

IV Visualizer: Entwurfsheft

Fraunhofer-Institut für Optronik, Systemtechnik und Bildauswertung IOSB

Josua Benjamin EYL, Lukas Friedrich,
Max Bretschneider, Nathaniel Till Hartmann, Robin Köchel

Betreut von: Mickael Cormier M.Sc., Stefan Wolf M.Sc

Wintersemester 2023/2024

Inhaltsverzeichnis

1 Einleitung	4
2 Architekturbeschreibung	5
2.1 gRPC	6
3 Model	11
3.1 Entwurf	11
3.2 Klassen	11
3.2.1 DataManager	11
3.2.2 DBManager	12
3.2.3 FSManger	13
3.2.4 Encoding	14
3.3 Datenspeicherung	15
3.3.1 Scylla	15
3.3.1.1 Tabellen	15
3.4 Ablaufbeschreibungen	16
4 Controller	18
4.1 Entwurf	18
4.2 Klassen	18
4.2.1 MessageHandler	18
4.2.2 ThreadHandler	19
4.2.3 ThreadIncoming	21
4.2.4 ThreadOutgoing	22
4.2.5 StreamReader	23
4.2.6 DataFrameFactory	24
4.2.7 TimeRange	24
4.2.8 LiveThread	25
4.2.9 Queue	26
4.2.10 Config	27
4.2.11 ConfigGStreamer	28
4.2.12 ConfigContainer	29
4.2.13 ConfigDetector	30
4.3 Ablaufbeschreibungen	32
4.3.1 Bilder aus dem Backend Anfragen	32
4.3.2 LiveStream	32
4.3.3 Starten eines Streams	33
4.3.4 Setzen der Config	34
5 View	35
5.1 Entwurf	35
5.2 Klassen	36
5.2.1 GUI	36
5.2.1.1 SettingsDialog	36
5.2.1.2 QPolygonPainterLabel	38
5.2.1.3 RegionOfInterestDialog	39
5.2.1.4 ExportDialog	40
5.2.1.5 ConfigDialog	41
5.2.1.6 ClusterFrame	42
5.2.1.7 CamFrame	43

5.2.1.8	ProfilerFrame	44
5.2.1.9	MainWindow	45
5.2.2	VideoPlayer	48
5.2.2.1	VideoPlayerWidget	48
5.2.2.2	VideoNavigatorWidget	50
5.2.2.3	SliderNavigator	51
5.2.3	ViewTypes	52
5.2.3.1	ViewTypeEnum	52
5.2.3.2	ViewTypeFrameWidget	53
5.2.3.3	ViewType	54
5.2.3.4	OnlyVideo	54
5.2.3.5	AnnotationWithoutVideo	55
5.2.3.6	AnnotationAndVideo	56
5.2.4	Pipeline	56
5.2.4.1	PipelineObject	57
5.2.4.2	VideoPipe	57
5.2.4.3	AnnotationPipe	58
5.2.4.4	RegionOfInterestPipe	59
5.2.5	BackendConnector	60
5.2.5.1	TimeRange	60
5.2.5.2	InitialisationMessage	60
5.2.5.3	DBConfig	61
5.2.5.4	Stream	61
5.2.5.5	Reader	62
5.2.5.6	GRPCReader	64
5.2.5.7	Writer	65
5.2.5.8	GRPCWriter	66
5.2.6	Data.Config	67
5.2.7	Data.FrameData	67
5.2.7.1	DataFrameStream	68
5.2.7.2	PipelineData	68
5.2.7.3	DataFrame	69
5.2.7.4	Image	70
5.2.7.5	Annotation	70
5.2.7.6	BoundingBox	71
5.2.7.7	RegionOfInterest(ROI)	72
5.2.8	Log	74
5.2.8.1	LoggerFrame	74
5.2.8.2	LogDatabaseConnector	75
5.2.8.3	Log	76
5.2.8.4	LogType	77
5.3	Ablaufbeschreibungen	78
5.3.1	Anzeigen eines neuen Streams	78
5.3.2	Anzeigen der Logdaten	79
5.3.3	Exportieren eines Videos ohne Annotationen	80
5.3.4	Füge eine neue Maske mit 3 Punkten hinzu	81
5.3.5	Füge ein neuen Stream hinzu	82
6	Änderungen Pflichtenheft	83
12	Glossar	84

1 Einleitung

Dieses Entwurfsheft beschreibt den Aufbau und Entwurf der Software IV Visualizer, welche im Rahmen des Moduls Praxis der Softwareentwicklung am Karlsruher Institut für Technologie im Wintersemester 2023/24 für das Fraunhofer-Institut für Optronik, Systemtechnik und Bildauswertung entwickelt wird.

Dabei wird Bezug genommen auf das Pflichtenheft und es werden Änderungen daran festgehalten.

Zudem werden in diesem Entwurfsheft Klassendiagramme und Sequenzdiagramme sowie das Architekturmuster genauer spezifiziert.

2 Architekturbeschreibung

Die Software des IV-Visualizers besteht aus einem Daten verwaltenden Modell, einer Kontrolleinheit und einer grafischen Benutzeroberfläche(GUI).

Dabei ist die Kontrolleinheit für das Entgegennehmen, Verarbeiten und Weiterleiten der Daten verantwortlich. Die Klassen der Kontrolleinheit sind im Subsystem Controller zusammengefasst.

Die Daten werden von der Kontrolleinheit an die Dateneinheit weitergeleitet. Die Dateneinheit kümmert sich um das Speichern der Daten und den Zugriff auf bereits abgelegte Daten. Die Klassen der Dateneinheit sind im Subsystem Model zusammengefasst.

Die GUI zeigt die Daten an und nimmt Benutzereingaben entgegen, um diese an die Kontrolleinheit weiterzuleiten und verarbeiten zu lassen. Die Klassen der GUI sind im Frontend zusammengefasst, um die Software erweiterbar, wartbar und skalierbar zu machen. Damit die Anliegen der Softwarekomponenten getrennt werden(Separation of Concerns), wird das Model-View-Controller Architekturmuster verwendet.

Nach der Erstellung aller Komponenten ist das Produkt einsatzbereit und die GUI nimmt erste Befehle des Nutzers entgegen. Wenn der Nutzer die Logs sehen will, stellt das „grafical interface“ eine Anfrage an den „log db connector“, um Log Daten zu bekommen. Der „log db connector“ baut dann eine Verbindung zur Loki Datenbank auf, holt die angefragten Daten und stellt sie dem „grafical interface“ zur Verfügung. Bei allen anderen Nutzereingaben wird die Anfrage an den „Backend connector“ weitergereicht. Dieser baut eine gRPC Verbindung zum Backend auf. Durch die gRPC Methodenaufrufe, die vom „gRPC api Frontend“ angenommen werden, werden vom „stream manager“ verschiedene Threads zur Anfrage der Daten erstellt und ausgeführt. Die Threads greifen im Model auf den „data manager“ zu, der dann entweder die Annotationsdaten aus der Datenbank, über den „database manager“, holt oder die Videodaten über den „file system manager“ aus dem Dateisystem holt. Der „file system manager“ wandelt Videodaten in Bilddaten um und sendet sie dem „stream manager“ im Controller. Dieser leitet sie wieder an das „grpc api Frontend“ weiter, um sie dem Frontend zur Verfügung zu stellen. Die Daten kommen beim „Backend connector“ im Frontend an und werden an das „grafical interface“ gesendet. Das „grafical interface“ schickt diese an die Pipeline, welche diese verarbeitet und zurückgibt. Nun können die Daten dem Benutzer angezeigt werden.

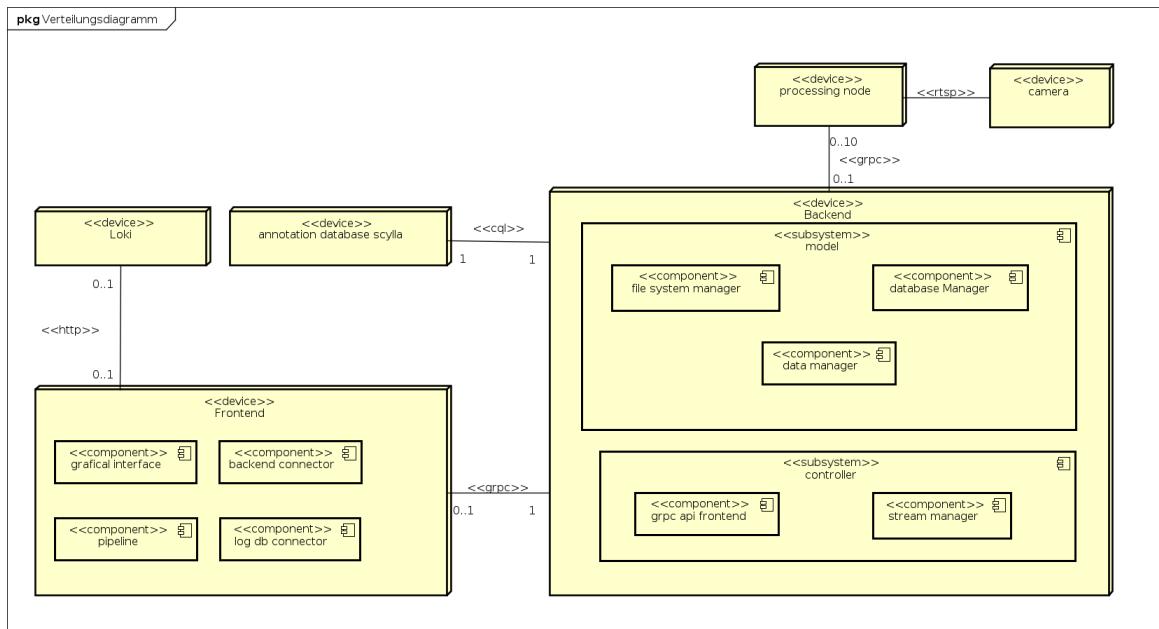


Abbildung 1: Verteilungsdiagramm

Abbildung 1 zeigt die Verteilung der Komponenten der Software und die grobe Struktur des Projekts auf Softwareebene. Während das Frontend den View Teil des Model-View-Controller-Modells repräsentiert, teilt sich das Backend das Model und den Controller auf. Es besteht eine niedrige Kopplung zwischen View und Model und erfüllt damit eine wichtige Eigenschaft des Model-View-Controller-Modells. Weil beim Entwurf großer Wert auf Erweiterbarkeit gelegt wurde, sind die viele Komponenten einfach austauschbar oder ergänzbar. So sind Komponenten, wie zum Beispiel die Pipeline, vollständig anpassbar und austauschbar.

Eine genaue Übersicht über alle Klassen findet man ab Kapitel 3.

2.1 gRPC

gRPC wird hier zur Kommunikation zwischen Backend und Processing Node sowie Frontend und Backend verwendet. Es sei erwähnt, dass zwei verschiedene Protodateien erstellt wurden, um beide Verbindungen zu realisieren.

Jeweils enthalten in den Protodateien sind:

- die Definition des gRPC-Services
- die Nachrichten zwischen Client und Server mit enthaltenen Attributen(in diesem Fall wie oben erwähnt in einer Datei zwischen Backend und Processing Node sowie Frontend und Backend in der anderen Datei)
- benötigte Enums

Der IvService wurde folgendermaßen in der Protodatei festgehalten:

RPC-Methoden zwischen Backend und Processing nodes

```
rpc Init(Empty) returns (ConfigResponseMessage)
Startet den Service
```

```
rpc Start(Empty) returns (stream DataStream)
Startet die Übertragung des Streams
```

```
rpc Stop(Empty) returns (InfoMessage)
Beendet den Stream und gibt eine Infonachricht zurück
```

```
rpc UpdateConfig(SetConfig) returns (ConfigResponseMessage)
Aktualisiert die Konfiguration von Container / GStreamer / Detector.
```

```
rpc GetConfig(Empty) returns (ConfigResponseMessage)
Fragt die aktuelle Konfiguration an
```

In der Protodatei zu der Verbindung zwischen Backend und Processing Node wurden folgende Nachrichten sowie Enums definiert:

Zunächst gibt es eine message Empty, die als Platzhalter fungieren soll.

Darüber hinaus gibt es die message ConfigContainer, die allgemeine Konfigurationseinstellungen enthält für die Pipeline. Sie enthält zwei boolesche Werte zur Wiedergabe von allgemeinen Informationen und einen int32 für die Qualität des Bildes.

Danach ist der message Point gegeben, welche einen zweidimensionalen Punkt definieren soll. Diese enthält zwei int32 für die Koordinaten.

Die darauffolgende message Roi definiert eine region of interest. Sie enthält einen String für den Namen,

einen int32 für die ID und einen RoiType, der den Typ definiert.

Die message RoiList ist eine Liste an Roi's.

Message ConfigDetector ist nun zur Spezifikation der Konfiguration des Detectors da. In ihr ist enthalten InferenceType, ein int32, zwei floats und eine RoiList.

Die Konfiguration für den GStreamer ist in der message ConfigGStreamer enthalten. Diese enthält einen String für Stream URI, einen int32 für die FPS und zweimal einen optional int32, jeweils für Breite sowie Höhe.

Modifikation wird ermöglicht durch die message setConfig. Diese enthält entweder den ConfigContainer, den ConfigDetector oder den ConfigGStreamer.

Durch die message ConfigResponseMessage ist die Rückgabe der aktuellen Konfiguration möglich. Sie enthält einen ConfigContainer, einen ConfigDetector und einen ConfigGStreamer.

Die message BoundingBoxData definiert eine Bounding Box innerhalb des Bildes. Sie enthält vier float Werte für Koordinanten, einen float Wert für die Confidence und einen String für das Label.

Zu der Versendung einer Liste von Bounding Boxen in einem einzelnen Bild gibt es die message BoundingBoxList. In ihr gibt es eine repeated BoudningBoxData.

Die message MetaData liefert zusätzliche Informationen zum Bild. Diese enthält zwei int32 für Breite und Höhe und einen uint64 für die Größe der Bilddatei.

Das Bild mit den dazugehörigen Metadaten kommt mit der message ImageData. Das image liegt in Form von bytes vor.

Die message ImageAndAnnotation enthält das Bild sowie einen uint64 timestamp und falls vorhanden noch eine Liste von Annotationen.

Zudem die message DataStream, die die Struktur des versendeten Streams festlegt.

Festgehaltene Enums sind InferenceType: ONNX_CPU, ONNX_GPU, TRT_FLOAT16, trt_model_handler, TRT_FLOAT32 und RoiType: INSIDE, OUTSIDE.

RPC-Methoden zwischen Frontend und Backend

```
rpc getImages(ImageRequest) returns (stream DataStream)
```

Gibt die Daten innerhalb des übergebenen Zeitraums zurück. Dabei werden nur die Daten des übergebenen Streams in Form eines Datenstreams zurückgegeben.

```
rpc getLiveImages(LiveImageRequest) returns (stream DataStream)
```

Gibt einen Datenstream zurück, der die Live Daten eines Streams enthält.

```
rpc startStream(StreamIdentifier) returns (InitialisationMessage)
```

Startet einen Stream im Backend und gibt die InitialisationMessage zurück.

```
rpc setConfig(Config) returns (ConfigResponseMessage)
```

Aktualisiert die Konfiguration eines Streams.

```
rpc getConfig(Empty) returns (ConfigResponseMessage)
```

Fragt die aktuelle Konfiguration eines Streams ab.

```
rpc setDBConfig(DBConfig) returns (ConfigDBResponseMessage)
```

Aktualisiert die Konfiguration des Models.

```
rpc getDBConfig() returns (ConfigDBResponseMessage)
```

Fragt die aktuelle Konfiguration des Models ab.

In der Protodatei zu der Verbindung zwischen Frontend und Backend wurden folgende Nachrichten sowie Enums definiert:

Es werden die gleichen Nachrichten und Enums definiert wie zwischen Backend und Processing Nodes. Hinzu kommen folgende Messages:

Die Message DBConfig besteht aus einer URL der Loki Datenbank, sowie zwei uint64. Einer steht für die Lebensdauer der Annotationen und der andere für die Lebensdauer der Videos.

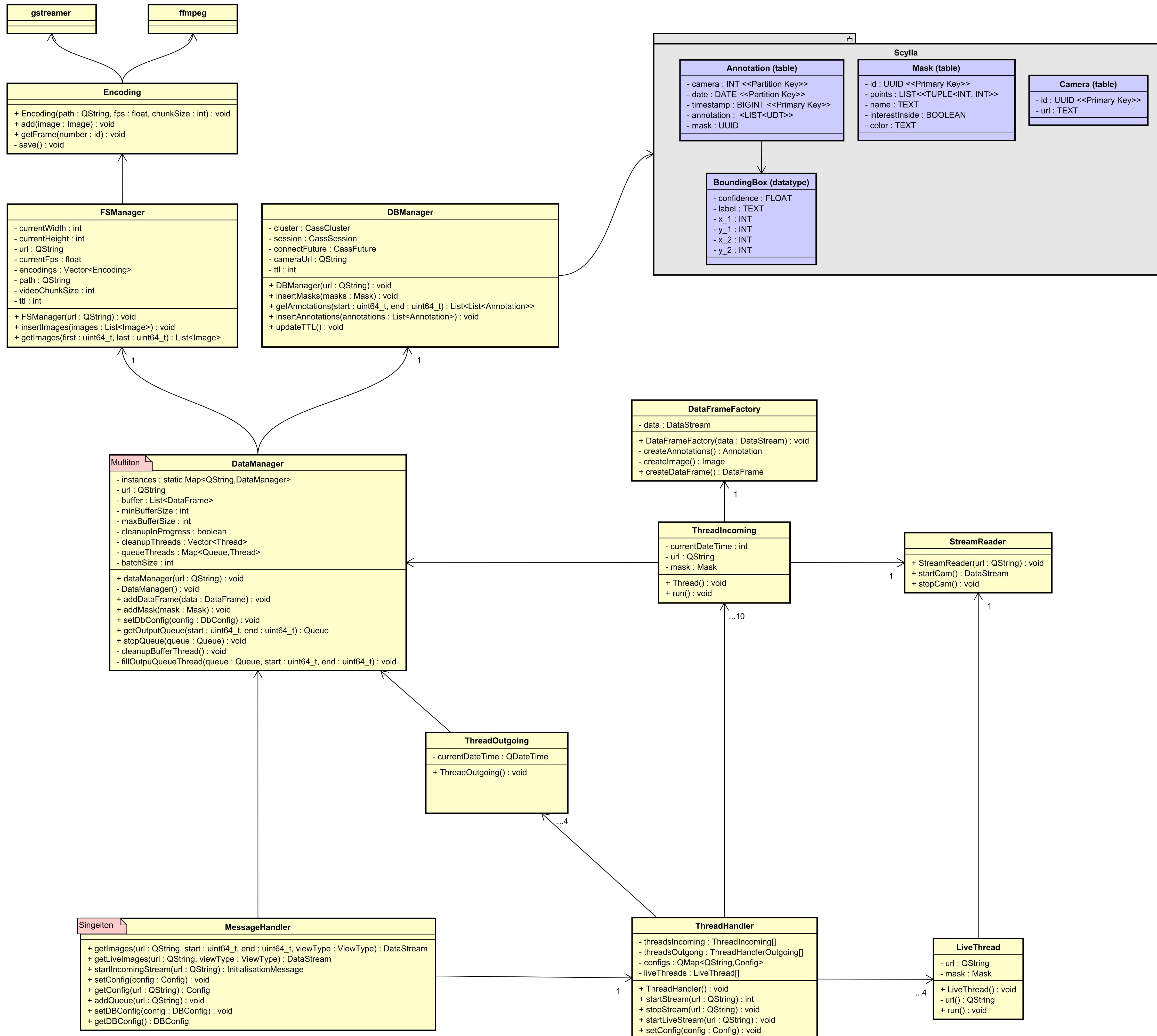
Die Message ImageRequest besteht aus zwei uint64, welche die Zeit beschreiben, in dem die Daten abgerufen werden sollen. Zudem ist ein StreamIdentifier enthalten, welcher den Stream in Form eines Strings identifiziert. Außerdem ist ein viewType enthalten, welcher bestimmt, in welcher Form die Daten geschickt werden sollen.

