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

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.

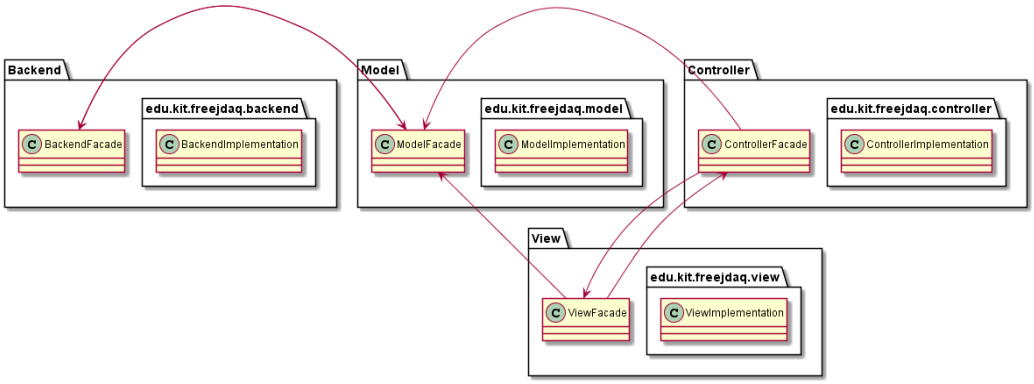

Abbildung 1: Die grobe Struktur des Entwurfs

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

Das Modul Model stellt den Model-Teil des MVC-Entwurfsmusters da und ist für die Erstellung und Verwaltung der Datenstrukturen der Anwendung verantwortlich. Außerdem verarbeitet sie die Sensordaten so, dass diese in Graphen und Tabellen angezeigt werden können. Das Modul besteht dem Modulmanager, den Paketen Core, BuildingBlockBuilder, Sensor-, Transformation-, Representation- und ChannelLogic und diverser Fassaden.

2.2.1 ModelManager

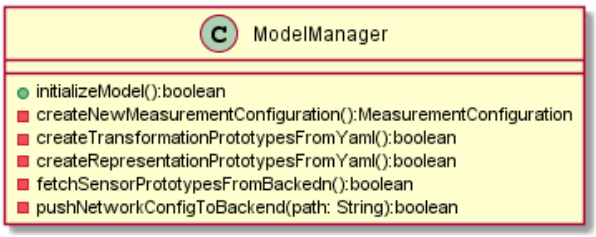

Abbildung 2: Darstellung der Klasse ModelManager

Die Klasse ModelManager ist in Abbildung 2 zu sehen. Sie hat die Aufgabe das Modul Model zu initialisieren. Dazu werden die folgenden sechs Methoden benutzt:

- Die Methode *initializeModel* initialisiert das Modul Model. Dabei nutzt sie vor allem die fünf anderen privaten Methoden.

- Die private Methode *createNewMeasurementConfiguration* erstellt eine neue leere Messkonfiguration für den Startbildschirm.

- Die private Methode *createTransformationPrototypesFromYaml* lässt den Director des BuildingBlockBuilder-Pakets alle von der Anwendung angebotenen Transformationsprototypen erbauen und fügt diese dann dem BuildingBlockDirectory hinzu.

- Die private Methode *createRepresentationPrototypesFromYaml* lässt den Director des BuildingBlockBuilder-Pakets alle von der Anwendung angebotenen Darstellungsprototypen erbauen und fügt diese dann dem BuildingBlockDirectory hinzu.

- Die private Methode *fetchSensorPrototypesFromBackend* holt die Yaml-Dateien von dem Backend und lässt den Director des BuildingBlockBuilder-Pakets alle von

der Anwendung angebotenen Sensorprototypen erbauen und fügt diese dann dem BuildingBlockDirectory hinzu.

- Die private Methode *pushNetworkConfigToBackend* liefert dem Backend die nötigen Netzwerkinformationen um mit dem RaspberryPi zu kommunizieren.

2.2.2 Core

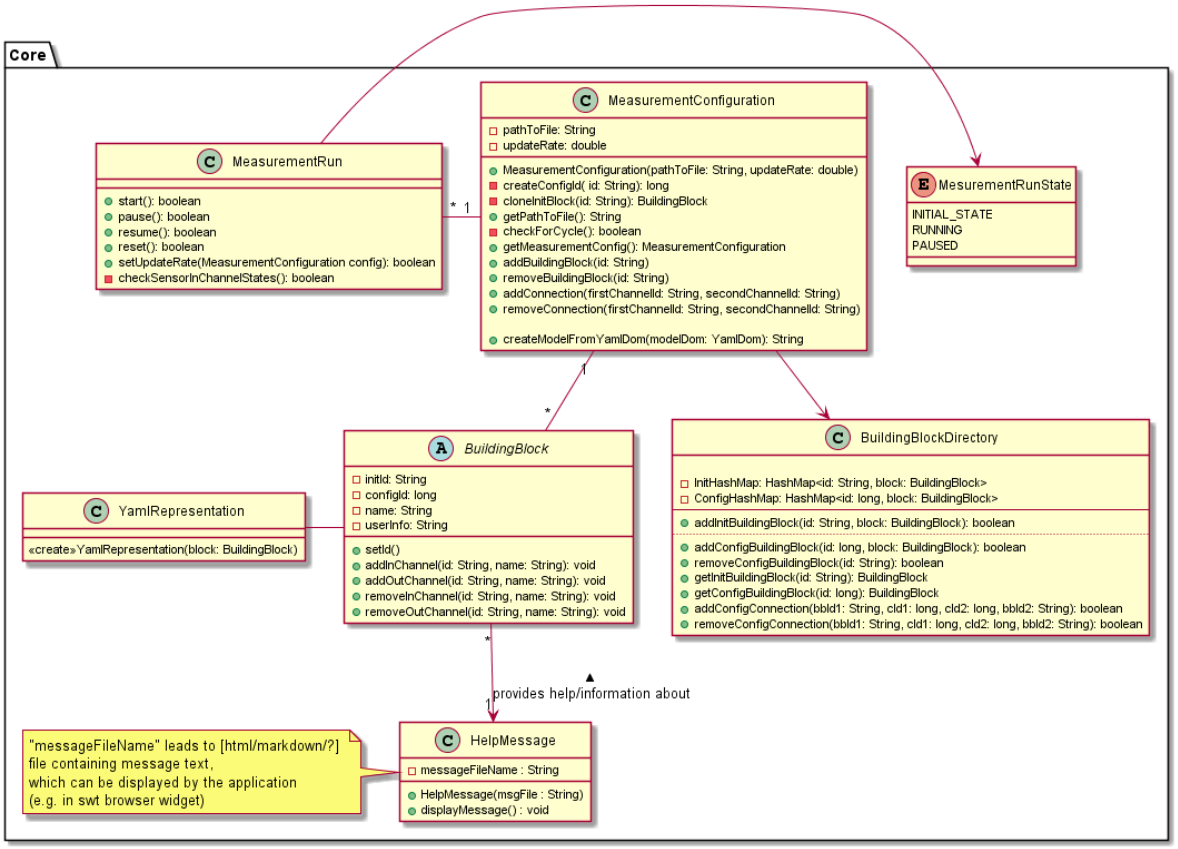

Abbildung 3: Aufbau des Pakets Core

Die Struktur des Pakets Core ist in Abbildung 3 zu sehen. Das Paket Core beinhaltet zentrale Klassen die für die Funktionalität des Models wichtig sind. TODO Mediator Messconfig??

BuildingBlockDirectory Die Klasse BuildingBlockDirectory ist in Abbildung 4 zu sehen. Das erste Attribut der Klasse ist InitHashMap. Sie dient zur Speicherung einer

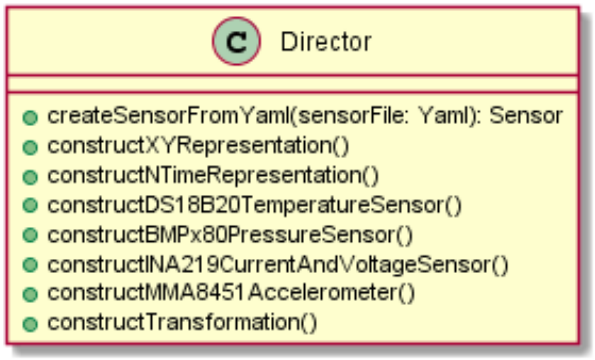

Abbildung 4: Darstellung der Klasse BuildingBlockDirectory

HashMap, die von allen Bausteinen einer Messkonfiguration die initId speichert und zuordnet. Als zweites Attribut ist ConfigHashMap. Sie dient zur Speicherung einer HashMap, die von verfügbaren Prototypbausteinen die configId speichert und zuordnet. Zur Verwaltung der Hashmaps sind folgende Methoden vorhanden:

- Die Methode *addInitBuildingBlock* fügt der InitHashmap einen Bausteinprototypen hinzu.
- Die Methode *addConfigBuildingBlock* fügt der ConfigHashmap einen Baustein hinzu.
- Die Methode *removeConfigBuildingBlock* entfernt von der ConfigHashmap einen Baustein.
- Die Methode *getInitBuildingBlock* gewährt den Zugriff auf einen bestimmen Baustein der Messkonfiguration.
- Die Methode *getConfigBuildingBlock* gewährt den Zugriff auf einen bestimmen Bausteinprototypen.
- Die Methode *addConfigConnection* fügt eine Verbindung zwischen zwei Bausteinen einer Messkonfiguration hinzu.
- Die Methode *removeConfigConnection* entfernt eine Verbindung zwischen zwei Bausteinen einer Messkonfiguration.

