

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>4</b>
<b>2</b>	<b>Klassenbeschreibung</b>	<b>5</b>
2.1	Backend . . . . .	6
2.2	Model . . . . .	7
2.2.1	Core . . . . .	7
2.2.2	BuildingBlock . . . . .	9
2.2.3	MRunReaction . . . . .	10
2.2.4	MRunInfo . . . . .	11
2.2.5	FacadeViewModel . . . . .	11
2.2.6	SensorLogic . . . . .	12
2.2.7	TransformationLogic . . . . .	12
2.2.8	RepresentationLogic . . . . .	13
2.2.9	ChannelLogic . . . . .	14
2.2.10	BuildingBlockBuilder . . . . .	15
2.3	Controller . . . . .	19
2.3.1	CommandManager . . . . .	20
2.3.2	Command bzw. konkrete Befehle . . . . .	20
2.3.3	Verbindung zum View . . . . .	23
2.3.4	Verbindung zum Model . . . . .	24
2.4	View . . . . .	25
2.4.1	MainWindow . . . . .	25
2.4.2	Menues . . . . .	26
2.4.3	Configuration . . . . .	27
2.4.4	BuildingBlockProperties . . . . .	31
2.4.5	Button . . . . .	35
2.4.6	OptionAndHelp . . . . .	37
2.4.7	Exception . . . . .	39
2.4.8	FacadeModelView . . . . .	41
2.4.9	FacadeControllerView . . . . .	41
<b>3</b>	<b>Sequenzdiagramme</b>	<b>44</b>
<b>4</b>	<b>Änderungen am Pflichtenheft</b>	<b>45</b>
<b>5</b>	<b>Formale Spezifikationen von Kernkomponenten</b>	<b>46</b>
<b>6</b>	<b>Weitere UML Diagramme</b>	<b>47</b>
<b>7</b>	<b>Anhang</b>	<b>48</b>
7.1	Vollständiges Klassendiagramm . . . . .	49