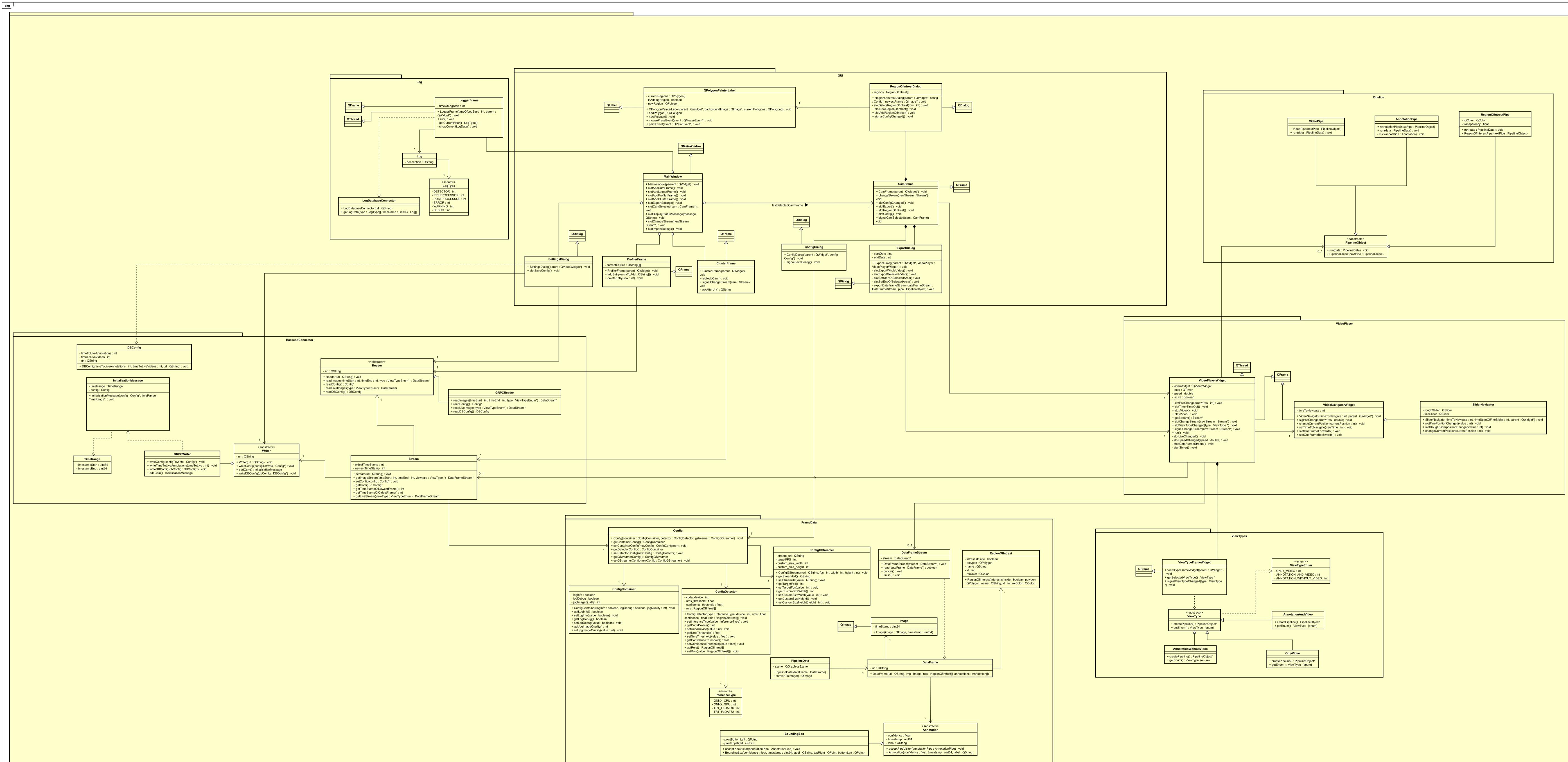
Die Message LiveImageRequest beinhaltet die gleichen Daten, wie ImageRequest, nur ohne die zeitliche Komponente.

Die Message InitialisationMessage beinhaltet den Zeitraum, in dem Daten von diesem Stream vorhanden sind, sowie die bisherige Konfiguration des Streams.

Zudem wurde der Enum viewType hinzugefügt, mit folgenden Werten: ONLY_VIDEO, ANNOTATION_AND_VIDEO und ONLY_ANNOTATION.

Anmerkung: Aus den folgenden Klassenbeschreibungen wird nicht ersichtlich sein, dass alle Klassen, die Signale und Slots verwenden, auch die Klasse QObject erweitern. Diese Information wurde aus formellen Gründen ausgelassen, um zu starke Repetitivität zu vermeiden. Zudem werden Qt-Klassen im Klassendiagramm aus Übersichtlichkeitsgründen öfter hinzugefügt. Des Weiteren werden nicht alle 'Use-Beziehungen' im Klassendiagramm aus dargestellt.





3 Model

3.1 Entwurf

Im Folgenden wird das Paket *Model*, ein Teil des Model-View-Controllers im genaueren beschrieben. Die Objekte, die gespeichert werden müssen, bestehen aus Bild-Daten und Koordinaten. Um die Datenbank nicht zu überlasten, wurde eine Aufteilung in Datenbank und Dateisystem geplant. Diese Aufteilung wird durch die Klasse *DataManager* versteckt. Des Weiteren enthält sie einen Buffer, um neue Daten mit geringer Verzögerung einzufügen und ohne Laufwerkzugriff schnell auslesen zu können.

3.2 Klassen

3.2.1 DataManager

Multiton	DataManager
	<ul style="list-style-type: none"> - instances : static Map<QString, DataManager> - url : QString - buffer : List<DataFrame> - minBufferSize : int - maxBufferSize : int - cleanupInProgress : boolean - cleanupThreads : Vector<Thread> - queueThreads : Map<Queue, Thread> - batchSize : int
	<ul style="list-style-type: none"> + dataManager(url : QString) : void - DataManager() : void + addDataFrame(data : DataFrame) : void + addMask(mask : Mask) : void + setDbConfig(config : DbConfig) : void + getOutputQueue(start : uint64_t, end : uint64_t) : Queue + stopQueue(queue : Queue) : void - cleanupBufferThread() : void - fillOutputQueueThread(queue : Queue, start : uint64_t, end : uint64_t) : void

Abbildung 2: DataManager.

Klassenbeschreibung

Die Klasse bietet eine Speicherschnittstelle für die *DataFrame* Objekte.

Dabei versteckt sie die Aufteilung auf Dateisystem und Datenbank, sowie die Umwandlung in Videodateien und Tabelleneinträge.

Attribute

- **instances** Speichert bestehende Instanzen des Multitons
- **url** Key für das Multiton, entspricht der Kamera ID
- **buffer** Buffer, damit *addDataFrame* nicht blockierend ist
- **batchSize** Die Anzahl an Elementen, die beim Persitieren auf einmal entnommen werden soll

Konstruktoren

dataManager(url : QString)

Erstellen des Multitons

Methoden

`addDataFrame(data : DataFrame)`

Fügt ein DataFrame zur Speicherung hinzu. (Nicht blockierend)

`addMask(mask : Mask) : void`

Fügt eine Maske zur Speicherung hinzu

`setDbConfig(config : DbConfig) : void`

Leitet die darin enthaltenen Einstellungen wie TTL an die entsprechenden Stellen weiter

`getOutputQueue(start : uint64_t, end : uint64_t) : Queue`

Gibt sofort eine Queue zurück, die mit DataFrame Objekten aus dem gewählten Zeitraum aufgefüllt wird

`stopQueue(queue : Queue) : void`

Unterbricht die Füllung der angegebenen Queue und zerstört das Objekt

`cleanupBufferThread() : void`

Thread, der Objekte aus dem buffer entfernt und abspeichert, wenn dieser zu groß wird

`fillOutputQueueThread(queue : Queue, start : uint64_t, end :`

`uint64_t) : void`

Thread, der Objekte aus dem Buffer oder dem Dateisystem bzw. der Datenbank holt und diese in die angefragte Queue legt.

3.2.2 DBManager

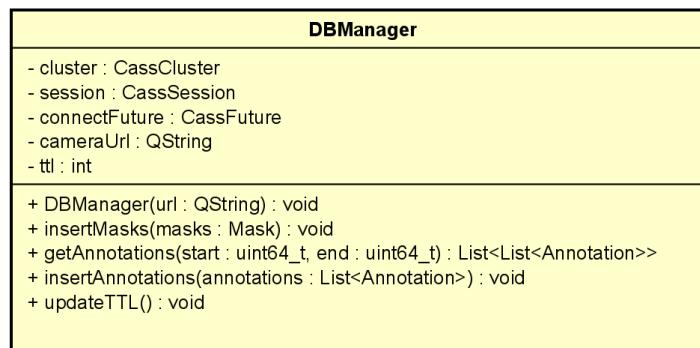


Abbildung 3: DBManager.

Klassenbeschreibung

Die Klasse stellt die Verbindung zur Datenbank dar und speichert objektorientierte Objekte in einer effizienten Weise.

Die Speicherung erfolgt auf einen Methodenaufruf und versteckt die Umwandlung der Objekte.

Attribute

- **ttl** Time to Live der Einträge, bevor sie automatisch vom DBMS entfernt werden

Konstruktoren

`DBManager(url : QString)`

DBManager Instanz mit neuer Verbindung zur Speicherung der Daten zur zugehörigen Kamera url

Methoden

- **insertAnnotations(annotations : List<Annotation>) : void** Bulk Insert aller gegebenen Annotations zur zugehörigen Kamera
- **insertMasks(masks : Mask) : void** Speichert eine neue Maskierung ab, falls sie noch nicht existiert
- **getAnnotations(start : uint64_t, end : uint64_t) : List<List<Annotation>>** Für jeden Frame zwischen den timestamps wird eine Liste mit allen Annotationen der entsprechenden Frames zurück gegeben
- **updateTTL** Weist jeder Annotation in der Datenbank eine neue TTL zu

3.2.3 FSManager

FSManager
- currentWidth : int - currentHeight : int - url : QString - currentFps : float - encodings : Vector<Encoding> - path : QString - videoChunkSize : int - ttl : int
+ FSManager(url : QString) : void + insertImages(images : List<Image>) : void + getImages(first : uint64_t, last : uint64_t) : List<Image>

Abbildung 4: FSManager.

Klassenbeschreibung

Die Klasse übernimmt das Speichern, Abfragen und Verwalten der Bilder auf dem Laufwerk.

Attribute

- **path** Speicherort der Dateien
- **videoChunkSize** Länge der Videos in Frames und Größe des Buffers für die Bilder auf dem Laufwerk

Konstruktoren

```
FSManager(url : QString)
```

Erstellen eines Speicherorts und Buffer

Methoden

- **insertImages(images : List<Image>) : void** Speichert die übergebenen Bilder ab
- **getImages(first : uint64_t, last : uint64_t) : Image** Gibt alle Bilder aus dem zugehörigen Zeitraum sortiert zurück

3.2.4 Encoding

Encoding
+ Encoding(path : QString, fps : float, chunkSize : int) : void + add(image : Image) : void + getFrame(number : id) : void - save() : void

Abbildung 5: Encoder.

Klassenbeschreibung

Konvertierung zwischen Video- und Bilddateien. Fassade, um existierende Bibliotheken für diesen Anwendungszweck dynamisch auszuwählen und nicht unterscheiden zu müssen, ob sich das Objekt im Arbeitsspeicher oder auf der Festplatte befindet.

Attribute

- -

Konstruktoren

```
Encoding( path : QString , fps : float , chunkSize : int ) : void
```

Erstellt ein Objekt, das zur Konvertierung und Speicherung von Bildern genutzt werden kann

Methoden

```
add(image : Image) : void
```

Fügt das Bild der Speicherung zum aktuellen Videoausschnitt hinzu. Möglichst komprimiert mit der gegebenen fps Rate

```
getFrame(number : id) : void
```

Liest einen einzelnen Frame aus dem aufgerufenem Encoding Videoausschnitt aus

3.3 Datenspeicherung

3.3.1 Scylla

Im Rahmen des Entwurfsprozesses wurde die Entscheidung getroffen, Scylla als Datenbanktechnologie zu verwenden. Diese Entscheidung basiert auf verschiedenen Kriterien und Anforderungen, die die Auswahl von Scylla als die optimale Lösung begründen.

Skalierbarkeit: Scylla wurde gewählt, da es die erforderliche Skalierbarkeit bietet, um mit den steigenden Anforderungen unseres Projekts Schritt zu halten. Diese Skalierbarkeit ermöglicht es, Ressourcen flexibel hinzuzufügen, um die Leistung der Datenbank nach Bedarf zu steigern, ohne Engpässe zu riskieren.

Flexibles Datenmodell: Ein wesentlicher Grund für die Auswahl von Scylla ist die Möglichkeit, Datentypen zu erstellen und Listen beliebiger Größe zu speichern, die zu einem Eintrag gehören. Diese Funktion erlaubt es, große und variable Datenmengen mit nur einer Anfrage effizient zu speichern, was die Handhabung komplexer Anforderungen erleichtert.

Leicht erweiterbare Datentypen: Scylla bietet die Möglichkeit, Datentypen einfach zu erweitern, wodurch Anpassungen an zukünftige Anforderungen ohne umfangreiche Änderungen an der bestehenden Datenbankstruktur vorgenommen werden können. Diese Flexibilität ermöglicht eine agile Entwicklung und schnelle Reaktion auf neue Anforderungen.

3.3.1.1 Tabellen

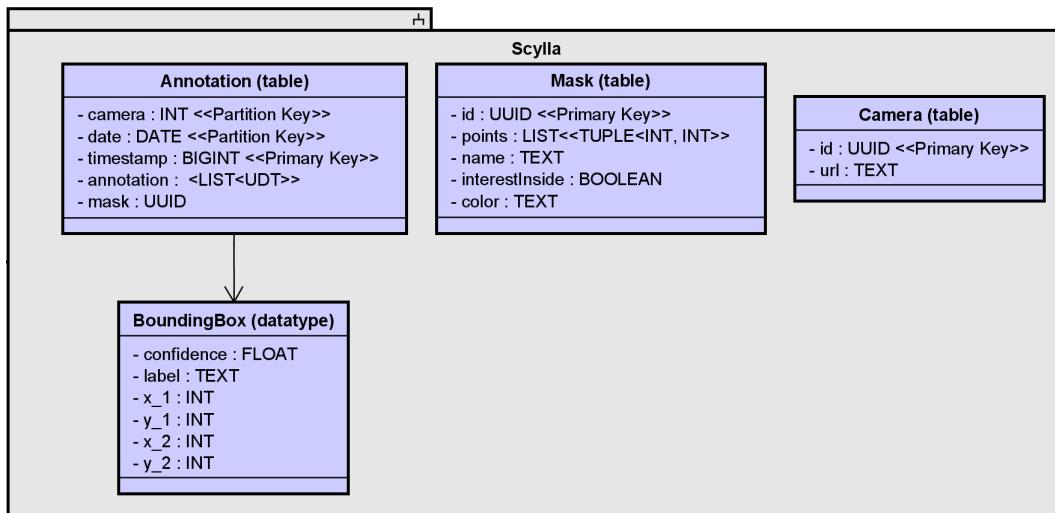


Abbildung 6: FSManager.

Scylla speichert Daten in Partitionen, die jeweils auf einer eigenen Node liegen können. Beim Design wurde die Entscheidung getroffen, dass jede Kamera für jeden Tag ihre eigene Partition erhält. Das hat den Hintergrund, dass Partitionen nicht zu groß werden dürfen.

Masken oder auch ReginOfInterests werden nur dann neu abgespeichert, wenn sie sich auch geändert haben. Dazu liegen sie in einer eigenen Tabelle, müssen dafür aber im Backend mit Annotations *gejoint* werden.

3.4 Ablaufbeschreibungen

addDataFrame

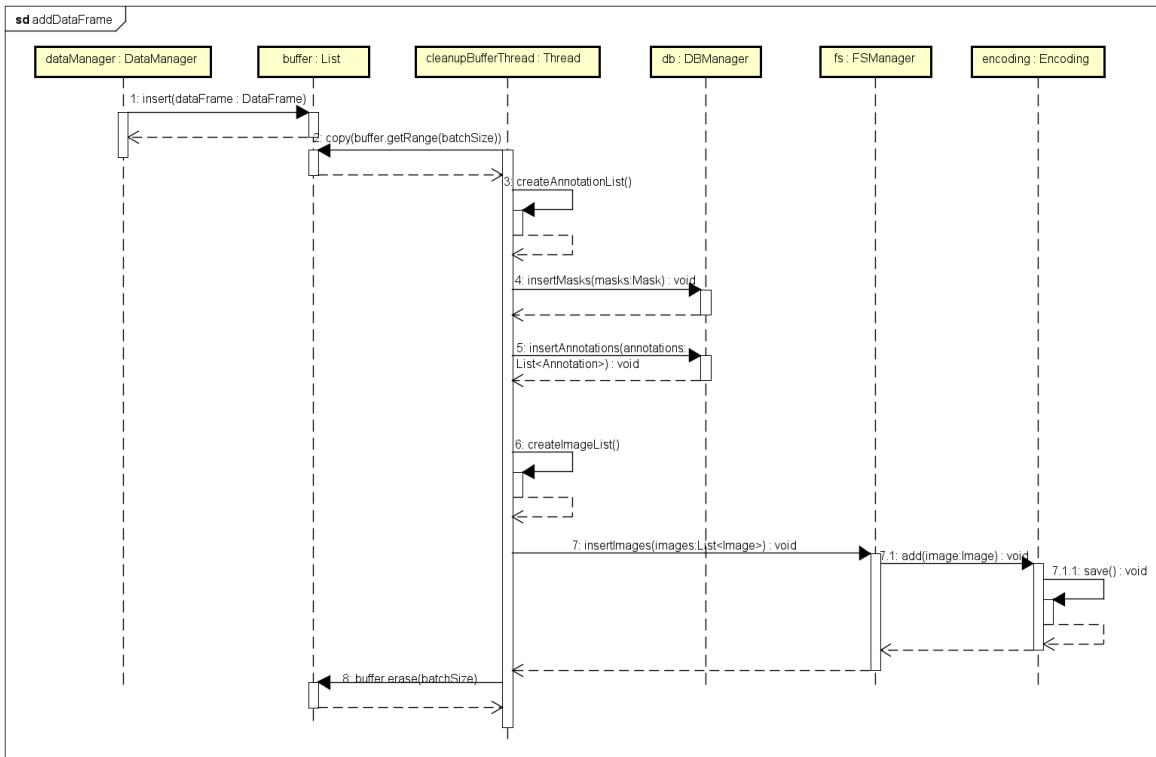


Abbildung 7: Sequenzdiagramm zum Speichern von DataFrames

Um einen Aufruf von `addDataFrame` auf `DataManager` zu gewährleisten, wird das Objekt direkt in einen Buffer geschrieben, das die Methode zurückgibt.

Sobald die `maxBufferSize` überschritten wurde, erwacht der `cleanupBufferThread` und kopiert `batchSize` Elemente.

Diese Kopie wird in weitere Listen zerlegt, die in den `DBManager` oder dem `FSManager` gegeben und dort abgearbeitet werden.

In `encoding` wird ein Buffer im H.264 Format gefüllt und nach erreichen von `videoChunkSize` auf die Festplatte geschrieben.

getOutputQueue

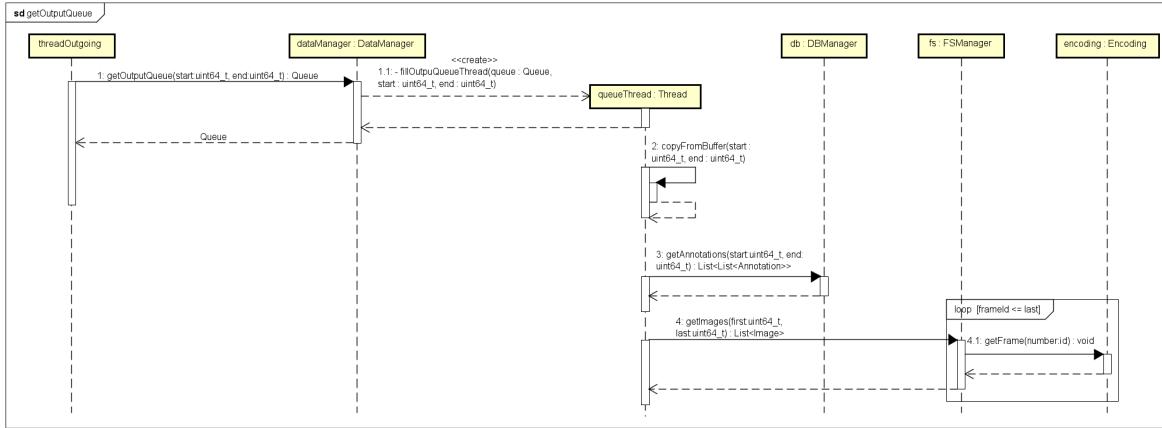


Abbildung 8: Sequenzdiagramm zum laden bereits eingefügter DataFrames

Zum Laden bereits gespeicherter *DataFrame* Objekte wird auf *DataManager* die Funktion *getOutputFrame* aufgerufen. Diese gibt sofort eine leere *Queue* zurück. Danach wird ein *queueThread* dazu angewiesen, die verfügbaren *DataFrames* aus dem Buffer in die *Queue* zu kopieren. Werden ältere Daten benötigt, werden diese aus *DBManager* und *FSManager* abgerufen.

Encoding prüft, ob die Daten auf der Festplatte oder dem Arbeitsspeicher liegen und extrahiert die Bilder daraus.

4 Controller

4.1 Entwurf

In diesem Abschnitt wird der Controller, der die Kommunikation zwischen den Processing Tools, der GUI, und der Datenbank darstellt, genauer beschrieben. Im Controller findet die Abarbeitungen der Nachrichten aus der GUI, sowie die Verarbeitung der Streams statt.

4.2 Klassen

4.2.1 MessageHandler

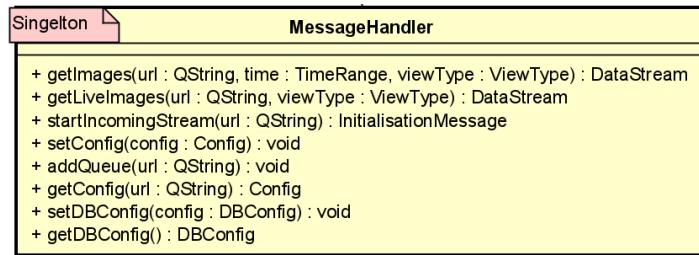


Abbildung 9: MessageHandler

Klassenbeschreibung

Der MessageHandler ist die Schnittstelle für einkommende Messages der GUI. Bei einer Anfrage aus der GUI wird über requestData() im DataManager die Füllung der entsprechenden Queues initiiert. Falls ein komplett neuer Stream hinzugefügt wird, wird eine neue Queue dafür angelegt. Bei Aufruf von getImages wird über gRPC ein neuer Thread erstellt, der die Bilder aus der Datenbank holt, und dann in den Stream zurückspielt, Bild für Bild.

