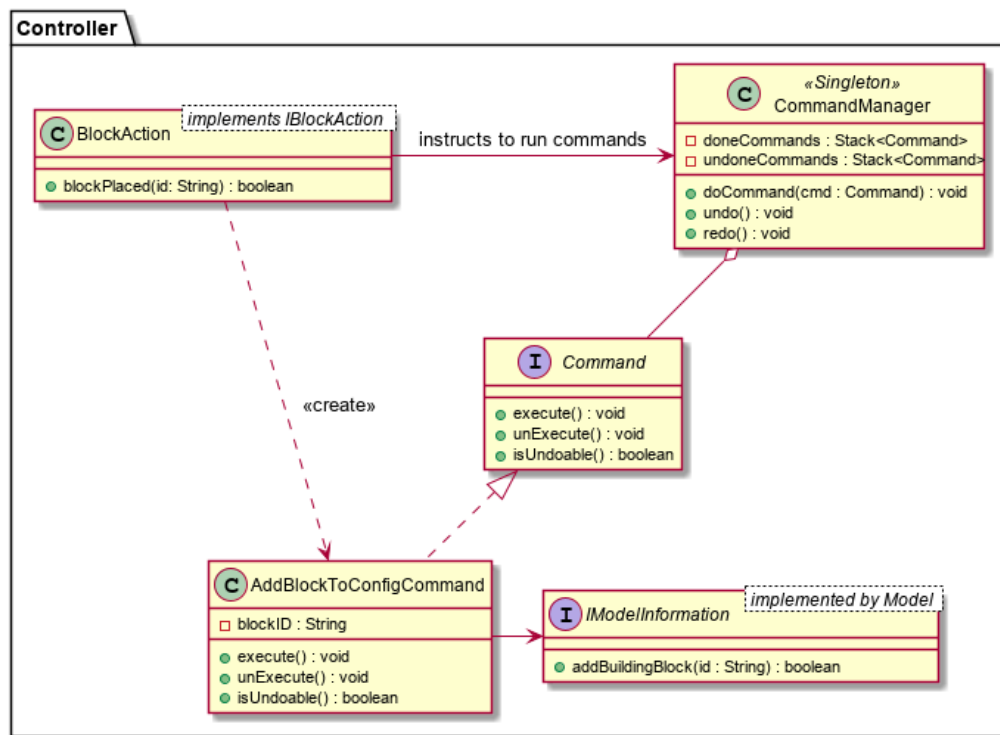


Abbildung 4: Die Struktur des Controllers (Ausschnitt)

Der Aufbau des Controllers setzt das Entwurfsmuster *Kommando* (*Command*) um. Die Rollen sind dabei folgendermaßen:

- Die Klasse *CommandManager* erfüllt die Rolle des *Aufrufers* (*Invoker*).
- Die Schnittstelle *Command* erfüllt die Rolle eines *Befehls* im abstrakten Sinne.
- Die Klassen, welche *Command* implementieren, erfüllen die Rolle der *konkreten Befehle*.
- Die Rolle des (bzw. der) *Klienten* wird durch die Klassen *ButtonAction*, *BlockAction*, bzw. *ConnectionAction* erfüllt. Diese bilden die Schnittstelle, über welche das *View*-Modul auf den Controller zugreift.
- Die Rolle der *Empfänger* wird durch die Schnittstelle(n) zum *Model*-Modul erfüllt.

### 2.3.1 CommandManager

Die Klasse `CommandManager` hat die Funktion, die Ausführung konkreter Befehle zu veranlassen. Der `CommandManager` ist als *Singleton* definiert um sicherzustellen, dass von allen Klienten auf dieselbe Instanz zugegriffen wird.

Anhand eines *Undo*- und eines *Redo-Stacks* bietet der `CommandManager` außerdem die Möglichkeit an, bereits ausgeführte Befehle rückgängig zu machen (bzw. rückgängig gemachte Aktionen wiederherzustellen). Nicht alle Befehle können rückgängig gemacht werden.

### 2.3.2 Command bzw. konkrete Befehle

Die Schnittstelle `Command` und die konkreten Klassen, welche die Schnittstelle implementieren, sind die Befehle des Controllers. Jeder konkrete Befehl kapselt eine genau definierte Funktionalität. Weitere Befehle können problemlos hinzugefügt werden, ohne bestehende Klassen verändern zu müssen.

Bestimmte Befehle können durch *unExecute()* rückgängig gemacht werden. In diesen Fällen wird durch *isUndoable()* immer *true* zurückgegeben.

Andere Befehle können nicht rückgängig gemacht werden. Dann ist die Methode *unExecute()* leer und *isUndoable()* gibt *false* zurück.

**AddBlockToConfigCommand** Dieser Befehl fügt einen gegebenen Baustein zum Konfigurationsfeld hinzu.

**RemoveBlockFromConfigCommand** Dieser Befehl entfernt einen Baustein aus dem Konfigurationsfeld.

**EditBlockPropertiesCommand** Dieser Befehl verändert die Eigenschaften eines Bausteins.

**CloneBlockCommand** Dieser Befehl kloniert einen bestehenden Bausteinprototyp.

**ExportBlockPrototypeCommand** Dieser Befehl exportiert einen Bausteinprototypen als Datei.

**CreateChannelConnectionCommand** Dieser Befehl erstellt eine Verbindung zwischen zwei gegebenen Kanälen (welche wiederum zu Bausteinen gehören).

**ModifyChannelConnectionCommand** Dieser Befehl verändert Start- und/oder Endpunkt einer Verbindung zwischen zwei Kanälen.

