

# Entwurfsdokument - VINJAB: VINJAB Is Not Just A Boardcomputer

Jonas Haas      David Grajzel      Nicolas Schreiber      Valentin Springsklee  
Yimeng Zhu

10. Januar 2016



# Inhaltsverzeichnis

0.1	Einleitung . . . . .	1
0.2	Legende . . . . .	1
<b>1</b>	<b>Vorgänge</b>	<b>2</b>
1.1	Datenrepräsentanten . . . . .	2
<b>2</b>	<b>Klassendiagramme</b>	<b>3</b>
2.1	Bus . . . . .	3
2.2	Virtuelle Sensoren . . . . .	5
2.3	Database Access . . . . .	7
2.4	ConfigFileReader & AvailableSignals . . . . .	8
2.5	Website . . . . .	9
2.6	User Interface . . . . .	10
2.7	Einparkhilfe Ultraschallsensoren . . . . .	12
2.8	Einparkhilfe Lenkradposition . . . . .	12
2.9	Einparkhilfe Serverseite . . . . .	13
2.10	Einparkhilfe Clientseite . . . . .	14
<b>3</b>	<b>Klassen</b>	<b>15</b>
3.1	Einparkhilfe Clientseite . . . . .	15
3.1.1	NodeComponent . . . . .	15
3.1.2	NodeComposite . . . . .	15
3.1.3	NodeLeaf . . . . .	15
3.1.4	PathNode . . . . .	15
3.1.5	PathRefresher . . . . .	15
3.1.6	ProximityDisplayNode . . . . .	16
3.1.7	ProximityDisplayRefresher . . . . .	16
3.1.8	ScreenManager . . . . .	16
3.2	Einparkhilfe Serverseite . . . . .	16
3.2.1	Path . . . . .	16
3.2.2	PathIterator . . . . .	16
3.2.3	PathCircle . . . . .	16
3.2.4	PathCircleIterator . . . . .	17
3.2.5	PathArray . . . . .	17
3.2.6	PathArrayIterator . . . . .	17
3.2.7	PathBezier . . . . .	17
3.2.8	PathBezierIterator . . . . .	17

3.2.9	Settings	18
3.2.10	PathCalculator	18
3.3	Bus	18
3.3.1	BusDevice	18
3.3.2	Message	19
3.3.3	MessageBus	19
3.3.4	Broker	20
3.3.5	Request	21
3.4	DBAccess	21
3.4.1	DBBusDevice	21
3.4.2	LevelDBAccess	21
3.4.3	DBEntry	22
3.5	User Interface	22
3.5.1	Website	22
3.5.2	ViewFrame	22
3.5.3	StatusBarFrame	23
3.5.4	GridView	23
3.5.5	SettingsView	23
3.5.6	Table	24
3.5.7	Widget	24
3.5.8	NumberDash	24
3.5.9	BooleanDash	24
3.5.10	MapView	24
3.5.11	MapWithGasView	24
3.5.12	MapWithPOIView	24
3.5.13	ListPOI	24
3.5.14	ListGasStation	25
3.5.15	POIItem	25
3.5.16	GasStationItem	25
3.6	Konfiguration und erhältliche Signale	25
3.6.1	ConfigFileReader	25
3.6.2	SignalCollector	25
3.7	Karte	26
3.7.1	Button	26
3.7.2	Map	26
3.7.3	Decorator	26
3.7.4	GasDecorator	26
3.7.5	POIDecorator	26
<b>4</b>	<b>Sequenzdiagramme</b>	<b>27</b>
4.1	Client mit Server verbinden	27
4.2	WebRTC-Verbindung	29
4.3	Livedaten und Konfiguration	30

4.4	Statistik übertragen . . . . .	31
4.4.1	Aggregierte Funktionen anfordern . . . . .	31

## 0.1 Einleitung

Dieses Dokument beschreibt den Entwurf des Projekts VINJAB (VINJAB Is Not Just A Boardcomputer). Es beinhaltet den Aufbau der Software, welche Bibliotheken und Frameworks wo eingesetzt werden, sowie die gesamte Klassenstruktur.

## 0.2 Legende

Vor allem in den Klassendiagrammen dieses Dokuments haben wir uns dazu entschlossen, verschiedene Konventionen zur Verbesserung der Übersicht und zum Vermeiden von Redundanzen zu benutzen. Konventionen, die nicht im UML-Standard enthalten sind, sind in der folgenden Tabelle erläutert.

Zeichen	Beschreibung
...	Eine Klasse, die nur drei Punkte enthält, ist in einem anderen Diagramm beschrieben und wird an dieser Stelle nur referenziert.
:Object	z.B. als Parameter einer Funktion. Der Name dieses Parameters ist gleich mit seinem Datentyp.
Getter/Setter	Aus Gründen der Übersicht werden offensichtliche Variablen, Getter, Setter und Entwurfsmusterspezifische Funktionen z.T. nicht explizit erwähnt.

# 1 Vorgänge

## 1.1 Datenrepräsentanten

Ein Hauptteil der Kommunikation auf dem Server wird über einen zentralen Bus mit Broker gehandelt. Auf ihm gibt es verschiedene Veröffentlicher und Abonnenten (sogenannte Publisher und Subscriber) die alle eine abstrakte Klasse 'BusDevice' erweitern.

Alle Nachrichten sind Objekte von Klassen, die die abstrakte Klasse 'Message' um konkrete Inhalte erweitern. Die Klasse Message und alle ihre Unterklassen sind serialisierbar.

Die Rohdaten stammen unter anderem vom OBD2-Bluetooth-Modul, dass über einen Proxy als Publisher die erhaltenen Daten auf den Bus legt.

Auf dem Server gibt es für jedes Endgerät ein Objekt, welches das physische Gerät repräsentiert. Diese Objekte sind Instanzen einer Klasse 'Proxy' und abonnieren die benötigten Signale auf dem Nachrichtenbus.

Außerdem besitzt jeder Proxy ein eigenes Objekt der Klasse 'PeerConnection', welches für die Verbindung zwischen Server und Client zuständig ist.

Die abonnierten Daten werden vom Proxy über die PeerConnection an das Endgerät verschickt. Kommen Informationen oder Anweisungen vom Endgerät über die PeerConnection am Proxy an, werden diese auf den Bus gelegt.

Aggregierte Funktionen und virtuelle Sensoren erben als Klassen von einer Oberklasse 'VirtualSensor'. Sie abonnieren die benötigten Nachrichten, berechnen daraus neue (virtuelle) Sensorwerte und stellen diese als neue Nachricht über den Bus zur Verfügung.

Zentral gespeichert werden alle Daten in einer LevelDB-Datenbank. Die Datenbank ist über ein Modul 'DBAccess' in die Software integriert, welches Zugang zum Bus und die Konvertierung der Nachrichten in Datenbankabfragen bzw. die Konvertierung von Ergebnissen von Queries an die Datenbank in Nachrichten ermöglicht.

Auf dem Endgerät wird der gleiche Datenbus genutzt wie auf dem Server. Auch hier gibt es einen Proxy mit einer PeerConnection.

## 2 Klassendiagramme

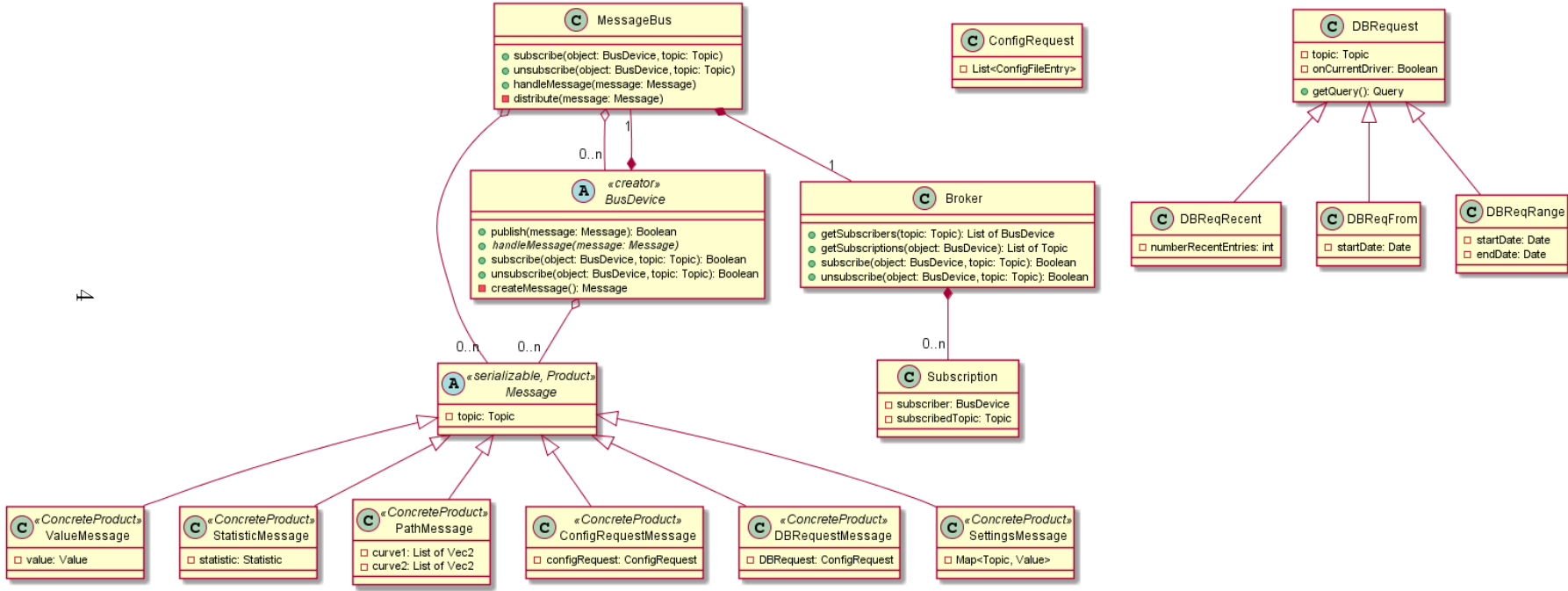
### 2.1 Bus

In diesem Diagramm ist der Aufbau des Moduls "Bus" zu sehen. Die Kommunikation zwischen allen Modulen ist entkoppelt und findet größtenteils über den Austausch von Nachrichten verschiedener Typen über den Bus statt. Um mit dem Bus arbeiten zu können, muss ein Modul eine Klasse besitzen, die von der abstrakten Klasse "BusDevice" erbt. Der Bus funktioniert als eine Variation des Beobachter-Entwurfsmusters. Eine von "BusDevice" ererbende Klasse kann über `subscribe()` bzw. `unsubscribe()` auf dem "MessageBus" Nachrichten eines bestimmten Topics (Sensortyp, Datenbanknachrichten, Konfigurationsnachrichten) abonnieren und Abonnements beenden.