Attribute

- -

Konstruktoren

`MessageHandler()`

Instantiiert den MessageHandler.

Methoden

```

DataStream getImages(const QString& url, const TimeRange& time,
const ViewType viewType)
Bei Aufruf von getImages wird über gRPC ein neuer Thread erstellt, der die Bilder
aus der Datenbank holt, und dann in den Stream zurückspielt, Bild für Bild.

```

```
DataStream getLiveImages(const QString& url, const ViewType viewType)
```

Bei Aufruf von `getLiveImages` wird über gRPC ein neuer Thread erstellt, der die Bilder aus der Datenbank holt, und dann in den Stream zurückspielt, Bild für Bild. Dabei sind es diesmal live Daten.

```
QDateTime startIncomingStream(const QString& url)
```

Durch diese Methode kann der Stream einer neuen Kamera gestartet werden.

```
void setConfig(const Config& config)
```

Über diese Methode lässt sich die Konfiguration der ProcessingTools anpassen. Durch die in der Config enthaltene URL wird bestimmt, wo die Konfiguration gesetzt wird.

```
Config getConfig(const QString& url)
```

Über diese Methode lässt sich die Konfiguration der ProcessingTools holen. Durch die übergebene URL wird bestimmt, wo die Konfiguration geholt wird.

```
void addQueue(const QString& url)
```

Fügt eine neue Queue hinzu, sobald ein neuer Stream hinzugefügt wird.

```
DBConfig getDBConfig()
```

Über diese Methode lässt sich die Konfiguration des Models holen.

```
void setDBConfig(const DBConfig& config)
```

Über diese Methode lässt sich die Konfiguration des Models setzen.

4.2.2 ThreadHandler

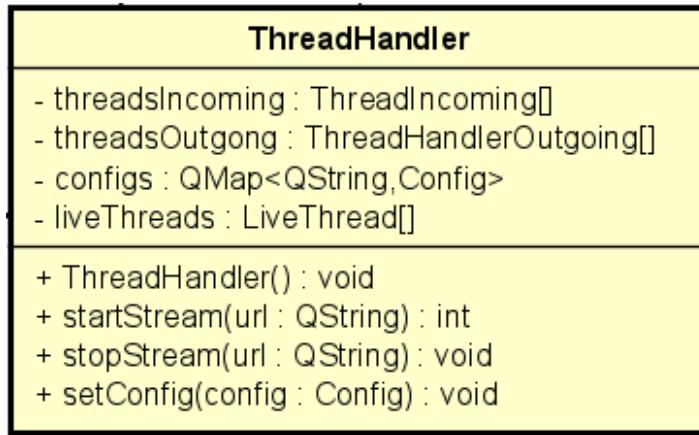


Abbildung 10: ThreadHandler

Klassenbeschreibung

Der ThreadHandler erstellt neue Threads um die Datenströme in und aus der Datenbank zu handhaben. Er reagiert direkt auf Aufrufe vom MessageHandler.

Attribute

- **threadsIncoming** Liste der ThreadsIncoming
- **threadsOutgoing** Liste der ThreadsOutgoing
- **threadsLive** Liste der LiveThreads
- **configs** Mappt die Konfigurationen zu den URLs, um auf getConfig Anfragen der GUI reagieren zu können

Konstruktoren

`ThreadHandler()`

Instantiiert den ThreadHandler.

Methoden

`void startStream(const QString& url)`

Diese Methode wird über die GUI Messages aufgerufen, um einen Stream zu starten.
Per URL wird eine bestimmte Kamera ausgewählt.

`void stopStream(const QString& url)`

Durch diese Methode kann der Stream einer bestimmten Kamera wieder beendet werden.

`void setConfig(Config& config)`

Durch diese Methode wird eine updateConfigMessage zu den ProcessingTools geschickt. Anschliessend wird zu dieser Kamera ein neuer Stream in die Datenbank erstellt.

4.2.3 ThreadIncoming

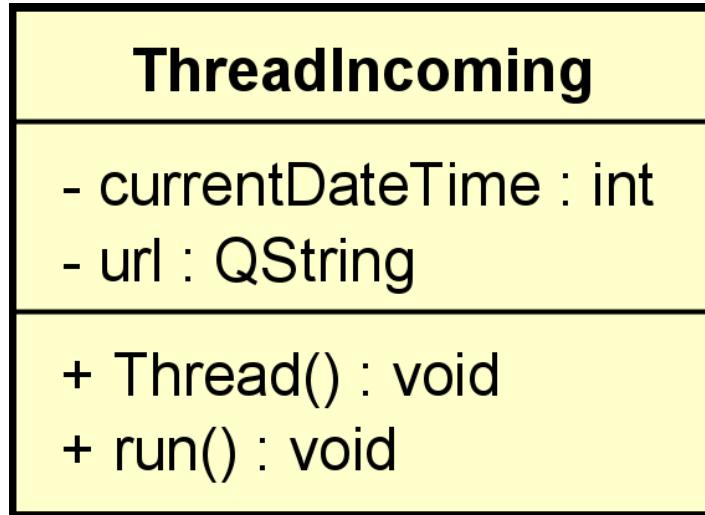


Abbildung 11: ThreadIncoming.

Klassenbeschreibung

Die ThreadIncoming Threads arbeiten die einzelnen Kamerastreams ab. Jeder Thread hat einen Stream-Reader, eine DataFrameFactory und eine Referenz zum DataManager, wo die Threads die Bilder in einer Queue pro Thread ablegen.

Attribute

- **currentDateTime** aktueller Zeitstempel
- **url** url des Streams

Konstruktoren

`ThreadData()`

Instantiiert ein ThreadData Objekt.

Methoden

`void run()`
Startet den Thread.

4.2.4 ThreadOutgoing

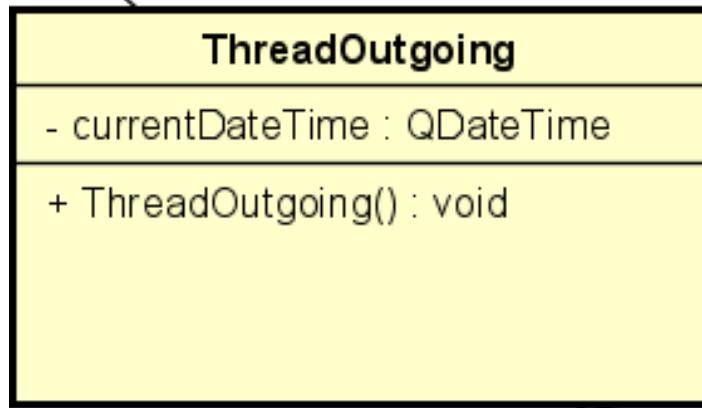


Abbildung 12: ThreadOutgoing.

Klassenbeschreibung

Die ThreadOutgoing Threads leiten die Bilder von der Datenbank zur GUI weiter.

Attribute

- **currentDateTime** Aktueller Zeitstempel

Konstruktoren

`ThreadOutgoing()`

Instantiiert ein ThreadData Objekt.

Methoden

`void run()`
Startet den Thread.

4.2.5 StreamReader

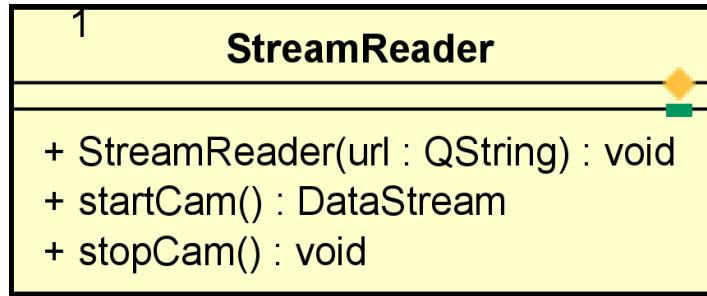


Abbildung 13: StreamReader.

Klassenbeschreibung

StreamReader kümmert sich um die Abhandlung des Kamerastreams pro Thread, also das Starten und Stoppen.

Attribute

- -

Konstruktoren

`StreamReader(const QString& str)`

Instantiierte einen StreamReader und übergibt ihm die Referenzen.

Methoden

`void startCam()`

Startet den Stream des Kamerabildes.

`void stopCam()`

Stoppt den Stream des Kamerabildes.

4.2.6 DataFrameFactory

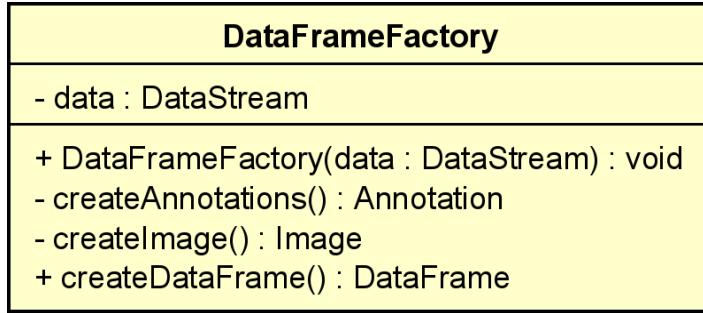


Abbildung 14: DataFrameFactory.

Klassenbeschreibung

DataFrameFactory erstellt neue DataFrame Objekte aus den einkommenden DataStream Messages

Attribute

- **data** Aktueller Datastream

Konuktoren

`DataStreamFactory (const DataStream& data)`

DataFrameFactory erstellt neue DataFrame Objekte aus den einkommenden DataStream Messages.

Methoden

`const DataFrame& createDataFrame()`

Erstellt ein DataFrame Objekt bestehend aus Image und Annotations.

`std::vector<Annotation> createAnnotations()`

Erstellt Annotation-Objekte aus den empfangenen Messages.

`std::vector<Image> createImage()`

Erstellt Image-Objekte aus den empfangenen Messages.

4.2.7 TimeRange

Klassenbeschreibung

Die Klasse TimeRange stellt einen Zeitraum, begrenzt mit einer Start- und Endzeit, dar.

Attribute

- **startTime** Startzeit des Streams
- **endTime** Endzeit des Streams

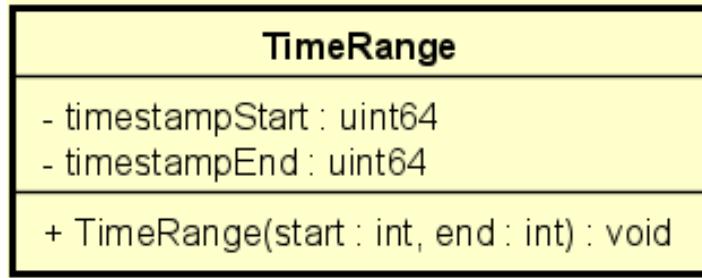


Abbildung 15: TimeRange

Konstruktoren

```
TimeRange(const uint64 start, const uint64 end)
```

Initiiert eine TimeRange und übergibt ihr die Referenzen.

Methoden

```
uint64 getStartTime() const  
Gibt die Startzeit der Zeitspanne zurück.
```

```
uint64 getEndTime() const  
Gibt die Endzeit der Zeitspanne zurück.
```

4.2.8 LiveThread

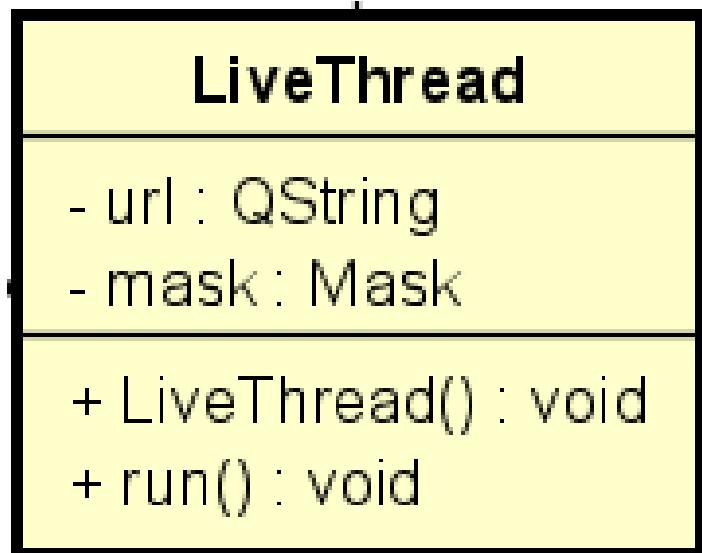


Abbildung 16: LiveThread

Klassenbeschreibung

LiveThreads kümmern sich um den Fall, dass der Kamerastream live zur GUI übertragen werden soll und leiten den Stream sofort weiter.

Attribute

- **url** identifiziert die Kamera
- **mask**

Konstruktoren

`LiveThread()`

Initiiert einen LiveThread.

Methoden

`void run()`
Startet den Thread.

4.2.9 Queue

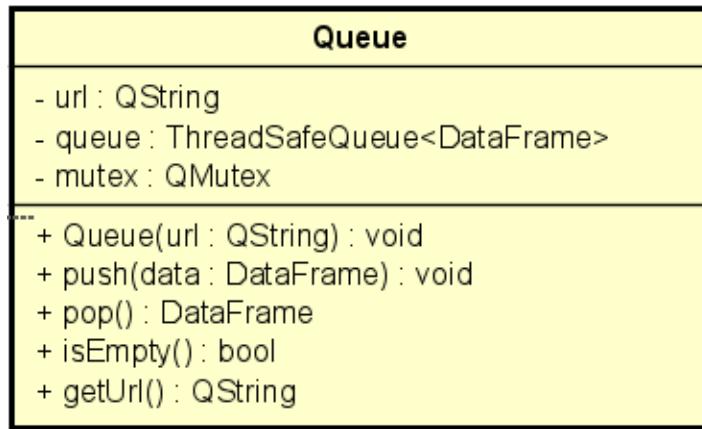


Abbildung 17: Queue

Klassenbeschreibung

Die Klasse Queue stellt eine Threadsichere Queue dar, die mittels URL zu einer Kamera zugeordnet werden kann.

Attribute

- **url** identifiziert die Kamera
- **queue**
- **mutex**

Konstruktoren

`Queue(QString& url)`

Initiiert eine Queue.

Methoden

`void push(const DataFrame& frame)`

Fügt ein DataFrame Objekt zur Queue hinzu.

`DataFrame& pop()`

Entfernt ein DataFrame Objekt aus der Queue.

`bool isEmpty()`

Gibt Auskunft darüber, ob die Queue leer ist.

`QString getUrl()`

Gibt die URL zurück.

4.2.10 Config

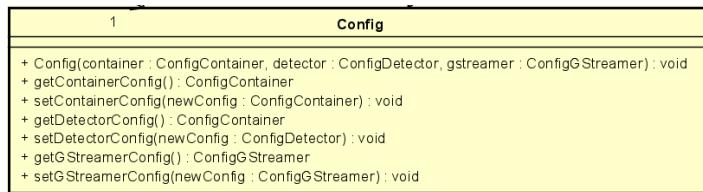


Abbildung 18: Queue

Klassenbeschreibung

Config ist die Oberklasse für weitere Config Objekte. Hier werden Konfigurationen, wie beispielsweise Einstellungen der Kameras, dargestellt.

Attribute

- Referenz zu **containerConfig**
- Referenz zu **gStreamerConfig**
- Referenz zu **detectorConfig**

Konstruktoren

`Config(const ConfigContainer& container, const ConfigDetector& detector, const ConfigGStreamer& gstreamer)`

Initiiert ein Config Objekt und übergibt ihm die Referenzen.

Methoden

```
ConfigContainer getContainerConfig() const
Gibt die Container Konfiguration der jeweiligen Gesamtkonfiguration zurück.
```

```
void setContainerConfig(const ConfigContainer& newConfig
Aktualisiert die Container Konfiguration.
```

```
ConfigDetector getDetectorConfig() const
Gibt die Container Konfiguration der jeweiligen Gesamtkonfiguration zurück.
```

```
void setDetectorConfig(const ConfigDetector& newConfig
Aktualisiert die Detector Konfiguration.
```

```
ConfigGStreamer getGStreamerConfig() const
Gibt die GStreamer Konfiguration zurück.
```

```
void setGStreamerConfig(const ConfigGStreamer& newConfig
Aktualisiert die GStreamer Konfiguration.
```

4.2.11 ConfigGStreamer

ConfigGStreamer	
- stream_url : QString	
- targetFPS : int	
- custom_size_width : int	
- custom_size_height : int	
+ ConfigGStreamer(url : QString, fps : int, width : int, height : int) : void	
+ getStreamUrl() : QString	
+ setStreamUrl(value : QString) : void	
+ getTargetFps() : int	
+ setTargetFps(value : int) : void	
+ getCustomSizeWidth() : int	
+ setCustomSizeWidth(value : int) : void	
+ getCustomSizeHeight() : int	
+ setCustomSizeHeight(height : int) : void	

Abbildung 19: Queue

Klassenbeschreibung

Die Klasse ConfigGStreamer stellt die Konfiguration eines GStreamers dar.

Attribute

- **streamURL** String in Form von: rtsp://user:pw@ip:port/media
- **targetFPS** Ziel fps des Streams
- **customSizeWidth** Breite des gestreamten Kamerabilds

- **customSizeHeight** Höhe des gestreamten Kamerabilds

Konstruktoren

```
ConfigGStreamer(const std::string& url, int fps, int width, int height)
```

Initiiert ein ConfigGStreamer Objekt und übergibt die Referenzen.

Methoden

```
std::string getStreamUrl() const  
Gibt die Stream-URL zurück.
```

```
void setStreamUrl(const std::string& value)  
Setzt die Stream-URL auf den angegebenen Wert.
```

```
int getTargetFps() const  
Gibt die Ziel-FPS (Bilder pro Sekunde) zurück.
```

```
void setTargetFps(int value)  
Setzt die Ziel-FPS auf den angegebenen Wert.
```

```
int getCustomSizeWidth() const  
Gibt die Breite der benutzerdefinierten Auflösung zurück. -1 bedeutet Standardauflösung.
```

```
void setCustomSizeWidth(int value)  
Setzt die Breite der benutzerdefinierten Auflösung auf den angegebenen Wert. -1 bedeutet Standardauflösung.
```

```
int getCustomSizeHeight() const  
Gibt die Höhe der benutzerdefinierten Auflösung zurück. -1 bedeutet Standardauflösung.
```

```
void setCustomSizeHeight(int value)  
Setzt die Höhe der benutzerdefinierten Auflösung auf den angegebenen Wert. -1 bedeutet Standardauflösung.
```

4.2.12 ConfigContainer

Klassenbeschreibung

Die Klasse ConfigContainer stellt die Konfiguration eines Containers dar.

Attribute

- **logInfo** Informationen Logging
- **logDebug** Debug Logging
- **JPGImageQuality** Bildqualität des JPGs

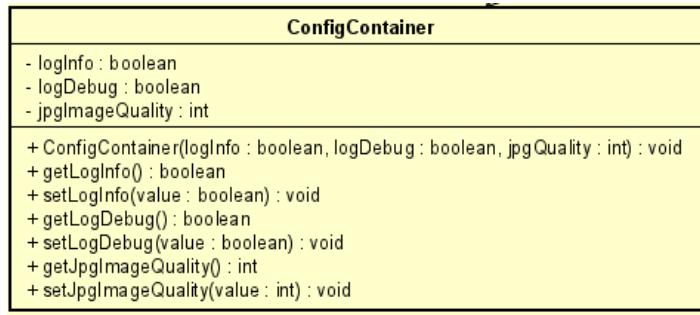


Abbildung 20: ConfigContainer

Konstruktoren

```
ConfigContainer(bool logInfo, bool logDebug, int jpgQuality)
```

Initiiert ein ConfigContainer Objekt und übergibt die Referenzen

Methoden

```
bool getLogInfo() const
```

Gibt den Wert des Log Info Flags zurück.

```
void setLogInfo(bool value)
```

Setzt den Wert des Log Info Flags auf den angegebenen Wert.

```
bool getLogDebug() const
```

Gibt den Wert des Log Debug Flags zurück.

```
void setLogDebug(bool value)
```

Setzt den Wert des Log Debug Flags auf den angegebenen Wert.

```
int getJpgImageQuality() const
```

Gibt die JPG-Bildqualität zurück. Der Wertebereich liegt zwischen 0 und 100 oder -1.

```
void setJpgImageQuality(int value)
```

Setzt die JPG-Bildqualität auf den angegebenen Wert. Der Wertebereich liegt zwischen 0 und 100 oder -1.

4.2.13 ConfigDetector

Klassenbeschreibung

Die Klasse ConfigDetector stellt die Konfiguration eines Detectors dar.

Attribute

- **InferenceType** Art der Inferenz

- **cudaDevice**

- **nmsThreshold**
- **confidenceThreshold** Threshold zur Erkennung
- **ROIs** Liste an Region Of Interests

Konstruktoren

```
ConfigDetector(InferenceType type, int device, float nms, float  
confidence, const RoiList& list)
```

Initiiert ein ConfigDetector Objekt und übergibt die Referenzen.