**DeleteChannelConnectionCommand** Dieser Befehl löscht eine Verbindung zwischen zwei Kanälen.

**StartRunCommand** Dieser Befehl startet einen Messlauf.

**StopRunCommand** Dieser Befehl beendet einen aktiven Messlauf.

**PauseRunCommand** Dieser Befehl pausiert einen aktiven Messlauf.

**ResumeRunCommand** Dieser Befehl setzt einen pausierten Messlauf fort.

**SaveConfigCommand** Dieser Befehl speichert die aktuelle Messkonfiguration an einem übergebenen Dateipfad.

**LoadConfigCommand** Dieser Befehl lädt eine Messkonfiguration von einem angegebenen Pfad.

**ResetConfigCommand** Dieser Befehl entfernt alle Elemente aus dem Konfigurationsfeld. (Verwendung optional.)

### 2.3.3 Verbindung zum View

Durch das View-Modul werden mehrere Schnittstellen definiert.

**BlockAction** Die Klasse BlockAction implementiert die Schnittstelle IBlockAction des Views.

**ButtonAction** Die Klasse ButtonAction implementiert die Schnittstelle IButtonAction des Views.

**ConnectionAction** Die Klasse ConnectionAction implementiert die Schnittstelle IConnectionAction des Views.

### 2.3.4 Verbindung zum Model

Es werden Schnittstellen vorgegeben, die durch das Model-Modul implementiert werden.

**IModelInformation** Diese Schnittstelle stellt Methoden zur Verfügung, anhand derer die Messkonfiguration und darin enthaltene Bausteine verändert werden können.

**IMeasurementRun** Diese Schnittstelle stellt Methoden zur Verfügung, anhand derer ein Messlauf gestartet, angehalten und fortgeführt werden kann.

## 2.4 View

Das Paket View, stellt gemäß des MVC- Entwurfsmusters die Darstellungen des Modells dar und realisiert Benutzerinteraktionen auf der graphischen Benutzeroberfläche.

### 2.4.1 MainWindow

Die Klasse MainWindow stellt den Rahmen der Benutzeroberfläche dar. Alle Restlichen graphischen Oberflächen werden durch das MainWindow instanziiert. Dazu gehört das Konfigurationsfeld, Buttonmenü, Konfigurationsbausteinmenü, Hilfe , Optionen und Fehlerfenster. Da MainWindow, das Entwurfsmuster Singleton verwendet, kann die Anwendung nur ein MainWindow besitzen soll. Bei Schließen des MainWindow wird ebenso die gesamte Anwendung beendet.

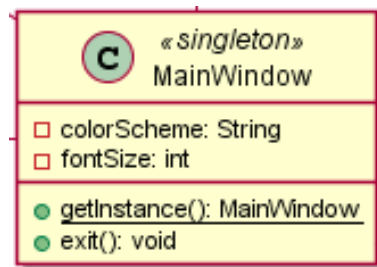


Abbildung 5: Die Klasse MainWindow

## 2.4.2 Menues

Menüs bieten dem Benutzer eine übersichtliche visuelle Zusammenfassung der Darstellungen der konkreten Bausteine und Knöpfe.

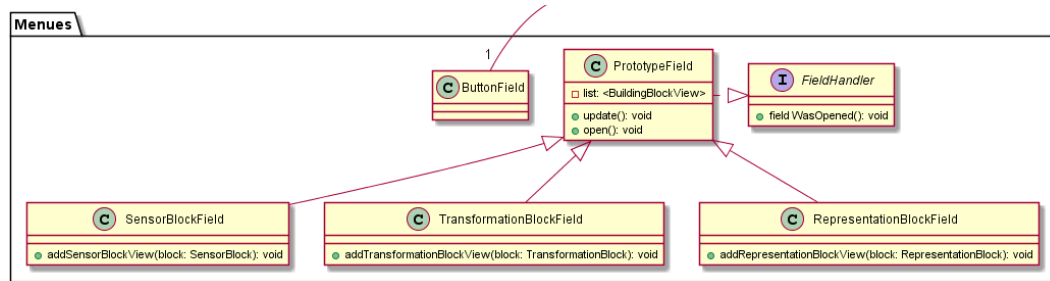


Abbildung 6: Aufbau des Menü-Paket

**PrototypeField** Die Klasse **PrototypeField** ist die Über-Klasse zu **SensorBlockField**, **TransformationBlockField** und **RepresentationBlockField**. Sie stellt die Menüfläche dar, in welcher vordefinierte Konfigurationsbausteine je nach Kategorie dargestellt werden und der Benutzer sie mit dem Mauszeiger in das Konfigurationsfeld ziehen und damit positionieren kann. Diese vordefinierten Bausteine werden über das Backend eingelesen und werden über das Model im Directory zur Verwendung auf der Benutzeroberfläche bereitgestellt.

**FieldHandler** Das Interface **FieldHandler** nimmt Benutzereingaben entgegen, in diesem Fall werden das Öffnen der Menüflächen registriert und an das Feld weitergeleitet.

**SensorBlockField** Die Klasse **SensorBlockField** stellt die Menüfläche dar, in welcher alle Sensorbausteine angezeigt werden.

**TransformationBlockField** Die Klasse **TransformationBlockField** stellt die Menüfläche dar, in welcher alle Transformationsbausteine angezeigt werden.

**RepresentationBlockField** Die Klasse **RepresentationBlockField** stellt die Menüfläche dar, in welcher alle Representationsbausteine angezeigt werden.