MeasurementConfiguration Die Klasse MeasurementConfiguration ist in Abbildung 5 zu sehen. TODO

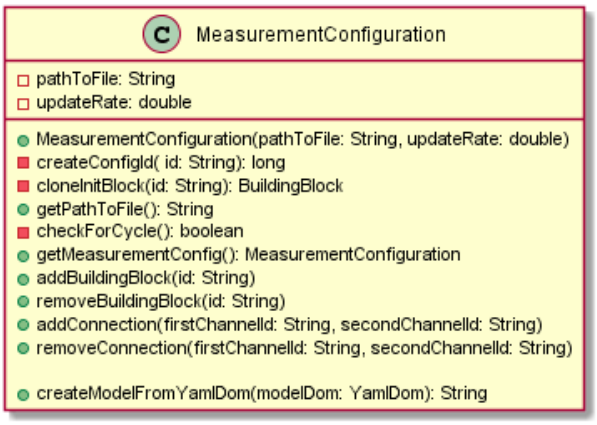

Abbildung 5: Darstellung der Klasse MeasurementConfiguration

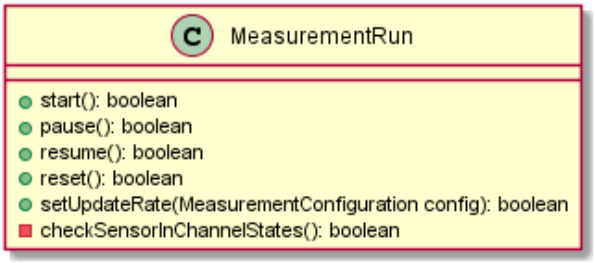

Abbildung 6: Darstellung der Klasse MeasurementRun

MeasurementRun Die Klasse MeasurementRun ist in Abbildung 6 zu sehen. TODO

2.2.3 BuildingBlock

Die abstrakte Klasse BuildingBlock ist in Abbildung 7 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.

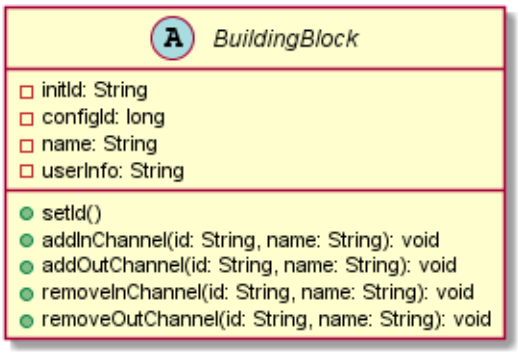

Abbildung 7: Darstellung der Klasse BuildingBlock

- 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 bestimmen 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.

2.2.4 MRunReaction

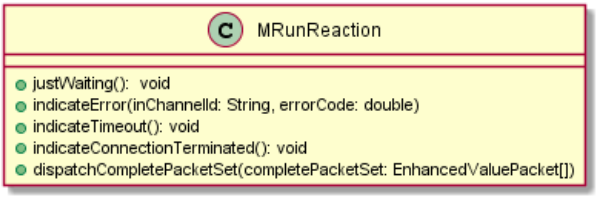

Abbildung 8: Darstellung der Klasse MRunReaction

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 Model 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 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 Eingangschannel jedes Sensors in der Messkonfiguration ein Packet erhält. Ein Datenpaket besteht hier aus Wert, Zielchannel und Zeitstempel.

2.2.5 MRunInfo

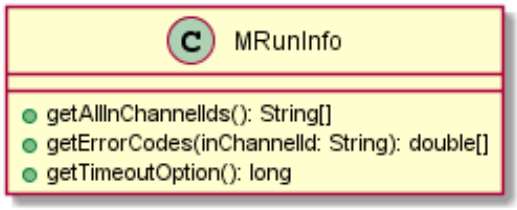

Abbildung 9: Darstellung der Klasse MRunInfo

Die Klasse MRunInfo ist in Abbildung 9 zu sehen. Sie implementiert das Interface MRunInfo vom Cache-Modul. Sie dient dazu, Informationen über den Datenfluss zu erhalten. Diese Informationen werden durch die drei folgenden Methoden erhalten:

- Die Methode *getAllInChannelIds* übergibt dem Model alle Ids von aktiven Kanälen.
- Die Methode *getErrorCodes* übergibt dem Model die ErrorCodes der Kanäle.
- Die Methode *getTimeoutOption* übergibt dem Model die Optionen für TimeOuts.

2.2.6 FacadeViewModel

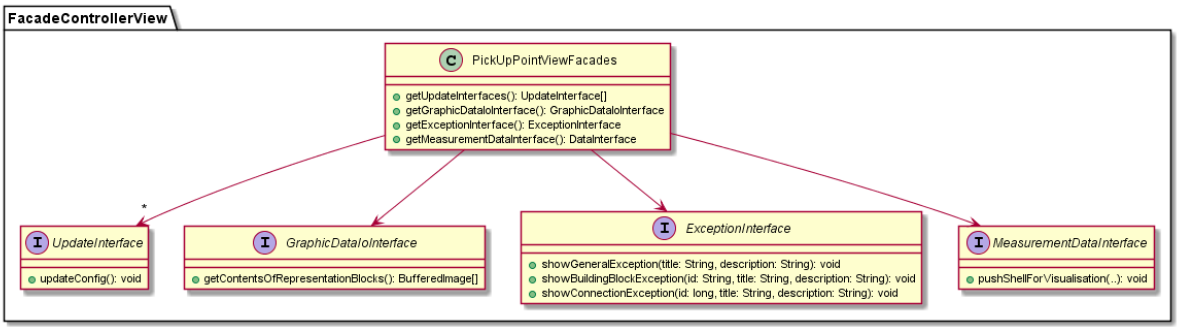

Abbildung 10: Aufbau des Pakets FacadeViewModel

Die Struktur des Pakets FacadeViewModel ist in Abbildung 10 zu sehen. Das Paket FacadeViewModel stellt die Schnittstelle dar, die vom View angeboten und vom Model benutzt wird. Die Facade ist intern in vier Interfaces aufgeteilt um den nutzenden Klassen nur die benötigten Methoden zu übergeben. Die Interfaces sind über die PickUp-Klasse mit dem Modul verbunden.

PickUpPointViewFacades Die Klasse PickUpPointViewFacades stellt die Schnittstelle zwischen den Interfaces und dem Model dar und kapselt diese von einander. Durch ihre Methoden wird das entsprechende geforderte Interface zurückgegeben, dessen Methoden dann verwendet werden können.

MeasurementDataInterface Das Interface MeasurementDataInterface ermöglicht es dem Model, durch die Methode *pushShellForVisualisation*, dem View einen XY-Graphen als Shell zu übergeben, der dann in der GUI dargestellt werden soll.

ExceptionInterface Das Interface ExceptionInterface ermöglicht es dem Model, dem View eine Exception zu übergeben. Durch die Methoden *showGeneralException*, *show-*

BuildingBlockException und *showConnectionException* kann die Art der Exception konkretisiert werden.

UpdateInterface Das Interface UpdateInterface ermöglicht es dem Model der View durch die Methode *updateConfig* die Veränderungen der Messkonfiguration zu signalisieren.

GraphicDataIoInterface TODO ??

2.2.7 SensorLogic

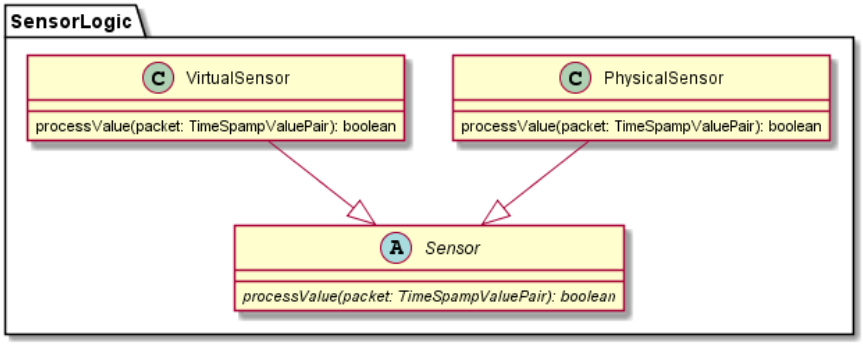

Abbildung 11: Aufbau des Pakets SensorLogic

Die Struktur des Pakets SensorLogic ist in Abbildung 11 zu sehen. Das Paket speichert die Datenstrukturen von verschiedenen Arten von Sensorbausteinen. Dabei wird zwischen physischen Sensoren und virtuellen Sensoren unterschieden.

Sensor Die abstrakte Klasse Sensor dient als Oberklasse aller Sensorbausteine im Model und gibt die Signatur der Methode *processValue* als abstrakte Methode vor.

VirtualSensor Die Klasse VirtualSensor stellt einen Sensorblock dar, der Daten aus einen Datei liest. Dabei enthält die Datei Messwerte aus vergangenen Messläufen oder maschinell erzeugte Daten. Die Methode *processValue* verarbeitet Daten, die über die Eingangskanäle eintreffen.

PhysicalSensor Die Klasse PhysicalSensor stellt einen Sensorblock dar, der Daten aus einem realen Sensor über das Backend erhält. Die Methode *processValue* verarbeitet Daten, die über die Eingangskanäle eintreffen.

2.2.8 TransformationLogic

Transformation

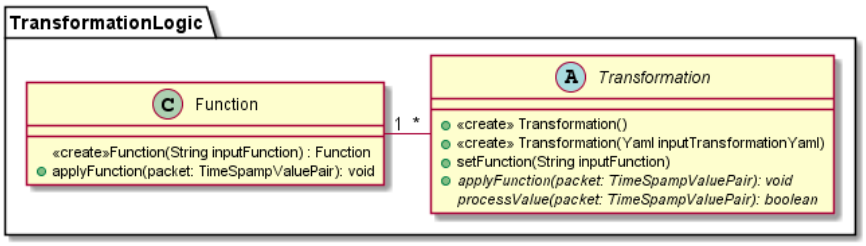

Abbildung 12: Aufbau des Pakets TransformationLogic

Function Die Struktur des Pakets TransformationLogic ist in Abbildung 12 zu sehen. Das Paket dient zur Repräsentation einer Transformation innerhalb des Models.

Transformation TODO

Function TODO

2.2.9 RepresentationLogic

Die Struktur des Pakets RepresentationLogic ist in Abbildung 13 zu sehen. Das Paket dient zur Repräsentation einer Darstellung innerhalb des Models. Dabei wird zwischen Graphen und Tabellen unterschieden.