Methoden

```
InferenceType getInferenceType() const  
Gibt den Typ der Inferenz zurück, z.B. ONNX_CPU, ONNX_GPU, TRT_FLOAT16  
oder TRT_FLOAT32.
```

```
void setInferenceType(InferenceType value)  
Setzt den Typ der Inferenz auf den angegebenen Wert.
```

```
int getCudaDevice() const  
Gibt die CUDA-Gerätenummer zurück, die für die Inferenz verwendet wird.
```

```
void setCudaDevice(int value)  
Setzt die CUDA-Gerätenummer für die Inferenz auf den angegebenen Wert.
```

```
float getNmsThreshold() const  
Gibt den Schwellenwert für die Nicht-Maximum-Unterdrückung (NMS) zurück.
```

```
void setNmsThreshold(float value)  
Setzt den Schwellenwert für die Nicht-Maximum-Unterdrückung (NMS) auf den  
angegebenen Wert.
```

```
float getConfidenceThreshold() const  
Gibt den Schwellenwert für die Konfidenz zurück.
```

```
void setConfidenceThreshold(float value)  
Setzt den Schwellenwert für die Konfidenz auf den angegebenen Wert.
```

```
RoiList getRois() const  
Gibt die Liste der Regionen von Interesse (ROIs) zurück.
```

```
void setRois(const RoiList& value)  
Setzt die Liste der Regionen von Interesse (ROIs) auf die angegebene Liste.
```

4.3 Ablaufbeschreibungen

4.3.1 Bilder aus dem Backend Anfragen

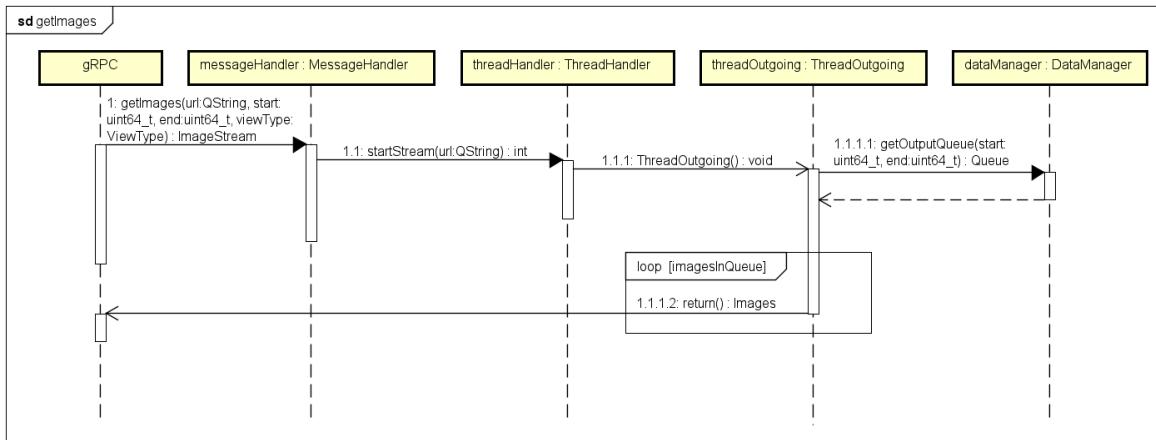


Abbildung 21: Sequenzdiagramm Bilder holen

Auf dem `MessageHandler` wird von der GUI per `gRPC` die Methode `getImages` aufgerufen. Im `ThreadHandler` wird die Methode `startStream` mit der entsprechenden URL aufgerufen. Der `ThreadHandler` erstellt einen neuen `ThreadOutgoing`, der sich die `DataFrame` Objekte aus der Datenbank holt und zurück in den `gRPC` Stream zur GUI spielt.

4.3.2 LiveStream

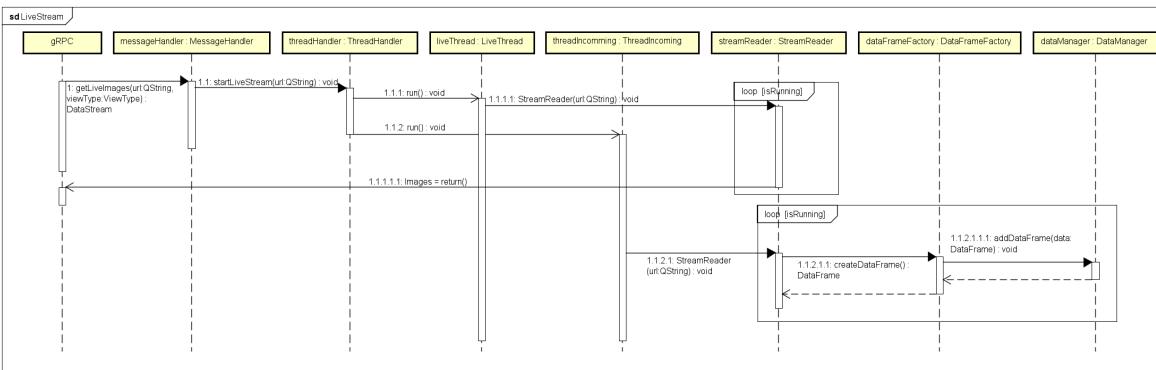


Abbildung 22: Sequenzdiagramm Livestream

Auf dem `MessageHandler` wird von der GUI über `gRPC` die `getLiveImages` Methode aufgerufen, mit der passenden URL. `MessageHandler` ruft auf `ThreadHandler` `startLiveStream` Methode auf in `startLiveStream` werden zwei Threads erstellt und gestartet, ein `ThreadIncoming` und einen `LiveThread`. Der `LiveThread` stellt über den `StreamReader` die `gRPC` Verbindung her und gibt die Bilder direkt an den `gRPC` Stream zurück. Der `ThreadIncoming` stellt ebenfalls die Verbindung über den `StreamReader` zur angefragten URL her, erstellt dann aber zuerst über seine `DataFrameFactory` aus den Stream Messages `DataFrame` Objekte, die dann in den Buffer des `DataManagers` gelegt werden.

4.3.3 Starten eines Streams

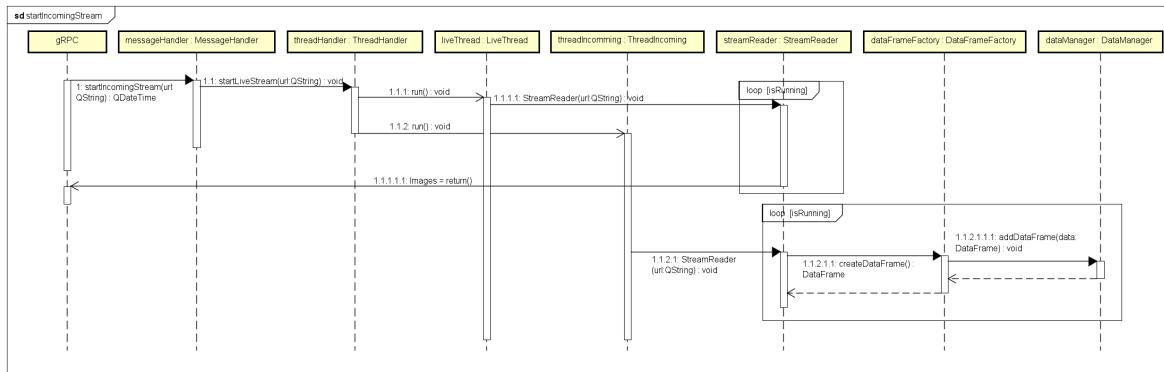


Abbildung 23: Sequenzdiagramm starten des Streams

Dieses Szenario beschreibt den Fall, in dem von der GUI zwar eine Kamera hinzugefügt wird, jedoch kein Bild in der GUI zu der Kamera angezeigt wird.

Über den MessageHandler wird per gRPC von der GUI die Methode startIncomingStream aufgerufen.

Ab hier ist das Szenario gleich, wie bei dem ThreadIncoming des LiveStream Szenarios

4.3.4 Setzen der Config

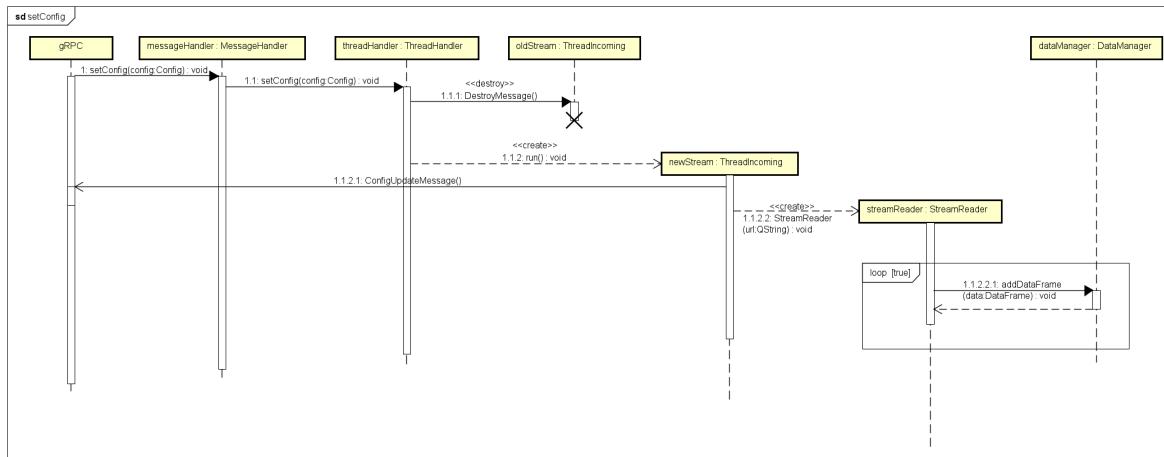


Abbildung 24: Sequenzdiagramm setze Konfig

Über die GUI wird per gRPC die Methode 'setConfig' aufgerufen. Der MessageHandler ruft auf dem ThreadHandler die gleichnamige Methode 'setConfig' auf. Nun wird der aktuelle Stream zu dieser URL geschlossen, also der Thread beendet. Jetzt wird ein neuer 'ThreadIncoming' erstellt, der zunächst eine 'ConfigUpdateMessage' per gRPC an die 'ProcessingTools' schickt und dann über den 'StreamReader' einen neuen Stream erstellt. Ab hier werden die Messages dann wieder zu DataFrames konvertiert und anschließend in die Datenbank eingegeben.

5 View

5.1 Entwurf

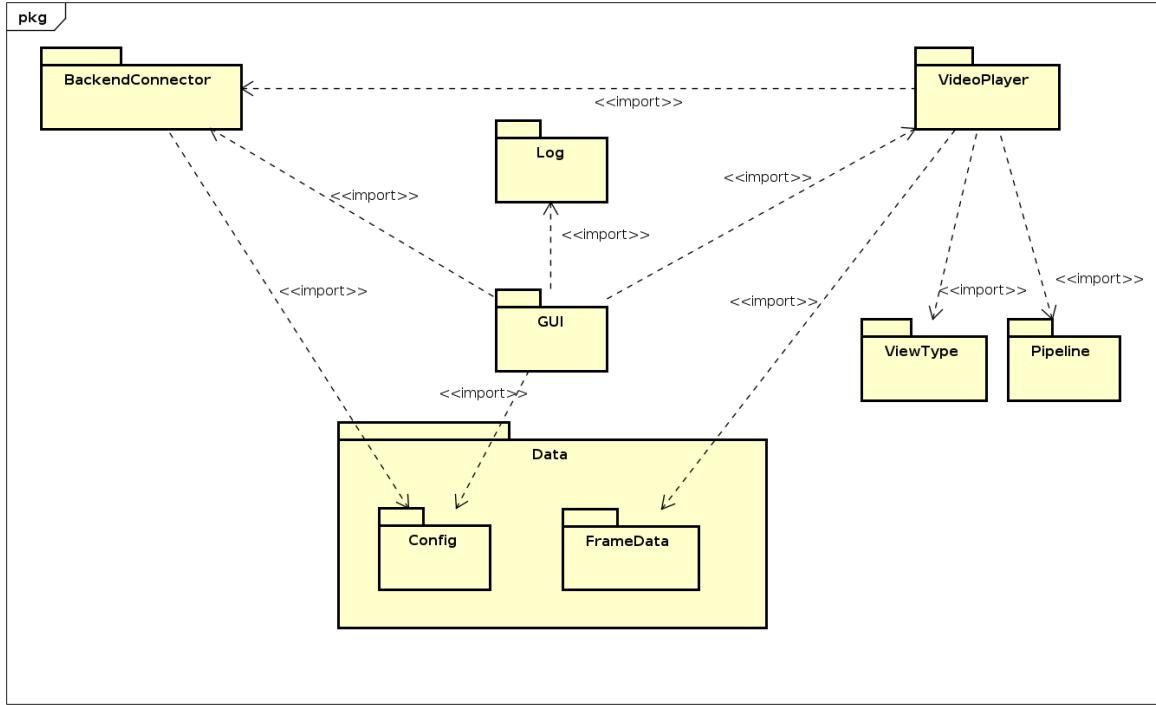


Abbildung 25: PaketDiagramm

Das Paket „GUI“ dient als Schnittstelle zum Benutzer. Das bedeutet, dass Benutzeranfragen hier entgegengenommen und weitergeleitet werden. Für sämtliche Anfragen bezüglich Logs übernimmt das Paket „Log“ die Verarbeitung. Möchte der Benutzer einen Stream oder eine Wiederholung sehen, wird dies dem „VideoPlayer“ mitgeteilt, der sich die Daten des Streams über den „BackendConnector“ vom „Backend“ holt. Diese Daten zeigt der VideoPlayer mithilfe einer Pipeline dem Benutzer an. Dabei werden die möglichen Videoformate, wie zum Beispiel das Video mit oder ohne Annotationen, über den „ViewType“ festgelegt. Das Paket „Data“ beinhaltet die Klassen für die Konfigurationsdaten sowie die Datenklassen der einzelnen anzulegenden Frames.

5.2 Klassen

5.2.1 GUI

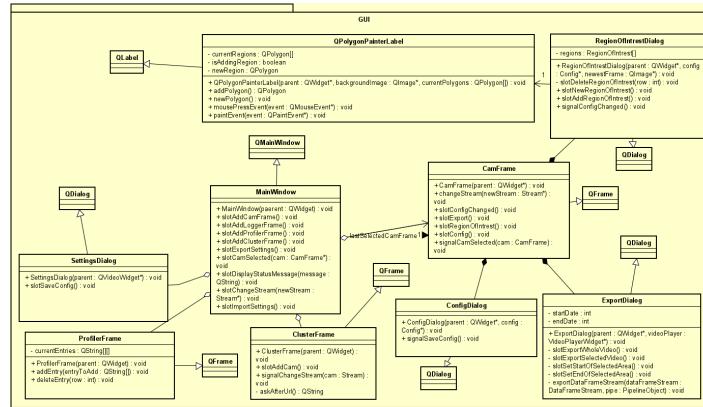


Abbildung 26: GUIPaket

Das Paket 'GUI' beinhaltet die Schnittstellen zum Benutzer.

5.2.1.1 SettingsDialog

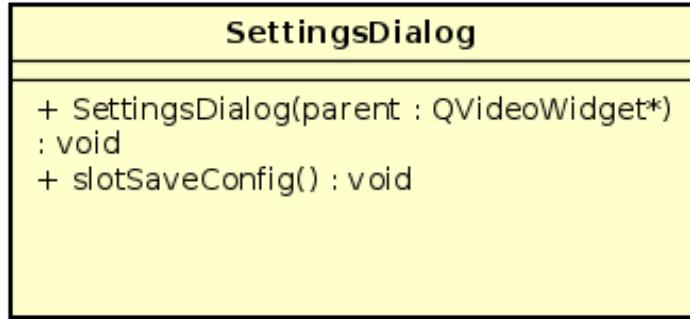


Abbildung 27: SettingsDialog

Klassenbeschreibung

Bietet dem Benutzer die Möglichkeit, die Modellkonfiguration zu verändern.

Attribute

- **writer** Mit dem die Konfiguration an das Backend weitergegeben werden kann.
- **reader** Mit dem die Konfiguration vom Backend gelesen werden kann.

Vererbung

Erbt von der Klasse `QDialog`, welche von QT bereitgestellt wird.

Konstruktoren

```
SettingsDialog(QWidget* parent)
```

Methoden

```
void slotSaveConfig()
```

wird vom Benutzer ausgelöst, sobald sich die Konfig ändert

5.2.1.2 QPolygonPainterLabel

QPolygonPainterLabel	
-	currentRegions : QPolygon[]
-	isAddingRegion : boolean
-	newRegion : QPolygon
+	QPolygonPainterLabel(parent : QWidget*, backgroundImage : QImage*, currentPolygons : QPolygon[]) : void
+	addPolygon() : QPolygon
+	newPolygon() : void
+	mousePressEvent(event : QMouseEvent*) : void
+	paintEvent(event : QPaintEvent*) : void

Abbildung 28: QPolygonPainterLabel

Klassenbeschreibung

Ist für das reine Zeichnen und erstellen von QPolygons zuständig.

Attribute

- **currentRegions** Eine Liste der jetzigen QPolygons.
- **isAddingRegion** Ein boolean, welcher dafür steht, ob gerade ein neues QPolygon vom Benutzer gezeichnet wird.
- **newRegion** Das unfertige QPolygon, welches vom Benutzer gerade gezeichnet wird.

Vererbung

Erbt von der Klasse QLabel, die von QT bereitgestellt wird.

Konstruktoren

```
QPolygonPainterLabel(QWidget* parent, QImage* backgroundImage, std ::  
vector<QPolygon> currentPolygons)
```

Initialisiert das Objekt mit den übergebenen Parametern und zeichnet die übergebenen QPolygons.

Methoden

```
void paintEvent(QPaintEvent* event)
```

Diese Methode zeichnet die bisherigen QPolygons auf das Label, sowie das unfertige QPolygon welches vom Benutzer gerade erstellt wird.

```
QPolygon addPolygon()
```

Beendet die Zeichnung des bisherigen QPolygon und gibt dieses zurück.

```
void newPolygon()
```

Startet die Zeichnung eines neuen QPolygons.

```
void mousePressEvent(QMouseEvent* event)
```

Wird aufgerufen wenn der Benutzer einen neuen Punkt hinzufügt. Dabei wird dieser Punkt in das zu zeichnende QPolygon hinzugefügt.

```
void paintEvent(QPaintEvent* event)
Wird aufgerufen wenn das Label neu gezeichnet werden soll.
```

5.2.1.3 RegionOfIntrestDialog

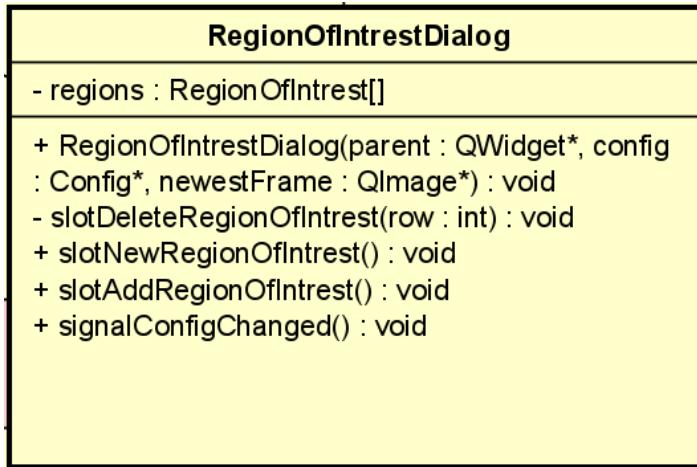


Abbildung 29: RegionOfIntrestDialog

Klassenbeschreibung

ist für das Hinzufügen und Löschen von neuen „RegionOfIntrests“ zuständig.

Attribute

- **regions** Eine Liste der bisherigen „RegionOfIntrests“.
- **polygonPainter** Ein 'QPolygonPainterLabel' auf dem die bisherigen und neuen QPolygone gezeichnet werden können.

Vererbung

Erbt von der Klasse QDialog, die von QT bereitgestellt wird.

Konuktoren

```
RegionOfIntrest(QWidget* parent, Config* config, QImage* newestFrame)
```

Initialisiert das Objekt mit den übergebenen Parametern und erstellt ein 'QPolygonPainterLabel'.

Methoden

```
void slotDeleteRegionOfIntrest(int row)
Diese Methode wird durch den Benutzer ausgelöst und löscht eine 'RegionOfIntrest'.
```

```
void slotNewRegionOfInterest()
```

Diese Methode wird durch den Benutzer ausgelöst und startet das Zeichnen einer neuen 'RegionOfInterest'.

```
void addRegionOfInterest()
```

Diese Methode wird durch den Benutzer ausgelöst und fügt die bisher gezeichnete 'RegionOfInterest' hinzu.

```
void signalConfigChanged()
```

Teilt dem CamFrame mit, dass eine neue 'RegionOfInterest' gezeichnet wurde.

5.2.1.4 ExportDialog

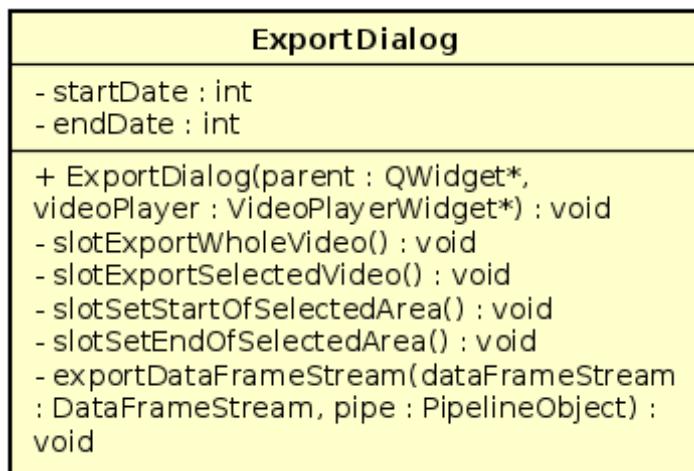


Abbildung 30: ExportDialog

Klassenbeschreibung

Das Fenster, in welchem der Benutzer einen Stream exportieren kann.