**ButtonField** Die Klasse ButtonField stellt die Menüfläche dar, in welcher alle konkreten Knöpfe platziert sind und diese für den Benutzer verwendbar sind.

### 2.4.3 Configuration

**ConfigurationField** Die Klasse KonfigurationField stellt das Konfigurationfeld dar, in welchem der Benutzer eine Messkonfiguration aufbauen kann. Konfigurationsbausteine, welche der Benutzer in das Konfigurationfeld platziert werden in einer Liste gespeichert. Konfigurationsbausteine, welche der Benutzer aus dem Konfigurationfeld entfernt, werden aus der Liste gelöscht. Beim Platzieren der Konfigurationsbausteine in das Konfigurationfeld wird dem Konfigurationsbaustein eine eindeutige Position zugeteilt, welche in Form von einer x-Koordinate und einer y-Koordinate dargestellt wird. Die Liste der Bausteine kann ebenfalls von außerhalb ausgelesen oder gesetzt werden, falls z.B. die Anordnung der Bausteine und Verbindungen gespeichert oder gesetzt werden soll.

**BuildingBlockView** Die Klasse BuildingBlockView ist die Überklasse der Darstellungen der Konfigurationsbausteine. Bausteine werden über das Directory erzeugt, indem zu jedem im Directory gespeicherten Baustein eine Darstellung dieses Bausteins erzeugt wird. Name und InitId bleiben bei Erzeugung des Bausteins gleich, jedoch wird der Baustein, um die visuellen Komponenten Koordinaten, Farbe, Form und Größe erweitert. Konfigurationsbausteine besitzen eine eindeutige Initialisierungs-ID, darunter versteht man die ID, welche der Baustein beim Erstellen durch das Model bekommt. Jede konkrete Instanz dieses Bausteins besitzt diese Initialisierungs-ID (InitId). Wenn ein Baustein mehrfach durch den Benutzer in das Konfigurationfeld gezogen wird, könnte dies dazu führen, dass diese Initialisierungs-ID nicht mehr eindeutig für diesen Baustein wäre. Deswegen besitzt jeder im Konfigurationfeld platzierte Baustein eine Konfigurations-ID. Diese ID ist eindeutig für diesen Baustein und somit ist dieser Baustein unterscheidbar von weiteren Bausteinen gleichem Prototyps. Ebenfalls besitzt ein Baustein einen Namen und falls sie im Konfigurationfeld platziert werden ihre Position anhand der Koordinaten x und y. Form, Farbe und Größe sind ebenfalls festgelegt. Bausteine werden über das Directory erzeugt, indem zu jedem im Directory gespeicherten Baustein eine Darstellung dieses Bausteins erzeugt wird. Name und ID bleiben bei Erzeugung des Bausteins gleich, jedoch wird der Baustein, um die visuellen Komponenten Koordinaten, Farbe, Form erweitert.

**SensorBlockView** Die Klasse SensorBlockView stellt einen Sensorbaustein dar. Sensorbausteine, welche in dem Konfigurationfeld platziert werden, können mit anderen Bausteinen verbunden werden, was im Messlauf einen Datenfluss über die verbundenen Bausteine erlaubt. Sensorbausteine besitzen, im Gegensatz zu anderen Konfigurations-