# 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.

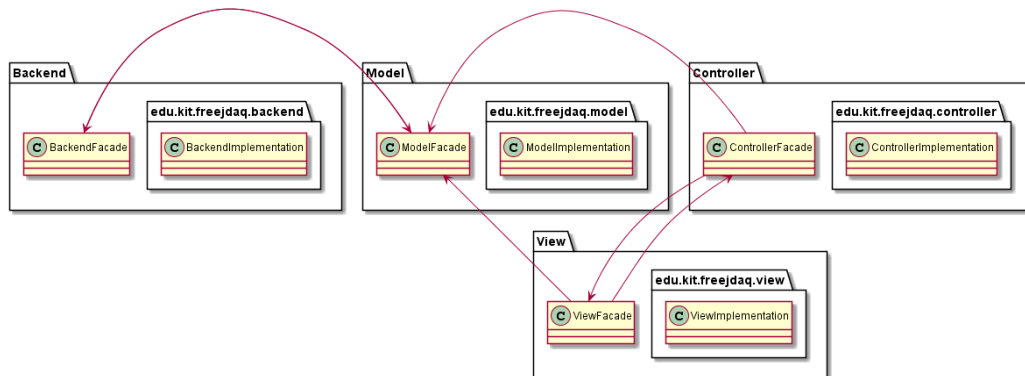


Abbildung 1: Die grobe Struktur des Entwurfs

## **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 Core

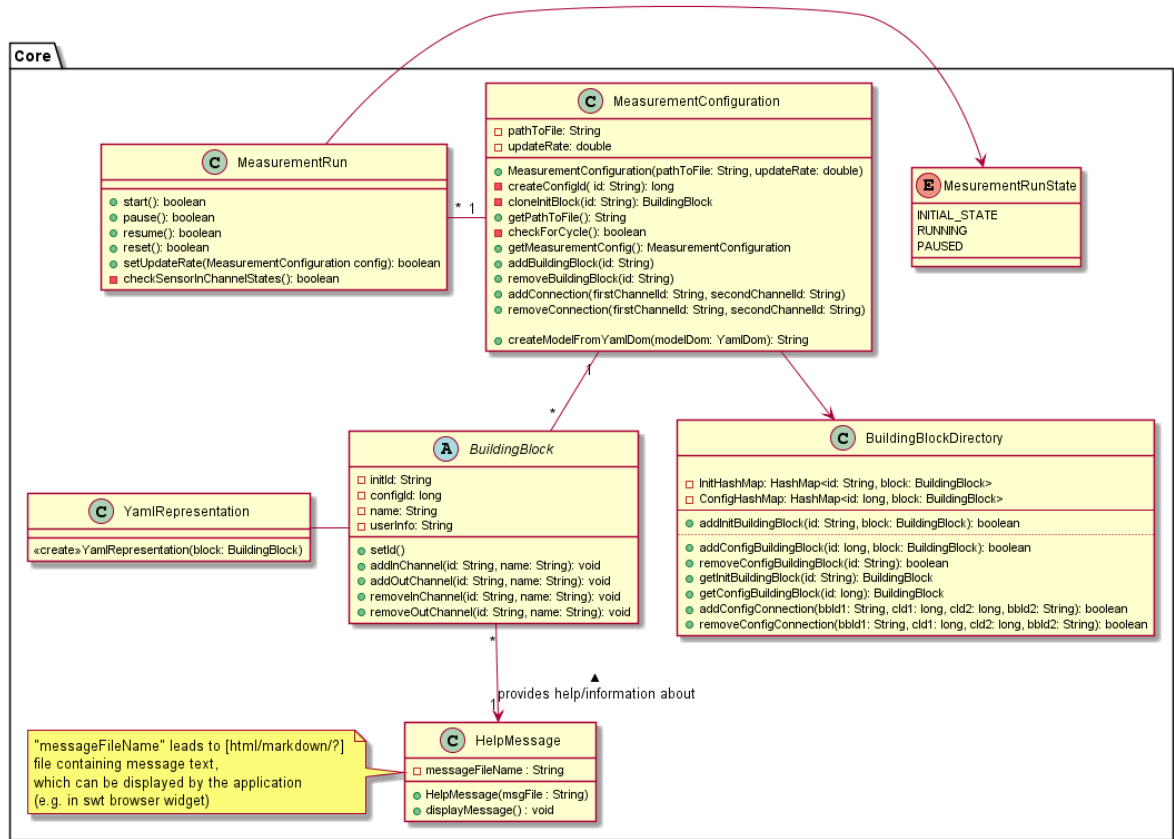


Abbildung 2: Aufbau des Pakets Core

Die Struktur des Pakets Core ist in Abbildung 2 zu sehen. TODO

**BuildingBlockDirectory** Die Klasse BuildingBlockDirectory ist in Abbildung 3 zu sehen. TODO

**MeasurementConfiguration** Die Klasse MeasurementConfiguration ist in Abbildung 4 zu sehen. TODO

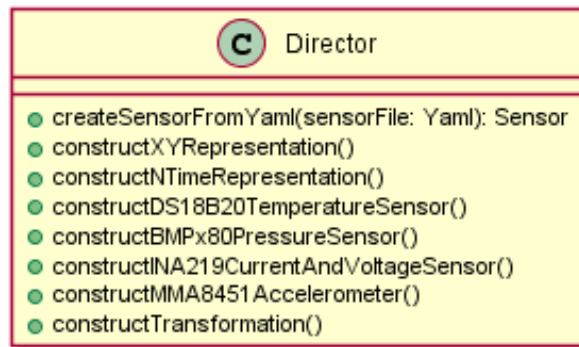


Abbildung 3: Darstellung der Klasse BuildingBlockDirectory

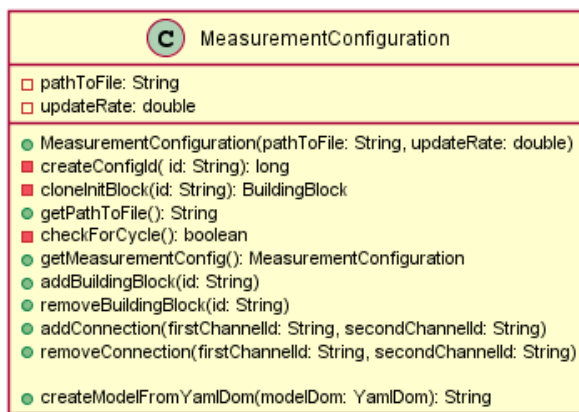


Abbildung 4: Darstellung der Klasse MeasurementConfiguration

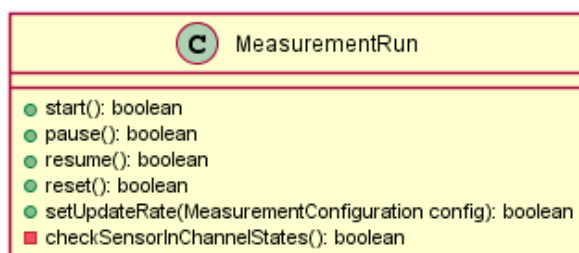


Abbildung 5: Darstellung der Klasse MeasurementRun



**MeasurementRun** Die Klasse MeasurementRun ist in Abbildung 5 zu sehen. TODO

### 2.2.2 BuildingBlock

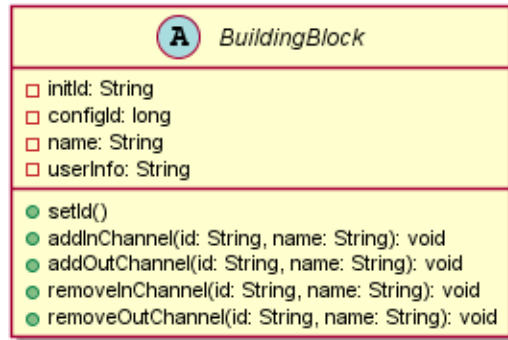


Abbildung 6: Darstellung der Klasse BuildingBlock

Die abstrakte Klasse BuildingBlock ist in Abbildung 6 zu sehen. Sie stellt im Model einen BuildingBlock dar und speichert die nötigen Daten als folgende Attribute:

- Das Attribut `initId` speichert als einen String eine eindeutige ID zur Unterscheidung von BuildingBlockPrototypen.
- Das Attribut `configId` dient als eindeutige ID zur Unterscheidung von BuildingBlockInstanzen innerhalb einer Messkonfiguration, da auch mehrere BuildingBlocks eines Typs innerhalb einer Messkonfiguration auftreten können und unterschieden werden müssen.
- Das Attribut `name` speichert den Namen einen BuildingBlocks als String dar.
- Das Attribut `userInfo` ?? TODO

Ein BuildingBlock hat eine beliebige Menge an In- und OutChannels, deren Anzahl durch die folgenden vier Methoden verwaltet kann:

- Die Methode `addInChannel` fügt dem BuildingBlock einen InChannel hinzu.
- Die Methode `removeInChannel` entfernt einen bestimmen InChannel von dem BuildingBlock.
- Die Methode `addOutChannel` fügt dem BuildingBlock einen OutChannel hinzu.

- Die Methode *removeOutChannel* entfernt einen bestimmten OutChannel von dem BuildingBlock.
- textitsetId TODO

**YamlRepresentation** TODO

**HelpMessage** Die Klasse HelpMessage enthält einen Konstruktor HelpMessage, der ein entsprechendes HelpMessage-Objekt erstellt. Ein HelpMessage-Objekt speichert zu einem zugehörigen BuildingBlock einen Tooltip mit Informationen über den entsprechenden BuildingBlock. Die Verbindung zu der Datei, die die Informationen enthält, ist als Attribut *messageFileName* gespeichert. Die Methode *displayMessage* ermöglicht es, die Informationen anzuzeigen.

### 2.2.3 MRunReaction

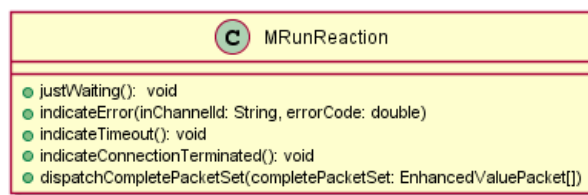


Abbildung 7: Darstellung der Klasse MRunReaction

Die Klasse MRunReaction ist in Abbildung ?? zu sehen. Sie implementiert das Interface MRunForward, welches im Cache-Modul zu finden ist. MRunReaction dient als Verbindung zwischen Cache und Modul. Der Datenfluss vom Cache zu den Sensorbausteinen im Modul wird durch die folgenden fünf Methoden 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 Eingangskanals 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.

#### 2.2.4 MRunInfo

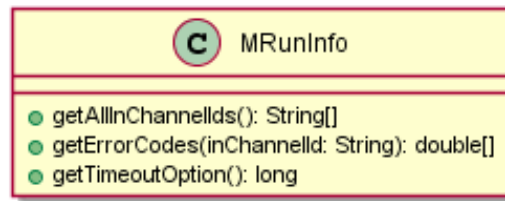


Abbildung 8: Darstellung der Klasse MRunInfo

Die Klasse MRunInfo ist in Abbildung 8 zu sehen.