Representation Die abstrakte Klasse Sensor dient als Oberklasse aller Darstellungsbausteine im Model und gibt die Signatur der Methode *processValue* als abstrakte Methode vor.

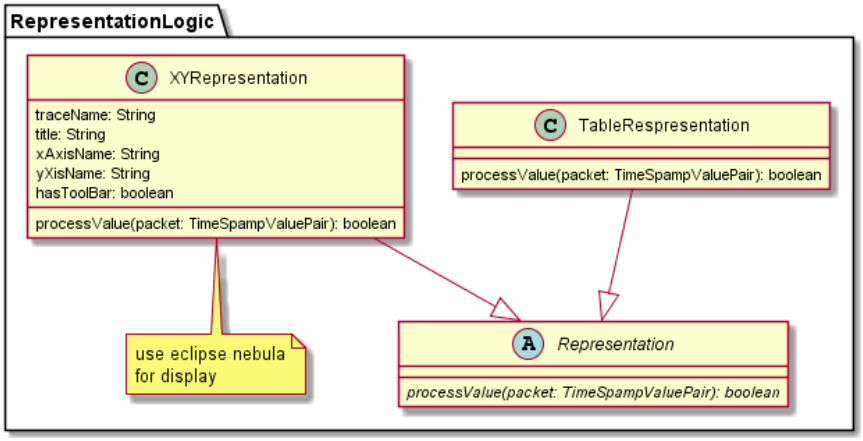

Abbildung 13: Aufbau des Pakets RepresentationLogic

TableRepresentation Die Klasse TableRepresentation stellt im Model eine Darstellung von Daten durch eine Tabelle dar. Die Methode *processValue* verarbeitet die eintreffenden Daten.

XYRepresentation Die Klasse XYRepresentation stellt im Model eine Darstellung von Daten durch einen Graph dar. Ein XYGraph die folgenden Attribute:

- Das Attribut traceName speichert den Namen des Traces.
- Das Attribut title speichert den Titel des Graphen.
- Das Attribut xAxisName speichtert den Namen der X-Achse.
- Das Attribut yAXisName speichtert den Namen der Y-Achse.
- Das Attribut hasToolBar gibt an, ob der Graph eine interaktive Toolbar anbietet.

Die Methode *processValue* verarbeitet die eintreffenden Daten und erstellt mit Hilfe der externen Library Eclipse Nebula einen Graphen.

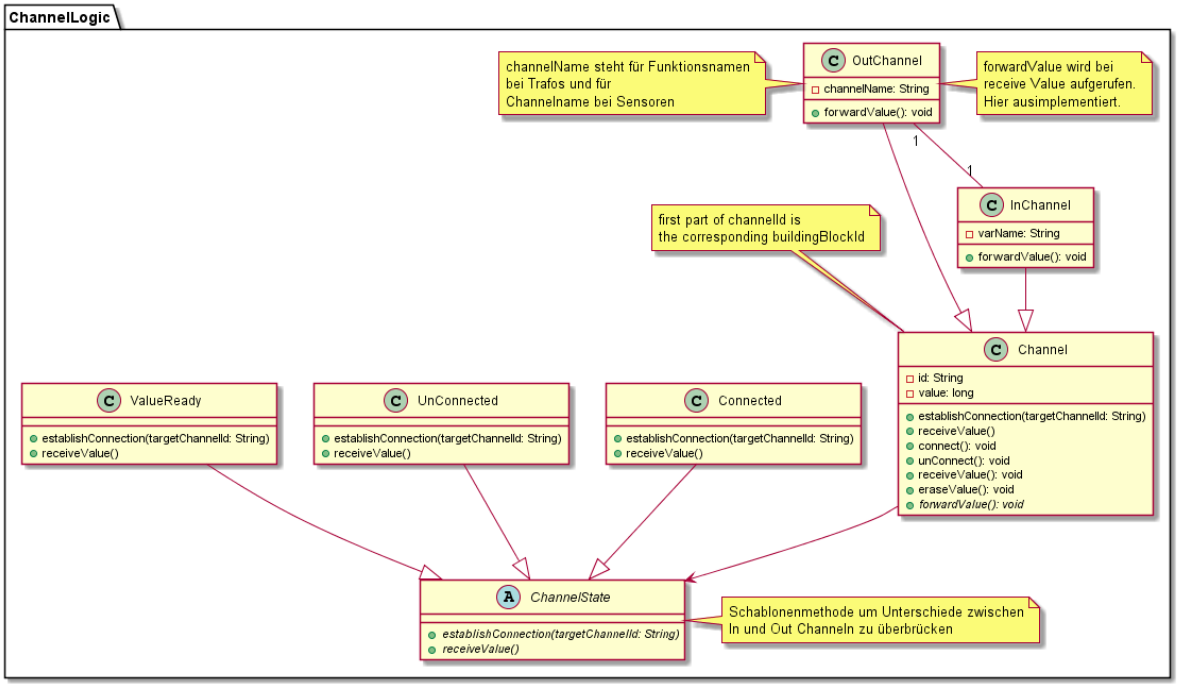

Abbildung 14: Aufbau des Pakets ChannelLogic

2.2.10 ChannelLogic

Die Struktur des Pakets ChannelLogic ist in Abbildung 14 zu sehen. Die Klasse repräsentiert die Datenstruktur eines Kanals. Dabei wird das Entwurfsmuster Zustand mit folgenden Rollen verwendet:

- Die Rolle der Clients wird durch die Klasse Channel erfüllt.
- Die Rolle des Zustandes wird von der abstrakten Klasse ChannelState erfüllt.
- Das Rolle des konkreten Zustandes wird durch die Klassen Connected, UnConnected und ValueReady erfüllt.

Das Entwurfsmuster ermöglicht es, die Anzahl der Zustände bei Bedarf zu erweitern.

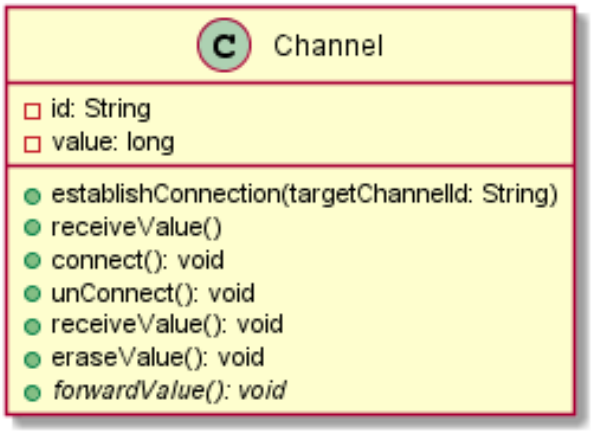

Abbildung 15: Darstellung der Klasse Channel

Channel Die Klasse Channel ist in Abbildung 15 zu sehen. Die Klasse Channel dient als Repräsentation eines Kanals im Model. Ein Kanal als Attribute eine eindeutige Id und einen Wert. Der Wert ist ein konkreter Wert eines Messlaufs und wird im Channel temporär zwischengespeichert. Ein Kanal hat die Aufgabe Daten zu empfangen und wieder zu versenden. Außerdem dient er als Ausgangspunkt einer Verbindung unter verschiedenen Bausteinen. Um die Verbindung und den Datenfluss zu verwalten, hat ein Kanal folgende Methoden:

- Die Methode *establishConnection* ermöglicht es dem Kanal eine Verbindung mit einem anderen Kanal einzugehen. Dabei ist nur eine Verbindung zwischen einem Eingangs- und einem Ausgangskanal möglich.

- Die Methode *receiveValue* ermöglicht es dem Kanal einen Wert zu empfangen.
- Die Methode *connect* wechselt den Zustand eines Kanals zu: vebundenen.
- Die Methode *unConnect* wechselt den Zustand eines Kanals zu: nicht vebundenen.
- Die Methode *eraseValue* löscht den Wert, der im Channel gespeichert ist.
- Die abstrakte Methode *forwardValue* dient dazu einen erhaltenen Wert weiter zu versenden. Die Implementierung der Methode ist in den Unterklassen zu finden.

InChannel Die Klasse InChannel ist eine Unterklasse von Channel und dient als Datenstruktur eines Eingangskanals im Model. Ein InChannel hat das zusätzliche Attribut varName, TODO Die Methode *forwardValue* wird hier implementiert.

OutChannel Die Klasse InChannel ist eine Unterklasse von Channel und dient als Datenstruktur eines Ausgangskanals im Model. TODO channelName Die Methode *forwardValue* wird hier implementiert.

ChannelState Die abstrakte Klasse ChannelState gibt als abstrakter Zustand die Signaturen der Methoden für der konkreten Zustände vor. Außerdem werden Unterschiede zwischen In- und OutChannel der Methoden *establishConnection* und *receiveValue* durch eine Schablonenmethode überbrückt.

Connected Die Klasse Connected repräsentiert den verbundenen Zustand eines Kanals.

UnConnected Die Klasse UnConnected repräsentiert den unverbundenen Zustand eines Kanals.

ValueReady Die Klasse ValueReady repräsentiert einen "bin bereitZustand des Kanal. Damit wird signalisiert, dass der Channel einen Wert erfolgreich empfangen hat und weiter versenden kann.