Attribute

- Ein VideoPlayerWidget, das dazu dient, den zu exportierenden Bereich auszuwählen.
- Der Startpunkt des zu exportierenden Videos.
- Der Endpunkt des zu exportierenden Videos.

Vererbung

Das Exportfenster erbt von der Klasse QDialog, die von QT bereitgestellt wird.

Konstruktoren

```
ExportDialog( parent: QWidget*, videoPlayer : VideoPlayerWidget *)
```

Methoden

```
void slotExportWholeVideo()
```

Diese Methode wird durch eine Aktion des Benutzers ausgeführt und führt dazu, dass das gesamte Video Exportiert wird.

```
void slotExportSelectedVideo()
```

Diese Methode wird durch eine Aktion des Benutzers ausgelöst und führt dazu, dass der markierte Bereich des Videos exportiert wird.

```
void slotSetStartOfSelectedArea()
```

Diese Methode wird durch eine Aktion des Benutzers ausgelöst und führt dazu, dass der Punkt, an dem das Video gestoppt wurde, als Startzeitpunkt festgelegt wird.

```
void slotSetEndOfSelectedArea()
```

Diese Methode wird durch eine Aktion des Benutzers ausgelöst und führt dazu, dass der Punkt, an dem das Video gestoppt wurde, als Endzeitpunkt festgelegt wird.

```
void exportDataFrameStream(DataFrameStream* dataFrameStream, Pipe-  
lineObject* pipe)
```

Diese Methode exportiert einen 'DataFrameStream'. Anhand der übergebenen Pipeline wird das Format des exportierten Streams festgelegt.

5.2.1.5 ConfigDialog

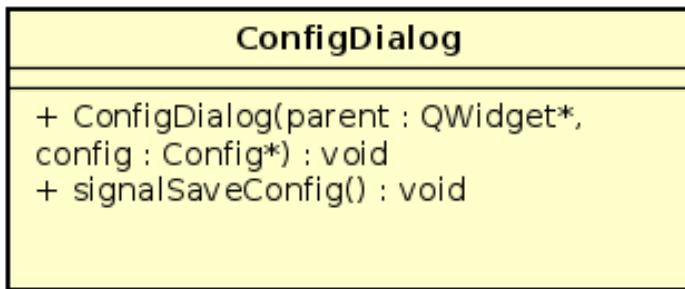


Abbildung 31: ConfigDialog

Klassenbeschreibung

Das Fenster, in welchem der Benutzer den Stream konfigurieren kann.

Attribute

- **currentConfig** Eine Config die zurzeit bearbeitet wird.

Vererbung

Das Config Fenster erbt von der Klasse QDialog, die von QT bereitgestellt wird.

Konstruktoren

```
ConfigDialog( parent: QWidget*, config : Config *)
```

Methoden

```
void signalSaveConfig()
```

Diese Methode signalisiert dem Kamerafenster eine Konfigurationsänderung.

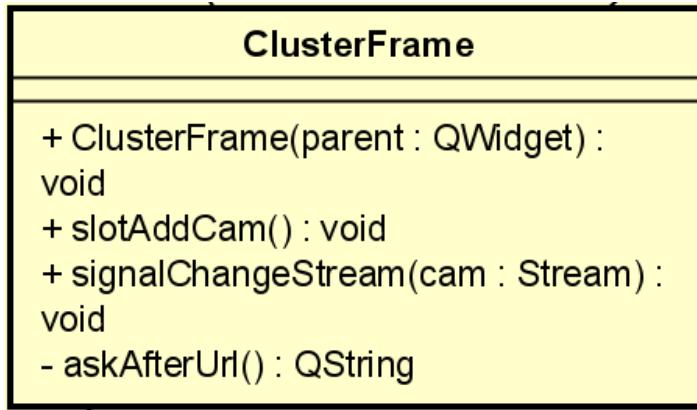
5.2.1.6 ClusterFrame

Abbildung 32: ClusterFrame

Klassenbeschreibung

Das Fenster, in welchem der Benutzer alle hinzugefügten Kameras sieht. In diesem Fenster kann er neue Kameras hinzufügen oder auswählen, welche Kamera angezeigt werden soll.

Attribute

- **streams** Liste aller hinzugefügten Streams beinhaltet eine Getter Methode für das Attribut.

Vererbung

Das Cluster Fenster erbt von der Klasse QFrame, die von QT bereitgestellt wird.

Konstruktoren

```
ClusterFrame(QWidget* parent = nullptr)
```

Methoden

```
void slotAddCam()
```

Diese Methode wird durch eine Aktion des Benutzers ausgeführt und führt dazu das eine neue Kamera aufgrund der eingegbenen URL hinzugefügt wird.

```
void signalChangeStream(Stream* cam)
```

Diese Methode benachrichtigt das Hauptfenster, dass ein neuer Stream ausgewählt wurde.

```
QString askAfterUrl()
```

Diese Methode fragt den Benutzer nach der URL einer Kamera.

5.2.1.7 CamFrame

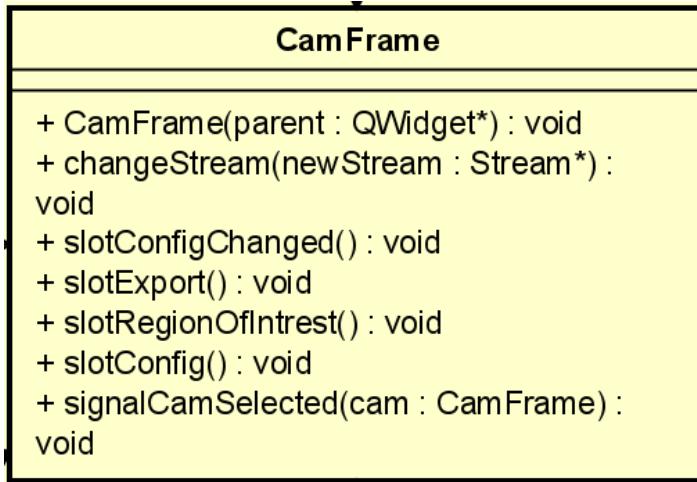


Abbildung 33: CamFrame

Klassenbeschreibung

Das Fenster in dem der Benutzer den Stream abspielen kann. Zudem kommt der Benutzer von diesem Fenster in die Konfig Einstellung des Streams.

Attribute

- **videoPlayerWidget** ein Video player Widget in dem das Video abgespielt wird.
- **configDialog** ein ConfigDialog in dem die Einstellungen zum Stream vorgenommen werden können
- **exportDialog** ein ExportDialog in dem der Stream Exportiert werden kann.
- **regionOfInterestDialog** ein RegionOfInterestDialog in dem die Regionen hinzugefügt oder gelöscht werden können

Vererbung

Das Kamera Fenster erbt von der Klasse QFrame die von QT bereitgestellt wird.

Konstruktoren

```
CamFrame(QWidget* parent = nullptr)
```

Methoden

```
void changeStream(Stream* newStream)
```

Diese Methode veranlasst einen Wechsel des angezeigten und zu konfigurierenden Stream.

```
void slotConfigChanged()
```

Über diesen Slot kann dem Kamera Fenster mitgeteilt werden, dass sich die Konfig des Streams geändert hat.

```
void slotExport()
```

Diese Methode wird durch eine Aktion des Benutzers ausgeführt und öffnet den Exportier Dialog.

```
void slotRegionOfIntrest()
```

Diese Methode wird durch eine Aktion des Benutzers ausgeführt und öffnet den Dialog zum Anpassen der Regionen.

```
void slotConfig()
```

Diese Methode wird durch eine Aktion des Benutzers ausgeführt und öffnet den Dialog zum Anpassen der Konfiguration.

```
void signalCamSelected(CamFrame* cam)
```

Diese Methode teilt dem Hauptfenster mit, dass ein neues Kamera Fenster ausgewählt wurde.

5.2.1.8 ProfilerFrame

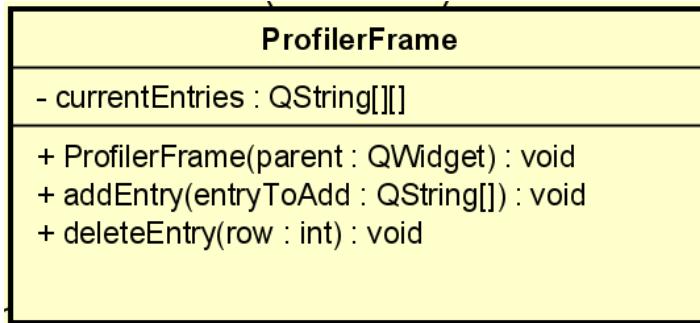


Abbildung 34: ProfilerFrame

Klassenbeschreibung

Das Fenster in dem der Benutzer Statistiken zur Performance der einzelnen Nodes sehen kann.

Attribute

- **currentEntries** die jetzigen Einträge der Tabelle.

Vererbung

Das 'Profiler' Fenster erbt von der Klasse QFrame, die von QT bereitgestellt wird.

Konstruktoren

```
ProfilerFrame(QWidget* parent = nullptr)
```

Methoden

```
void addEntry(std::vector<QString> entryToAdd)
```

Diese Methode fügt dem Profiler eine neue Zeile hinzu.

```
void deleteEntry(int row)
```

Löscht eine Zeile aus dem Profiler.

5.2.1.9 MainWindow

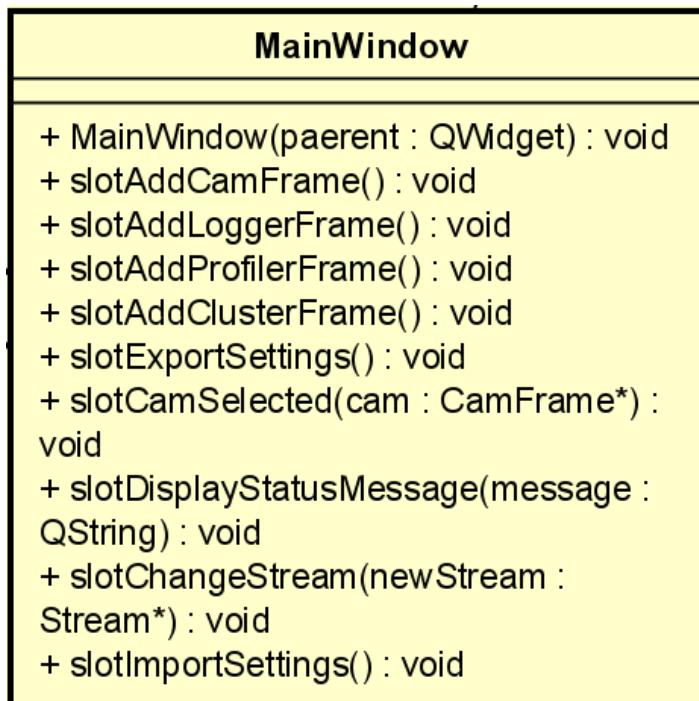


Abbildung 35: MainWindow

Klassenbeschreibung

Das Hauptfenster fasst alle Fenster, in dem der Benutzer Aktionen durchführen kann, zusammen. Damit hat der Benutzer ein zentrales Fenster, in dem alle Aktionen durchgeführt werden können.

Attribute

- **camFrames** Liste aller Cam Frames

- **logFrames** Liste aller Logger Frames
- **clusterFrames** Liste aller Cluster Frames
- **ProfilerFrame** Liste aller Profiler Frames
- **SettingsDialog** Ein Dialog zum einstellen der Einstellungen

Vererbung

Das Hauptfenster erbt von der Klasse QMainWindow die von QT bereitgestellt wird.

Konstruktoren

```
MainWindow(QWidget* parent = nullptr)
```

Methoden

```
void slotAddCamFrame()
```

Diese Methode wird durch eine Aktion des Benutzers ausgeführt und fügt ein Cam Frame ohne Stream dem MainWindow hinzu.

```
void slotAddLoggerFrame()
```

Diese Methode wird durch eine Aktion des Benutzers ausgeführt und fügt ein Logger Frame dem MainWindow hinzu.

```
void slotAddProfilerFrame()
```

Diese Methode wird durch eine Aktion des Benutzers ausgeführt und fügt ein Profiler Frame dem MainWindow hinzu.

```
void slotAddClusterFrame()
```

Diese Methode wird durch eine Aktion des Benutzers ausgeführt und fügt ein Cluster Frame dem MainWindow hinzu.

```
void slotCamSelectedFrame(CamFrame* cam)
```

Slot über welchem dem MainWindow mitgeteilt werden kann, welcher Cam Frame ausgewählt wurde.

```
void slotDisplayStatusMessage(QString message)
```

Slot über welchem dem MainWindow mitgeteilt werden kann, welche Mitteilung dem Benutzer angezeigt werden soll.

```
void slotChangeStream(Stream* newStream)
```

Slot über welchem dem MainWindow mitgeteilt werden kann, dass ein neuer Stream angezeigt werden soll.

```
void slotImportSettings()
```

Diese Methode wird durch eine Aktion des Benutzers ausgeführt und importiert vom Benutzer ausgewählte Einstellungen.

```
void slotExportSettings()
```

Diese Methode wird durch eine Aktion des Benutzers ausgeführt und exportiert die jetzigen Einstellungen.

5.2.2 VideoPlayer

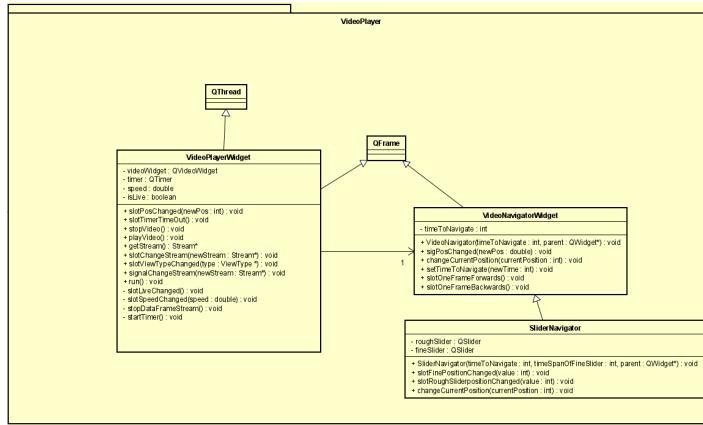


Abbildung 36: VideoPlayerPaket

Das 'VideoPlayerPaket' ist dafür zuständig, ein Video anzuzeigen sowie innerhalb des Videos zu navigieren. Dabei wird eine abstrakte Klasse zur Verfügung gestellt, mit der navigiert wird, und eine konkrete Implementierung in Form von zwei 'QSlidern' bereitgestellt.

5.2.2.1 VideoPlayerWidget

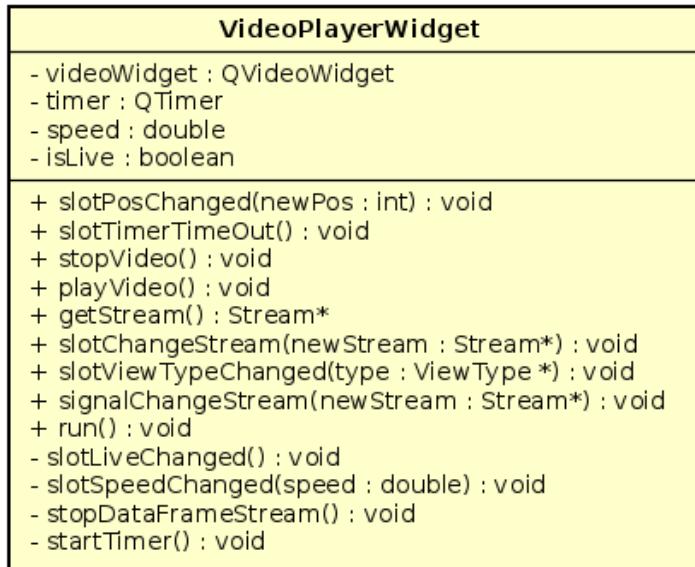


Abbildung 37: VideoPlayerWidget

Klassenbeschreibung

Die Klasse VideoPlayerWidget ist für die Steuerung und das Anzeigen eines Streams zuständig. Dabei bekommt sie durch den 'DataFrameStream' den aktuellen DataFrame. Mithilfe dessen nun ein Timer eingestellt wird, welcher genau dann auslöst, wenn das zugehörige Bild angezeigt werden soll. Dabei geht der DataFrame durch eine Pipeline, die aus dem 'ViewType' erstellt wird.

Attribute

- **currentPipeline** die Pipeline, welche die DataFrames bearbeitet
- **videoNavigatorWidget**, welcher für das spulen innerhalb eines Videos zuständig ist.
- **viewTypeFrameWidget** ein Widget um den eingegebenen 'ViewType' des Benutzers herauszufinden
- **dataFrameStream** der jetztige 'DataFrameStream' der angezeigt wird.
- **stream**, welcher angezeigt wird
- **videoWidget** in dem das Video abgespielt wird.
- **timer** welcher auslöst wenn ein neues Bild angezeigt werden soll.
- **speed** die jetztige Geschwindigkeit mit dem das Video abgespielt werden soll.
- **isLive** sagt aus ob der Stream gerade Live angezeigt wird oder nicht.

Vererbung

Erbt von der Klasse QThread und QFrame, die von QT bereitgestellt wird.

Konstruktoren

`VideoPlayerWidget()`

Methoden

`void slotPosChanged(int newPos)`

Diese Methode wird ausgeführt falls der Stream eine neue position einnimmt.

`void slotTimerTimeOut()`

Wird ausgelöst falls ein neues Bild angezeigt werden soll.

`void stopVideo()`

Stoppt das Video.

`void playVideo()`

Spielt das Video ab.

`Stream* getStream()`

Gibt den Stream zurück welcher Stream gerade angezeigt wird.

`void slotChangeStream(Stream* newStream)`

Wird durch das Signal in der gleichen Klasse ausgelöst und ändert den Stream.

`void slotViewTypeChanged(ViewType* type)`

Ein verfügbarer Slot zum ändern des ViewTypes.

`void signalChangeStream(Stream* newStream)`

Wird von außen ausgelöst, falls ein Stream wechsel gemacht werden soll.

```
void run()
Startet den Stream.
```

```
void slotLiveChanged()
Wird vom Benutzer ausgeführt falls er innerhalb des Streams spulen will oder er den Livestream wieder sehen will.
```

```
void slotSpeedChanged(double newSpeed)
Verändert die Geschwindigkeit mit dem das Video abgespielt wird.
```

```
void stopDataFrameStream()
Beendet den jetzigen Stream.
```

```
void startTimer()
Startet den Timer.
```

5.2.2.2 VideoNavigatorWidget

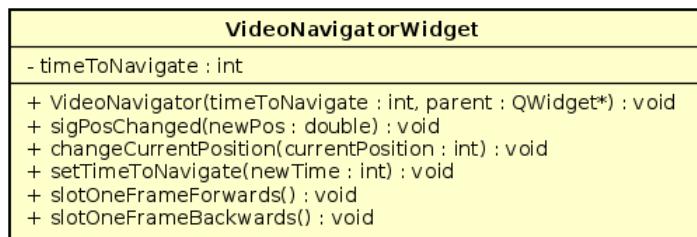


Abbildung 38: VideoNavigatorWidget

Klassenbeschreibung

Ein Interface für ein Widget, das die Funktion bereitstellt, in einem Video zu navigieren.

Attribute

- **timeToNavigate** Eine Zahl, die festlegt, über welchen Zeitraum der Navigator navigieren muss. Beinhaltet ein Getter und Setter Methoden für das Attribut.

Vererbung

Das VideoNaviagtorWidget erbt von der Klasse QWidget, die von QT bereitgestellt wird.

Konstruktoren

```
VideoNaviagtorWidget(int timeToNavigate, QWidget* parent = nullptr)
```

Methoden

```
void sigPosChanged(double newPos)
Dieses Signal wird gesendet, wenn es im Video einen Positionswechsel gab.
```

```
void changeCurrentPosition(int currentPosition )
Ändert die Position auf den übergebenen Wert.
```

```
void setTimeToNavigate(int newTime)
Setzt die Zeit in der navigiert wird auf den übergebenen Wert.
```

```
void slotOneFrameForwards()
Es wird ein Frame nach vorne gespult.
```

```
void slotOneFrameBackwards()
Es wird ein Frame nach hinten gespult.
```

5.2.2.3 SliderNavigator

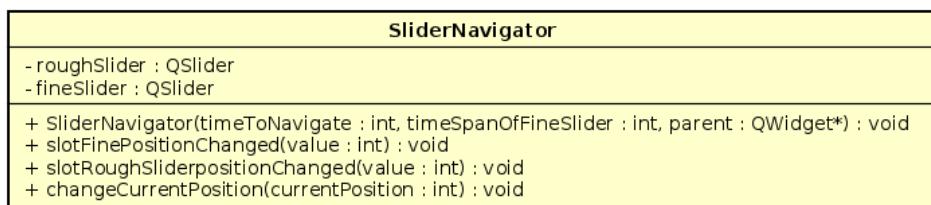


Abbildung 39: SliderNavigator

Klassenbeschreibung

Bietet die Möglichkeit, in einem Video mithilfe von zwei Slidern zu navigieren: einem Slider für die gesamte Länge des Videos und einem Slider für eine feinere Einstellung.

Attribute

- **roughSlider** Einen QSlider für die grobe Navigation innerhalb eines Videos
- **fineSlider** Einen QSlider für eine feinere Navigation innerhalb eines Videos

Vererbung

Der SliderNavigator erbt von der Klasse VideoNavigatorWidget.

Konuktoren

```
SliderNavigator( int timeToNavigate , QWidget* parent = nullptr , int timeSpanOfFineSlider )
```

Setzt die Zeitspanne, in dem der feinere Slider navigieren soll.