#### 2.2.5 FacadeViewModel

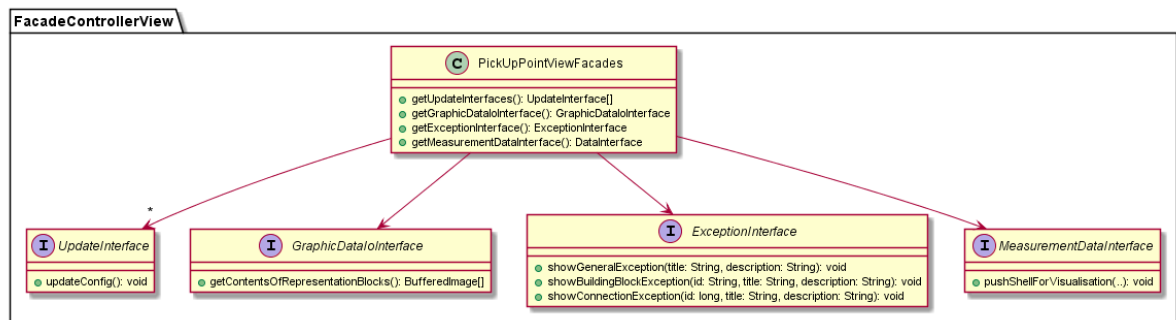


Abbildung 9: Aufbau des Pakets FacadeViewModel

Die Struktur des Pakets FacadeViewModel ist in Abbildung 9 zu sehen. TODO

#### PickUpPointViewFacades

#### MeasurementDataInterface

#### ExceptionInterface

**UpdateInterface**

**GraphicDataIoInterface**

### 2.2.6 SensorLogic

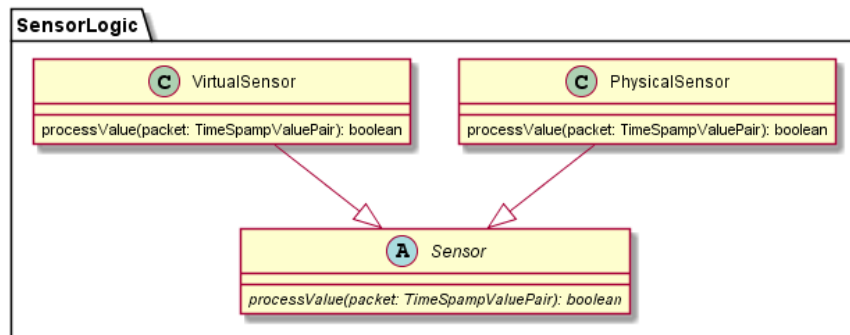


Abbildung 10: Aufbau des Pakets SensorLogic

Die Struktur des Pakets SensorLogic ist in Abbildung 10 zu sehen. TODO

**Sensor**

**VirtualSensor**

**PhysicalSensor**

### 2.2.7 TransformationLogic

**Transformation**

**Function** Die Struktur des Pakets TransformationLogic ist in Abbildung 11 zu sehen.  
TODO

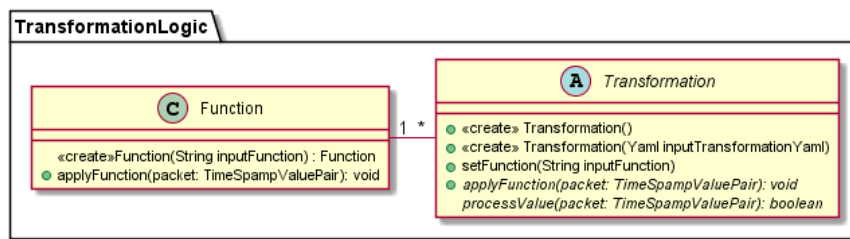


Abbildung 11: Aufbau des Pakets TransformationLogic

## Transformation

## Function

### 2.2.8 RepresentationLogic

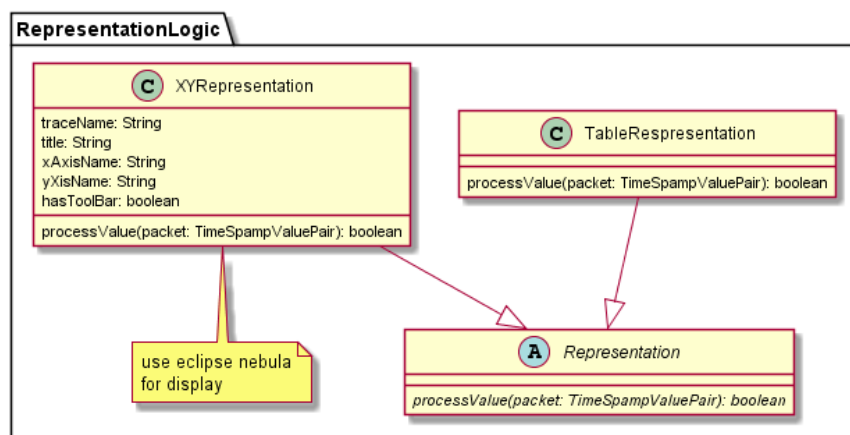


Abbildung 12: Aufbau des Pakets RepresentationLogic

Die Struktur des Pakets RepresentationLogic ist in Abbildung 12 zu sehen. TODO

## Representation

## TableRepresentation

## XYRepresentation

### 2.2.9 ChannelLogic

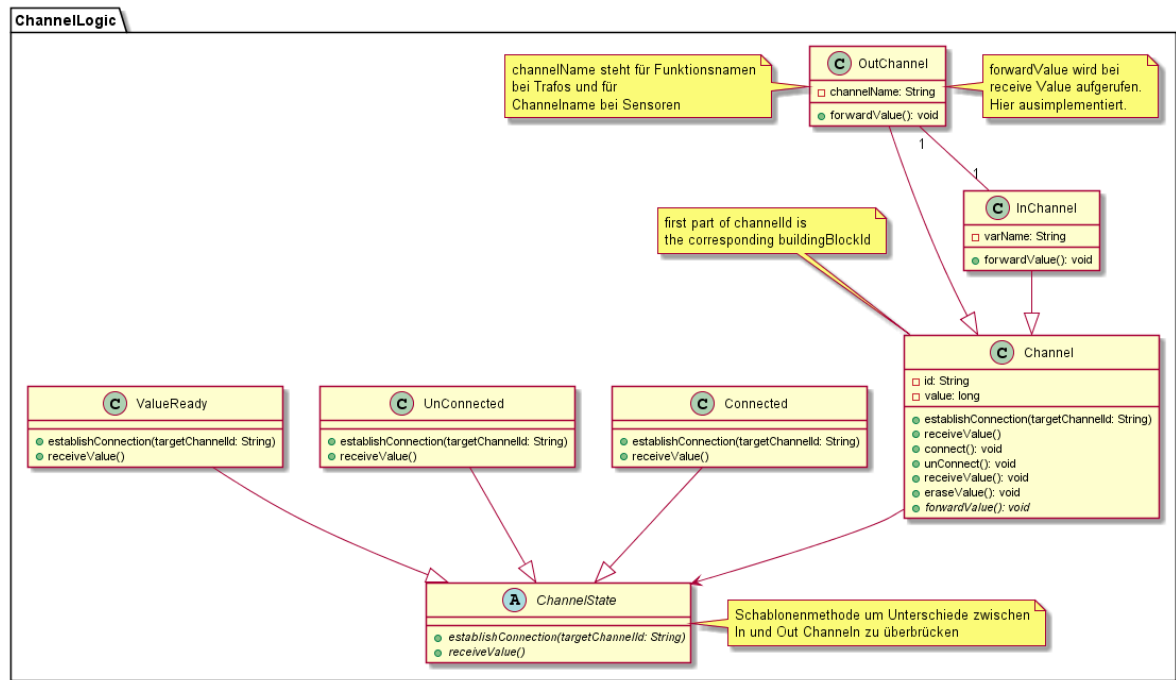


Abbildung 13: Aufbau des Pakets ChannelLogic

Die Struktur des Pakets ChannelLogic ist in Abbildung 13 zu sehen. TODO

## Channel

## InChannel

## OutChannel

ChannelState

Connected

UnConnected

ValueReady

## 2.2.10 BuildingBlockBuilder

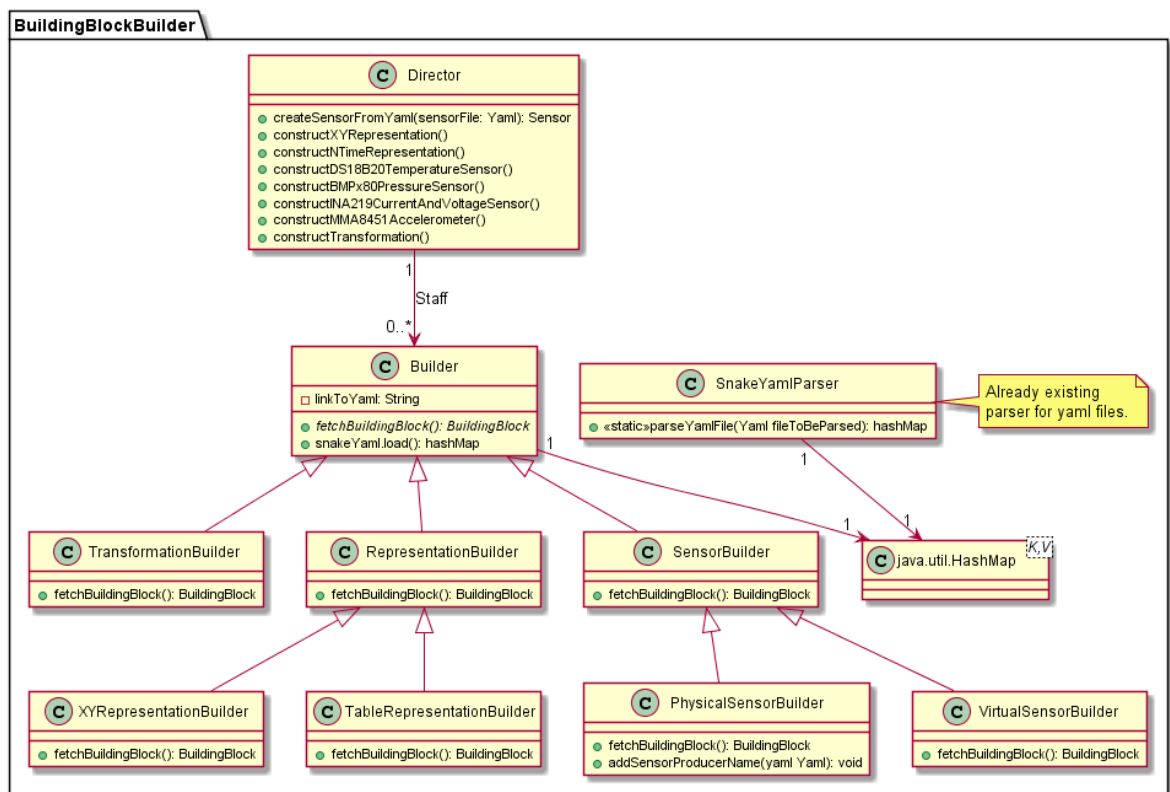


Abbildung 14: Aufbau des Pakets BuildingBlockBuilder

Der Aufbau des Pakets BuildingBlockBuilder, zu sehen in Abbildung 14, setzt das Entwurfsmuster Erbauer um. Die Rollen sind dabei folgendermaßen umgesetzt:

- Die Klasse *Director* erfüllt die Rolle des *Direktors*.
- Die Klasse *Builder* erfüllt die Rolle eines *Erbauers*.
- Die Klassen *SensorBuilder*, *TransformationBuilder*, *RepresentationBuilder*, *VirtualSensorBuilder*, *PhysicalSensorBuilder*, *XYRepresentationBuilder* und *TableRepresentationBuilder* erfüllen die Rolle der *konkreten Erbauer*.
- Die Rollen der *Produkte* werden in den Paketen *SensorLogic*, *TransformationLogic* und *RepresentationLogic* als entsprechende Bausteine erfüllt.