Diese Abonnements werden in der Klasse "Broker" verwaltet. Sendet eine Instanz von "BusDevice" über `publish()` eine Nachricht an "MessageBus", so holt sich letzterer von seinem Broker eine Liste über alle Abonnements für das Topic der Nachricht und sendet diese dann an alle abonnierenden Objekte. Das Versenden einer Nachricht sowie deren Empfang sind asynchrone Funktionsaufrufe.

Die abstrakte Klasse "Message" besitzt eine Reihe von ererbenden Klassen. Dies ist nötig, um alle möglichen Typen von Nachrichten darzustellen: "SensorValueMessage" wird für das einfache Versenden von Messwerten benutzt, "ConfigFileRequest" für Konfigurationsdateien usw. Dieser Aufbau ermöglicht das leichte Ergänzen um ggf. zusätzlich benötigte Nachrichtentypen. Ein besonderer Nachrichtentyp ist die 'DatabaseRequestMessage', die Anfragen an die Datenbank beinhaltet. Die Anfragen vom Typ "DBRequest" sind unterteilt in verschiedene Typen von Anfragen: `DBReqRecent` fordert die letzten `n` Einträge an, `DBReqFrom` alle Einträge seit einem bestimmten Datum, und `DBReqRange` fordert alle Datenbankeinträge zwischen zwei Daten an.

Das Konstrukt der Klassen "BusDevice" sowie ererbende Klassen, "Message" und die von "Message" ererbenden Klassen ergeben gemeinsam das Entwurfsmuster der Fabrikmethode.



4

Abbildung 2.1: Bus Klassendiagramm



## 2.2 Virtuelle Sensoren

Virtuelle Sensoren berechnen aggregierte Funktionen aus Sensorwerten. Die Klasse "VirtualSensor" ist ein BusDevice. Es gibt (grundlegend unterschiedliche) Arten von Virtuelle Sensoren. Der momentane Benzinverbrauch ist ein Live-Datum, welches aus mehreren Live-Sensordaten berechnet wird. Die generierten Daten sind insbesondere unabhängig vom Zustand der Datenbank. Ein virtueller Sensor, der einen Durchschnittswert über eine bestimmte Zeit berechnet, benötigt Daten aus der Datenbank. Die Anfrage wird über den Bus geschickt ( 4.4.1). Der virtuelle Sensortyp "TopicOverTime" schickt nach einem Datenbankrequest nacheinander Messages mit Tupeln aus Zeit und Wert auf den Bus. Ein solcher Virtueller Sensor (wie Durchschnittswert und "TopicOverTime") ist abhängig vom Zustand der Datenbank.

Die Berechnung einer zustandsabhängigen aggregierten Funktion erfolgt durch eine einmalige Initialisierung des Virtuellen Sensors mit Daten aus der Datenbank. Danach werden die Daten direkt vom virtuellen Sensor inkrementell berechnet.

Es gibt die virtuellen Sensoren:

- TopicOverTime Die einfachste Art eines zustandsabhängigen virtuellen Sensors. Schickt, wenn requested, Tupel aus Zeit und Wert des spezifizierten Topics.
- TripMileage Berechnet den Kilometerstand der Fahrt ab einem spezifiziertem Zeitpunkt.
- FuelConsumption Berechnet den Momentanverbrauch
- RemainingMiles Berechnet die verbleibenden Kilometer. Die Berechnung ist abhängig vom Fahrverhalten des aktuellen Fahrers.
- AverageComputation Liefert den Durchschnittswert eines spezifizierten Topics.

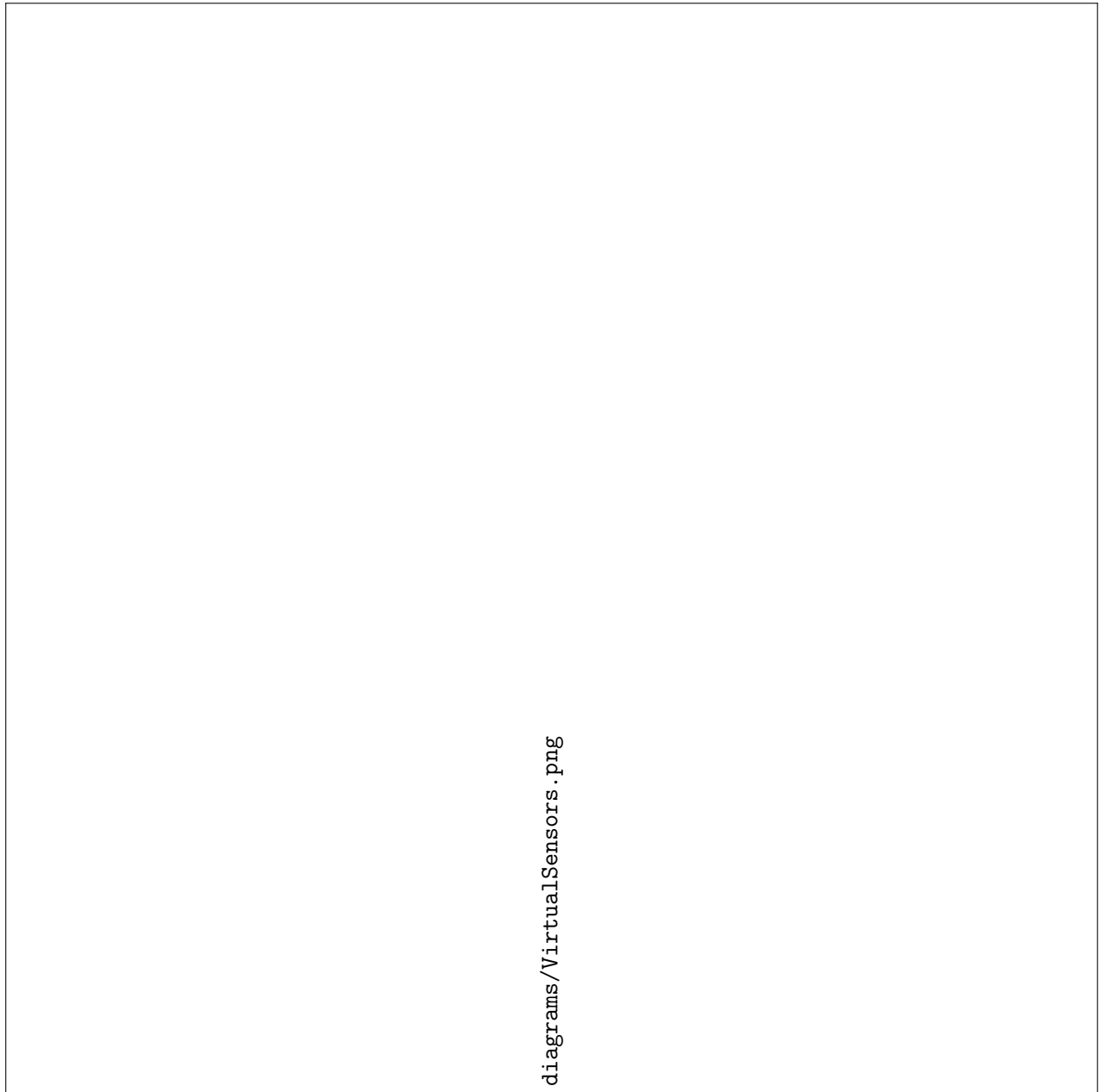


Abbildung 2.2: Klassendiagramm der virtuellen Sensoren

## 2.3 Database Access

Im folgenden Diagramm ist der Aufbau der Datenbank und des Datenbankzugriffs zu sehen.