Methoden

```
void slotFinePositionChanged(int value)
Sendet das Signal 'sigPosChanged' von der Oberklasse mit dem Übergabewert, der
nach dem Verschieben des feinen Sliders entstanden ist.
```

```
void slotRoughSliderPositionChanged(int value )
```

Sendet das Signal 'sigPosChanged' von der Oberklasse mit dem Übergabewert, der nach dem Verschieben des groben Sliders entstanden ist.

```
void changeCurrentPosition(int currentPosition)
```

Überschreibt die Methode der Oberklasse und setzt die übergebene Position.

5.2.3 ViewTypes

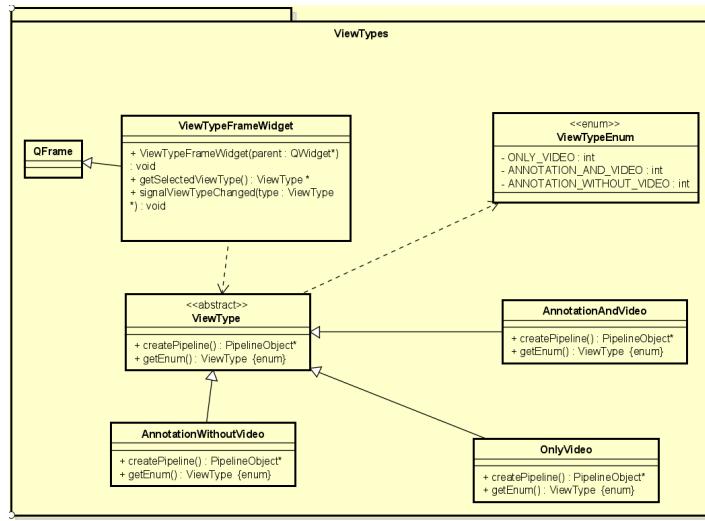


Abbildung 40: ViewTypePaket

Das Paket 'ViewTypes' beinhaltet alle Möglichkeiten des Benutzers den Stream anzuschauen. Zudem beinhaltet das Paket eine Klasse zum Auswählen eines 'ViewTypes'. Dabei muss jeder 'ViewType' eine zugehörige Pipeline zurückgeben. Damit können dynamisch weitere ViewTypes hinzugefügt werden und es muss nur die dazugehörige Pipeline angegeben werden.

5.2.3.1 viewTypeEnum

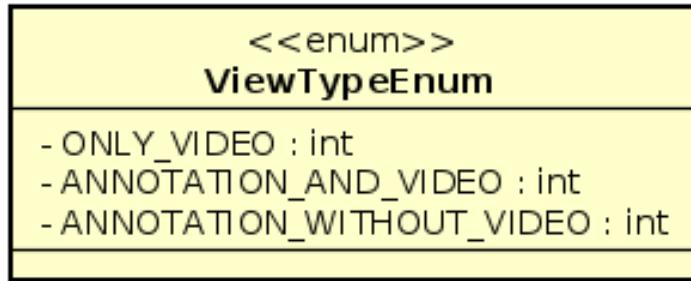


Abbildung 41: viewTypeEnum

Klassenbeschreibung

beinhaltet alle möglichen viewType Optionen.

Attribute

- **ONLY_VIDEO**
- **ANNOTATION_AND_VIDEO**
- **ANNOTATION_WITHOUT_VIDEO**

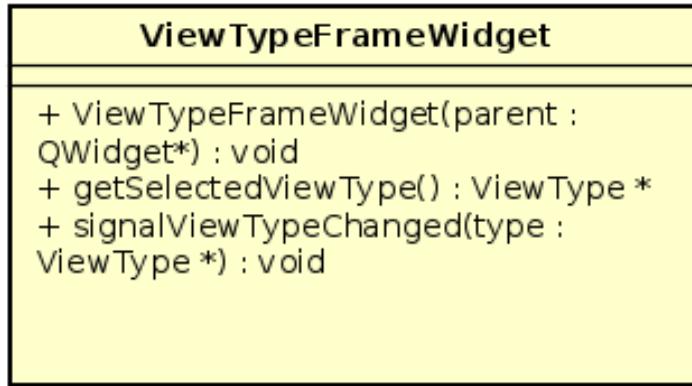
5.2.3.2 ViewTypeFrameWidget

Abbildung 42: viewTypeFrameWidget

Klassenbeschreibung

Ein Widget mit dem der Benutzer ein viewType auswählen kann.

Vererbung

Das viewTypeFrameWidget Fenster erbt von der Klasse QFrame die von QT bereitgestellt wird.

Konstruktoren

`ViewTypeFrameWidget (QWidget* parent)`

Methoden

`ViewType* getSelectedViewType()`

Diese Methode gibt den aktuell ausgewählten viewType zurück.

`void signalViewTypeChanged(ViewType* type)`

Das Signal wird gesendet, wenn der Benutzer den viewType ändert.

5.2.3.3 viewType

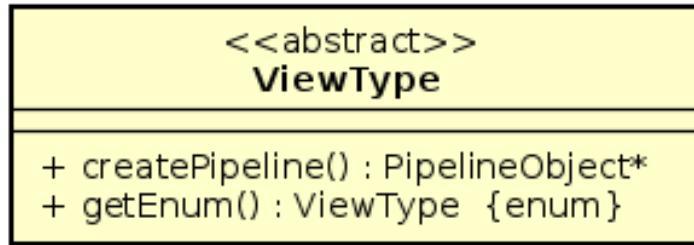


Abbildung 43: viewType

Klassenbeschreibung

Der viewType ist die Klasse die representiert wie der Benutzer den Stream anzeigen. Dabei ist viewType die Abstrakte Klasse.

Methoden

`PipelineObject* createPipeline()`
Diese Methode gibt die zum viewType gehörige Pipeline zurück.

`ViewType enum getEnum()`
Gibt das zum viewType passende Enum zurück.

5.2.3.4 OnlyVideo

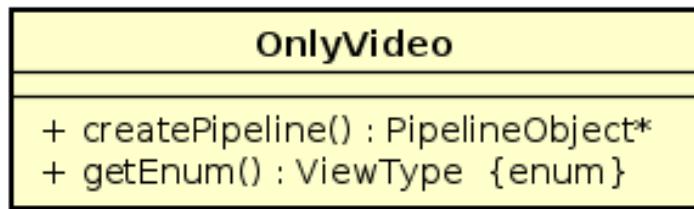


Abbildung 44: OnlyVideo

Klassenbeschreibung

OnlyVideo representiert den viewType in welchem nur das Video angezeigt wird.

Vererbung

Erbt von der Klasse viewType.

Methoden

```
PipelineObject* createPipeline()
```

Diese Methode gibt das Pipelineobjekt 'VideoPipe' verkettet mit 'RegionOfInterestPipe' zurück.

```
ViewTypeEnum getEnum()
```

Gibt das Enum 'ONLY_VIDEO' zurück.

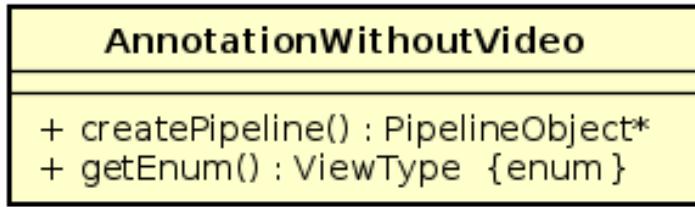
5.2.3.5 AnnotationWithoutVideo

Abbildung 45: AnnotationWithoutVideo

Klassenbeschreibung

AnnotationWithoutVideo representiert den viewType in welchem nur die Annotationsen angezeigt werden.

Vererbung

Erbt von der Klasse viewType.

Methoden

```
PipelineObject* createPipeline()
```

Diese Methode gibt das Pipelineobjekt 'AnnotationPipe' verkettet mit 'RegionOfInterestPipe' zurück.

```
ViewTypeEnum getEnum()
```

Gibt das Enum 'ANNOTATION_WITHOUT_VIDEO' zurück.

5.2.3.6 AnnotationAndVideo

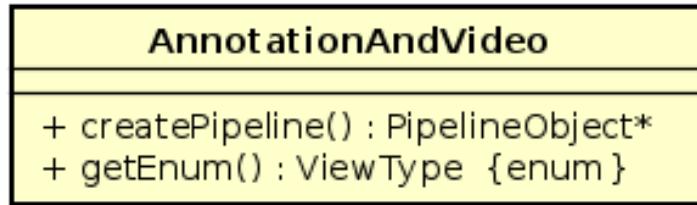


Abbildung 46: AnnotationAndVideo

Klassenbeschreibung

AnnotationAndVideo representiert den viewType in welchem die Annotationen und das Video angezeigt werden.

Vererbung

Erbt von der Klasse viewType.

Methoden

`PipelineObject* createPipeline()`

Diese Methode gibt die Verkettung von folgenden Pipelineobjekten zurück: 'AnnotationPipe', 'VideoPipe', 'RegionOfInterestPipe'.

`ViewType enum getEnum()`

Gibt das Enum 'ANNOTATION_AND_VIDEO' zurück.

5.2.4 Pipeline

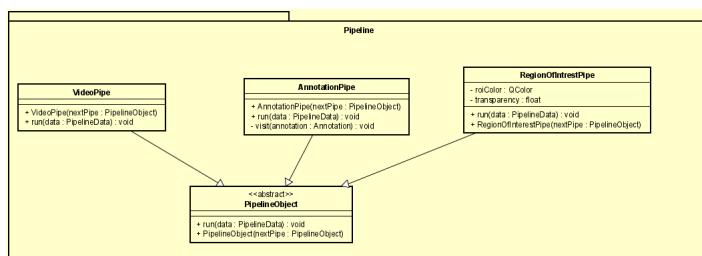


Abbildung 47: PipeLinePaket

Das Paket „Pipeline“ bietet die Möglichkeit, Annotationen und Masken auf das Bild zu projizieren. Dabei werden verschiedene 'PipelineObjects' hintereinander angeordnet. Auf dem ersten 'PipelineObject' wird 'run(pipelineData)' aufgerufen. Dabei wird 'pipelineData' von jedem Objekt modifiziert und an das nächste weitergegeben.

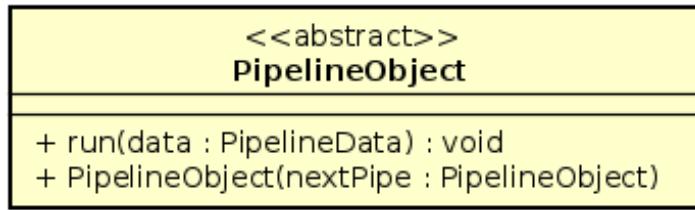


Abbildung 48: PipelineObject

5.2.4.1 PipelineObject

Klassenbeschreibung

Die Klasse PipelineObject ist die abstrakte Oberklasse für alle weiteren, konkreten Pipeline Klassen.

Konstruktoren

```
virtual PipelineObject( PipelineObject nextPipe )
```

Es wird das nächste Pipelineobjekt übergeben. Ist das übergebene Objekt NULL, dann ist dieses Pipelineobjekt das letzte Teil der Pipeline. Die Klasse ist abstrakt, es kann also kein konkretes Objekt der Klasse PipelineObject erstellt werden.

Attribute

- ein Pipelineobjekt, dass das den nächsten Schritt der Pipe darstellt und die verarbeiteten Daten dieses Pipelineobjekts entgegennimmt. Für dieses Attribut gibt es eine Getter-Methode.

Methoden

```
virtual void run(PipelineData data)
eine abstrakte Methode, die für die Ausführung eines jeden Pipelineobjekts zuständig
ist.
```

5.2.4.2 VideoPipe

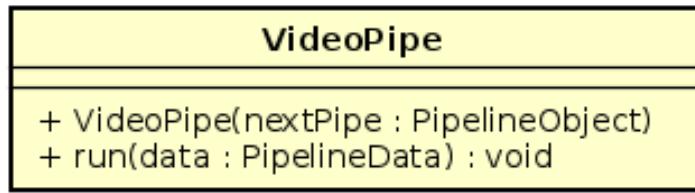


Abbildung 49: VideoPipe

Klassenbeschreibung

Die Klasse zeichnet ein bestehendes Bild auf eine bestehende QGraphicsSzene.

Vererbung

die Video Pipe Klasse erbt von der Klasse PipeObject und implementiert die Methode run().

Konstruktoren

```
VideoPipe( PipelineObject nextPipe )
```

Weil VideoPipe von PipelineObject erbt, wird im Konstruktor das nächste Pipelineobjekt festgelegt.

Methoden

```
void run(PipelineData data) override  
diese Methode schreibt jeweils ein Bild auf eine QSzene und gibt das Resultat an das  
nächste Pipelineobjekt weiter.
```

5.2.4.3 AnnotationPipe

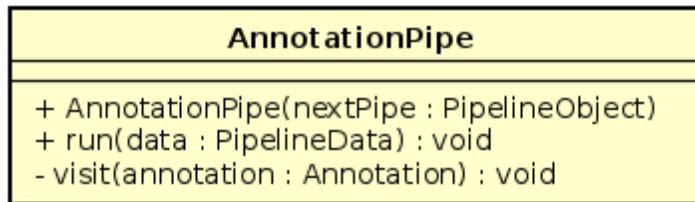


Abbildung 50: AnnotationPipe

Klassenbeschreibung

Die Klasse implementiert zum Handhaben der verschiedenen Reaktionen auf verschiedene Annotationstypen das Visitor pattern und zeichnet die Annotationen(BoundingBoxen, etc.) auf die bereits vorhandene QGraphicsSzene.

Vererbung

die Video Pipe Klasse erbt von der Klasse PipeObject und implementiert die Methode run().

Attribute**Konstruktoren**

```
AnnotationPipe( PipelineObject nextPipe )
```

Weil AnnotationPipe von PipelineObject erbt, wird im Konstruktor das nächste Pipelineobjekt festgelegt.

Methoden

```
void run(PipelineData data) override  
diese Methode zeichnet alle Annotationen aus data auf die QGraphicsSzene und gibt  
das Objekt dann an das nächste PipelineObject weiter.
```

5.2.4.4 RegionOfInterestPipe

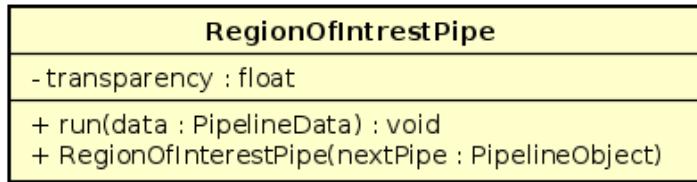


Abbildung 51: RegionOfInterestPipe

Klassenbeschreibung

Die Klasse zeichnet eine Region von Interesse bzw. markiert den irrelevanten Teil auf ein Bild.

Vererbung

die Video Pipe Klasse erbt von der Klasse PipeObject und implementiert die Methode run.

Attribute

- **float transparency** der Wert wie durchsichtig die ROI sein soll (0:=keine ROI zu sehen; 100:= ROI ist undurchsichtig)

für transparency stehen Getter und Setter Methoden zur Verfügung.

Konuktoren

```
RegionOfInterestPipe(PipelineObject nextPipe)
```

Weil RegionOfInterestPipe von PipelineObject erbt, wird im Konstruktor das nächste Pipelineobjekt festgelegt.

Methoden

```
void run(PipelineData data) override
diese Methode zeichnet eine ROI mit auf die bereits vorhandene QGraphicsScene und
gibt die PipelineDataObjekte an das nächste PipelineObject weiter
```

5.2.5 BackendConnector

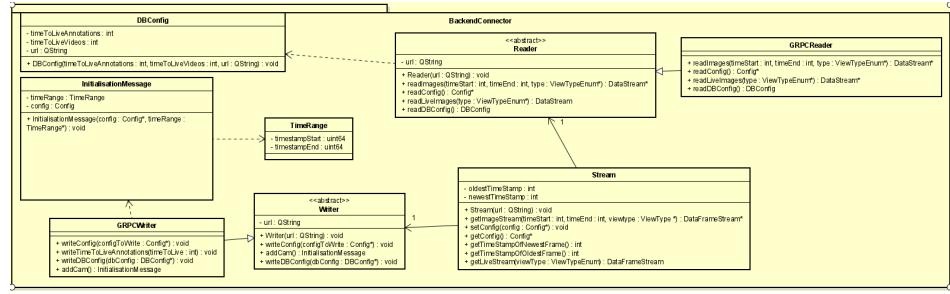


Abbildung 52: BackendConnectorPaket

Das 'BackendConnector' Paket ist dafür da eine Verbindung zum Backend aufzubauen. Dabei werden generische Klassen wie 'Writer/Reader' zu Verfügung gestellt, sowie die konkrete Verbindung über GRPC implementiert.

5.2.5.1 TimeRange

TimeRange wurde bereits im Controller definiert.

5.2.5.2 InitialisationMessage



Abbildung 53: InitialisationMessage

Klassenbeschreibung

Beinhaltet eine Konfiguration eines Streams, sowie die Zeitspanne der bisher eingegangenen Daten.

Attribute

- **timerange** die Zeitspanne der bisher eingegangenen Daten.
- **config** Konfiguration des hinzugefügten Streams.

Konstruktoren

`InitialisationMessage(Config* config , TimeRange* timeRange)`

5.2.5.3 DBConfig

DBConfig
<ul style="list-style-type: none"> - timeToLiveAnnotations : int - timeToLiveVideos : int - url : QString
+ DBConfig(timeToLiveAnnotations : int, timeToLiveVideos : int, url : QString) : void

Abbildung 54: DBConfig

Klassenbeschreibung

Beinhaltet die Konfiguration des Models.

Attribute

- **timeToLiveAnnotations** Sagt dem Model, wie lange die Annotationen gespeichert werden.
- **timeToLiveVideos** Sagt dem Model, wie lange die Videos gespeichert werden.
- **url** Repräsentiert die URL der Loki Datenbank.

Konuktoren

DBConfig(**int** timeToLiveAnnotations, **int** timeToLiveVideos, **QString** url)

initialisiert das Objekt mit den übergebenen Parametern.

5.2.5.4 Stream

Stream
<ul style="list-style-type: none"> - newestTimeStamp : int - newestTimeStamp : int
<ul style="list-style-type: none"> + Stream(url : QString) : void + getImageStream(timeStart : int, timeEnd : int, viewType : ViewType *) : DataFrameStream* + setConfig(config : Config*) : void + getConfig() : Config* + getTimestampOfNewestFrame() : int + getTimestampOfOldestFrame() : int + getLiveStream(viewType : ViewTypeEnum) : DataFrameStream

Abbildung 55: Stream

Klassenbeschreibung

repräsentiert einen Stream. Dieser Stream hat eine Konfiguration und bietet die Möglichkeit einen 'DataFrameStream' zurückzugeben.

Attribute

- **newestTimeStamp** Den neuesten TimeStamp.
- **oldestTimeStamp** Den ältesten TimeStamp.
- **writer** Einen 'Writer' auf den Informationen auf das Backend geschrieben werden kann.

- **reader** Einen „Reader“, mit dem Informationen vom Backend gelesen werden kann.
- **config** die Konfiguration des Streams.

Beinhaltet einen Getter für die beiden TimeStamps. Und einen Getter und Setter für die Konfiguration.

Konstruktoren

`Stream(QString url)`

erstellt einen „Writer“ und „Reader“ aufgrund der übergebenen URL und startet einen Stream im Backend über den „Writer“.

Methoden

`DataFrameStream* getImageStream(int timeStart, int timeEnd, ViewType* viewType)`

Diese Methode holt sich einen 'DataStream' aufgrund den übergebenen Daten und wandelt diesen in einen 'DataFrameStream' um.

`void setConfig(Config* config)`

Diese Methode setzt die neue 'Config' und teilt dies dem Backend über den 'Writer' mit.

`Config* getConfig()`

Gibt die bisherige Konfiguration zurück.

`void getTimeStampOfNewestFrame()`

Fragt das Backend nach dem neuesten TimeStamp und gibt diesen zurück.

`void getTimeStampOfOldestFrame()`

Gibt den TimeStamp vom ältesten Frame zurück.

`DataFrameStream getLiveStream(ViewType viewType)`

Fragt einen Livestream aus dem Backend an und wandelt diesen in einen 'DataStream' um.

5.2.5.5 Reader

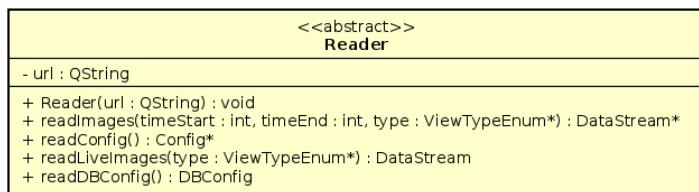


Abbildung 56: Reader

Klassenbeschreibung

Ist eine abstrakte Klasse, welche eine Möglichkeit bietet Information vom Backend zu lesen.

Attribute

- **url** Die Url von einem Stream.

Konstruktoren

```
Reader( QString url )
```

Methoden

```
DataStream* readImages(uint64 timeStart, uint64 timeEnd, ViewType* type)
```

Diese Methode holt sich einen DataStream von Annotationen und Bildern vom Backend. Dabei schreibt das Backend, beginnend beim übergebenen Zeitpunkt, so lange Bilder und Annotationen in den Stream, bis der übergebene Endzeitpunkt erreicht ist.

```
Config* readConfig()
```

Holt die Config eines Streams aus dem Backend.

```
DataStream readLiveImages(ViewType* type)
```

Holt sich aus dem Backend einen DataStream, in welchem das Backend Live Bilder und Annotationen reinschreibt.