TODO Begründe warum Erbauer und kein anderes.

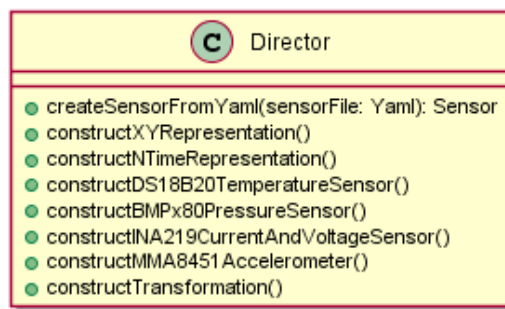


Abbildung 15: Darstellung der Klasse Director

**Director** Die Klasse Director ist in Abbildung 15 zu sehen. Sie bietet eine Reihe an *construct...* Methoden an, mit denen konkrete BuildingBlocks erstellt werden können. Der resultierende BuildingBlock wird dann in einer HashMap im BuildingBlockDirectory abgelegt. Dabei kann die Auswahl an Methoden durch neue Methoden erweitert werden, um neue Arten von Blöcken erbauen zu können. Dabei muss auch jeweils ein neuer konkreter Erbauer implementiert werden. In erster Linie hat der Director die Aufgabe, beim Start der Anwendung die angebotenen Prototypen zu erstellen und im BuildingBlockDirectory zu speichern.

**Builder** Die Klasse Builder ist die Oberklasse aller konkreten Builder. Sie hat als Attribut eine Verbindung zu einer Yaml-Datei. Mit Hilfe dieser Yaml-Datei kann ein entsprechender BuildingBlock erstellt werden. Die Yaml-Datei wird durch die Methode *snakeYaml.load* geladen, welche durch das externe Paket SnakeYamlParser angeboten wird. Die abstrakte Methode *fetchBuildingBlock* gibt die Signatur für die konkreten Erbauer vor. In den konkreten Erbauern kann der Director durch diese Methode einen ent-



sprechenden BuildingBlock anfordern und als Rückgabewert erhalten, sobald er erbaut wurde.

**SensorBuilder** Die Klasse erstellt als konkreter Erbauer durch Aufrufen der Methode *fetchBuildingBlock* aus einer Yaml-Datei einen TransformationsBlock und gibt ihn als Rückgabewert zurück. Die Verbindung zu der entsprechenden Yaml-Datei wird als Attribut von der Oberklasse Builder geerbt.

**VirtualSensorBuilder** Die Klasse erstellt als konkreter Erbauer durch Aufrufen der Methode *fetchBuildingBlock* aus einer Yaml-Datei einen VirtualsensorBlock und gibt ihn als Rückgabewert zurück. Die Verbindung zu der entsprechenden Yaml-Datei ist als geerbtes Attribut vorhanden.

**PhysicalSensorBuilder** Die Klasse erstellt als konkreter Erbauer durch Aufrufen der Methode *fetchBuildingBlock* aus einer Yaml-Datei einen PhysicalSensorBlock und gibt ihn als Rückgabewert zurück. Die Verbindung zu der entsprechenden Yaml-Datei ist als geerbtes Attribut vorhanden. Der PhysicalSensorBuilder fügt dem Sensorblock durch die Methode *addSensorProducerName* zusätzlich noch den Werknamen des Sensors hinzu.

**TransformationBuilder** Die Klasse erstellt als konkreter Erbauer durch Aufrufen der Methode *fetchBuildingBlock* aus einer Yaml-Datei einen TransformationsBlock und gibt ihn als Rückgabewert zurück. Die Verbindung zu der entsprechenden Yaml-Datei wird als Attribut von der Oberklasse Builder geerbt.

**RepresentstationBuilder** Die Klasse erstellt als konkreter Erbauer durch Aufrufen der Methode *fetchBuildingBlock* aus einer Yaml-Datei einen RepresentationBlock und gibt ihn als Rückgabewert zurück. Die Verbindung zu der entsprechenden Yaml-Datei wird als Attribut von der Oberklasse Builder geerbt.

**TableRepresentstationBuilder** Die Klasse erstellt als konkreter Erbauer durch Aufrufen der Methode *fetchBuildingBlock* aus einer Yaml-Datei einen TableRepresentationBlock und gibt ihn als Rückgabewert zurück. Die Verbindung zu der entsprechenden Yaml-Datei wird als Attribut von der Oberklasse Builder geerbt.

**XYRepresentstationBuilder** Die Klasse erstellt als konkreter Erbauer durch Aufrufen der Methode *fetchBuildingBlock* aus einer Yaml-Datei einen XYRepresentationBlock und

gibt ihn als Rückgabewert zurück. Die Verbindung zu der entsprechenden Yaml-Datei wird als Attribut von der Oberklasse Builder geerbt.

**java.util.HashMap**<sub>*K,V*</sub> TODO

**SnakeYamlParser** Diese Klasse stellt ein externes Paket dar, welches den SnakeYaml-Parser implementiert.

## 2.3 Controller

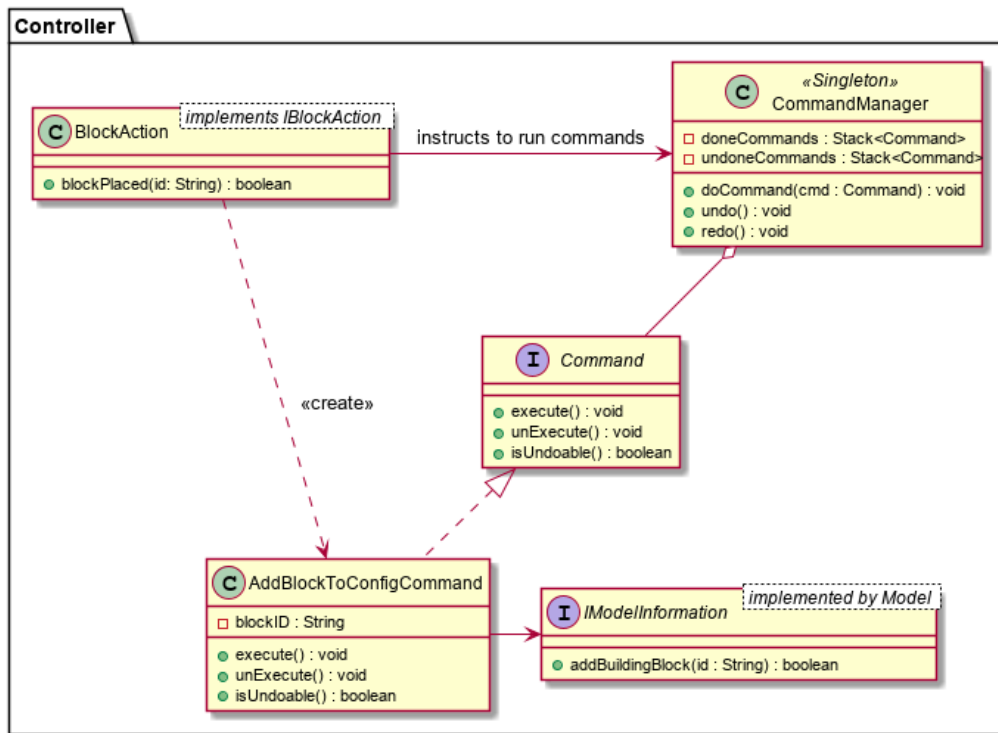


Abbildung 16: Die Struktur des Controllers (Ausschnitt)

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

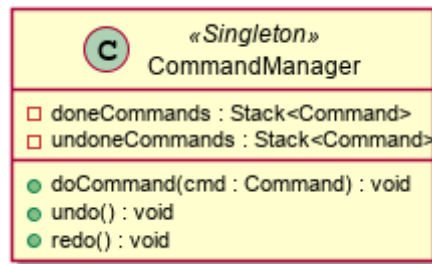


Abbildung 17: Die Klasse CommandManager

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 und kann rückgängig gemacht werden.