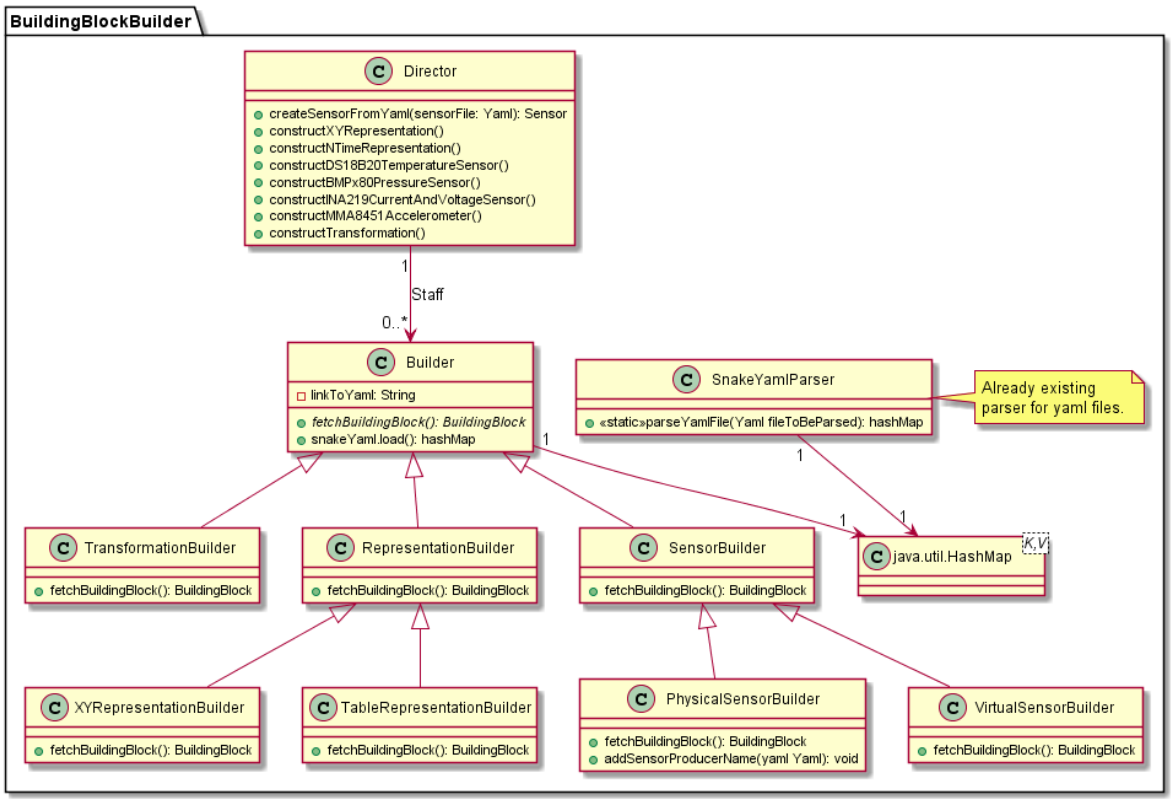

Abbildung 16: Aufbau des Pakets BuildingBlockBuilder

2.2.11 BuildingBlockBuilder

Der Aufbau des Pakets BuildingBlockBuilder, zu sehen in Abbildung 16, 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.

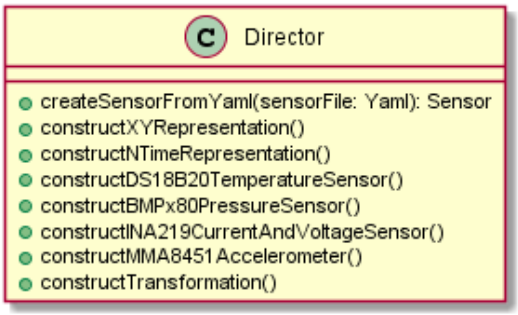

Abbildung 17: Darstellung der Klasse Director

Director Die Klasse Director ist in Abbildung 17 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 entsprechenden 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.

RepresenstationBuilder 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.

TableRepresenstationBuilder 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.

XYRepresenstationBuilder 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¡K,V¿ TODO

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

2.3 Controller

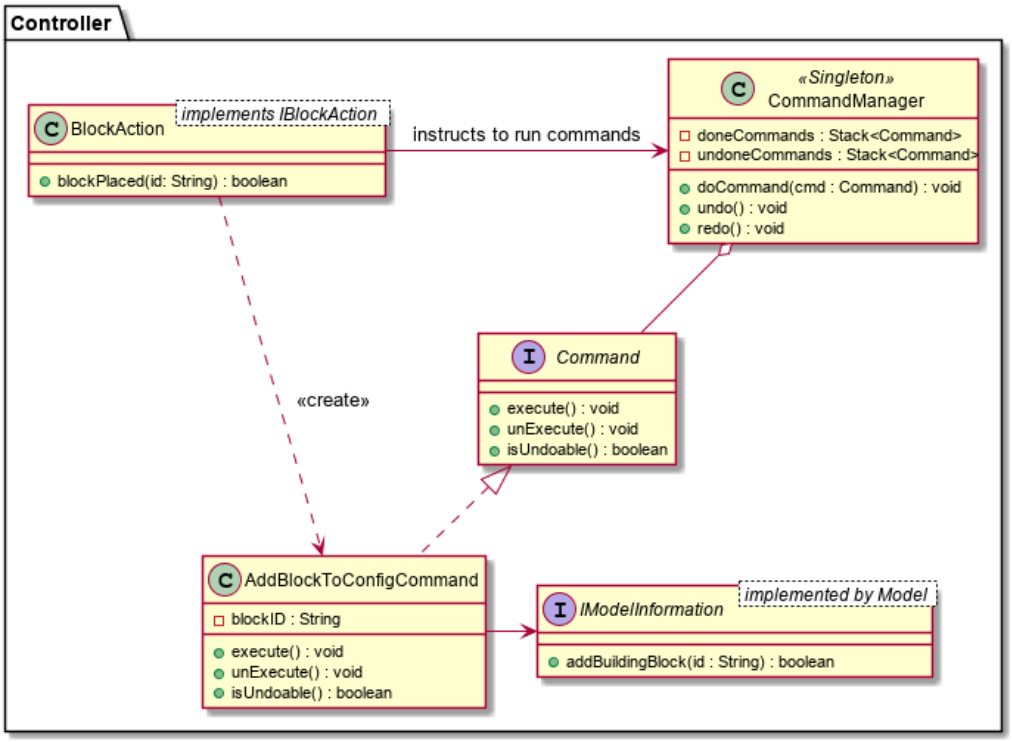

Abbildung 18: Die Struktur des Controllers (Ausschnitt)

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

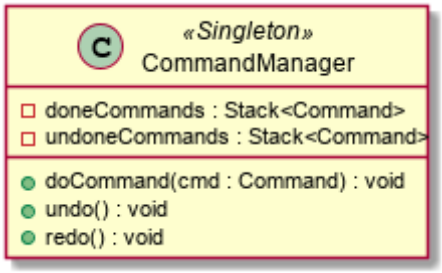

Abbildung 19: Die Klasse CommandManager

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.

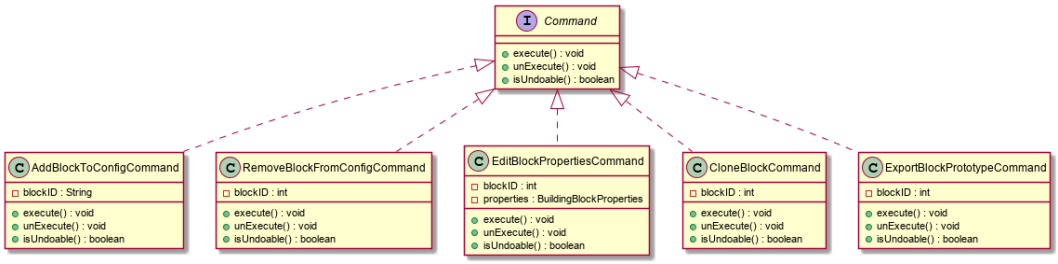

Abbildung 20: Baustein-Befehle

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 klont einen bestehenden Bausteinprototyp.

ExportBlockPrototypeCommand Dieser Befehl exportiert einen Bausteinprototypen als Datei.

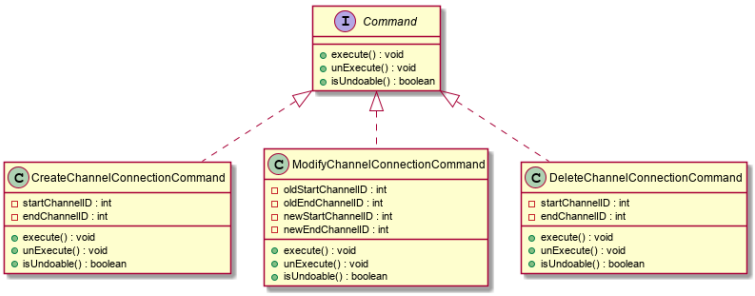

Abbildung 21: Verbindungs-Befehle

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.

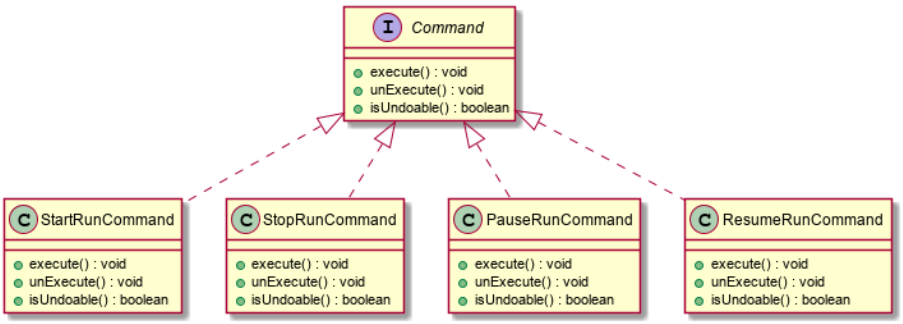

Abbildung 22: Messlauf-Befehle

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.)