5.2.5.6 GRPCReader

GRPCReader	
- url : QString	
+ GRPCReader(url : QString)	
+ readImages(timeStart : uint64, timeEnd : uint64, type : ViewTypeEnum*) : DataStream*	
+ readConfig() : Config*	
+ readLiveImages(type : ViewTypeEnum*) : DataStream*	
+ readDBConfig() : DBConfig	

Abbildung 57: GRPCReader

Klassenbeschreibung

Implementiert eine konkrete Leseverbindung mit dem Backend mithilfe von GRPC.

Attribute

- **url** Die Url von einem Stream.

Vererbung

Erbt von der Klasse Reader.

Konstruktoren

`GRPCReader(QString url)`

Methoden

`DataStream* readImages(uint64 timeStart, uint64 timeEnd, ViewType* type)`

Diese Methode holt sich einen ProtoBuff-Stream von Annotationen und Bildern vom Backend. Dabei schreibt das Backend, beginnend beim übergebenen Zeitpunkt, so lange Bilder und Annotationen in den Stream, bis der übergebene Endzeitpunkt erreicht ist.

`Config* readConfig()`

Holt die Config eines Streams aus dem Backend.

`DataStream readLiveImages(ViewType* type)`

Holt sich aus dem Backend einen ProtoBuff-Stream in welchem sie Live Bilder und Annotationen reingeschrieben werden.

`DBConfig readDBConfig()`

Holt die Config des Streams

5.2.5.7 Writer

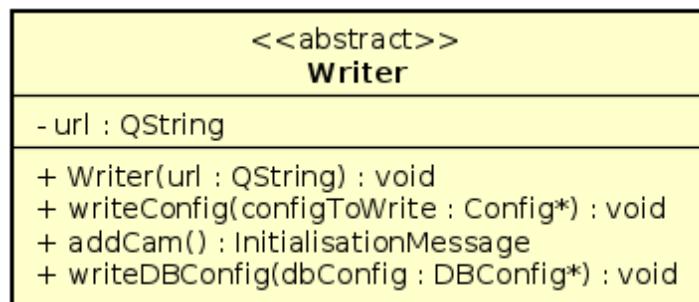


Abbildung 58: Writer

Klassenbeschreibung

Ist eine abstrakte Klasse, welche eine Möglichkeit bietet Information an das Backend zu schreiben.

Attribute

- **url** Die Url von einem Stream.

Konstruktoren

Writer(QString url)

Methoden

void writeConfig(Config* configToWrite)

Diese Methode teilt dem Backend mit, dass es eine Veränderung einer Konfiguration gegeben hat.

InitialisationMessage addCam()

Fügt dem Backend eine Kamera hinzu und gibt die InitialisationMessage des Backends zurück.

void writeDBConfig(DBConfig* dbConfig)

Teilt dem Backend die veränderte Datenbank Konfiguration mit.

5.2.5.8 GRPCWriter

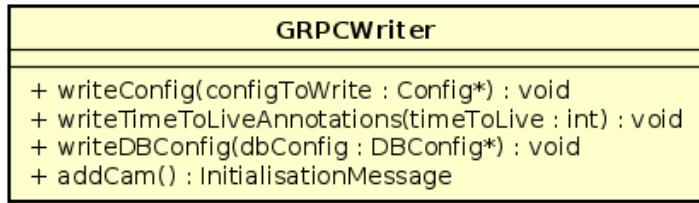


Abbildung 59: GRPCWriter

Klassenbeschreibung

Implementiert eine konkrete Schreibverbindung mit dem Backend mithilfe von GRPC.

Attribute

- **url** Die Url von einem Stream.

Vererbung

Erbt von der Klasse Writer.

Konuktoren

GRPCWriter(QString url)

Methoden

void writeConfig(Config* configToWrite)

Diese Methode teilt dem Backend mit, dass es eine Veränderung einer Konfiguration gegeben hat über eine Serviceroutine in der Protodatei.

InitialisationMessage addCam()

Fügt eine Kamera dem Backend hinzu und gibt die InitialisationMessage des Backends zurück über eine Serviceroutine in der Protodatei.

void writeDBConfig(DBConfig* dbConfig)

Teilt dem Backend die veränderte Datenbank Konfiguration mit über eine Serviceroutine in der Protodatei.

5.2.6 Data.Config

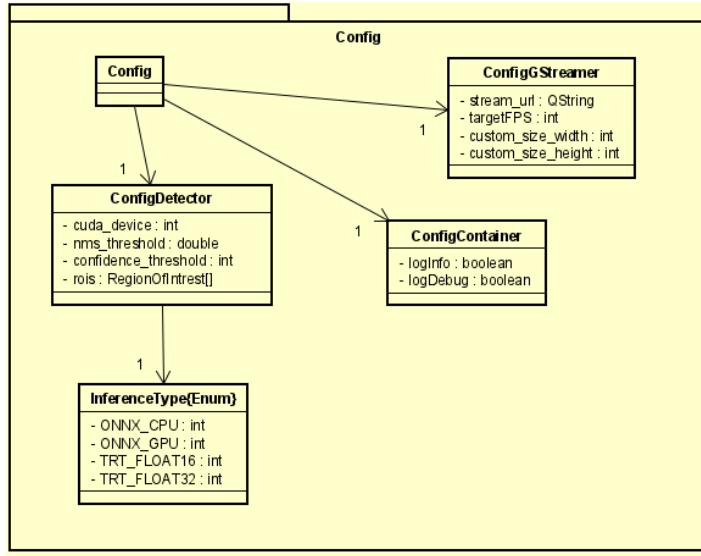


Abbildung 60: ConfigPaket

Das Paket 'Data.Config' beinhaltet die Konfiguration des Streams. Config-Klassen sind im Backend beschrieben.

5.2.7 Data.FrameData

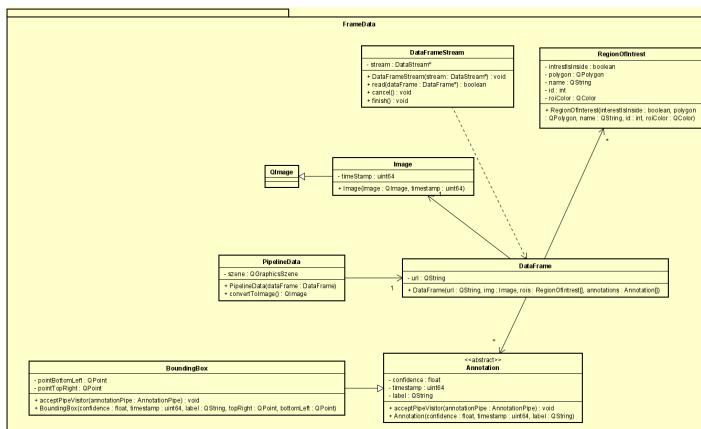


Abbildung 61: FrameDataPaket

Das Paket 'FrameDataPaket' beinhaltet alle Klassen, die die Daten eines „Frames“ repräsentieren.

5.2.7.1 DataFrameStream

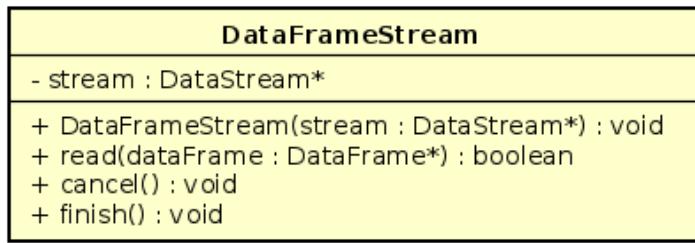


Abbildung 62: DataFrameStream

Klassenbeschreibung

Wandelt den Protobuf DataStream in einen DataFrameStream um.

Attribute

- **stream** Einen DataStream der Umgewandelt wird.

Konstruktoren

DataFrameStream(DataStream* stream)

Methoden

boolean read(DataFrame* dataFrame)

Diese Methode schreibt in den übergebenen DataFrame die Informationen aus dem Stream. Gibt bei Erfolg true zurück sonst false.

void cancel()

Bricht den Stream ab.

void finish()

Beendet den Stream.

5.2.7.2 PipelineData

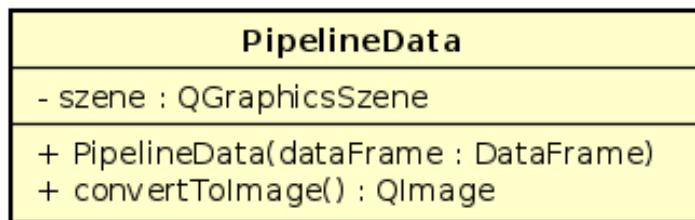


Abbildung 63: PipelineData

Klassenbeschreibung

Speichert die QGraphicsScene und einen DataFrame. Die QGraphicsScene ist standardmäßig schwarz und wird auch so angezeigt, wenn keine Bilddaten bereitgestellt werden.

Attribute

- **DataFrame** **dataFrame** alle Daten, die von der Pipeline verarbeitet werden
- **QGraphicsScene** **scene** das Objekt, auf das die Pipeline in jedem Schritt zeichnet und was am Ende angezeigt wird

sowohl für dataFrame als auch für scene werden Getter-Methoden bereitgestellt.

Konstruktoren

```
PipelineData (DataFrame dataFrame)
```

Es wird ein DataFrame übergeben und eine neue QGraphicsScene erstellt.

Methoden

```
QImage convertToImage()
konvertiert die QGraphicsScene in QImage und gibt es zurück
```

5.2.7.3 DataFrame

DataFrame
- url : QString
+ DataFrame(url : QString, img : Image, rois : RegionOfInterest[], annotations : Annotation[])

Abbildung 64: DataFrame

Klassenbeschreibung

Speichert alle wichtigen Daten, die nötig sind, um einen Frame einer Kamera zu beschreiben. Dazu gehören das eigentliche Bild, die Kamera URL, eine Menge an von Annotationen und eine Menge von ROIs. Für alle diese Attribute werden Getter und Setter-Methoden zur Verfügung gestellt.

Attribute

- **QString** **url** die URL der Kamera
- **Image** **image** die Bilddaten
- **RegionOfInterest[]** **rois** die Menge der ROIs
- **Annotation[]** **annotations** die Menge der Annotationen

für alle Attribute stehen Getter und Setter-Methoden bereit.

Konstruktoren

```
DataFrame(QString url, Image img, RegionOfInterest[] rois, Annotation[] annotations)
DataFrame()
```

Es werden entweder alle Attribute über den Konstruktor festgelegt und dann nicht mehr geändert oder es werden keine Attribute übergeben (Standard Konstruktor) und die Attribute werden von über Setter gesetzt.

5.2.7.4 Image

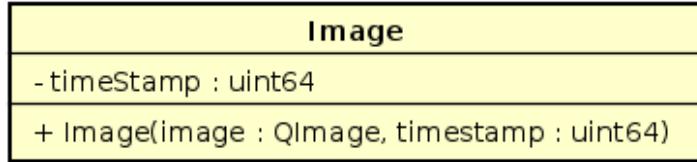


Abbildung 65: Image

Klassenbeschreibung

erweitert die Klasse QImage um einen Timestamp. Sowohl das Bild als auch der Timestamp können über Getter-Methoden erreicht werden.

Vererbung

Die Klasse erbt von QImage und ergänzt die Klasse um ein Attribut timestamp.

Attribute

- **uint64 timeStamp** der Timestamp, zu dem das Bild erstellt wurde
für das Attribut timestamp steht eine Getter-Methode zur Verfügung.

Konstruktoren

```
Image(QImage image, uint64 timestamp)
```

Es werden das eigentliche Bild und der dazugehörige Timestamp übergeben

5.2.7.5 Annotation

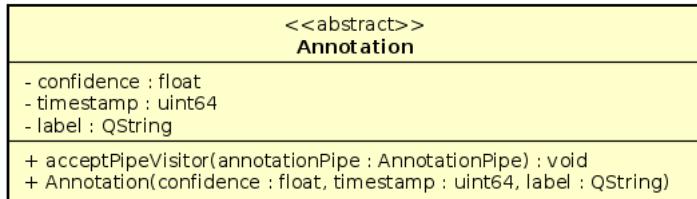


Abbildung 66: Annotation

Klassenbeschreibung

Speichert Attribute wie Zuverlässigkeit, eine Bezeichnung und einen Timestamp. Die Klasse ist Teil eines Visitor Patterns.

Attribute

- float confidence
- uint64 timestamp
- QString label

Für alle Attribute stehen Getter-Methoden zur Verfügung.

Konstruktoren

```
Annotation( float confidence, uint64 timestamp, QString label )
```

Es werden Metadaten wie Zuverlässigkeit, den Aufnahmezeitpunkt und die Bezeichnung übergeben und gespeichert.

Methoden

```
void acceptPipeVisitor(AnnotationPipe visitor)
```

Diese Methode ruft die AnnotationPipe auf und übergibt sich selbst. Dadurch muss nicht für jede Tochterklasse eine eigene Fallunterscheidung in der AnnotationPipe Klasse gemacht werden.

5.2.7.6 BoundingBox

BoundingBox
- pointBottomLeft : QPoint
- pointTopRight : QPoint
+ acceptPipeVisitor(annotationPipe : AnnotationPipe) : void
+ BoundingBox(confidence : float, timestamp : uint64, label : QString, topRight : QPoint, bottomLeft : QPoint)

Abbildung 67: BoundingBox

Klassenbeschreibung

Erweitert die Klasse Annotation und fügt zwei Punkte (oben rechts und unten links – für ein Rechteck) hinzu

Vererbung

BoundingBox erbt von der Klasse Annotation.

Attribute

- **QPoint pointTopRight** markiert die obere rechte Ecke der Bounding Box
- **QPoint pointBottomLeft** markiert die untere, linke Ecke der Bounding Box

für beide Attribute werden Getter-Methoden zur Verfügung gestellt.

Konstruktoren

```
BoundingBox(float confidence, uint64 timestamp, QString label, QPoint  
topRight, QPoint bottomLeft)
```

Es werden Metadaten wie Zuverlässigkeit, den Aufnahmezeitpunkt und die Bezeichnung übergeben und gespeichert. Außerdem werden die Eckpunkte für die Bounding Box übergeben.

Methoden

```
void acceptAnnotationVisitor(Annotation Pipe visitor)  
Diese Methode ruft die AnnotationPipe auf und übergibt sich selbst. Dadurch muss  
nicht für jede Tochterklasse eine eigene Fallunterscheidung in der AnnotationPipe  
Klasse gemacht werden.
```

5.2.7.7 RegionOfInterest(ROI)

RegionOfInterest	
- interestIsInside : boolean	
- polygon : QPolygon	
- name : QString	
- id : int	
- roiColor : QColor	
+ RegionOfInterest(interestIsInside : boolean, polygon : QPolygon, name : QString, id : int, roiColor : QColor)	

Abbildung 68: RegionOfInterest

Klassenbeschreibung

Eine Klasse, die eine Region von Interesse beschreibt.

Konstruktoren

```
RegionOfInterest(bool interestIsInside, QPolygon polygon, QString  
name, int id, QColor roiColor, int roiTransparency)
```

Es werden Metadaten wie Name, ID, Farbe der ROI, Transparenz der ROI und ob das innere oder äußere des Polygons gebraucht wird, übergeben und gespeichert. Außerdem wird ein Polygon, dass die ROI beschreibt, übergeben

Attribute

- **QString name** der Name der ROI
- **int id** die ID der ROI
- **QColor roiColor** die Farbe, mit der die ROI angezeigt wird
- **bool interestIsInside** bestimmt, ob die ROI im Polygon ist oder außerhalb

- **QPolygon polygon** bestimmt die Position und Form der ROI
für alle Attribute steht eine Getter-Methode zur Verfügung. Für das Namensattribut gibt es eine Setter-Methode.

5.2.8 Log

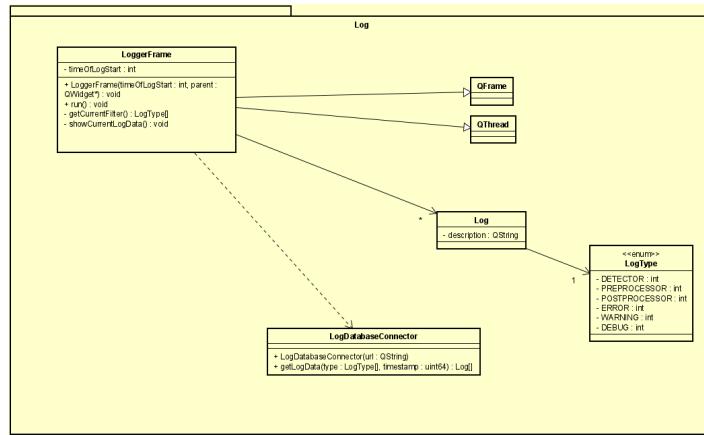


Abbildung 69: PaketLog

Das Paket 'Log' ist dafür zuständig eine Verbindung mit der Logdatenbank aufzunehmen, sowie die Logs gefiltert anzuzeigen. Dabei sind die Verschiedenen Filteroptionen als Enum festgelegt.

5.2.8.1 LoggerFrame

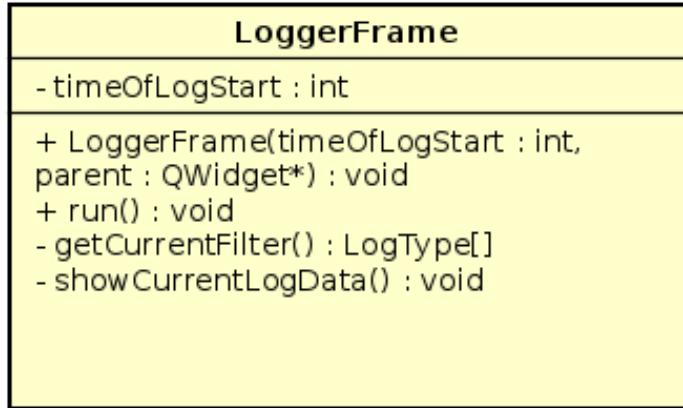


Abbildung 70: LoggerFrame

Klassenbeschreibung

Das Fenster, in welchem der Benutzer die gefilterten Logs betrachten kann.

Attribute

- **currentLogs** Liste aller angezeigten Logs
- **logDB** Einen 'LogDataBaseConnector' von dem er die Logsauslesen kann.
- **timeOfLogStart** der Zeitpunkt, ab dem die Logs angezeigt werden

Vererbung

Das Log Fenster erbt von der Klasse QFrame, die von QT bereitgestellt wird. Und von der Klasse QThread, die ebenfalls von QT bereitgestellt wird.

Konstruktoren

```
LoggerFrame(int timeOfLogStart, QWidget* parent = nullptr)
```

Erstellt den Logger und setzt die Zeit, von dem aus die Logdaten angezeigt werden sollen, auf timeOfLog-Start.

Methoden

```
std::vector<LogType> getCurrentFilter()
```

Diese Methode gibt die eingestellten Filter einstellungen des Benutzers zurück.

```
void showCurrentLogData()
```

Diese Methode holt sich die Logdaten aus der Datenbank aufgrund der eingestellten Filter und zeigt sie dem Benutzer an.

```
void run()
```

Diese Methode wird beim Starten ausgeführt des Threads und holt sich standardmäßig alle Logdaten startend bei der ausgewählten Zeit.

5.2.8.2 LogDatabaseConnector

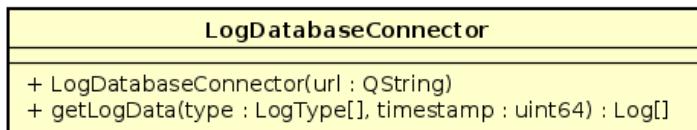


Abbildung 71: LogDatabaseConnector

Klassenbeschreibung

Diese Klasse bildet das Interface zur Log-Datenbank, stellt Anfragen an diese und nimmt die Antworten entgegen.

Konstruktoren

```
LogDatabaseConnector(QString url)
```

es wird die URL zur Log-Datenbank übergeben.

Methoden

Log[] getLogData(LogType[] types, uint64 timestamp)

Die Methode nimmt eine Menge an Typen und einen Timestamp entgegen. Die Menge an Typen representieren die Filtereinstellungen, die von der Datenbank bei der Ausgabe berücksichtigt werden sollen. Der Timestamp representiert den Zeitpunkt ab dem die Daten der Datenbank ausgelesen werden.

5.2.8.3 Log



Abbildung 72: LogDatabaseConnector

Klassenbeschreibung

Diese Klasse ist ein Wrapper für einen Eintrag aus der Log-Datenbank.

Attribute

- **QString description** der Inhalt des Eintrags

für das Attribut stehen Getter und Setter Methoden zur Verfügung.

5.2.8.4 LogType

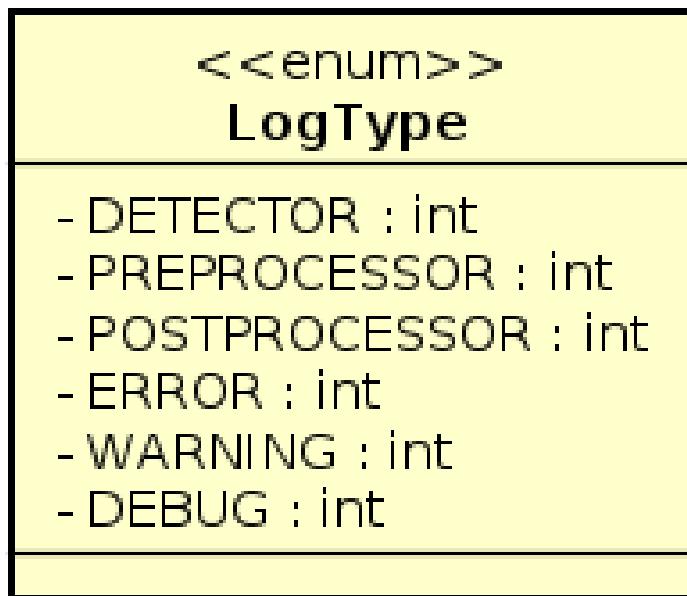


Abbildung 73: LogDatabaseConnector

Klassenbeschreibung

LogType ist ein Enum, das alle Filteroptionen für die Logs Speichert.