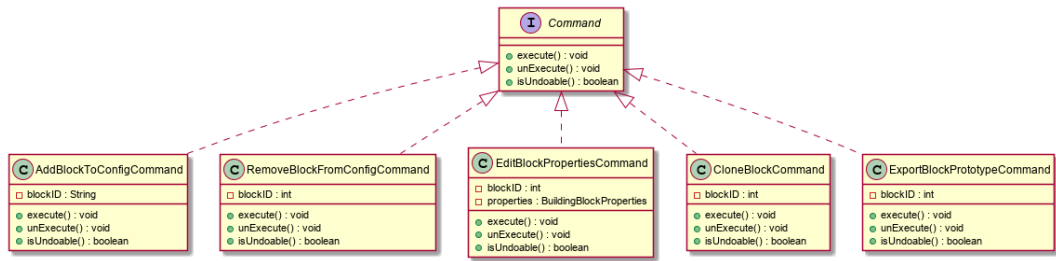


Abbildung 18: Baustein-Befehle

**RemoveBlockFromConfigCommand** Dieser Befehl entfernt einen Baustein aus dem Konfigurationsfeld und kann rückgängig gemacht werden.

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

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

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

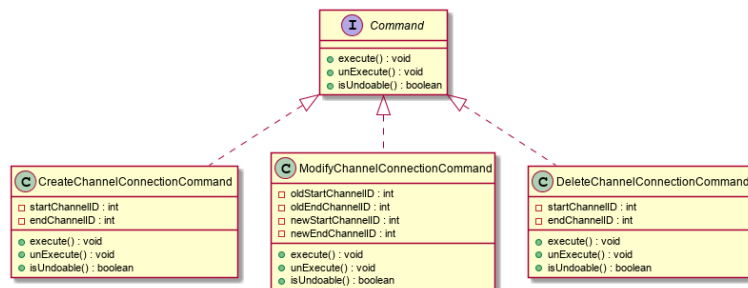


Abbildung 19: Verbindungs-Befehle

**CreateChannelConnectionCommand** Dieser Befehl erstellt eine Verbindung zwischen zwei gegebenen Kanälen (welche wiederum zu Bausteinen gehören) und kann rückgängig gemacht werden.

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

**DeleteChannelConnectionCommand** Dieser Befehl löscht eine Verbindung zwischen zwei Kanälen und kann rückgängig gemacht werden.

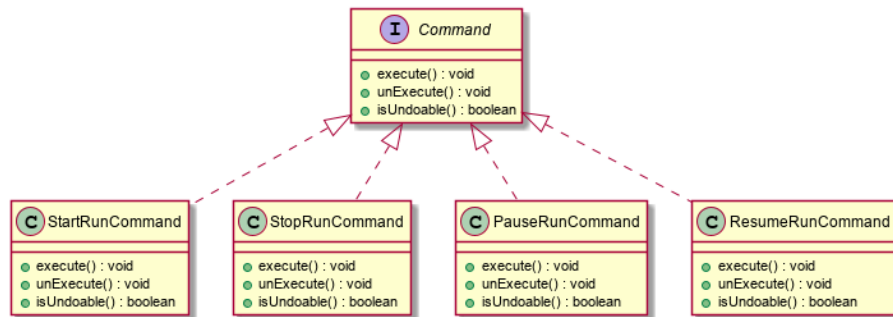


Abbildung 20: Messlauf-Befehle

**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.)

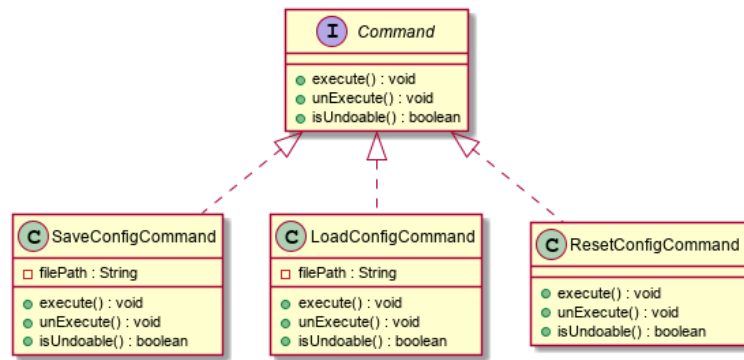


Abbildung 21: Messkonfigurations-Befehle

### 2.3.3 Verbindung zum View

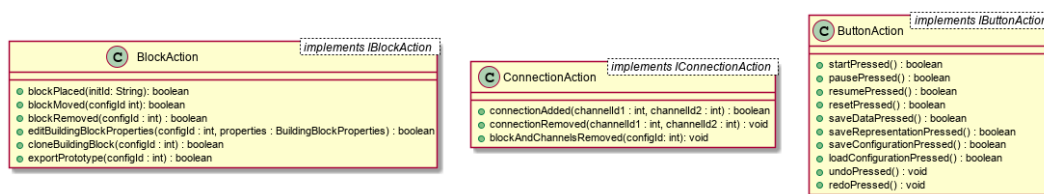


Abbildung 22: Schnittstellenimplementierung View zu Controller

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.

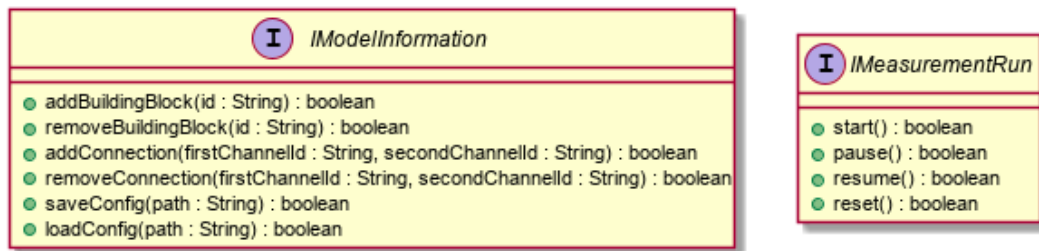


Abbildung 23: Schnittstelle Controller zu Model

### 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, beendet, 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.

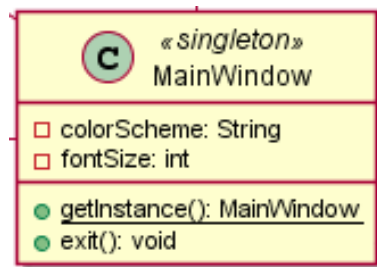


Abbildung 24: Die Klasse MainWindow

## 2.4.2 Menues

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

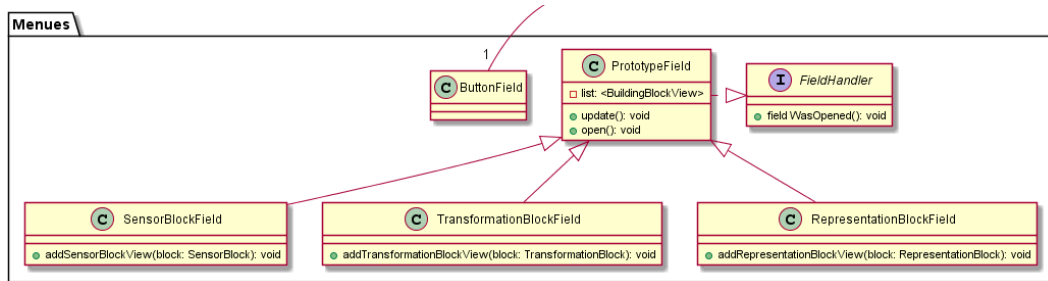


Abbildung 25: Aufbau des Menü-Paket

**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

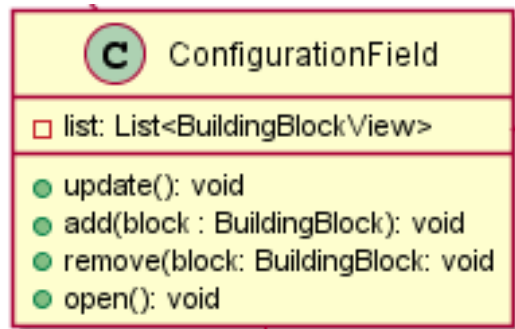


Abbildung 26: Aufbau des Konfigurationsfeld

**ConfigurationField** Die Klasse KonfigurationField stellt das Konfigurationfeld dar, in welchem der Benutzer eine Messkonfiguration aufbauen kann. Konfigurationsbausteine, welche der Benutzer in das Konfigurationsfeld platziert werden in einer Liste gespeichert. Konfigurationsbausteine, welche der Benutzer aus dem Konfigurationsfeld entfernt, werden aus der Liste gelöscht. Beim Platzieren der Konfigurationsbausteine in das Konfigurationsfeld 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 Baustein besitzt diese Initialisierungs-ID (InitId). Wenn ein Baustein mehrfach durch den Benutzer in das Konfigurationsfeld gezogen wird, könnte dies dazu führen, dass diese Initialisierungs-ID nicht mehr eindeutig für diesen Baustein wäre. Deswegen besitzt jeder im Konfigurationsfeld 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