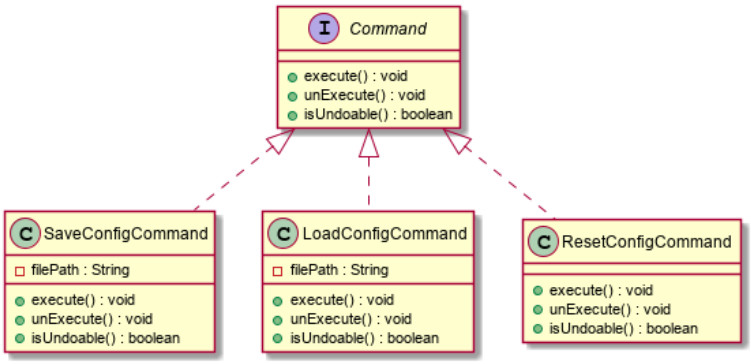

Abbildung 23: Messkonfigurations-Befehle

2.3.3 Verbindung zum View

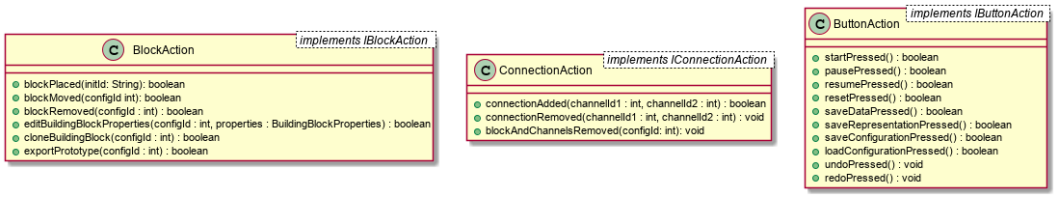

Abbildung 24: Schnittstellenimplementierung View zu Controller

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.

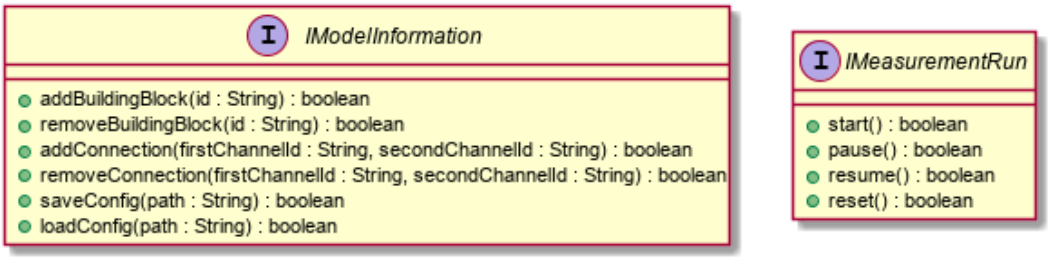

Abbildung 25: Schnittstelle Controller zu Model

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- Entwurfmusters die Darstellungen des Modells dar und realisiert Benutzerinteraktionen auf der graphischen Benutzeroberfläche.

2.4.1 MainWindow

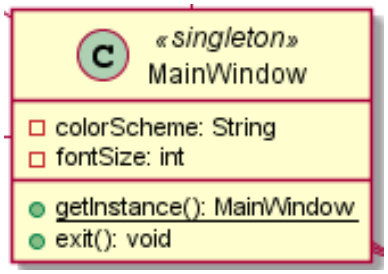

Abbildung 26: Die Klasse MainWindow

Die Klasse MainWindow, zu sehen in Abbildung 26, stellt den Rahmen der Benutzeroberfläche dar. Alle Restlichen graphischen Oberflächen werden durch das MainWindow instanziert. Dazu gehört das Konfigurationsfeld, Buttonmenü, Konfigurationsbausteinmenü, Hilfe , Optionen und Fehlerfenster. Da MainWindow, dass Entwurfsmuster Singleton verwendet, kann die Anwendung nur ein MainWindow besitzen soll. Bei Schließen des MainWindow wird ebenso die gesamte Anwendung beendet.

2.4.2 Menues

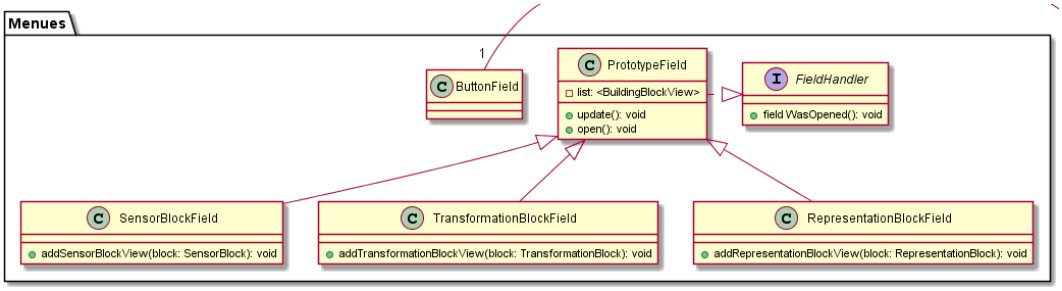

Abbildung 27: Aufbau des Menü-Paket

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

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 registiert 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

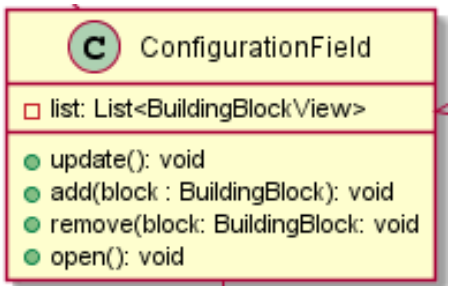

Abbildung 28: Aufbau des Konfigurationsfeld

ConfigurationField Die Klasse KonfigurationField, zu sehen in Abbildung 28 stellt das Konfigurationsfeld 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.

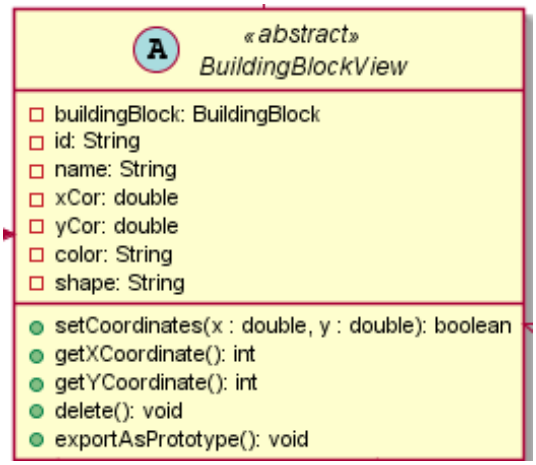

Abbildung 29: Aufbau des BuildingBlockView

BuildingBlockView Die Klasse BuildingBlockView ist in Abbildung 29 zu sehen. Sie stellt 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 Initialsierungs-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 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.

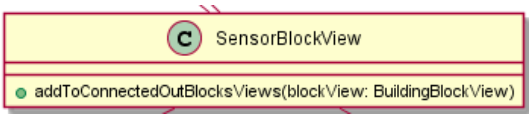

Abbildung 30: Aufbau des SensorBlockView

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.

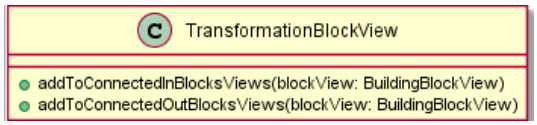

Abbildung 31: Aufbau des TransformationBlockView

TransformationBlockView Die Klasse TransformationBlockView ist die visuelle Darstellung einer Transformation dar. Transformationsbausteine besitzen eine vordefinierte Funktion, welche die Messdaten nach der Funktion transformiert. Transformationbausteinen besitzen Eingänge, welche Daten von Sensorenbausteinen oder anderen Transformationenbausteinen empfangen können. Ausgänge der Transformationsbausteine können nur an weitere Transformationsbausteine oder Darstellungsbausteine angebunden werden, um einen sinnvollen Datenfluss zu ermöglichen.

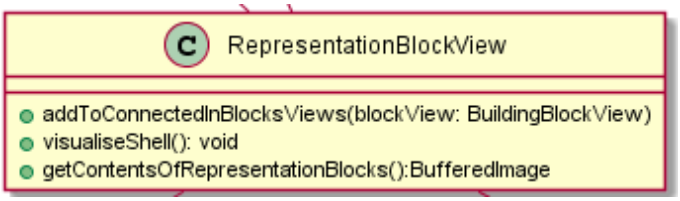

Abbildung 32: Aufbau des RepresentationBlockView

RepresentationBlockView Die Klasse RepresentationBlockView stellt einen Darstellungbaustein 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 visualisierte Messdaten nicht mehr verarbeitet werden.

Abbildung 33: Aufbau des BlockDragAndDropHandler

BlockDragAndDropHandler Über das Interface BlockDragAndDropHandler, welches in Abbildung 33 zu sehen ist, 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.

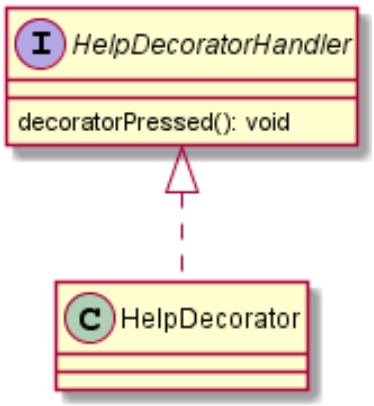

Abbildung 34: Klasse HelpDecorator mit Interface HelpDecoratorHandler

HelpDecorator Jeder Konfigurationsbaustein besitzt eine kurze Beschreibung seiner Art und Funktionalität. Diese Beschreibung wird über das Fenster der Klasse HelpDecorater dem Benutzer dargestellt. Die Abbildung der Klasse ist in Abbildung 34 zu sehen.

HelpDecoratorHandler Das Interface HelpDecoraterHandler, zu sehen in Abbildung 34 wird von der Klasse HelpDecorater implementiert und registriert wenn der Benutzer den HelpDecorater öffnen will, in dem er darauf drückt.