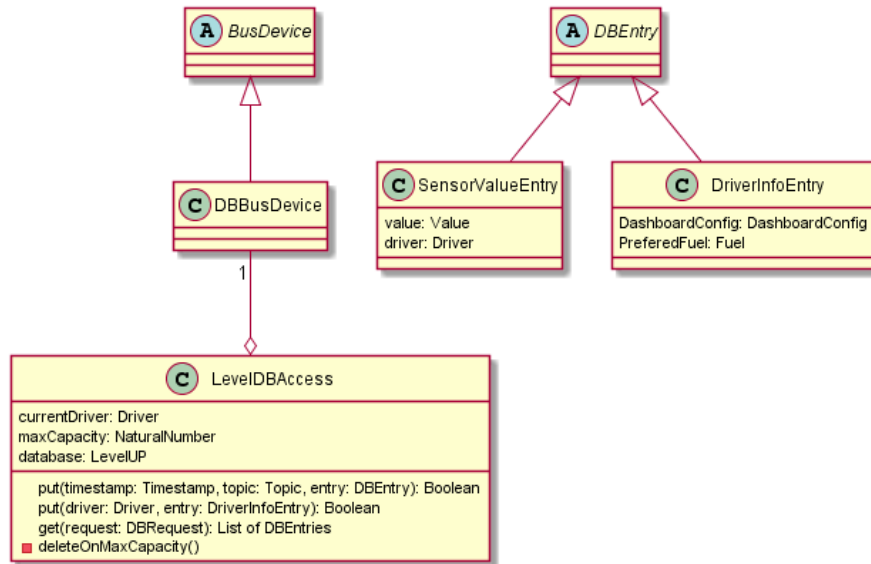


Abbildung 2.3: Klassendiagramm des Datenbankzugriffs

Als Datenbanksystem wurde, wie im Pflichtenheft spezifiziert, LevelDB benutzt. In der LevelDBAccess-Klasse wird außer der Referenz auf die LevelUp-Instanz auch der aktuelle Fahrer sowie die spezifizierte maximale Kapazität der Datenbank gespeichert.

In der Datenbank, im Fall von LevelDB ein einfacher Key-Value-Store, werden zwei Typen von Objekten gehalten: Zum einen werden mit der Zeit und dem Typ des Sensors als Key Objekte von Sensorwerten und derzeitigem Fahrer gespeichert. Zum anderen existieren fahrerbezogene Einträge, die den präferierten Kraftstoff des Fahrers sowie eine Referenz auf die Konfiguration des virtuellen Armaturenbretts beinhalten und mit der Identifikation des Fahrers als Key gespeichert werden. Zur Kommunikation mit anderen Modulen besitzt der Datenbankzugriff mit 'DBBusDevice' eine Klasse, die den Zugang zum Bus möglich macht und Nachrichten vom Bus dekodiert.

## 2.4 ConfigFileReader & AvailableSignals

Verschiedene selten geänderte Informationen werden auch in einer Konfigurationsdatei gespeichert. Diese Informationen kann der ConfigFileReader ändern. Er reagiert auf bestimmte Requests auf dem Messagebus.

Eine weitere wichtige Information ist welche Datensignale allgemein zur Verfügung stehen. Dies wird im SignalCollector gespeichert. Er reagiert ebenso auf bestimmte Signale auf dem Bus

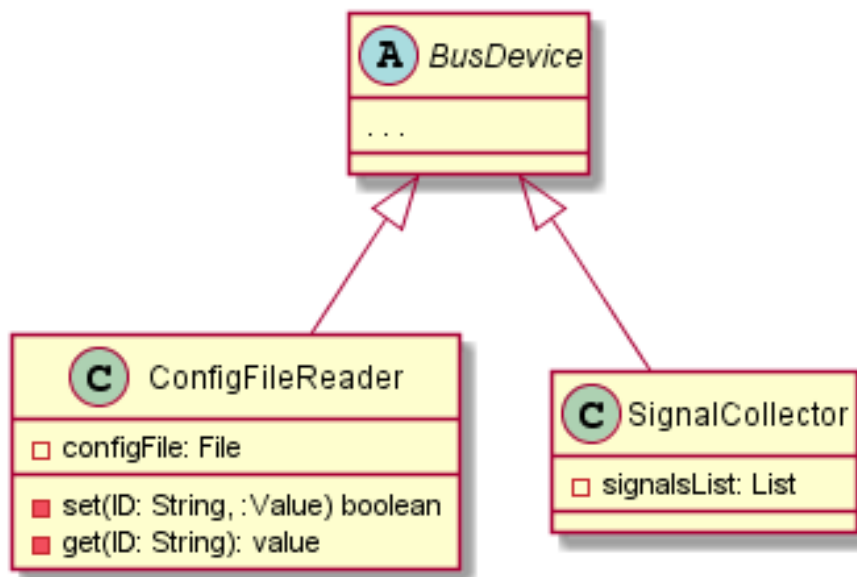


Abbildung 2.4: Klassendiagramm des Config-Readers und SignalsCollector

## 2.5 Website

Hier ist zu sehen wie die Seite allgemein aufgebaut ist. Die Website besteht aus zwei Frames. In einem Frame wird die Statusleiste angezeigt, im anderen werden können verschiedene Sichten angezeigt werden, darunter unter anderem das GridView, auf welchem die Dashes angezeigt werden. Für weiteres siehe: GridView, SettingsView.

Die Elemente der Statusleiste können unter anderem Text, Knöpfe und Icons sein.

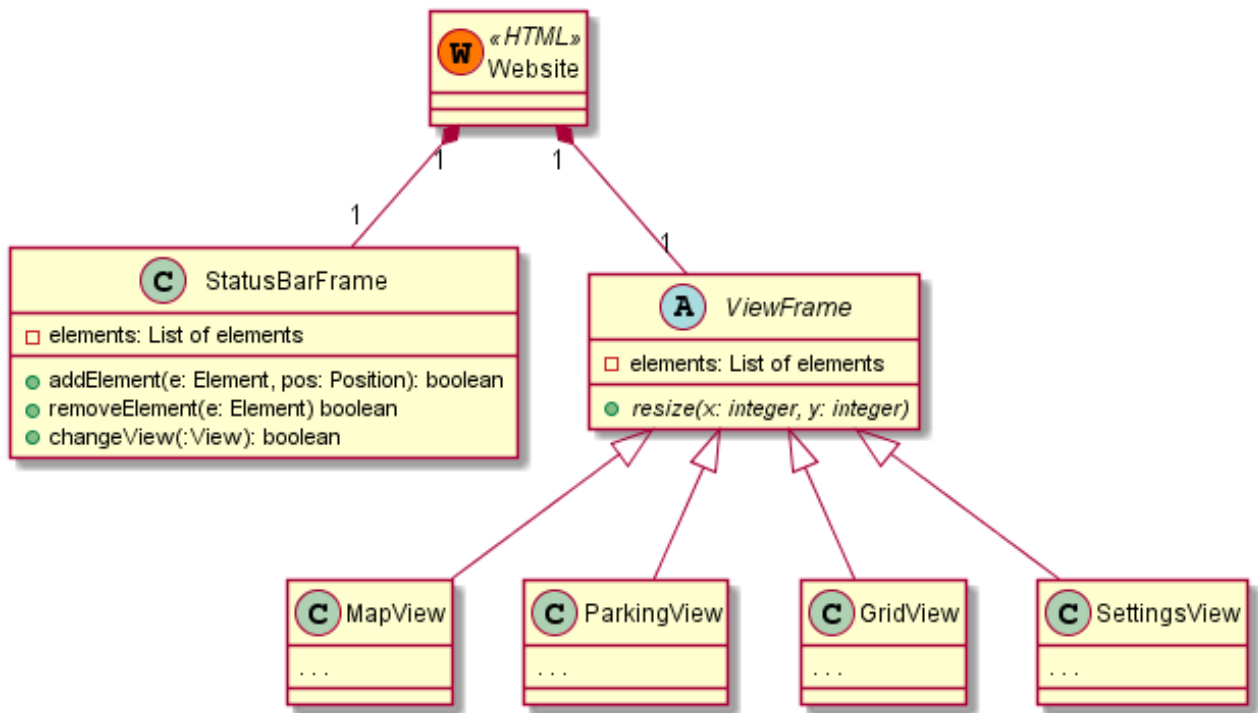


Abbildung 2.5: Klassendiagramm der GUI

## 2.6 User Interface

Wie in 2.5 zu sehen ist besteht ein Hauptteil des User Interface aus dem ViewFrame, der von verschiedenen Views vertreten werden kann. Hier sieht man den Aufbau des GridViews und des SettingViews. Das Grid nutzt ein Objekt der Klasse 'Gridster' aus der Bibliothek 'Gridster'. Das GridView besteht hauptsächlich aus diesem Objekt. Es ermöglicht die Anzeige verschiedener Widgets in einem Grid mit Elementen verschiedener Größe.

Hierbei sind die Widgets selbst Abonnenten des angezeigten Signals, weshalb sie die abstrakte Klasse BusDevice implementieren müssen.

Über die SettingsView ist es möglich, verschiedene Widgets in diesem Grid hinzuzufügen oder zu ändern. Aus diesem Grund speichert SettingView immer die Instanz des Grids als Objekt. Dies wird benutzt, wenn z.B. ein früherer Zustand des Grids wiederhergestellt werden soll. Um früher gespeicherte Konfigurationen vom Bus zu empfangen und neue Einstellungen an den Server zu schicken implementiert auch die SettingsView das BusDevice.