Attribute

- DETECTOR
- PREPROCESSOR
- POSTPROCESSOR
- ERROR
- WARNING
- DEBUG

5.3 Ablaufbeschreibungen

5.3.1 Anzeigen eines neuen Streams

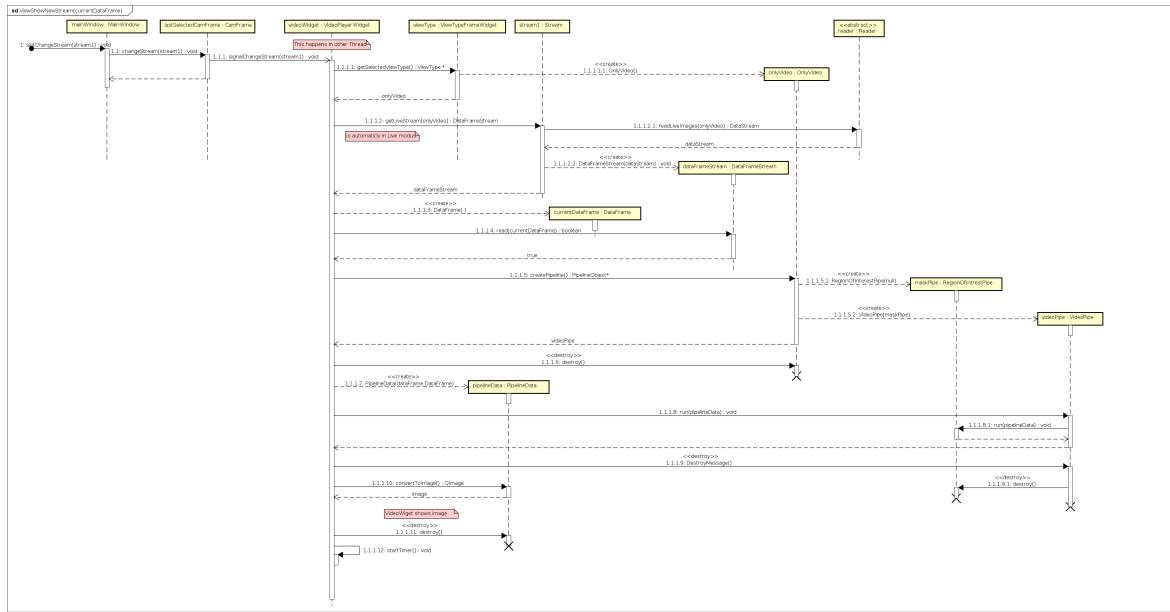


Abbildung 74: Sequenzdiagramm zum Anzeigen eines neuen Streams

Im Hauptfenster wird der letzte Stream ausgewählt. Das Cam Frame sendet daraufhin eine Nachricht an das VideoPlayerWidget, um den Stream zu wechseln. In einem separaten Thread wird der aktuelle viewType anhand der Radiobuttons als 'onlyVideo' erstellt und zurückgegeben. Das VideoWidget fragt den DataFrameStream nach einem Livestream ohne Annotationen (onlyVideo). Der DataFrameStream liest die Live-Bilder, verarbeitet sie und gibt einen DataFrameStream zurück.

Ein DataFrame wird erstellt und mit den Daten des DataFrameStreams gefüllt. Anschließend wird dem viewType 'onlyVideo' befohlen, eine Pipeline zu erstellen. Dieses Objekt generiert zwei Pipelineobjekte: Eine RegionOfInterestPipe und eine VideoPipe. Die Videopipe wird mit der ROI-Pipe verbunden, und die VideoPipe wird zurückgegeben.

Die Daten für die Pipeline werden vorbereitet, indem ein pipelineData-Objekt erstellt wird. Dieses Objekt durchläuft die Pipeline, wird dort bearbeitet, und die resultierenden Daten werden in Bilder umgewandelt. Die fertigen Bilder werden zurückgegeben, und gleichzeitig wird ein Timer gestartet.

5.3.2 Anzeigen der Logdaten

Die Methode `showCurrentLogData()` ruft die Klasse `LoggerFrame` auf, um die aktuellen Logdaten anzuzeigen. Das `LoggerFrame`-Objekt führt einen internen Methodenaufruf durch, um die aktuell eingestellten Filtereinstellungen zu erhalten. Dabei sind die Filter „`DETECTOR`“ und „`DEBUG`“ ausgewählt.

Anschließend erfolgt eine Anfrage an die Klasse `LogDatabaseConnector`, um alle Logdateien aus der Datenbank abzurufen, die den ausgewählten Filtern entsprechen. Innerhalb der Methode wird das Datenbank-Query für die Abfrage erstellt, die eigentliche Anfrage an die Datenbank durchgeführt, und schließlich werden die endgültigen Log-Objekte erstellt.

Die erstellten Log-Objekte werden von der Methode zurückgegeben und daraufhin angezeigt.

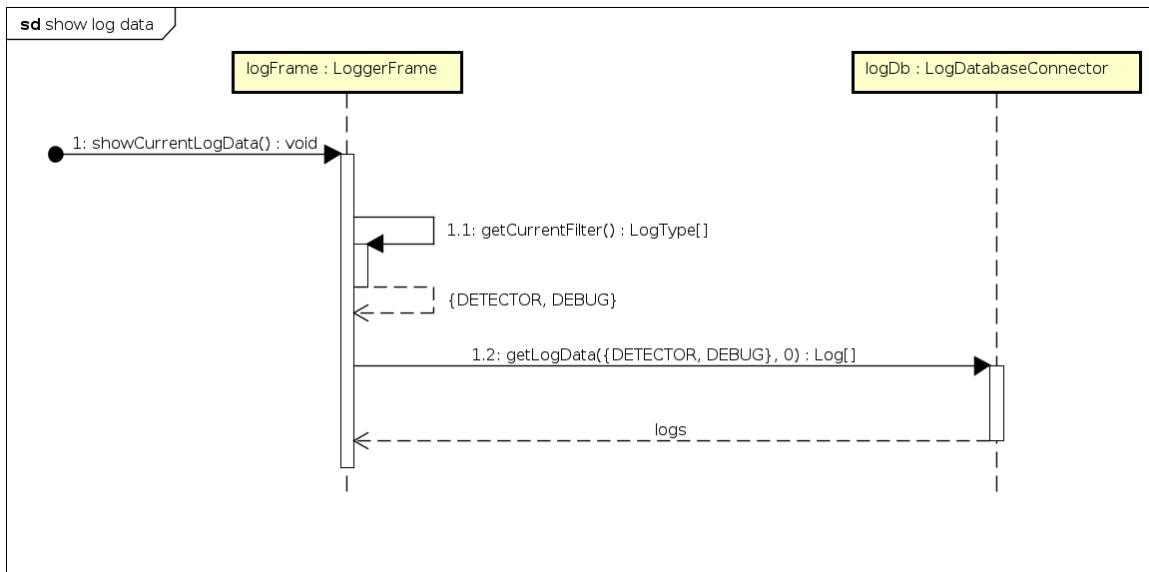


Abbildung 75: Sequenzdiagramm zum Anzeigen der Logdaten

5.3.3 Exportieren eines Videos ohne Annotationen

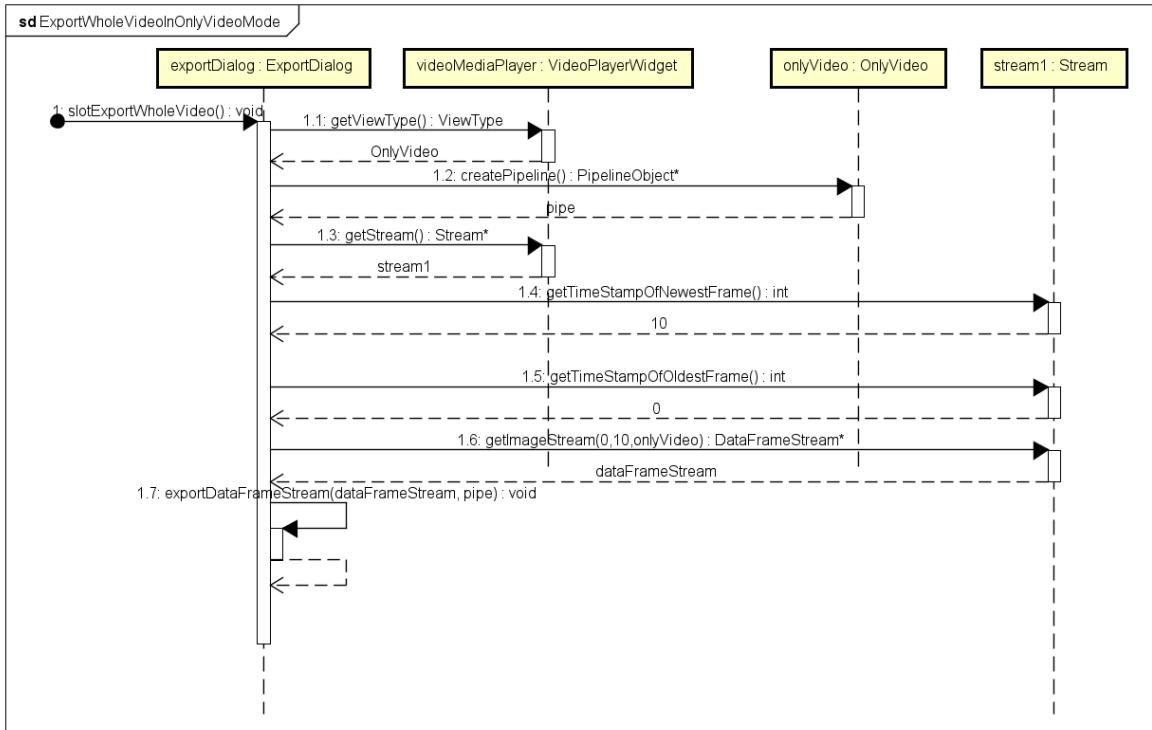


Abbildung 76: Sequenzdiagramm zum Exportieren eines Videos ohne Annotationen

Der Benutzer löst den Slot 'slotExportWholeVideo' aus. Zuerst wird der 'ViewType' aus dem 'VideoPlayerWidget' abgerufen. In diesem Fall hat der Benutzer ihn auf 'OnlyVideo' eingestellt. Anschließend wird aus dem 'ViewType' die entsprechende Pipeline erstellt. Als Nächstes wird der Stream aus dem 'videoMediaPlayer' entnommen, um die Zeitspanne des zu exportierenden Videos zu bestimmen. Danach wird ein 'DataFrameStream' für die festgelegte Zeitspanne angefordert. Zum Schluss wird der Stream exportiert.

5.3.4 Füge eine neue Maske mit 3 Punkten hinzu

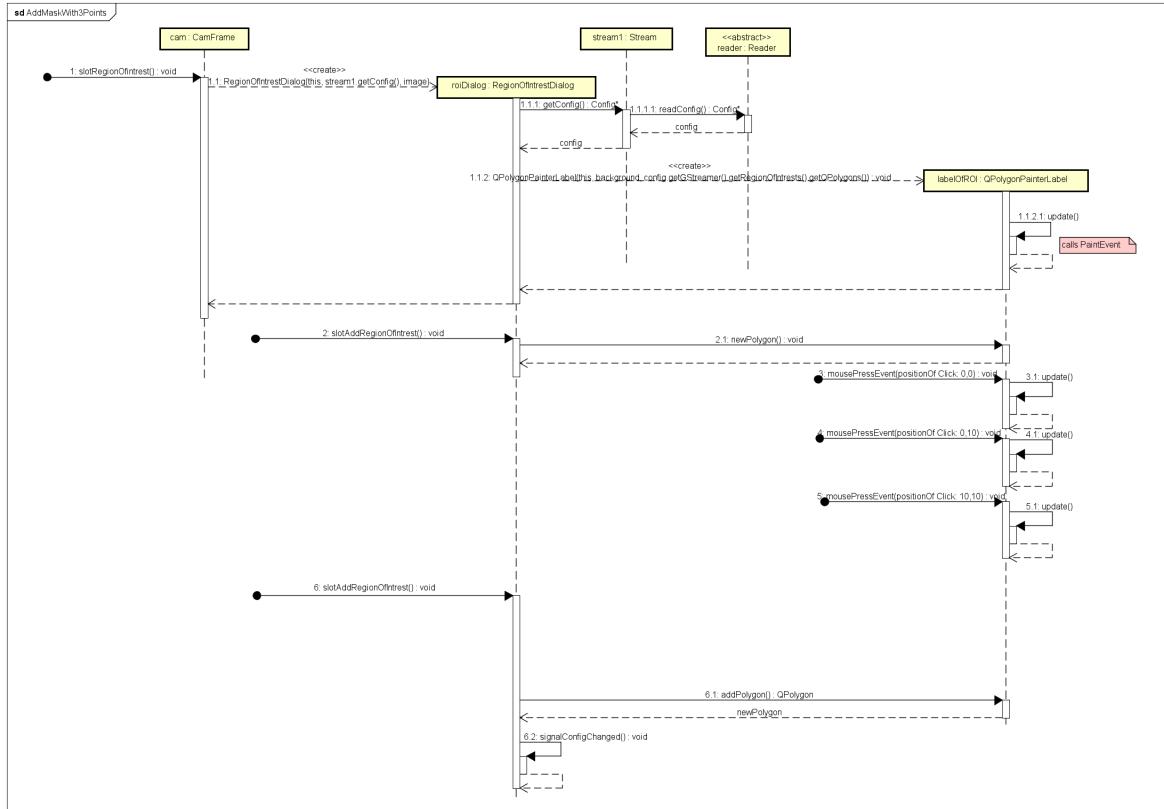


Abbildung 77: Sequenzdiagramm zum Hinzufügen einer neuen Maske mit 3 Punkten

Der Benutzer löst den 'slotRegionOfInterest' im Stream aus, um eine neue Maske hinzuzufügen. Ein neues 'RegionOfInterestDialog' wird erstellt und erhält den zu ändernden Stream sowie das aktuell angezeigte Bild. Daraufhin holt sich der 'RegionOfInterestDialog' die Konfiguration. Mithilfe dieser Konfiguration wird ein 'QPolygonPainterLabel' erstellt, das die bisherigen 'Regions of Interest' auf das übergebene Bild zeichnet. Der Benutzer fügt nun eine neue 'Region of Interest' hinzu. Dabei wird dem 'QPolygonPainterLabel' mitgeteilt, dass ein neues Polygon gezeichnet wird. Der Benutzer markiert drei Punkte, wobei jeder Punkt auf dem Label gezeichnet und miteinander verbunden wird. Abschließend fügt der Benutzer die 'Region of Interest' hinzu. Dabei wird das bisherige Polygon vom 'QPolygonPainterLabel' übernommen. Zuletzt wird ein Signal gesendet, um die Konfiguration zu speichern.

5.3.5 Füge ein neuen Stream hinz

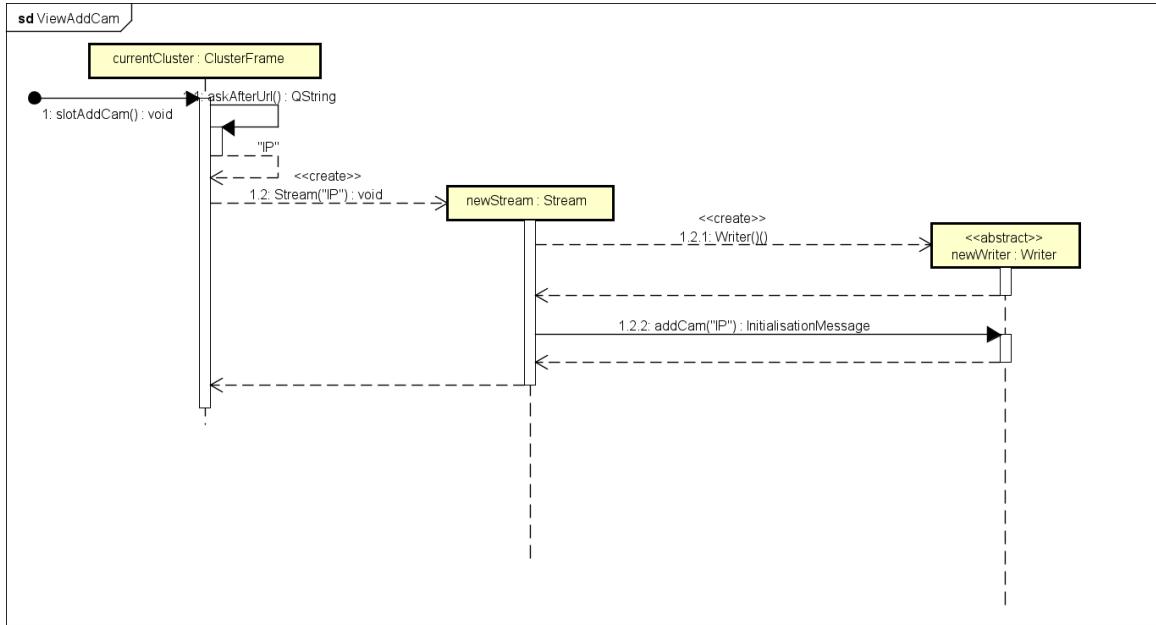


Abbildung 78: Sequenzdiagramm zum hinzufügen eines neuen Streams

Im 'ClusterFrame' löst der Benutzer den Slot 'slotAddCam' aus. Anschließend wird der Benutzer nach der URL des hinzuzufügenden Streams gefragt. Daraufhin wird ein neuer Stream erstellt, der wiederum einen neuen 'Writer' erzeugt. Über diesen Writer wird dann der neue Stream hinzugefügt.

6 Änderungen Pflichtenheft

Folgende Punkte werden rückwirkend am Pflichtenheft geändert:

- Liniendiagramme sind Teil der Kann-Kriterien
- Projekte sind Teil der Kann-Kriterien
- Datenpersistierung passiert in den Processing Nodes
- Produktumgebung ist angepasst

Glossar

Backend Backend (auch Back End oder Back-End) ist ein Begriff aus der Informationstechnik, welcher die Datenverarbeitung und- speicherung im Hintergrund von Software, Apps, Webseiten und mehr umfasst.. 5–8

Bulk Insert Das Einfügen von mehren Daten auf einmal. 13

Client Ein Client ist ein Computer oder eine Software, das mit einem Server kommuniziert und Daten und Dienste anfordert oder bereitstellt.. 6

Controller Eine Komponente im MVC-Muster, die Eingaben des Benutzers verarbeitet, mit dem Modell interagiert und die Ansicht aktualisiert.. 5, 6

DBMS Database Management System, ein System zur Erstellung, Verwaltung und Nutzung von Datenbanken.. 13

Enum ein Datentyp für Variablen mit einer endlichen Wertemenge. 6–8

fps Bilder pro Sekunde. 14, 28

Frontend Ein Frontend ist eine grafische Benutzeroberfläche zur Bedienung einer Website oder App.. 5–8

GUI Eine Benutzerschnittstelle, die es Nutzern ermöglicht, mit einem System über visuelle Elemente wie Fenster, Icons und Menüs zu interagieren.. 5, 18, 20, 22, 26, 32–34

H.264 gewöhnliches komprimiertungs Format. 16

Model Im MVC-Muster die Komponente, die die Geschäftslogik und die Daten repräsentiert und verwaltet.. 5–7

Model-View-Controller Ein Entwurfsmuster für die Implementierung von Benutzerschnittstellen, das die Anwendung in drei miteinander verbundene Komponenten teilt: Modell, Ansicht und Controller.. 5, 6

Multiton Ein Entwurfsmuster, das eine Erweiterung des Singleton-Musters ist und eine begrenzte Anzahl von Instanzen mit gleichem Attribut verwaltet.. 11

Pipeline Eine Folge von Verarbeitungsschritten. 5, 6

Protodatei Entwicklerdatei, die im Protokollpufferformat von Google erstellt wurde. 6, 8

QObject QObject wird von Qt zur Verfügung gestellt, um Benutzung von Signalen und Slots zu ermöglichen. Diese implementieren sehr leicht das Entwurfsmuster „Beobachter“, welches es ermöglicht, unbekannte Empfänger zu benachrichtigen.. 8

Separation of Concerns Ein Entwurfsprinzip, das darauf abzielt, ein System in verschiedene Teile aufzuteilen, wobei jeder Teil eine separate Aufgabe oder Sorge behandelt.. 5

Server Ein Server ist ein leistungsstarker Netzwerkrechner, der seine Ressourcen für andere Computer oder Programme bereitstellt.. 6

Signal Signale und Slots werden verwendet, um zwischen Objekten zu kommunizieren. Sobald Objekte Signale aussenden werden die verbunden Solts ausgelöst.. 8

Slot Signale und Slots werden verwendet, um zwischen Objekten zu kommunizieren. Sobald Objekte Signale aussenden werden die verbunden Slots ausgelöst.. 8

TTL Time To Live, die Lebensdauer von Annotationen oder Videos, bevor die gelöscht werden. 12, 13

View Die Komponente im MVC-Muster, die für die Darstellung der Daten und die Interaktion mit dem Benutzer verantwortlich ist.. 6