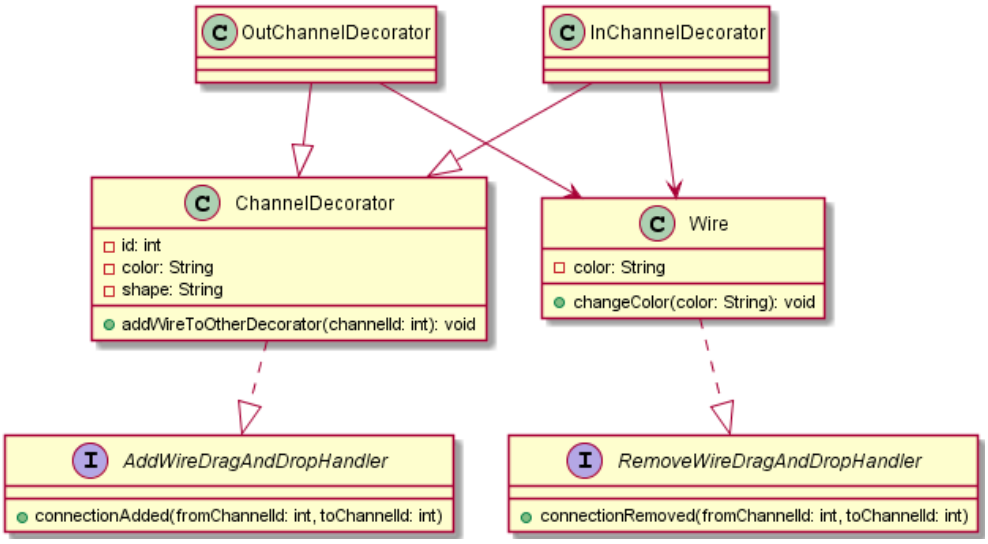

Abbildung 35: Aufbau der ChannelDecorator-Logik

ChannelDecorator Die Klasse ChannelDecorater stellt die Überklasse zu den Klassen InChannelDecorator und OutChannelDecorator dar. Ein ChannelDecorater ist ein Objekt, welches durch Konfigurationsbausteine instanziiert werden. Ein ChannelDecorater stellt die Schnittstelle zu anderen ChannelDecorater dar, um zwei Bausteine miteinander zu verbinden. Dies geschiet durch die Methode addWireToOtherDecorator. Um diese ChannelDecorater eindeutig unterscheiden zu können besitzten diese eine eindeutige ID. Zur visuellen Repräsentation der Ein- und Ausgänge besitzen diese eine Farbe und Form.

InChannelDecorator Die Klasse InChannelDecorator erbt von der Klasse ChannelDecorater und stellt einen Kanaleingang eines Konfigurationsbausteins dar. Über einen Kanaleingang können nur Messdaten in den Konfigurationsbaustein reinkommen und werden ja nach Bausteinart verarbeitet.

OutChannelDecorator Die Klasse OutChannelDecorator ist ebenfalls eine Unterklasse der Klasse ChannelDecorator und stellt einen Kanalausgang eines Konfigurationsbaustein dar. Über diese Kanalausgänge werden Messdaten aus einem Sensor oder bereits bearbeitete Messdaten an den nächsten Konfigurationsblock weitergebe.

Wire Die Klasse Wire stellt die visuelle Darstellung einer Verbindung zwischen einem InChannelDecorator und einem OutChannelDecorator dar. Jede Verbindung besitzt eine

Farbe, welche sich z.B in einem Fehlerfall ändern kann, um den Benutzer auf den Fehler bei dieser Verbindung aufmerksam zu machen.

AddWireDragAndDropHandler Damit der Benutzer zwei Konfigurationsbausteine mit einer Verbindung verbinden kann, gibt das Interface AddWireDragAndDropHandler an die Anwendung weiter, wenn der Benutzer zwei ChannelDecorater miteinander verbunden hat, indem er die eindeutige ID dieser mitgibt.

RemoveWireDragAndDropHandler Um eine Verbindung zu entfernen gibt das Interface RemoveWireDragAndDropHandler an die Anwendung weiter, wenn eine Verbindung entfernt wurde. Mitgegeben werden als Parameter hierbei die zwei eindeutigen IDs der Kanal Ein- und Ausgänge.

2.4.4 BuildingBlockProperties

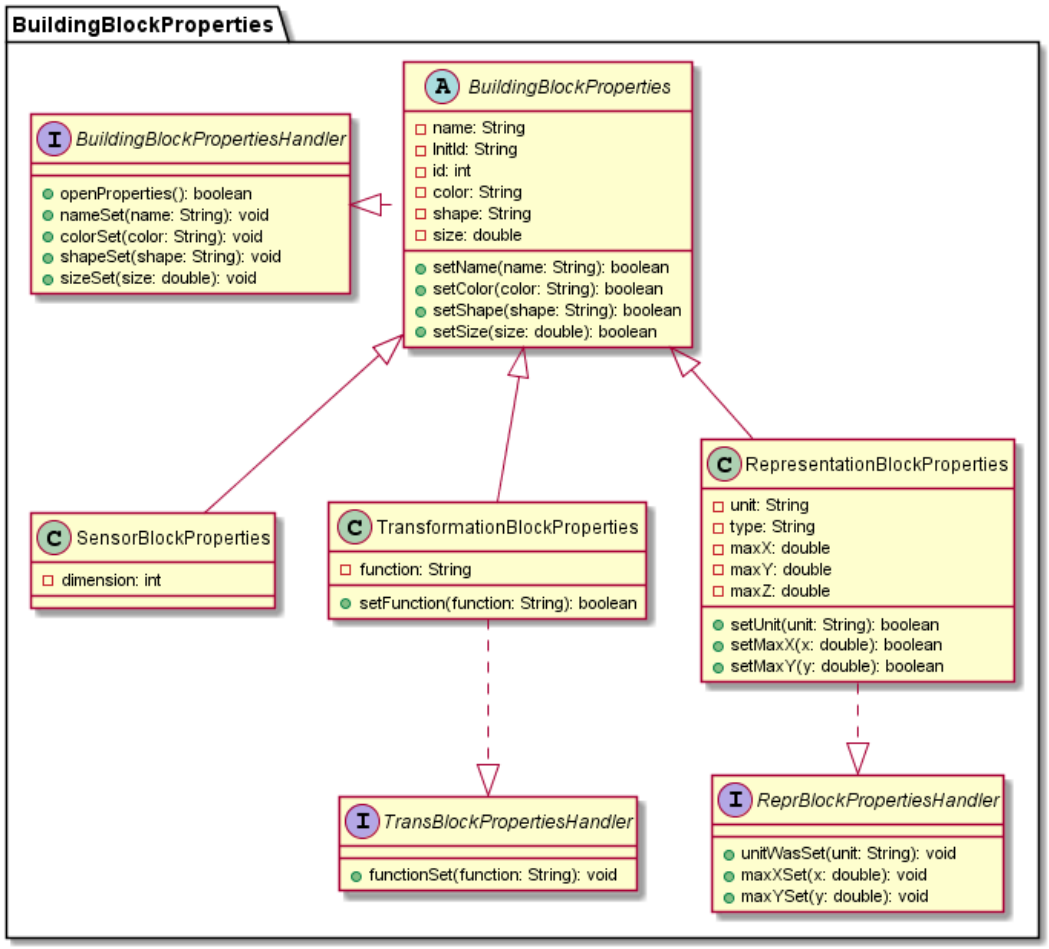

Abbildung 36: Aufbau des BuildingBlockProperties-Paket

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 Ëigenschaften-Menü"bereit, in welchem dem Benutzer die wichtigsten Eigenschaften zu sehen bekommt.

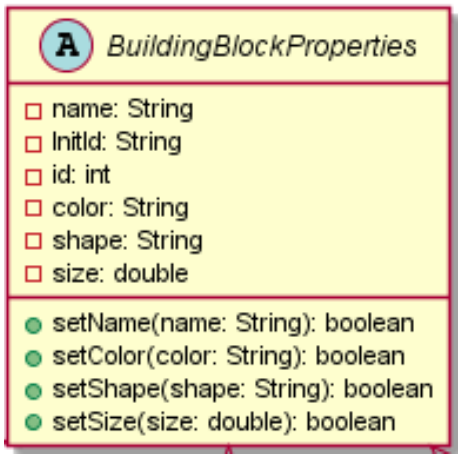

Abbildung 37: Aufbau der BuildingBlockProperties

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.

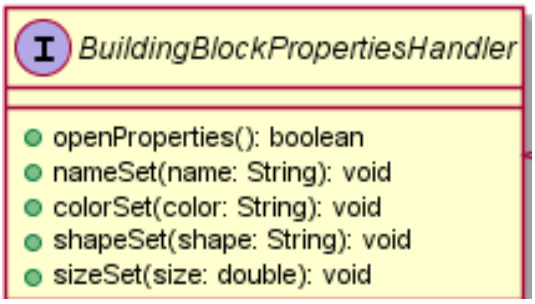

Abbildung 38: Aufbau des BuildingBlockPropertiesHandler

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.

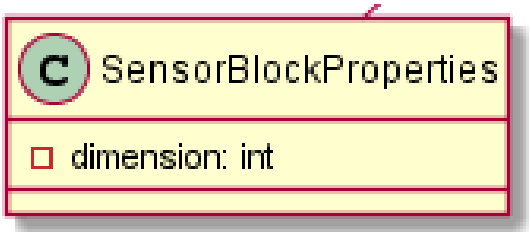

Abbildung 39: Klasse SensorBlockProperties

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.

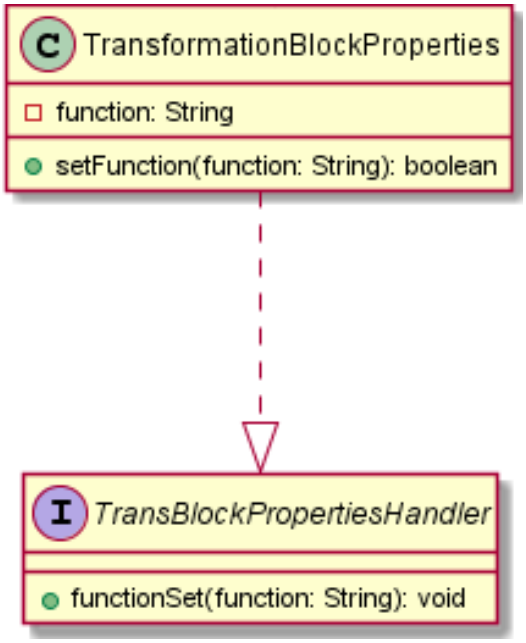

Abbildung 40: Aufbau der Klasse TransformationBlockProperties und dem Interface TransBlockPropertiesHandler

TransformationBlockProperties Transformationsbausteine besitzen neben den Standart-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.