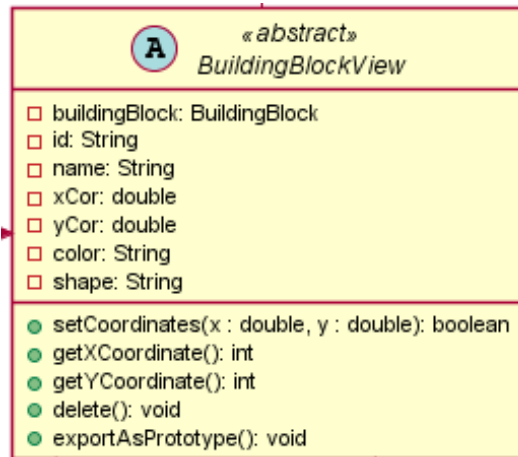


Abbildung 27: Aufbau des BuildingBlockView

Namen und falls sie im Konfigurationsfeld 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 InitId bleiben bei Erzeugung des Bausteins gleich, jedoch wird der Baustein, um die visuellen Komponenten Koordinaten, Farbe, Form erweitert.

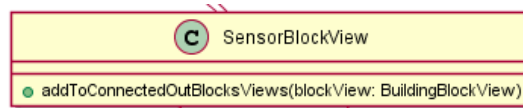


Abbildung 28: Aufbau des SensorBlockView

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

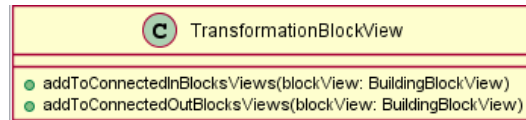


Abbildung 29: Aufbau des TransformationBlockView

bausteine empfangen können. Ausgänge der Transformationsbausteine können nur an weitere Transformationsbausteine oder Darstellungsbausteine angebunden werden, um einen sinnvollen Datenfluss zu ermöglichen.

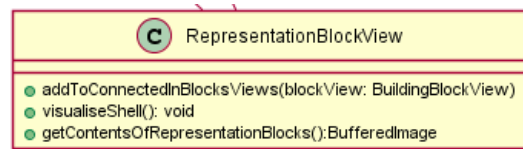


Abbildung 30: Aufbau des RepresentationBlockView

**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.

**IDragAndDropHandler** Über das Interface IDragAndDropHandler wird der Anwendung mitgeteilt, dass ein Konfigurationsbaustein durch Drag-and-Drop in der GUI bewegt wurde. Es wird übergeben, um welchen eindeutigen Konfigurationsbaustein es sich handelt, indem bei Konfigurationsbausteinen, welche zuvor nicht im Konfigurationsfeld platziert die Initialisierungs-ID übergeben wird. Bei Konfigurationsbausteinen, welche bereits im Konfigurationsfeld liegen und nicht mehr durch die Initialisierungs-ID eindeutig unterscheidbar sind, wird die Konfigurations-ID übergeben. Ebenfalls wird die Position auf welche der Konfigurationsbaustein gesetzt wurde durch die x- und y- Koordinaten mitübergeben.

**HelpDecorator** Jeder Konfigurationsbaustein besitzt eine kurze Beschreibung seiner Art und Funktionalität. Diese Beschreibung wird über das Fenster HelpDecorater dem Benutzer dargestellt.

**HelpDecoratorHandler** Das Interface HelpDecoratorHandler wird von der Klasse HelpDecorator implementiert und registriert wenn der Benutzer den HelpDecorator öffnen will, in dem er darauf drückt.

**ChannelDecorator**

**InChannelDecorator**

**OutChannelDecorator**

**Wire**

**AddWireDragAndDropHandler**

**RemoveWireDragAndDropHandler**

#### 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.

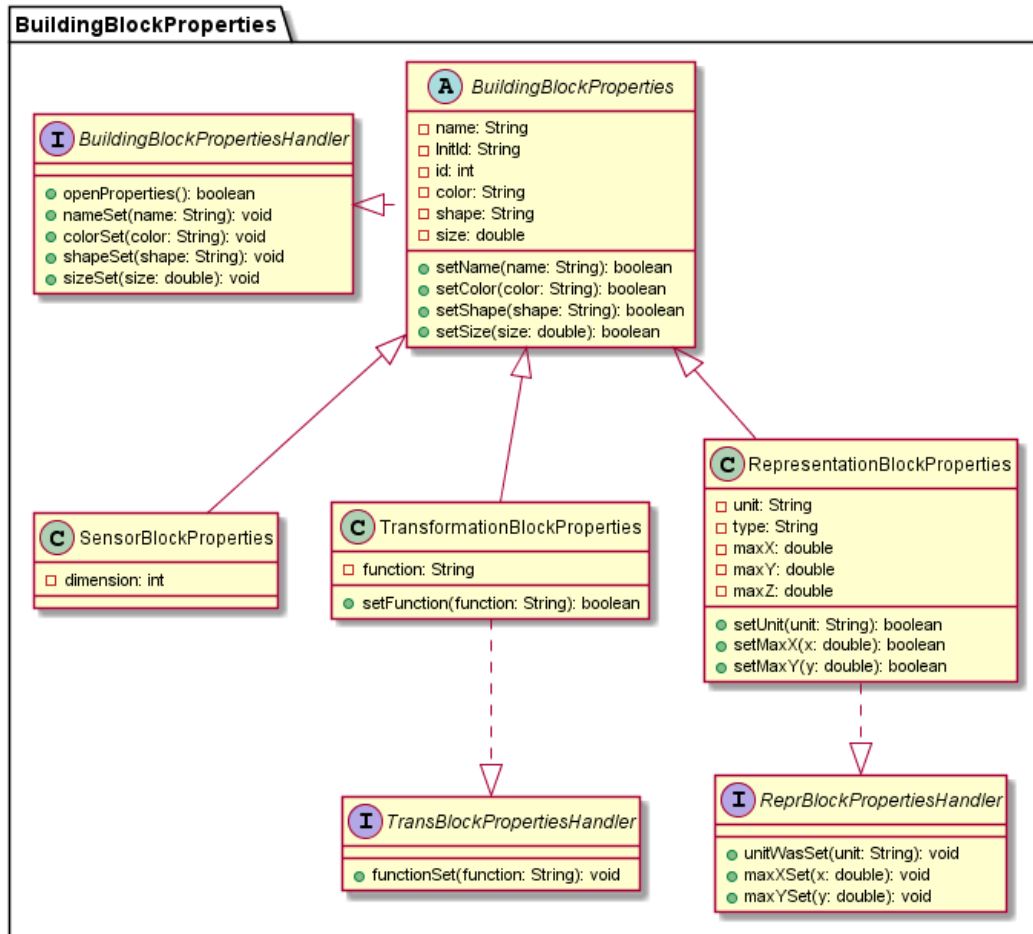


Abbildung 31: Aufbau des BuildingBlockProperties-Paket

**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



## *BuildingBlockProperties*

- ❑ name: String
- ❑ InitId: String
- ❑ id: int
- ❑ color: String
- ❑ shape: String
- ❑ size: double

- setName(name: String): boolean
- setColor(color: String): boolean
- setShape(shape: String): boolean
- setSize(size: double): boolean