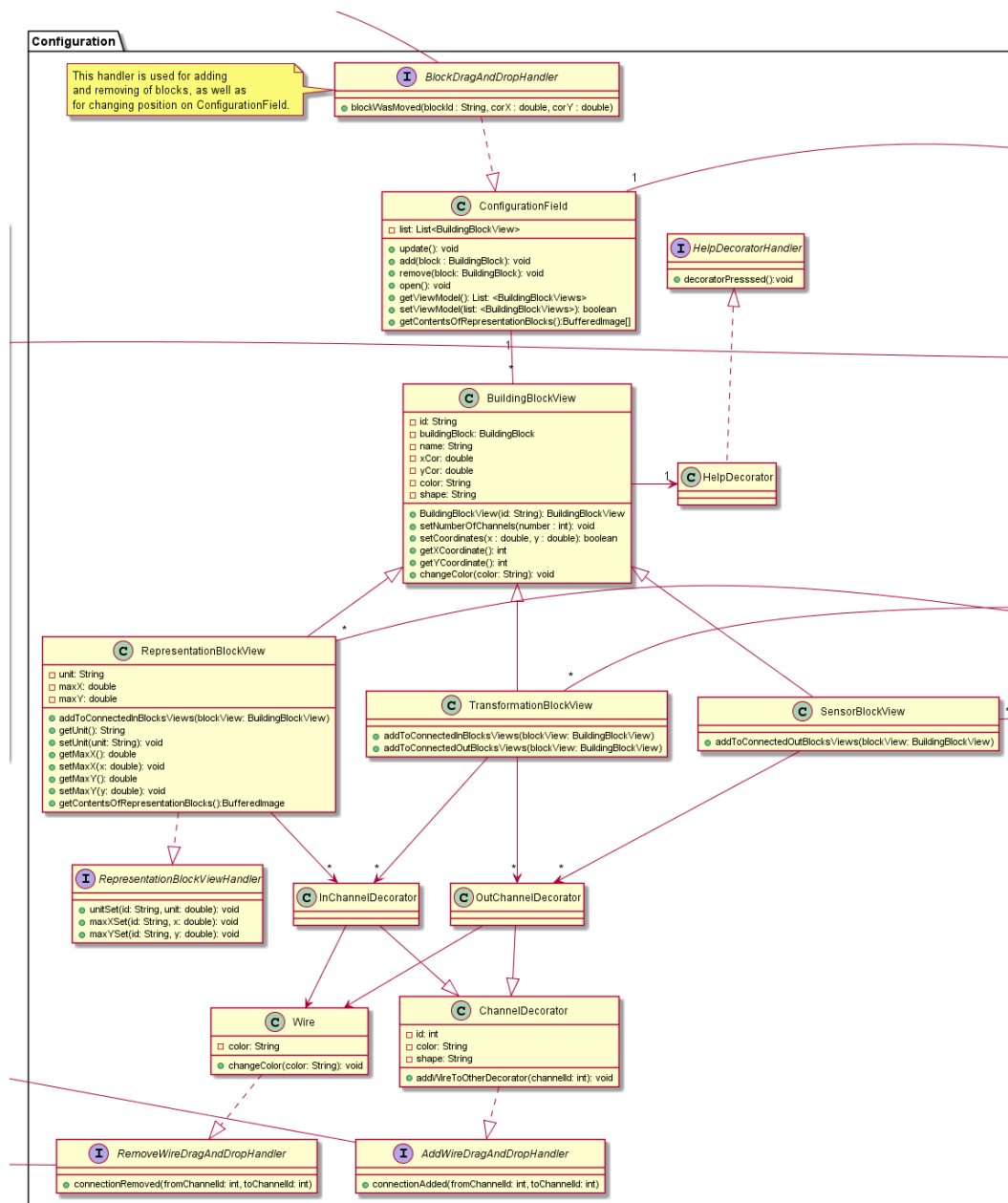


Abbildung 7: Aufbau des Konfigurations Paket

bausteinen nur Datenausgänge, über welche sie verbunden werden können, da Sensoren, gemäß physikalischer Repräsentation nur Datenausgänge besitzen.

**TransformationBlockView** Die Klasse `TransformationBlockView` stellt einen Transformationsbaustein dar. Transformationsbausteine besitzen eine vordefinierte Funktion, welche die Messdaten nach der Funktion transformiert. Transformationbausteinen besitzen Eingänge, welche Daten von Sensorenbausteinen oder anderen Transformationsbausteine empfangen können. Ausgänge der Transformationsbausteine können nur an weitere Transformationsbausteine oder Darstellungsbausteine angebunden werden, um einen sinnvollen Datenfluss zu ermöglichen.

**RepresentationBlockView** Die Klasse `RepresentationBlockView` stellt einen Darstellungsbaustein dar, dieser bestimmt, wie die Messdaten visualisiert werden. Dafür bekommt der Repräsentationsbaustein die visuelle Darstellung in dem Darstellungsgerüst (z.B: Graph, Tabelle) mit den dargestellten Messdaten. Zur Speicherung der visuellen Darstellung der Daten muss ein Bild erstellt werden, welches zum Speichern weiter geleitet wird. Darstellungsbausteine besitzen nur Eingänge, da dargestellt Messdaten nicht mehr verarbeitet werden.

#### 2.4.4 BuildingBlockProperties

Damit Benutzer Informationen über einzelne Bausteine bekommt, welche ihm das Benutzen der Anwendung erleichtern würden, wie auch eine tiefere Einsicht über die Funktionsweise bietet, stellt jeder Baustein ein eigenes „Eigenschaften-Menü“ bereit, in welchem dem Benutzer die wichtigsten Eigenschaften zu sehen bekommt.