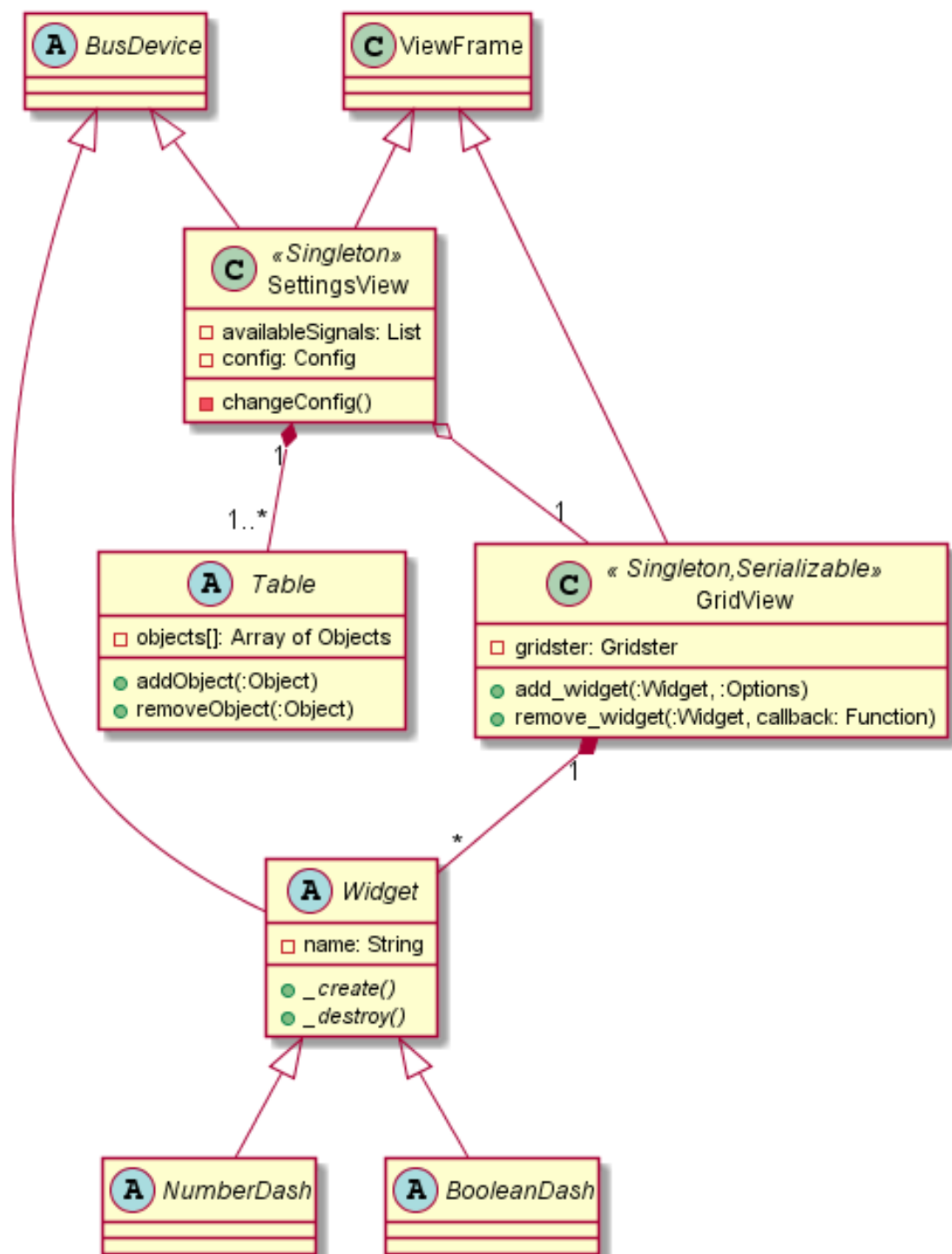


Abbildung 2.6: Klassendiagramm der GUI

## 2.7 Einparkhilfe Ultraschallsensoren

Keine Berechnungen sind auf dem Server für die Ultraschallsensoren notwendig. Die neuen Werte werden ständig via WebRTC auf das Endgerät geschickt und dort werden sie interpretiert. Auf dem Endgerät werden auch keine Berechnungen durchgeführt, es werden lediglich Messwerte (Distanz) in Farben umgewandelt (ProximityDisplayRefresher).

## 2.8 Einparkhilfe Lenkradposition

Die Berechnungen der Einparkhilfe werden auf dem Server durchgeführt (Thin-Client-Prinzip). Bei einer Änderung der Lenkradposition wird der Pfad (Path) neu berechnet. Der PathCalculator erzeugt einen neuen Pfad. Die PKW-bezogene Daten, die in Settings gespeichert sind, werden dafür verwendet (carWidth, ... usw.). Falls die Daten nicht vorhanden sind, werden die von der Datenbank geholt. Danach wird der Pfad projiziert (es wird ein Pfad mit den neuen Daten erzeugt) und wird berechnet, wie die Kurve auf dem Bildschirm angezeigt werden soll. Die benötigten Daten sind ebenfalls in Settings gespeichert (camPosition, ... usw.). Danach wird der Pfad auf den Bus gelegt und Richtung Endgerät weitergeleitet. Auf dem Endgerät erfolgen keine Berechnungen. Wenn die Daten (SVG-Path-Koordinaten oder Koordinaten für Bezierkurven) ankommen, wird das SVG-Bild neu gezeichnet (PathRefresher).



## 2.9 Einparkhilfe Serverseite

Im folgenden Diagramm ist der Aufbau des Rückfahrsistenzsystems auf der Serverseite dargestellt.

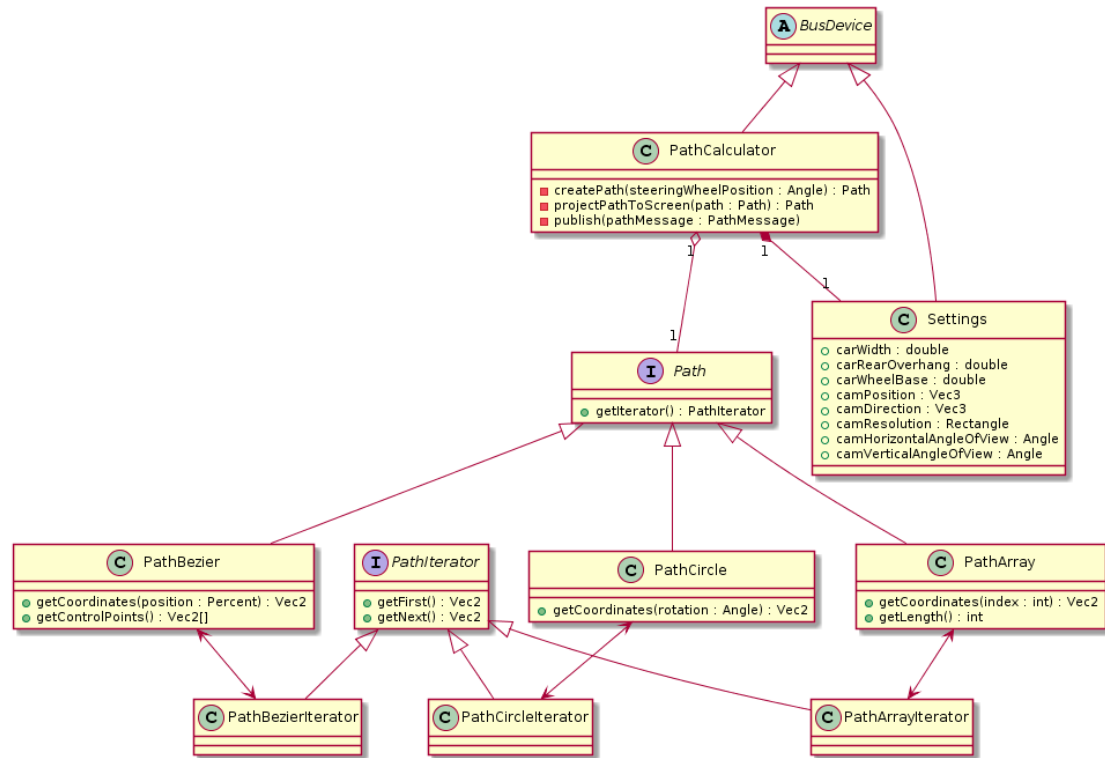


Abbildung 2.7: Klassendiagramm des Rückfahrsistenzsystems (1)

## 2.10 Einparkhilfe Clientseite

Im folgenden Diagramm ist der Aufbau des Rückfahrasistenzsystems auf der Clientseite dargestellt