Abbildung 32: Aufbau der BuildingBlockProperties



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

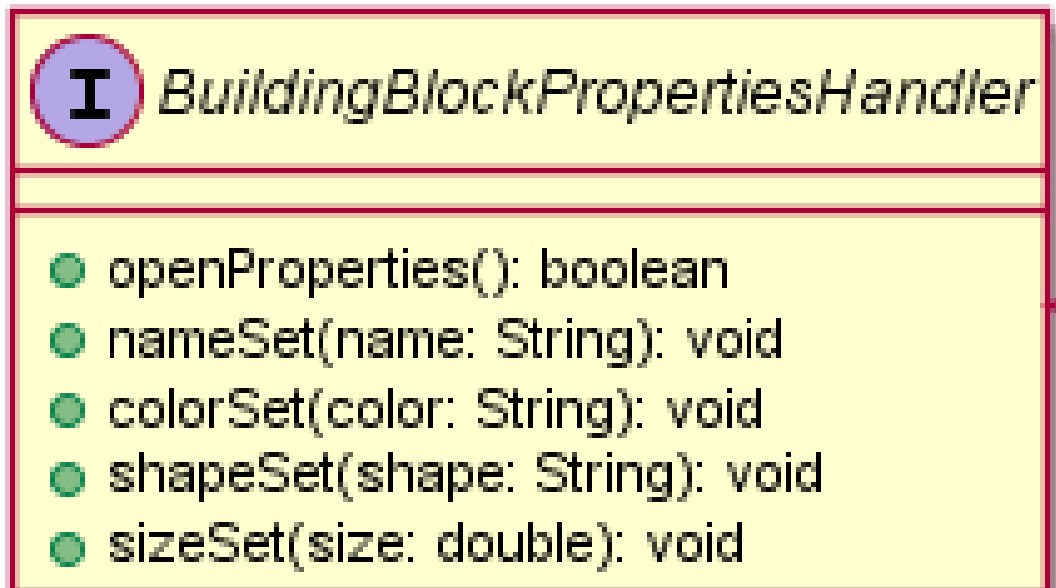


Abbildung 33: Aufbau des `BuildingBlockPropertiesHandler`

**BuildingBlockPropertiesHandler** Das Interface `BuildingBlockPropertiesHandler` wird von der Klasse `BuildingBlockProperties` implementiert und lässt den Benutzer durch mehrere Methoden das Eigenschaften-Fenster öffnen und Attribute der Baustein-Eigenschaften verändern. Bei allen Darstellungen von Konfigurationsbausteinen lässt sich der 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.

**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.

**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.

**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.

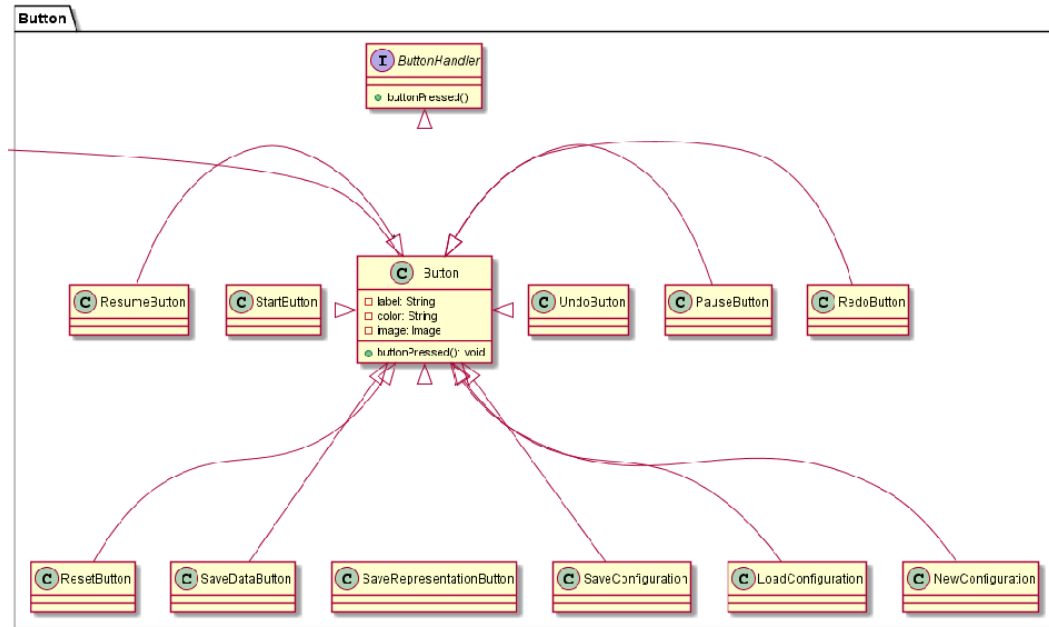


Abbildung 34: Aufbau des Button-Paket

**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.

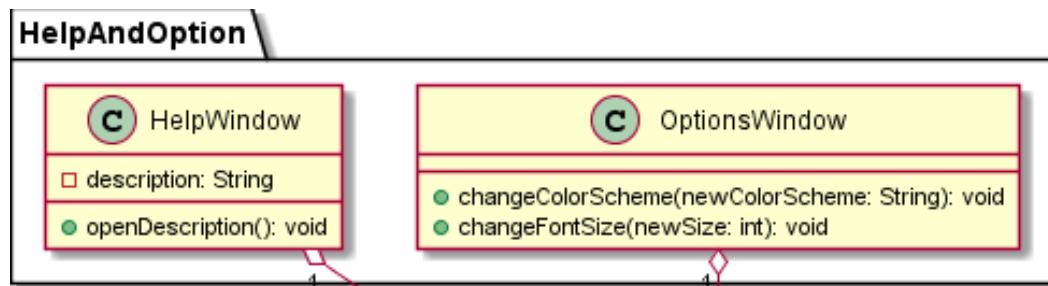


Abbildung 35: Aufbau des HelpAndOption-Paket

**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.

**HelpWindowHandler** Das Interface HelpWindowHandler, welches von der Klasse HelpWindow implementiert wird, liefert die Benutzereingabe im Falle des Drückens des Hilfe-Knopfes an die Anwendung weiter. Beim Drücken des Hilfe-Knopfes soll die Hilfe sofort erscheinen.

**OptionsWindowHandler** Das Interface OptionsWindowHandler, welches von der Klasse OptionsWindow implementiert wird, erkennt die Benutzereingabe im Falle des Drückens

des Einstellungen- oder Optionen Knopf. In dem geöffneten Fenster kann der Benutzer dann das Farbschema und die Schriftgröße anpassen.

## 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.

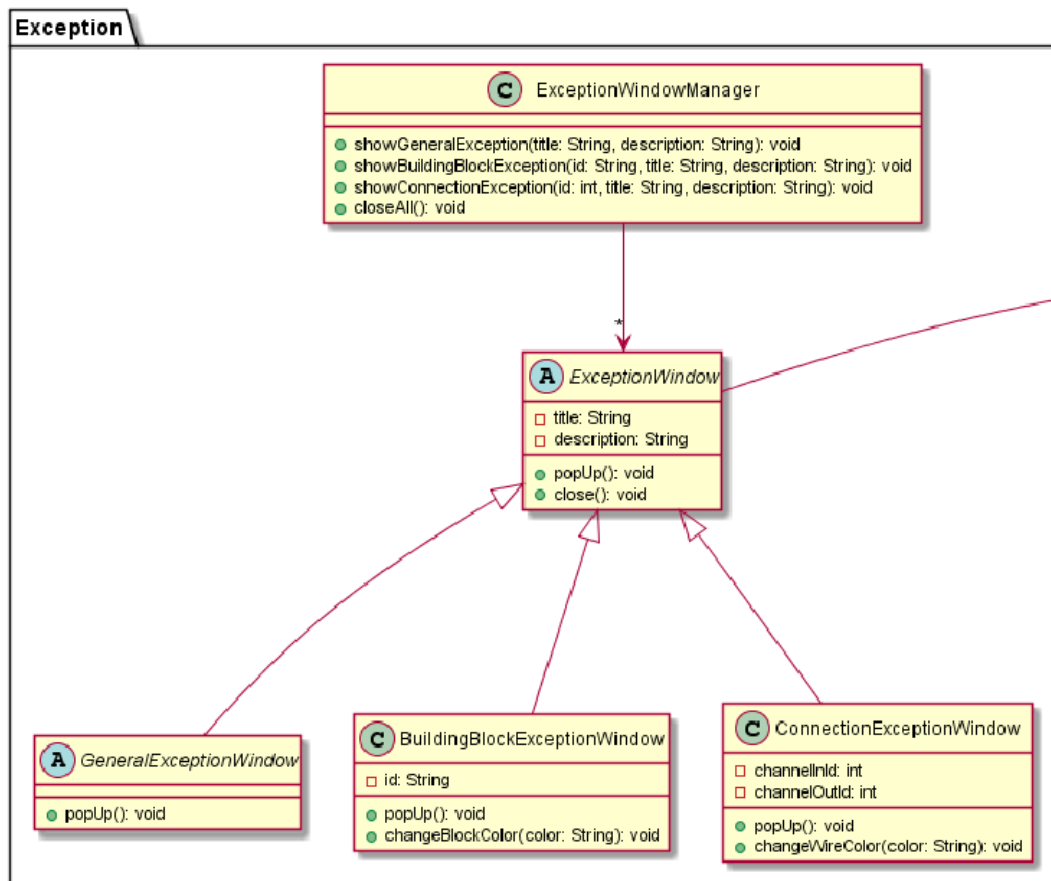


Abbildung 36: Aufbau des Exception-Paket

**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 eine 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 Fehlerbehe-