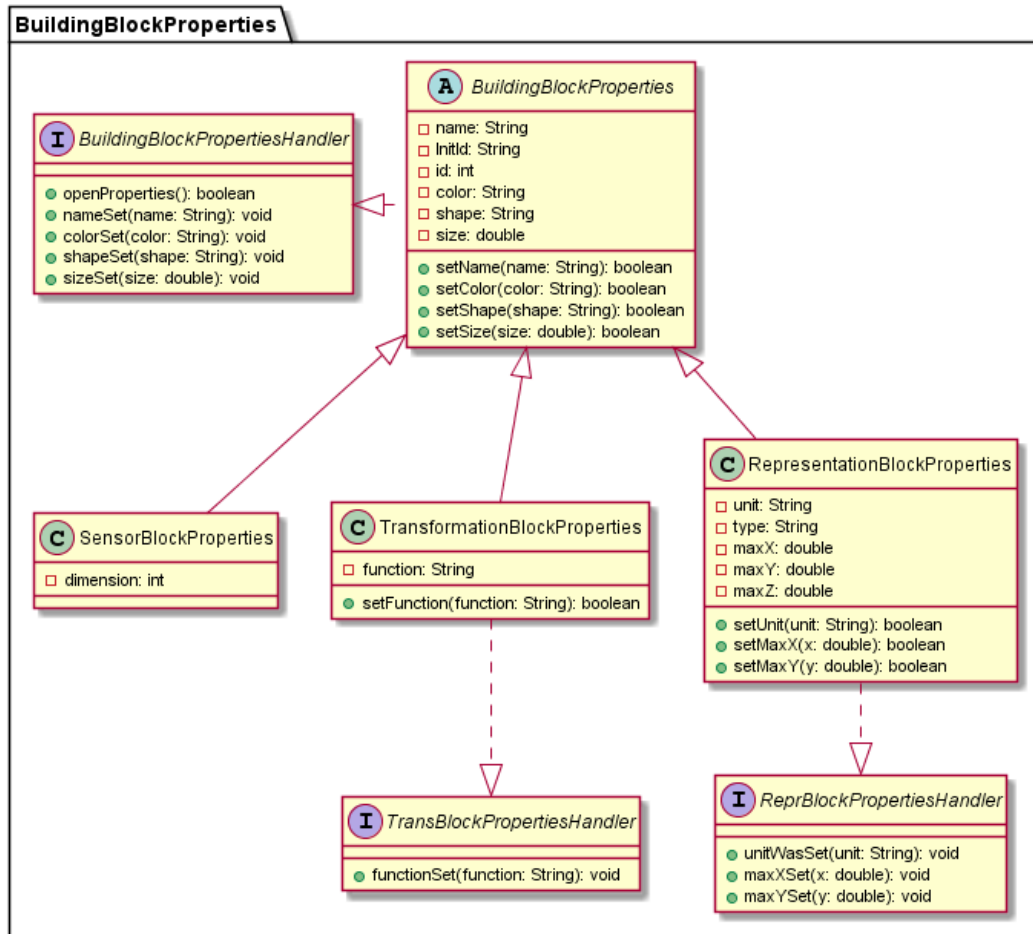


Abbildung 8: Aufbau des BuildingBlockProperties-Paket

**BuildingBlockProperties** Die abstrakte Klasse **BuildingBlockProperties** stellt alle Eigenschaften der konkreten Bausteine dar, welche alle Arten von Bausteinen (Sensor, Transformation, Darstellung) gemeinsam haben. Dazu gehören der Name, Initialisierungs-ID, eindeutige Konfigurations-ID, Farbe, Form und Größe. Da einzelne Eigenschaften

unveränderlich sein sollen, wie die IDs lassen sich nur Name, Farbe, Form und Größe verändern.

**SensorBlockProperties** Neben den gemeinsamen Eigenschaften besitzt die Unterklasse `SensorBlockProperties` ebenfalls das Attribut der Dimension, welches darstellt über wie viele Kanäle dieser Sensor Messdaten liefert. Da dieses Attribut für Sensoren unveränderlich ist, kann diese ebenfalls vom Benutzer nicht verändert werden.

**TransformationBlockProperties** Transformationsbausteine besitzen neben den Standard-Eigenschaften noch eine vordefinierte Funktion für jeden Transformationsbaustein. Um dem Benutzer zu erlauben neue Transformationsbausteine zu definieren ist die Funktion veränderbar.

**RepresentationBlockProperties** Da Repräsentationsbausteine die visuelle Repräsentation beschreiben besitzen diese für die Darstellung notwendige Eigenschaften, wie Einheit, Maximalwerte der Achsen und Art der Darstellung. Um den Benutzer die Möglichkeit zu geben die Darstellung auf die Messwerte anzupassen, lassen sich Einheit und Maximalwerte vom Benutzer einstellen.

**BuildingBlockPropertiesHandler** Damit der Benutzer Attribute der Konfigurationsbausteine ändern kann, bietet der `BuildingBlockPropertiesHandler` Methoden an, um die gemeinsamen Attribute, wie Name, Farbe, Form, Größe zu verändern. Das heißt, er nimmt diese Benutzereingaben entgegen.

**TransBlockPropertiesHandler** Da der Benutzer bei einem Transformationsbaustein nur die Funktion verändern kann nimmt dieser Handler ebenfalls nur eine Benutzereingabe für eine Funktion entgegen und setzt diese in den Transformationsbaustein-Eigenschaften.

**ReprBlockPropertiesHandler** Bei Repräsentationsbaustein-Eigenschaften ist der Benutzer in der Lage Einheit und Grenzwerte zu setzen. Daher nimmt der Handler diese Benutzer eingaben entgegen.

## 2.4.5 Button

Knöpfe bieten dem Benutzer eine Anzahl von Funktion zur Bedienung der Anwendung an. Das Paket ButtonLayer enthält die unterschiedlichen Knöpfe, das Feld, in welchem die Knöpfe dargestellt werden und eine Annahmestelle für die Benutzerinteraktion.