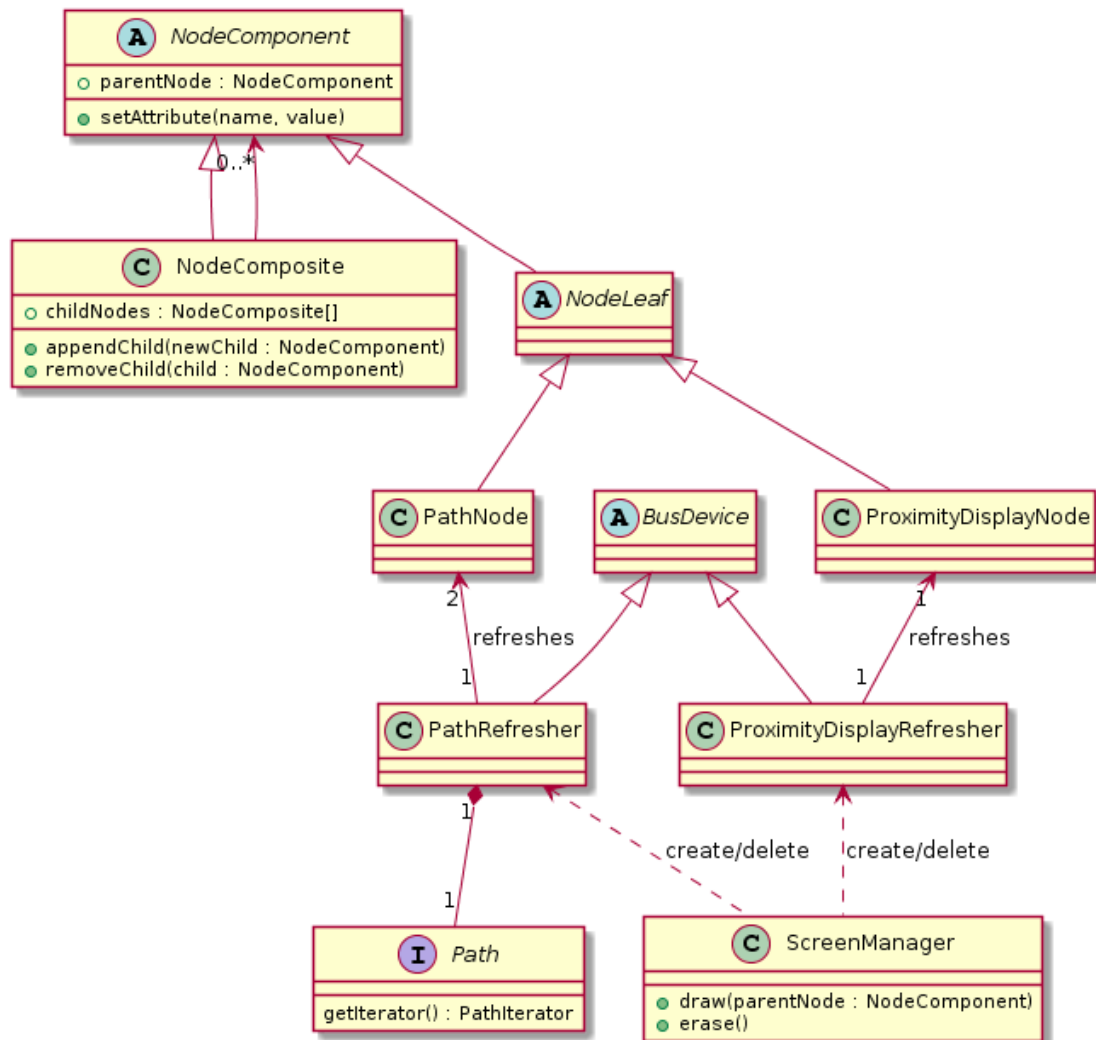


Abbildung 2.8: Klassendiagramm des Rückfahrasistenzsystems (2)

## 3 Klassen

### 3.1 Einparkhilfe Clientseite

#### 3.1.1 NodeComponent

Eine DOM-Komponente, es kann entweder Kompositum oder Blatt sein (Kompositum-Entwurfsmuster).

`public parentNode: NodeComponent` Der Elternknoten dieser NodeComponent.

`public setAttribute(name: var, value: var)` Setzt den Wert eines Attributs in NodeComponent.

**Parameter:** Der Name und der Wert des Attributs.

#### 3.1.2 NodeComposite

Ein DOM-Kompositum (Kompositum-Entwurfsmuster).

`public childNodes: NodeComposite[]` Die Liste aller Kindknoten.

`public appendChild(newChild : NodeComponent)` Fügt einen neuen Kindknoten hinzu.

`public removeChild(child : NodeComponent)` Entfernt einen bereits vorhandenen Kindknoten.

#### 3.1.3 NodeLeaf

Ein DOM-Blatt (Kompositum-Entwurfsmuster).

#### 3.1.4 PathNode

Ein DOM-Blatt, das einen Pfad auf dem Bildschirm anzeigt.

#### 3.1.5 PathRefresher

Zeichnet die zwei PathNodes neu, wenn die Lenkradposition geändert wird.

### 3.1.6 ProximityDisplayNode

Ein DOM-Blatt, das den Wert eines Ultraschallsensors auf dem Bildschirm visualisiert. Wenn es mehrere Ultraschallsensoren gibt, dann sind mehrere ProximityDisplayNodes notwendig.

### 3.1.7 ProximityDisplayRefresher

Aktualisiert das ProximityDisplayNode-Blatt, wenn der Wert des Sensors geändert wird.

### 3.1.8 ScreenManager

Verwaltet die gesamte Darstellung, zeichnet die grafische Elemente

`public draw(parentNode: NodeComponent)` Zeichnet alle Anzeigeelemente auf den Bildschirm und initialisiert die Webkamera.

**Parameter:** Der Elternknoten, wo die Anzeigeelemente gezeichnet werden sollen.

`erase()` Löscht alle Anzeigeelemente und stoppt die Webkamera-Videoübertragung.

## 3.2 Einparkhilfe Serverseite

### 3.2.1 Path

Das Interface für alle Pfade.

`public getIterator() : PathIterator` Holt den Iterator, der die aktuelle Implementierung von Path unterstützt. **Rückgabewert:** Ein PathIterator, der die Punkte dieses Pfades durchlaufen kann.

### 3.2.2 PathIterator

Das Interface für den Pfad-Iteratoren.

`public getFirst(): Vec2` Holt den ersten Punkt des Pfades. **Rückgabewert:** Die Koordinaten des Punktes.

`public getNext(): Vec2` Holt den nächsten Punkt des Pfades. **Rückgabewert:** Die Koordinaten des Punktes.

### 3.2.3 PathCircle

Eine Path-Implementierung, kann verwendet werden um effizient kreisförmige Pfade zu beschreiben. Damit können die einzelne Punkte des Pfades bei Bedarf einfach berechnet werden.

`public getCoordinates(rotation: Angle) : Vec2` Holt den Koordinaten eines Punktes, der sich auf dem Kreis befindet. Der Punkt ist wie folgt spezifiziert: es gibt ein Vektor, der vom Mittelpunkt des Kreises zu dem Pfad-Startpunkt zeigt (der Startpunkt ist auf der Kreislinie). Dieser Vektor wird dann gedreht (um `rotation`) und der Punkt zu dem der Vektor dann zeigt soll zurückgegeben werden. Der Iterator soll die Funktion falls `getFirst()` aufgerufen wurde mit `rotation=0` aufrufen. Bei `getNext()` soll der Winkel jeweils inkrementiert werden. **Parameter:** Die Drehung des Vektors. Alle mögliche Winkel sind gültig.  
**Rückgabewert:** Die Koordinaten des Punktes auf der Kreislinie.

### 3.2.4 PathCircleIterator

Ein Iterator, der PathCircle unterstützt.

### 3.2.5 PathArray

Eine Path-Implementierung, kann verwendet werden um allgemeine Pfade einfach speichern zu können. Die Punkte des Pfades werden als Polygonzug gespeichert.

`public getCoordinates(index: int) : Vec2` Holt den n-ten Punkt des Polygonzuges, falls `index = n`. **Parameter:**Spezifiziert den Punkt im Polygonzug. Falls der Polygonzug aus m Punkten besteht, dann sind die index-Werte zwischen 0 und m-1 gültig. Falls der index-Parameter ungültig ist, soll null zurückgegeben werden.  
**Rückgabewert:** Die Koordinaten des Punktes.

`public getLength()` Holt den Anzahl der Punkte im Polygonzug. **Rückgabewert:** Es wird der Anzahl der Punkte im Polygonzug zurückgegeben.

### 3.2.6 PathArrayIterator

Ein Iterator, der PathArray unterstützt.

### 3.2.7 PathBezier

Eine Path-Implementierung, kann verwendet werden um bei der Speicherung von Kurven Platz zu sparen.

`public getCoordinates(position: Percent) : Vec2` Holt die Koordinaten eines Punktes, der sich auf der Bezierkurve befindet. **Parameter:** Bezier-Parameter t der Kurve [0, 1]. **Rückgabewert:** Die Koordinaten des Punktes auf der Bezierkurve.

`public getControlPoints() : Vec2[]` Holt die Koordinaten aller Kontrollpunkten der Bezierkurve. **Rückgabewert:** Die Koordinaten aller Punkten auf der Bezierkurve.

### 3.2.8 PathBezierIterator

Ein Iterator, der PathBezier unterstützt.

### 3.2.9 Settings

Hier werden die Kraftfahrzeug- und Kamerabezogene Parameterwerte bei Bedarf von der Datenbank geholt und gepuffert.

```
public carWidth : double Die Breite des Fahrzeugs in Meter.
public carRearOverhang : double Der hintere Überhang des Fahrzeugs in Meter.
public carWheelBase : double Der Radstand des Fahrzeugs in Meter.
public camPosition : Vec3 Die Position der Kamera in Fahrzeugkoordinatensystem.
public camDirection : Vec3 Die Richtung der Kamera in Fahrzeugkoordinatensystem.
public camResolution : Rectangle Die Auflösung der Kamera in Pixeln.
public camHorizontalAngleOfView : Angle Der horizontale Winkel der Kamera.
public camVerticalAngleOfView : Angle Der vertikale Winkel der Kamera.
```

### 3.2.10 PathCalculator

Hört den Bus ab und falls die Position des Lenkrads geändert wurde, erzeugt einen neuen Pfad und leitet den an das Endgerät weiter.

```
private createPath(steeringWheelPosition : Angle) : Path Erzeugt einen Pfad
    der die Bewegung des Fahrzeugs in der Zukunft beschreibt. Der Pfad ist in Fahr-
    zeugkoordinatensystem. Die zwei gespeicherte Koordinaten sind entlang der x- und
    z-Achsen, die y-Koordinaten sind 0 für alle Punkte und müssen deswegen nicht ge-
    speichert werden. Parameter: Die Position des Lenkrads (Uhrzeigersinn). Rück-
gabewert: Der Pfad, der die Bewegung des Fahrzeugs in der Zukunft beschreibt.

private projectPathToScreen(path : Path) : Path Projiziert einen Pfad auf den
    Kamerabildschirm. Parameter: Der Pfad der Fahrzeugbewegung in Fahrzeugko-
    ordinatensystem. Rückgabewert: Der Pfad der Fahrzeugbewegung in Kamerako-
    ordinatensystem.

private publish(pathMessage: PathMessage) Sendet eine Nachricht mit dem Pfad
    an das Endgerät. Parameter: Die Nachricht, die den Pfad (in Kamerakoordina-
    tensystem) enthält.
```

## 3.3 Bus

### 3.3.1 BusDevice

Eine abstrakte Klasse, deren erbende Klassen Zugriff auf das Bus-Modul haben.

```
private messages: List of Messages Noch zu versendende Nachrichten
```

**public publish(message: Message): Boolean** Sendet eine Nachricht asynchron an eine Instanz von MessageBus.

**Parameter:** Die zu versendende Nachricht.

**Rückgabewert:** Boolean, der beschreibt, ob die Nachricht beim Bus angekommen ist.

**abstract public handleMessage(message: Message): Void** Funktion zum asynchronen Empfangen einer Nachricht von einer Instanz von MessageBus.