bung. Damit der Benutzer auf die Fehlnachricht aufmerksam wird, bewirkt die Methode `popUp()`, dass die Fehlnachricht zu sehen ist. Damit der Benutzer weiterarbeiten kann oder den Fehler beheben will kann die Fehlnachricht wieder geschlossen werden.

**BuildingBlockExceptionWindow** Die Klasse `BuildingBlockExceptionWindow` ist eine Konkretisierung der Überklasse `ExceptionWindow` und stellt eine Fehlnachricht im Bezug zu Konfigurationsbausteinen dar. Neben einem Titel und einer Beschreibung wird zur Erzeugung dieser Fehlnachricht 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 Fehlnachricht bei Verbindungen zwischen Konfigurationsbausteinen dar. Zur Identifizierung der Fehlerquelle wird neben Titel und Beschreibung ebenfalls die IDs der Ein- und Ausgangskanäle der Konfigurationsbausteine mitgegeben. Die Methode `popUp()` soll ebenfalls die Fehlnachricht 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 Fehlnachricht dar. Diese werden zum Beispiel bei Messfehlern oder Fehler bei der Messkonfiguration ausgelöst. Diese Fehlnachrichten 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 unterschiedlichen 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.

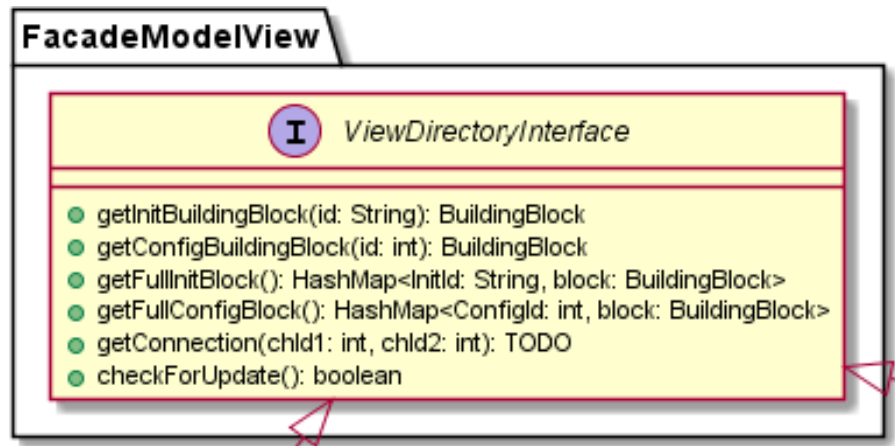


Abbildung 37: Aufbau des FacadeModelView-Paket

**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.

## 2.4.9 FacadeControllerView

Das Paket FacadeControllerView stellt die Schnittstelle dar, welche der Controller anbietet und vom View benutzt wird. Zur Übersicht ist die Fassade intern in 3 Interfaces auf-

geteilt, welche jeweils Funktionen eines Objektes darstellen (Button, Block, Connection). Eine Pickup-Klasse stellt die Anbindung zum View dar und kapselt die Interfaces.

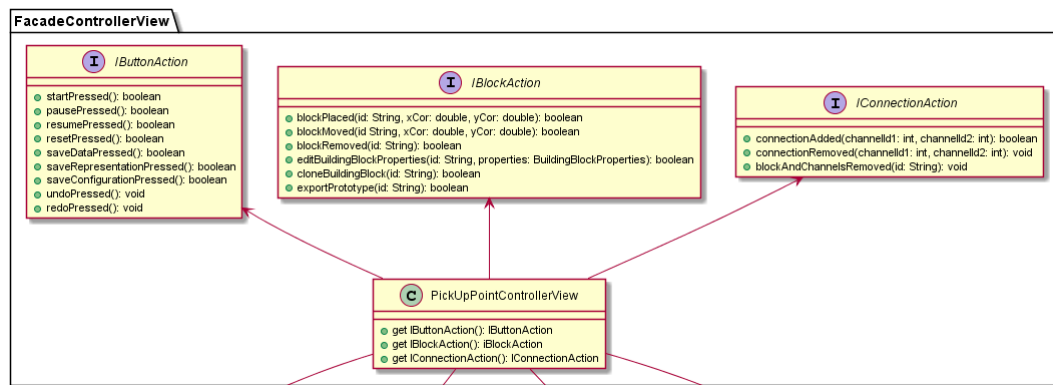


Abbildung 38: Aufbau des FacadeControllerView-Paket

**PickUpPointControllerView** Die Klasse PickUpPointControllerView stellt die Schnittstelle zwischen den Interfaces und den Klassen, welche auf diese zugreifen dar. Ihre Methoden liefern jeweils das gewollte Interface zurück, über welches dann Aktionen an den Controller übergeben werden können.

**IButtonAction** Das Interface IButtonAction liefert eine Schnittstelle für alle Knopf-Aktionen. Dass heißt, wenn durch den ButtonHandler eine Benutzereingabe in Form des Drücken eines konkreten Knopfes registriert wird, wird in diesem Interface die für den Knopf spezifische Methode aufgerufen, um den Controller zu benachrichtigen. Dabei gibt es für jeden konkreten Knopf eine Interface-methode.

**IBlockAction** Das Interface IBlockAction liefert eine Schnittstelle für alle Benutzerinteraktionen mit einem Konfigurationsbaustein. Diese gibt das Interface weiter an den Controller, in dem das Interface implementiert ist. Die Methode blockPlaced gibt hierbei weiter, wenn ein Konfigurationsbaustein aus einem dem Prototypen Menüs auf das Konfigurationsfeld per Drag-and-Drop platziert wurde. Hierbei wird die Prototyp-spezifische ID mitgegeben und die Koordinaten, welche die Position des Bausteins eindeutig bestimmen. Die Methode blockMoved wird dann benutzt, wenn ein Konfigurationsbaustein innerhalb des Konfigurationsfeldes die Position ändert. Wenn der Benutzer einen Konfigurationsbaustein aus dem Konfigurationsfeld entfernt wird dies über die Methode blockRemoved mit Übergabe der eindeutigen ID an den Controller überliefert. Wenn der Benutzer die Eigenschaften eines Bausteinprototyps ändert wird dies über die Methode editBuildingBlockProperties mit der eindeutigen ID und den neuen Eigenschaften

übergeben. Wenn ein Baustein geklont oder exportiert werden soll, wird dies mit der Übergabe der eindeutigen ID an den Controller übergeben.

**ICconnectionAction** Das Interface `ICconnectionAction` liefert eine Schnittstelle für Aktionen, welche der Benutzer mit Verbindungen macht. Die Methode `connectionAdded` übergibt dem Controller zwei eindeutige Channel-IDs, welche der Benutzer miteinander verbunden hat, um eine Messkonfiguration aufzubauen. Wenn der Benutzer eine Verbindung zwischen zwei Konfigurationsbausteinen entfernt übergibt die Methode `connectionRemoved` die zwei Channel-IDs an den Controller um diese Verbindung aus der Messkonfiguration zu entfernen. Falls der Benutzer einen Konfigurationsbaustein entfernt, welcher bereits mit weiteren Konfigurationsbausteinen verbunden war, werden diese Verbindungen ebenfalls gelöscht, daher wird bei der Methode `blockAndChannelsRemoved` nur die eindeutige Konfigurationsblock-ID übergeben.

### 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..