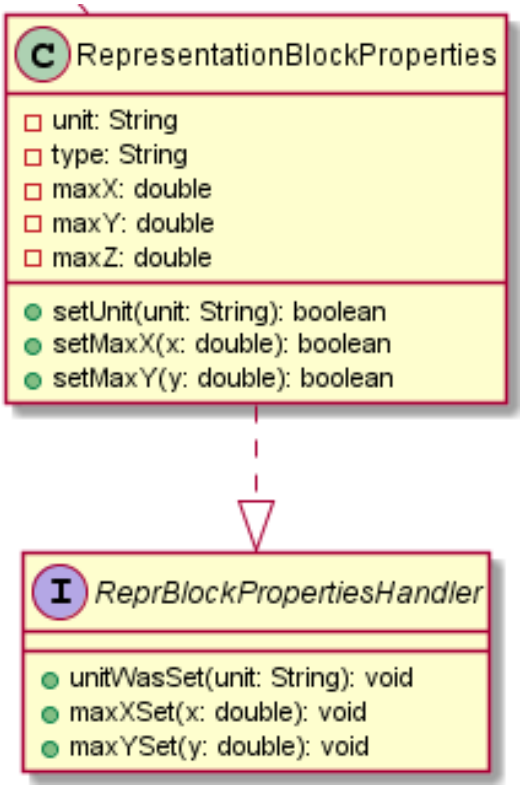

Abbildung 41: Aufbau der Klasse RepresentationBlockProperties und Interface ReprBlockPropertiesHandler

RepresentationBlockProperties Da Repräsentatonsbausteine 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 setzten. 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.

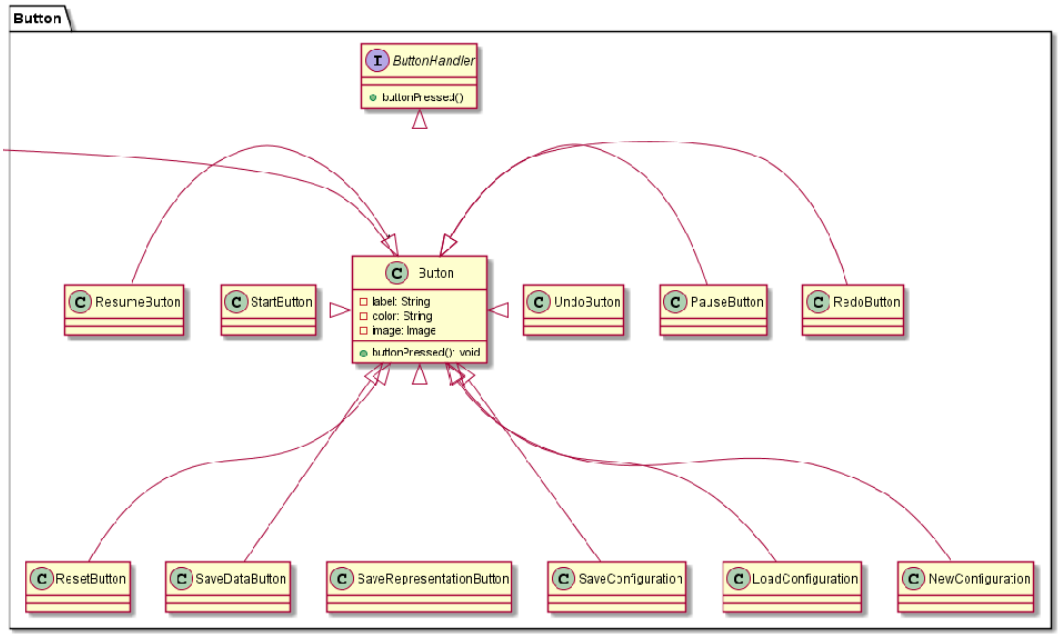

Abbildung 42: Aufbau des Button-Paket

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

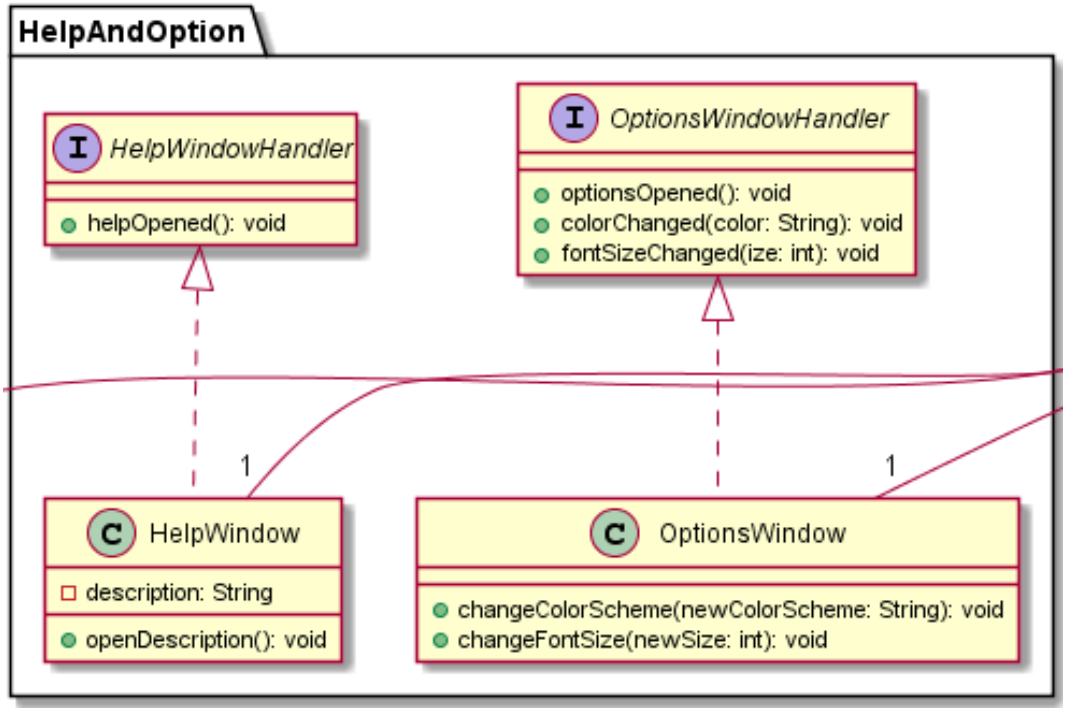

Abbildung 43: Aufbau des HelpAndOption-Paket

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 Einstellungsmöglichkeiten gibt, um eine möglichst barrierefreie Benutzung zu ermöglichen.

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. Eben-

falls 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 Einstellungenöder Optionen Knopf. In dem geöffneten Fenster kann der Benutzer dann das Farbschema und die Schriftgröße anpassen.

2.4.7 Exception

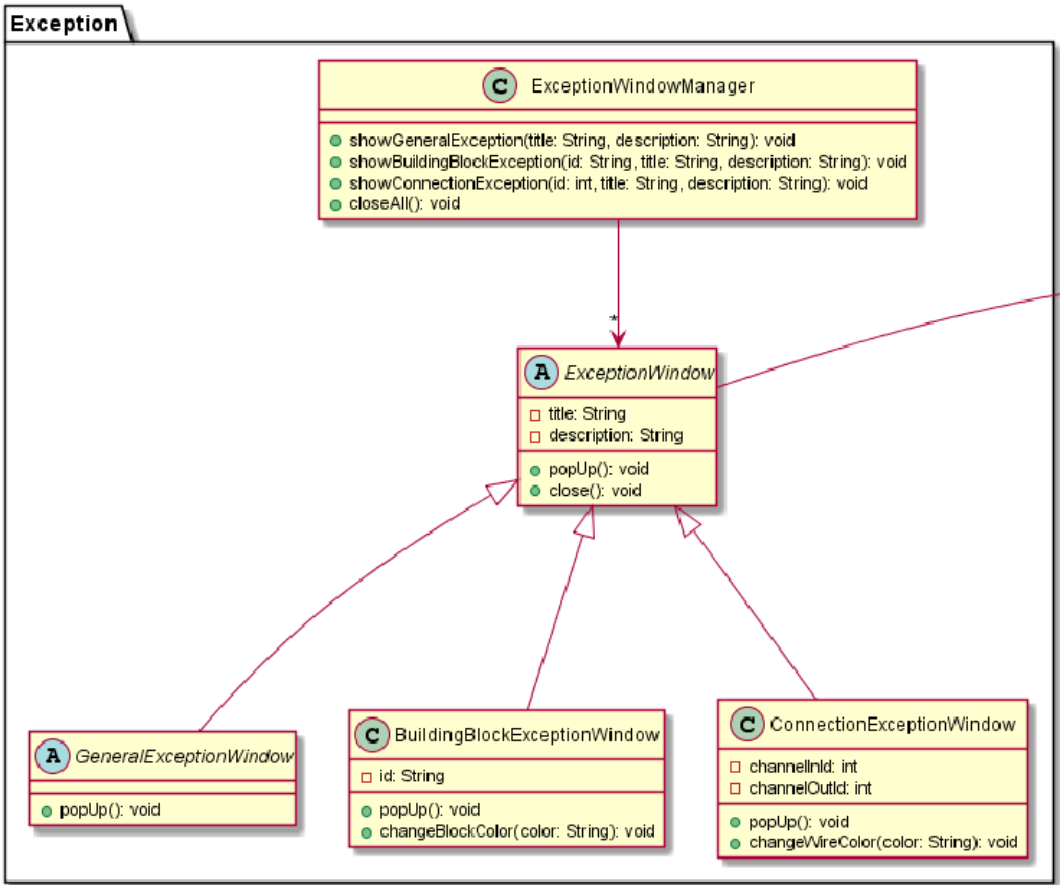

Abbildung 44: Aufbau des Exception-Paket

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.