**public subscribe(object: BusDevice, topic: Topic): boolean** Funktion zum abonnieren von Nachrichten eines bestimmten Themas.

**Parameter:** Der Abonnent und das abonnierte Thema.

**Rückgabewert:** Boolean, der beschreibt, ob das Thema erfolgreich abonniert wurde.

**public unsubscribe(object: BusDevice, topic: Topic): boolean** Funktion zum Beenden eines Abonnements zu einem bestimmten Thema.

**Parameter:** Der Abonnent und das abonnierte Thema.

**Rückgabewert:** Boolean, der beschreibt, ob das Thema erfolgreich aus der Liste der Abonnements entfernt wurde.

**private createMessage(): Message** Funktion zum Erzeugen einer neuen Nachricht. Die Funktion sollte in den meisten Fällen von der erbenden Klasse überschrieben werden.

**Rückgabewert:** Die neu erzeugte Nachricht.

### 3.3.2 Message

Eine abstrakte Klasse, die als Muster für alle Nachrichten dient, die über den Bus versendet werden können.

**protected topic: Topic** Das Thema der Nachricht; wird zur Identifikation des Nachrichtentyps sowie des Inhalts der Nachricht benutzt.

**Inhalt** Der Inhalt der Nachricht, ist je nach erbender Klasse von einem unterschiedlichen Typ.

### 3.3.3 MessageBus

Die Klasse, die die Funktionalität des Busses beinhaltet.

**private broker: Broker** Der Broker des Busses; siehe Beschreibung der Broker-Klasse.

**protected subscribe(object: BusDevice, topic: Topic): Boolean** Funktion, die für eine Instanz von BusDevice auf ein Thema eine Subscription in der Broker-Instanz des MessageBus anlegt.

**Parameter:** Der Abonnent und das abonnierte Thema.

**Rückgabewert:** Boolean, der den Erfolg des Eintrags der Subscription angibt.

`protected unsubscribe(object: BusDevice, topic: Topic): Boolean` Funktion, die für eine Instanz von `BusDevice` auf ein Thema eine Subscription in der Broker-Instanz des `MessageBus` löscht, sofern diese existiert.

**Parameter:** Der Abonnent und das un-abonnierte Thema.

**Rückgabewert:** Boolean, der angibt, ob keine Subscription (mehr) von dem angegebenen Abonnenten auf das angegebene Thema im Broker eingetragen ist.

`protected handleMessage(message: Message): Void` Funktion, die asynchron eine Nachricht von einer Instanz von `BusDevice` empfängt, sich von der lokalen Broker-Instanz alle Subscriptions auf das Thema der Nachricht holt, und die Nachricht dann über einen Aufruf von `distribute(message: Message, subscribers: List of BusDevice)` an Instanzen von `BusDevice` verteilt.

**Parameter:** Die versendete Nachricht.

`private distribute(message: Message, subscribers: List of BusDevice): Void` Funktion, die eine Nachricht an alle angegebenen Instanzen von `BusDevice` versendet. **Parameter:** Die zu versendende Nachricht und eine Liste aller Instanzen von `BusDevice`, die die Nachricht bekommen sollen.

### 3.3.4 Broker

Die Klasse, die im Bus alle Abonnements hält und auf Bedarf an den `MessageBus` zurückgibt.

`getSubscribers(topic: Topic): List of Subscriptions` Funktion, die die Liste aller Abonnenten auf ein bestimmtes Thema zurückgibt.

**Parameter:** Das Thema, auf welchem alle Abonnenten zurückgegeben werden sollen.

**Rückgabewert:** Eine Liste aller Abonnenten auf das Thema.

`getSubscriptions(object: BusDevice): List of Topics` Funktion, die die Liste aller Abonnements einer bestimmten Instanz von `BusDevice` zurückgibt.

**Parameter:** Die Instanz von `BusDevice`, für die alle Abonnements zurückgegeben werden sollen.

**Rückgabewert:** Eine Liste aller Themen, für die die Instanz von `BusDevice` Abonnements hält.

`public subscribe(object: BusDevice, topic: Topic): Boolean` Funktion, die für eine Instanz von `BusDevice` auf ein Thema eine Subscription anlegt.

**Parameter:** Der Abonnent und das abonnierte Thema.

**Rückgabewert:** Boolean, der den Erfolg des Eintrags der Subscription angibt.

`public unsubscribe(object: BusDevice, topic: Topic): Boolean` Funktion, die für eine Instanz von `BusDevice` auf ein Thema eine Subscription löscht, sofern diese existiert.



**Parameter:** Der Abonnent und das un-abonnierte Thema.

**Rückgabewert:** Boolean, der angibt, ob keine Subscription (mehr) von dem angegebenen Abonnenten auf das angegebene Thema eingetragen ist.

### 3.3.5 Request

Eine "Familie" von Klassen, die zum Anfordern von Informationen benutzt werden. Requests werden in der Regel über Nachrichten versendet und beinhalten z.B. Anfragen an die Datenbank.

## 3.4 DBAccess

### 3.4.1 DBBusDevice

Die Klasse, die im Modul DBAccess den Zugriff auf den Bus ermöglicht und Nachrichten an die Klasse LevelDBAccess weiterleitet.

`public handleMessage(message: Message): Void` Überschriebene Funktion, die die eingegangene Nachricht dekodiert und als konkreten Request, konkrete Konfigurationsänderung oder DBEntry an die zugehörige Instanz von LevelDBAccess weiterleitet.

**Parameter:** Die eingegangene Nachricht

### 3.4.2 LevelDBAccess

Die Klasse, die im Modul die Instanz von LevelUP hält und den konkreten Zugriff auf die Datenbank ausführt.

`currentDriver: Driver` Der momentan als Fahrer eingetragene Benutzer

`maxCapacity: NaturalNumber` Die momentan gesetzte maximale Kapazität der Datenbank.

`database: LevelUP` Die Instanz von LevelUP, die als Datenbank für das System benutzt wird.

`protected put(timestamp: Timestamp, topic: Topic, entry: DBEntry): Boolean` Funktion, die eine Instanz von DBEntry mit einem Key aus Thema und Zeitstempel in der Datenbank ablegt.

**Parameter:** Der Zeitstempel des Eingangs der Nachricht im Modul, das Thema der Nachricht sowie der konkrete Datenbankeintrag.

**Rückgabewert:** Boolean, der angibt, ob der Eintrag in der Datenbank abgelegt wurde.

`protected put(driver: Driver, entry: DriverInfoEntry): Boolean` Funktion, die eine Instanz von `DriverInfoEntry` mit einem Benutzer als Key in der Datenbank ablegt.

**Parameter:** Der Benutzer, dessen Fahrerinformationen geändert bzw. gespeichert werden sollen, sowie der konkrete Eintrag in die Datenbank.

**Rückgabewert:** Boolean, der angibt, ob der Eintrag in der Datenbank abgelegt wurde.

`protected get(request: DBRequest): List of DBEntry` Funktion, die eine Instanz von `DBRequest` entgegennimmt und eine Liste von dem Request entsprechenden Datenbankeinträgen zurückgibt.

**Parameter:** Eine Instanz von `DBRequest`

**Rückgabewert:** Eine Liste aller `DBEntries` aus der Datenbank, die dem Request entsprechen.

`private deleteOnMaxCapacity(): Void` Funktion, die die in `maxCapacity` festgelegte Obergrenze der Datenbank überprüft und ggf. Einträge aus der Datenbank löscht.

### 3.4.3 DBEntry

Die abstrakte Klasse, deren erbende Klassen Datenbankeinträge darstellen. `SensorValueEntries` beinhalten die Werte von Sensorwerten und den zum Zeitpunkt der Messung eingetragenen Fahrer; `DriverInfoEntries` beinhalten die Konfiguration des Dashboards eines Benutzers sowie dessen präferierten Kraftstoff.

## 3.5 User Interface

### 3.5.1 Website

Da es sich um eine statische Website handelt, kann man dies nicht als Objekt betrachten. Es besitzt zwei Frames die mit den gewünschten JS oder TS-Elementen gefüllt werden können.

### 3.5.2 ViewFrame

Abstrakte Oberklasse für alle Views. Dazu gehören unter anderem das Dashboard, die Settings, die Karte und die Rückfahrkamera.

`private elements: List of elements` Liste aller angezeigten Elemente.

`abstract public resize(x: integer, y: integer): boolean` Ändert die Größe des Frames und passt die Größe aller darin befindlichen Elemente der neuen Größe des Frames an.

### 3.5.3 StatusBarFrame

Zweiter Anzeigeframe in Form einer Leiste am oberen Rand der Website.

`private elements: List of elements` Liste aller angezeigten Elemente.