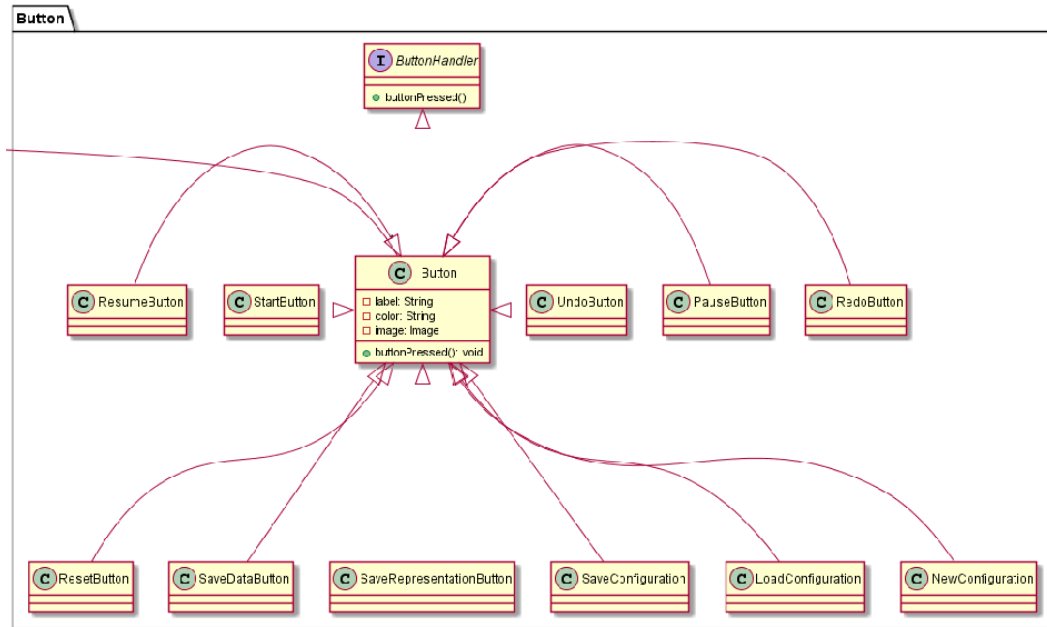


Abbildung 9: Aufbau des Button-Paket

**Button** Die Klasse Button ist die Überklasse zu den konkreten Knöpfen. Jeder Knopf enthält einen eindeutigen Namen und zur Unterscheidung der Knöpfe und zur einfachen Benutzung eine Farbe und ein aussagekräftiges Bild, welches die Funktionalität des Knopfes darstellt.

**StartButton** Die Klasse StartButton erbt von der Überklasse Button und stellt den Knopf dar, welcher bei Betätigung den Messlauf starten soll.

**PauseButton** Die Klasse PauseButton ist eine weitere Konkretisierung von Button und stellt den Knopf dar, welcher einen Messlauf pausiert.

**ResumeButton** Die Klasse ResumeButton stellt den Knopf dar, welcher einen Messlauf fortsetzt.

**ResetButton** Die Klasse ResetButton stellt den Knopf dar, welcher bei Betätigung den Messlauf auf den Ausgangszustand zurücksetzt.

**SaveDataButton** Die Klasse SaveDataButton stellt den Knopf dar, welcher dem Benutzer ermöglicht die Messwerte aus einem Messlauf zu speichern.

**SaveRepresentationButton** Die Klasse SaveRepresentationButton stellt den Knopf dar, welcher eine Momentaufnahme der graphischen Visualisierung der Messwerte speichern lässt.

**SaveConfiguration** Die Klasse SaveConfiguration stellt den Knopf dar, welcher dem Benutzer erlaubt seine eigene Messkonfiguration zu speichern.

**LoadConfiguration** Die Klasse LoadConfiguration repräsentiert den Knopf, welcher eine gespeicherte Messkonfiguration in das Konfigurationsfeld laden lässt.

**NewConfiguration** Die Klasse NewConfiguration stellt den Knopf dar, welcher dem Benutzer die Funktion bietet eine neue Konfiguration zu erstellen.

**UndoButton** Die Klasse UndoButton stellt den Undo-Knopf dar, welcher bei Betätigung die letzte Benutzeraktion rückgängig macht.

**RedoButton** Die Klasse RedoButton stellt den Redo-Knopf dar, welcher die letzte rückgängig gemachte Aktion wiederherstellt.

**Interface ButtonHandler** Das Interface ButtonHandler registriert Benutzerinteraktionen auf der Benutzeroberfläche und löst die Methode ButtonPressed() aus, über welche die Anwendung die Benutzerinteraktion weiterverarbeitet.

## 2.4.6 OptionAndHelp

Das Paket OptionAndHelp soll dem Benutzer die Benutzung der Anwendung vereinfachen. Getrennt wurde das Paket in die Funktionsspezifische Klasse HelpWindow, welche dem Benutzer Hilfe zur Bedienung gibt und in die Klasse OptionsWindow, welche dem Benutzer Auswahlmöglichkeiten gibt, um eine möglichst Barrierefreie Benutzung zu ermöglichen.

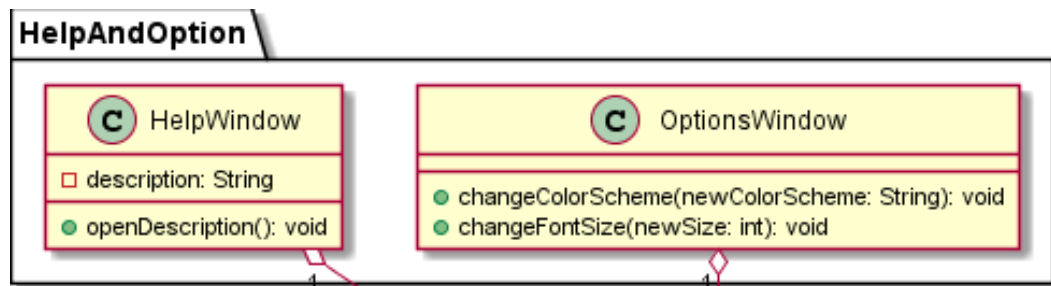


Abbildung 10: Aufbau des HelpAndOption-Paket

**HelpWindow** Die Klasse HelpWindow beschreibt das Hilfe-Fenster der Anwendung. Der Benutzer bekommt bei Öffnen des Hilfe-Fensters eine allgemeine Erklärung zur Funktionalität und zur Bedienbarkeit der gesamten Anwendung. Ebenfalls könnte in dem Hilfstext ein einfaches Anwendungsbeispiel erklärt werden, um dem Benutzer erste Schritte zu vereinfachen.

**OptionsWindow** Die Klasse OptionsWindow stellt das Einstellungen-Fenster der Anwendung dar. Der Benutzer soll hierbei das verwendete Farbschema ändern können, um die Bedienung der Anwendung trotz möglichen Farbschwächen zu ermöglichen. Ebenfalls soll die Schriftgröße der Textelemente verändert werden können, um Sehschwächen auszugleichen.