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

BuildingBlockExceptionWindow Die Klasse BuildingBlockExceptionWindow ist eine Konkretisierung der Überklasse ExceptionWindow und stellt eine Fehlernachricht im Bezug zu Konfigurationsbausteinen dar. Neben einem Titel und einer Beschreibung wird zur Erzeugung dieser Fehlernachricht 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 Konfigurationsbaustein 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 Fehlernachricht bei Verbindungen zwischen Konfigurationsbausteinen dar. Zur Identifizierung der Fehlerquelle wird neben Titel und Beschreibung ebenfalls die IDs der Ein- und Ausgangskanäle der Konfigurationsbaustein mit übergeben. Die Methode popUp() soll ebenfalls die Fehlernachricht in der Nähe der Fehlerquelle im Konfigurationsfeld plazieren. 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 Fehlernachricht dar. Diese werden zum Beispiel bei Messfehlern oder Fehler bei der Messkonfiguration ausgelöst. Diese Fehlernachrichten sollen sichtbar in der Mitte der Anwedung 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.

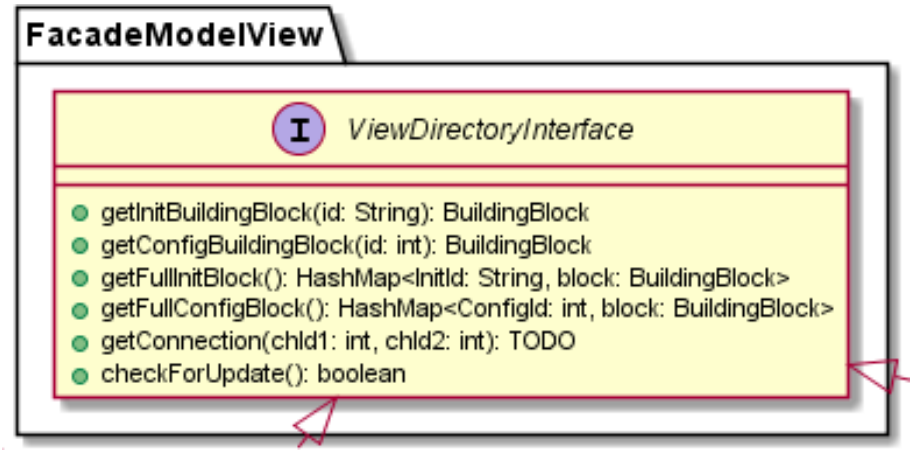

Abbildung 45: Aufbau des FacadeModelView-Paket

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.

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 Konfigurationsbausteinen enthält zurück um daraus die Prototypenmenüs zu erstellen. Um einzelne Bausteine mit bestmmter 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 ChannelDecorater-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

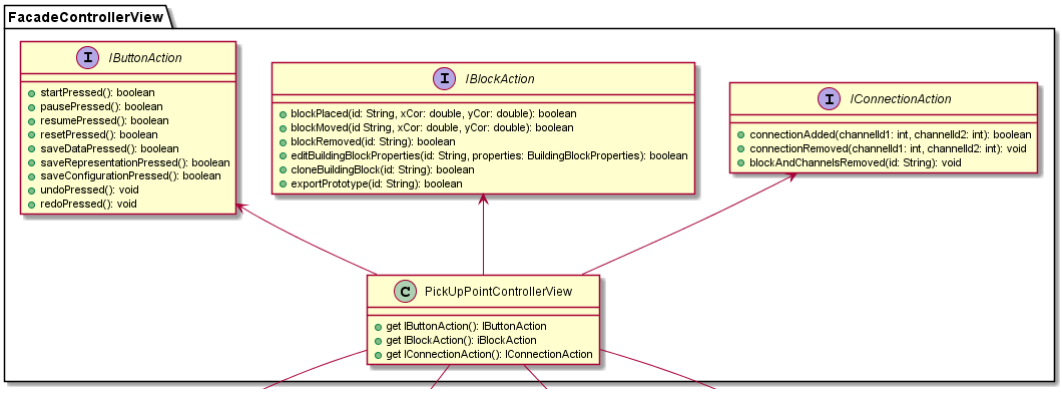

Abbildung 46: Aufbau des FacadeControllerView-Paket

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

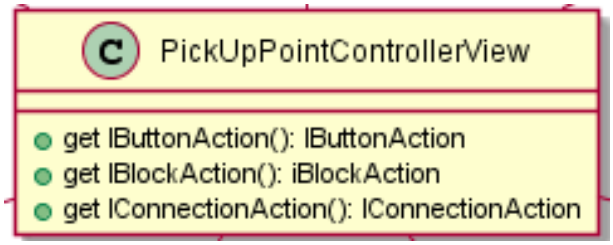

Abbildung 47: Die Klasse PickUpPointControllerView

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.

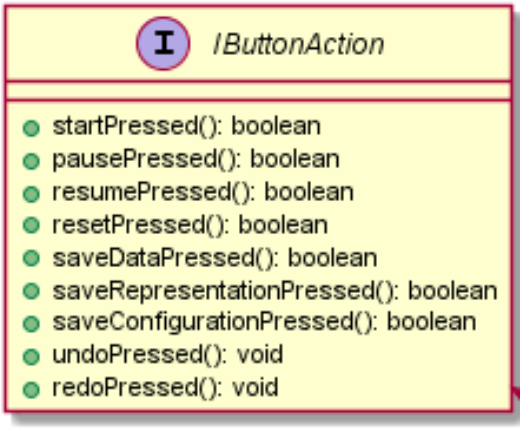

Abbildung 48: Das Interface IButtonAction

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.

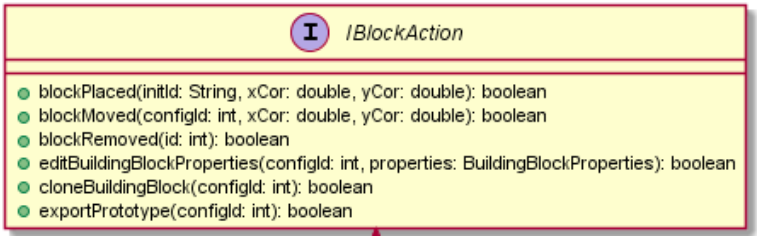

Abbildung 49: Das Interface IBlockAction

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-Drap 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 Konfiguratonsfeld 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.

Abbildung 50: Das Interface IConnectionAction

IConnectionAction Das Interface IConnectionAction 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

3.1 Block hinzufügen, undo, redo

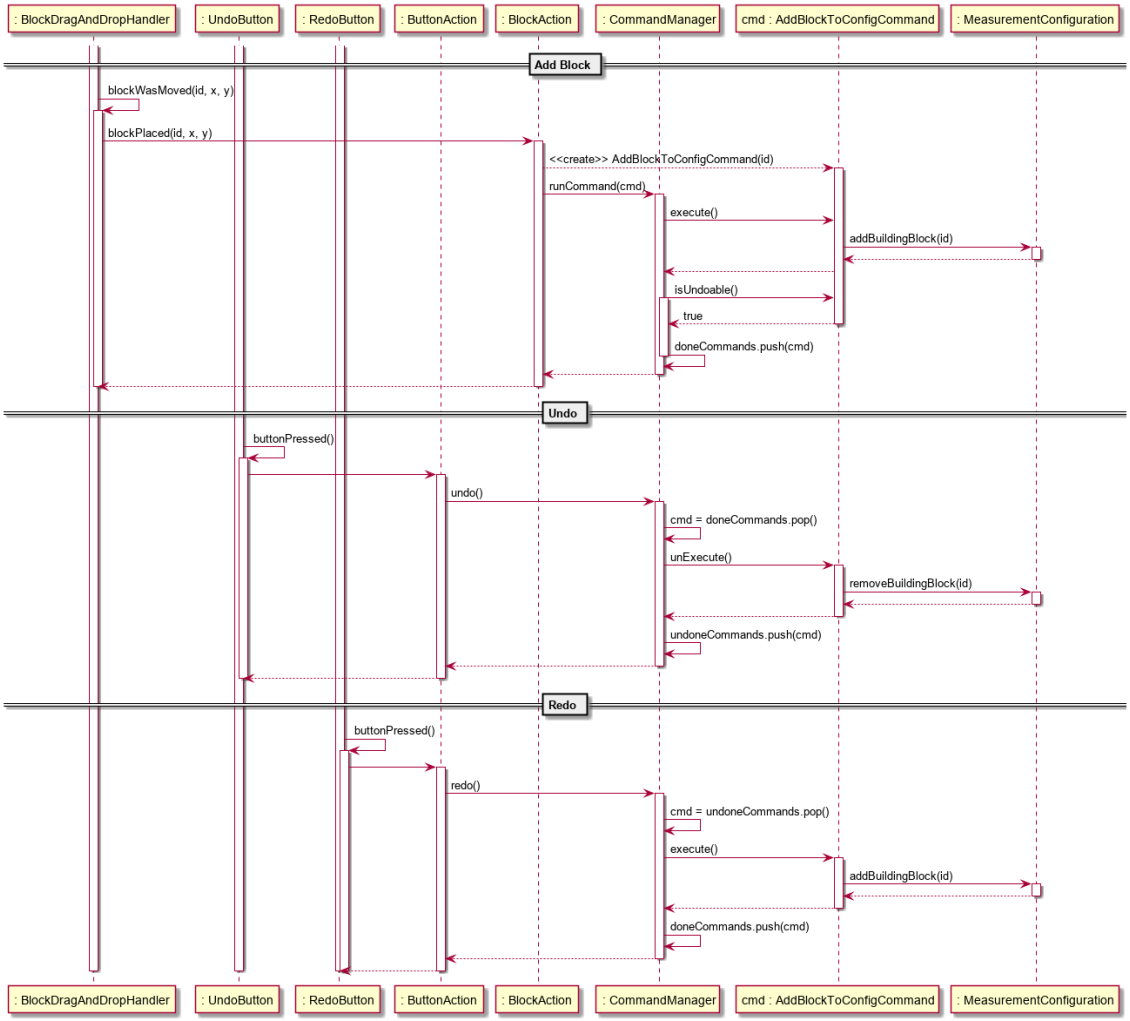

Abbildung 51: Das Hinzufügen eines Bausteins zur Messkonfiguration, außerdem das rückgängig machen und wiederherstellen dieser Aktion.

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..