`public addElement(e: Element, pos: Position): boolean` Fügt ein neues Anzeigeelement zur Leiste hinzu. **Parameter:** Mit dem Parameter `pos` lässt sich die Position des Elements `e` auf der Leiste bestimmen. **Rückgabewert:** Boolean der angibt, ob das Hinzufügen des Elements funktioniert hat.

`public removeElement(e: Element): boolean` Entfernt ein Anzeigeelement aus der Leiste. **Parameter:** Das zu entfernende Element `e`. **Rückgabewert:** Boolean der angibt, ob das Entfernen funktioniert hat.

### 3.5.4 GridView

Das Dashboard. Die gesamte Anzeige besteht aus diesem Grid. Da die Klasse serialisierbar ist, lässt sich sein Zustand speichern und wiederherstellen.

`private gridster: Gridster` Objekt des Plugins, welches das gesamte Grid darstellt.

`public addWidget(:Widget, :Options): boolean` Fügt ein Widget in das Grid hinzu. **Parameter:** Das hinzuzufügende Widget und Optionen (z.B. Positions- und Größenangaben). **Rückgabewert:** Boolean, der angibt, ob das Hinzufügen ein Erfolg war.

`public removeWidget(:Widget): boolean` Entfernt ein Widget vom Grid. **Parameter:** Das zu entfernende Widget. **Rückgabewert:** Boolean, der angibt, ob das Entfernen ein Erfolg war.

### 3.5.5 SettingsView

Die Einstellungsanzeige.

`private availableSignals: List` Liste aller verfügbaren Signale

`private config: config` Aktuelle Konfiguration im Config-File

`private changeConfig()` Wird aufgerufen wenn die Einstellungen angepasst werden. Übermittelt die neuen Einstellungen über den Bus an den Server.

### 3.5.6 Table

Abstrakte Klasse einer Tabelle. Die Klasse ermöglicht es, verschiedene Objekte in Zeilen und Spalten anzuzeigen. Generell soll immer eine ganze Zeile als Objekt hinzugefügt werden. Ein eigenes Zeilenobjekt muss dafür implementiert werden.

### 3.5.7 Widget

Entspricht den Vorgaben von JQuery, ein neues Widget zu erstellen. Diese werden für das GridView benötigt. Es existieren außer den beschriebenen Funktionen noch solche, die von JQuery bei bestimmten Aktionen aufgerufen werden.

`private name: String` Bezeichner für das Widget

`abstract public _create()` Wird beim Erzeugen des Widgets aufgerufen. Zur Initialisierung von Variablen u.ä.

`abstract public _destroy()` Wird beim Löschen des Widgets aufgerufen.

### 3.5.8 NumberDash

Überklasse für alle Dashes, die einen Zahlenwert anzeigen.

### 3.5.9 BooleanDash

Überklasse für alle Dashes, die einen Wahrheitswert anzeigen.

### 3.5.10 MapView

Eine Karteanzeige.

`map: Map` Die anzuzeigende Karte.

`updateMap()` Die Karte wird aktualisiert.

### 3.5.11 MapWithGasView

Eine Karteanzeige mit den Tankstellen in der Umgebung; besteht aus einer Liste der Tankstellen und der Karte

### 3.5.12 MapWithPOIView

Eine Karteanzeige mit den POI; besteht aus einer Liste der POI und der Karte.

### 3.5.13 ListPOI

Eine Liste der POI.

### 3.5.14 ListGasStation

Eine Liste der Tankstellen.

### 3.5.15 POIItem

Der Eintrag in der POI-Liste.

```
private name: String Name der POI
private address: String Adresse der POI
private distance: float Der Abstand zu POI
```

### 3.5.16 GasStationItem

Der Eintrag in der Liste für die Tankstellen.

```
private name: String Name der Tankstelle
private address: String Adresse der Tankstelle
private price: float Der Preis vom voreingestellten Brennstoff in der Tankstelle
private distance: float Der Abstand zu der Tankstelle
```

## 3.6 Konfiguration und erhältliche Signale

### 3.6.1 ConfigFileReader

Ermöglicht das Lesen und Schreiben einer Konfigurations-Datei über den Datenbus. Es reagiert auf bestimmte RequestMessages.

```
private configFile: File Die zu bearbeitende und zu lesende Datei

setConfig(name: String, value: Value) Verändert eine Einstellung. Parameter: Der
    Name der Einstellung sowie der neue Wert dieser Einstellung.
    Rückgabewert: Boolean das angibt ob es diese Einstellung bereits gibt.

getConfig(name: String) Liest eine Einstellung aus. Parameter: Der Name der Ein-
    stellung.
    Rückgabewert: Der Wert der Einstellung oder null.
```

### 3.6.2 SignalCollector

Registriert, welche Informationen erhältlich sind und reagiert auf bestimmte Request-Messages, entweder durch Hinzufügen oder Auslesen dieser Signale. Ein Beispiel ist hier das die OBD2-Schnittstelle, die auf Anfrage die gesamte Liste seiner Signale auf den Bus legen kann.

```
signalsList Die Liste der erhältlichen Signale.
```

## 3.7 Karte

### 3.7.1 Button

Die abstrakte Klasse enthält die Lokalisierungsinformation des Autos und bekommt die Information über Tankfüllstand vom Bus. Es kann auch die Karte der Umgebung von Google Maps heruntergeladen und die Karte angezeigt werden.

`private gasLevel: double` Der aktuelle Tankfüllstand.

`private location: location` Die Lokalisierungsinformation des Engeräts des Fahrers.

`downloadMap()` Die Karte der Umgebung wird heruntergeladen.

`view()` Die heruntergeladene Karte wird angezeigt.

### 3.7.2 Map

Enthält die Karte der Umgebung.

### 3.7.3 Decorator

Die Decorator-Klasse ist nach Entwurfsmuster Decorator entworfen. Diese Klasse dient dazu, in einem gewissen Kontext eine spezielle Karte(z.B die Karte mit Tankstelle oder POI) herunterzuladen.

### 3.7.4 GasDecorator

In einem GasDecorator-Objekt wird die Karte mit den Tankstellen in der Umgebung heruntergeladen.

`downloadMap()` Die Karte, auf der die Tankstellen der Umgebung markiert sind, wird heruntergeladen.

### 3.7.5 POIDecorator

In einem POIDecorator-Objekt wird die Karte mit interessanten Positionen in der Umgebung heruntergeladen.

`downloadMap()` Die Karte, auf der die interessanten Positionen der Umgebung markiert sind, wird heruntergeladen.

## 4 Sequenzdiagramme

### 4.1 Client mit Server verbinden

In dem folgenden Diagramm sieht man den Ablauf des Verbindens eines Endgeräts mit dem Server. Auf die Ausführung des Verbindens folgt in der Regel "Livedaten und Konfiguration"

In diesem Fall handelt es sich beim Terminal um das Endgerät, auf dem über einen Browser die VINJAB-Seite aufgerufen wird. Der Befehl `openWebpage()` soll das Aufrufen der Website darstellen. Es handelt sich sozusagen um eine Anfrage über HTTP nach den genutzten Dateien und Programmteilen. Bei der Referenz für die WebRTC-Connection wird "WebRTC-Verbindung" ausgeführt.