### 2.4.7 Exception

Fehlernachrichten sind ein wichtiger Teil der Anwendung, um dem Benutzer eine möglichst benutzerfreundliche Umgebung zu liefern und eine möglichst einfache und verständliche Bedienung zu ermöglichen. Damit der Benutzer aussagekräftige Fehlermeldungen erhält unterscheiden wir im Entwurf zwischen drei Typen von Fehlerarten aus verschiedenen Fehlerquellen.

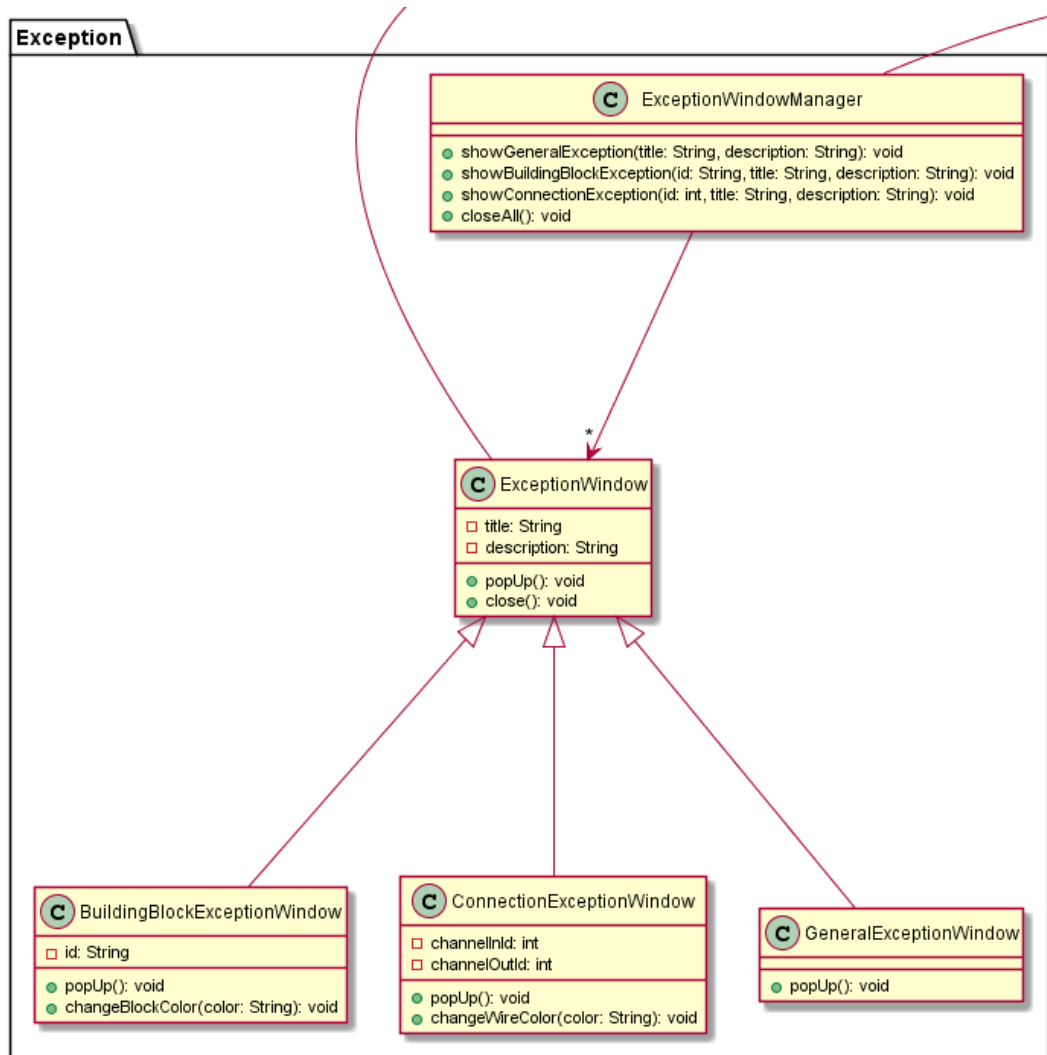


Abbildung 11: Aufbau des Exception-Paket

**ExceptionWindow** Die Klasse `ExceptionWindow` stellt die Überklasse der drei verschiedenen Unterklassen dar und enthält die gemeinsamen Attribute, welche die konkreten Fehlermeldungen enthalten. Eine Fehlermeldung besitzt immer einen Titel, der wünschenswerter Weise bereits die Fehlermeldung aussagekräftig und kurz beschreibt. Die Beschreibung der Fehlermeldung wiederum liefert eine genauere und explizite Erklärung zur Fehlerquelle, Fehlerursache und möglicherweise ebenfalls zur Fehlerbehebung. Damit der Benutzer auf die Fehlermeldung aufmerksam wird, bewirkt die Methode `popUp()`, dass die Fehlermeldung zu sehen ist. Damit der Benutzer weiterarbeiten kann oder den Fehler beheben will kann die Fehlermeldung wieder geschlossen werden.

**BuildingBlockExceptionWindow** Die Klasse `BuildingBlockExceptionWindow` ist eine Konkretisierung der Überklasse `ExceptionWindow` und stellt eine Fehlermeldung im Bezug zu Konfigurationsbausteinen dar. Neben einem Titel und einer Beschreibung wird zur Erzeugung dieser Fehlermeldung die eindeutige ID des Konfigurationsbausteins benötigt. Dadurch erfährt der Benutzer sofort, bei welchem Konfigurationsbaustein ein Fehler aufgetreten ist. Die Methode `popUp()` aus der Überklasse wird hier überschrieben. Damit soll bewirkt werden, dass die Fehlermeldung als Pop-Up Nachricht direkt neben dem Konfigurationsbaustein im Konfigurationsfeld erscheint und somit dem Benutzer sofort die Fehlerquelle signalisiert. Ebenfalls wird zur Darstellung des Fehlers die Farbe des Konfigurationsbausteins im Konfigurationsfeld geändert, um dem Benutzer nochmal auf die Fehlerquelle hinzuweisen.