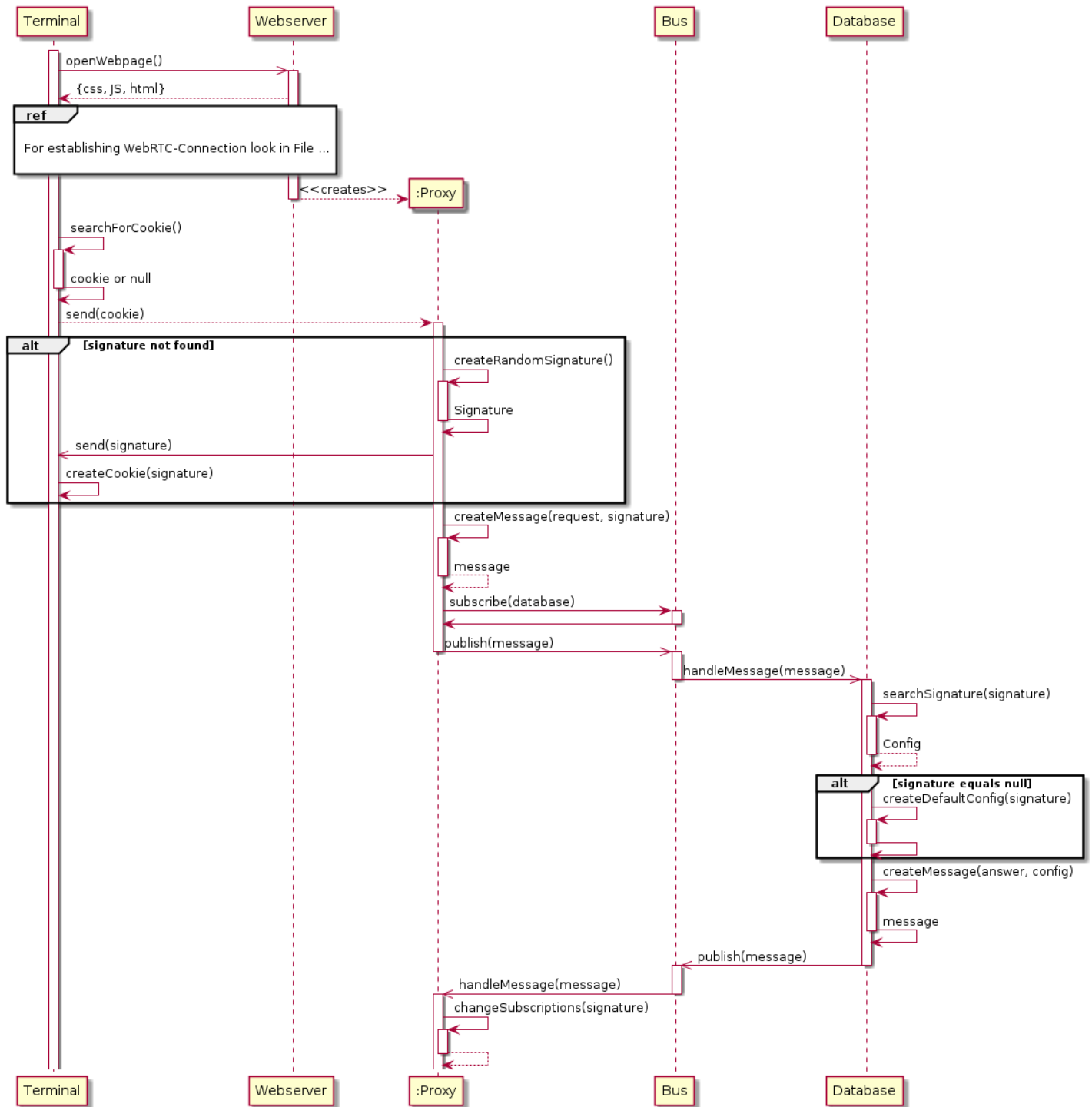


Abbildung 4.1: Herstellen einer Verbindung zwischen Client und Server



## 4.2 WebRTC-Verbindung

In dem folgenden Diagramm wird dargestellt, wie eine WebRTC-Verbindung zwischen dem Endgerät und dem Webserver hergestellt wird. Bei unserem Projekt initiiert der Client die Verbindung.

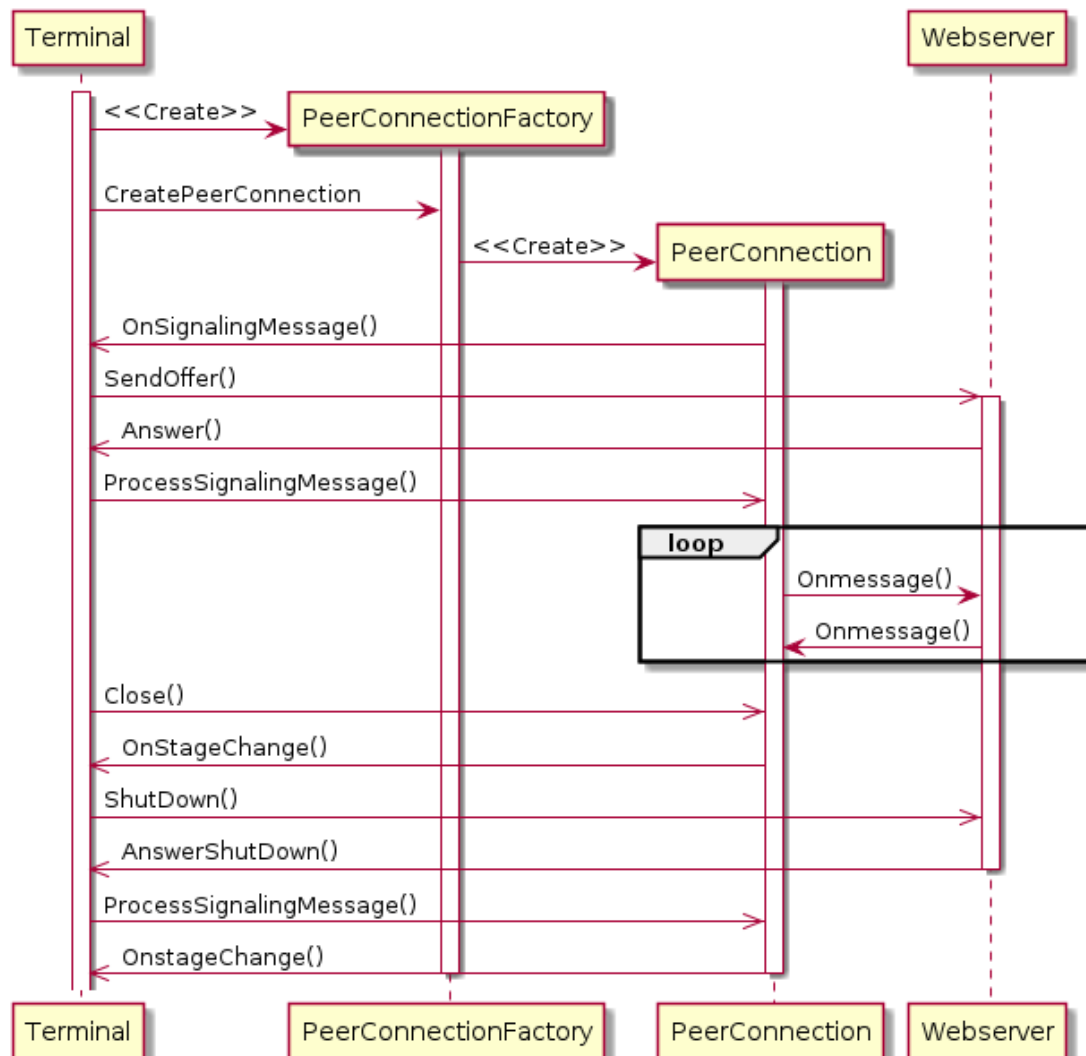


Abbildung 4.2: Herstellen einer WebRTC-Verbindung

### 4.3 Livedaten und Konfiguration

Nach hergestellter Verbindung befindet sich das System in einer Schleife, in der eigentlich nur noch Live-Daten vom Server zum Client geschickt werden. Nur wenn der Nutzer die Anzeigeeinstellungen ändert, wird diese Schleife für kurze Zeit unterbrochen, um die neue Konfiguration an den Server zu übermitteln, damit von diesem keine nicht benötigten Daten mehr versendet werden und die Konfiguration permanent gespeichert werden kann.

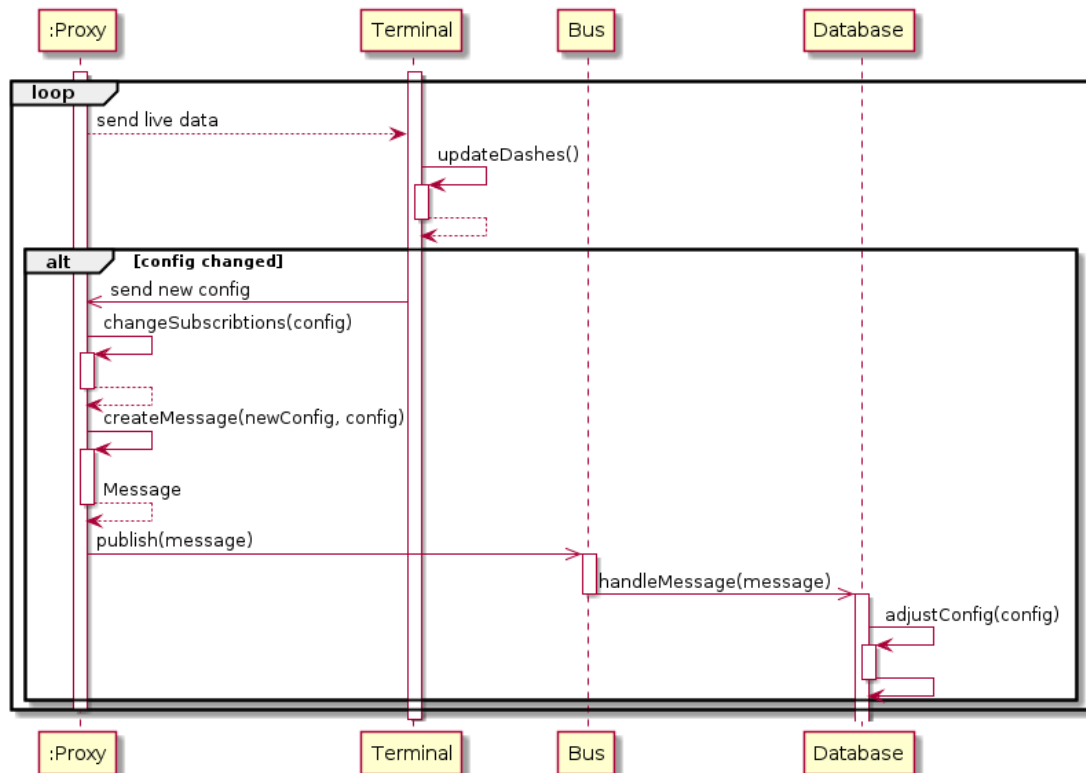
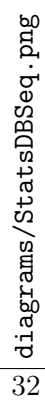


Abbildung 4.3: Livedatenübertragung und Konfigurationsänderung

## 4.4 Statistik übertragen

### 4.4.1 Aggregierte Funktionen anfordern

Der User fordert ein Dash zur Anzeige an, welches bisher noch nicht angezeigt wurde. Über die Peer Connection von WebRTC wird eine "RequestMessage" an den Server geschickt. Auf Serverseite wird vom entsprechenden TerminalProxy eine RequestMessage für den entsprechenden Wert auf den Bus gelegt. Der Erzeuger verarbeitet die Nachricht und veröffentlicht ab sofort die erzeugten Werte auf dem Bus. Handelt es sich beim Erzeuger um einen virtuellen Sensor, muss dieser noch einen Request an die Datenbank senden, sofern ein Diagramm oder Durchschnittswert gefragt ist. Die Datenbank hat eine Fassade, die ein Bus Device ist.



diagrams/StatsDBSeq.png