**ConnectionExceptionWindow** Die Klasse `ConnectionExceptionWindow` ist eine weitere Konkretisierung der Überklasse `ExceptionWindow` und stellt eine Fehlermeldung bei Verbindungen zwischen Konfigurationsbausteinen dar. Zur Identifizierung der Fehlerquelle wird neben Titel und Beschreibung ebenfalls die IDs der Ein- und Ausgangskanäle des Konfigurationsbausteins mitgegeben. Die Methode `popUp()` soll ebenfalls die Fehlermeldung in der Nähe der Fehlerquelle im Konfigurationsfeld platzieren. Ebenfalls wird die Farbe des Drahtes sinnvoll verändert um die Fehlerquelle zu signalisieren.

**GeneralExceptionWindow** Die Klasse `GeneralExceptionWindow` stellt neben den zwei konkreten Fehlermeldungen `ConnectionExceptionWindow` und `BuildingBlockExceptionWindow` eine allgemeinere Fehlermeldung dar. Diese werden zum Beispiel bei Messfehlern oder Fehler bei der Messkonfiguration ausgelöst. Diese Fehlermeldungen sollen sichtbar in der Mitte der Anwendung geöffnet werden, um dem Benutzer auf diesen Fehler hinzuweisen.

**ExceptionWindowManager** Die Klasse `ExceptionWindowManager` nimmt Fehlermeldungen entgegen und stößt die Visualisierung der jeweilig nach Fehlermeldung unter-

schiedlichen Fenster an. Der Klasse werden die für eine Fehlermeldung notwendigen Parameter Titel und Beschreibung übergeben. Je nach Art der Fehlermeldung wird auch die Baustein- oder Verbindung-ID der Fehlerquelle übergeben.

#### 2.4.8 FacadeModelView

Das Paket FacadeModelView enthält das Interface, welches das Model anbietet und vom View verwendet wird. Da durch das Directory eine Art Zwischenschicht zwischen Model und View darstellt ersetzt die Fassade zum Directory eine unübersichtliche Fassade zu dem Model.

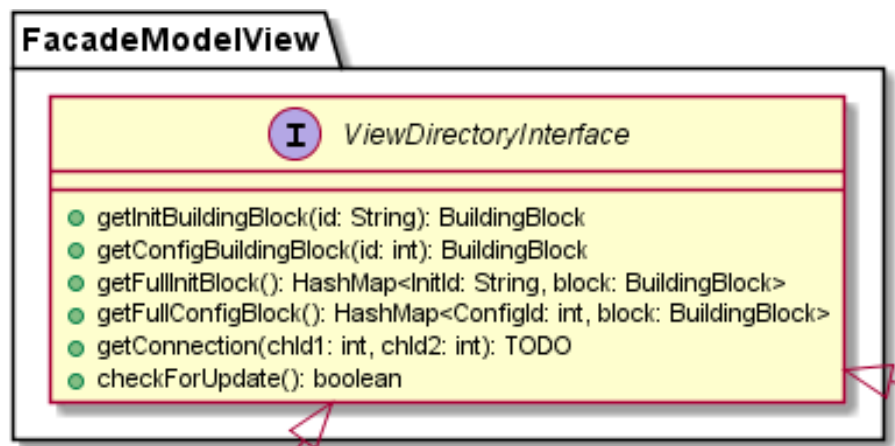


Abbildung 12: Aufbau des FacadeModelView-Paket

**ViewDirectoryInterface** Das Interface ViewDirectoryInterface bietet wichtige Funktionen an, um Änderungen am Model in die GUI zu übertragen. Bei dem Starten der Anwendung werden alle über das Backend übertragenen Bausteine durch das Model in das Directory geladen. Um alle Bausteine in die GUI zu laden gibt die Methode getFullInitBlock() die gesamte Hash-Map, welche die Konfigurationsbausteine enthält zurück um daraus die Prototypenmenüs zu erstellen. Um einzelne Bausteine mit bestimmter aus dem Directory zu laden gibt es die Methoden getInitBuildingBlock() und getConfigBuildingBlock(). Um eine gespeicherte Verbindung zu bekommen gibt es die Methode getConnection(chId1: int, chId2: int), welche zwei ChannelDecoratorer-IDs mit übergibt und die Verbindung zurückgibt. Damit das View bei Benachrichtigung über ein Update des Models überprüfen kann, ob das Directory Änderungen enthält gibt es die Methode checkForUpdates(), welche einen Wahrheitswert zurückgibt, welcher eine Aussage über die Änderungen am Directory enthält.

### 3 Sequenzdiagramme

## **4 Änderungen am Pflichtenheft**

## **5 Formale Spezifikationen von Kernkomponenten**

## 6 Weitere UML Diagramme

## **7 Anhang**



## 7.1 Vollständiges Klassendiagramm

## 8 Glossar

**Model-View-Controller** Architekturmuster, dass die Software in die drei Komponenten: Model, View und Controller unterteilt. Dadurch sollen die einzelnen Komponenten unabhängig von einander verändert werden können..