Entwurfsdokument - VINJAB: VINJAB Is Not Just A Boardcomputer

Jonas Haas David Grajzel Nicolas Schreiber Valentin Springsklee
Yimeng Zhu

20. Dezember 2015

Inhaltsverzeichnis

0.1 Legende

Vor allem in den Klassendiagrammen dieses Dokuments haben wir uns dazu entschlossen verschiedene Konventionen zur Übersicht und zum Verhindern von redundanten Informationen zu verwenden. Die selbst definierten sind in der folgenden Tabelle zu sehen.

Zeichen	Beschreibung
. . .	Eine Klasse mit nur drei Punkten ist in einem anderen Diagramm weiter erklärt.
:Object	z.B. als Parameter einer Funktion. Hier wäre der Name des Parameters praktisch der selbe wie der Typ.
Getter/Setter	Bei Prototypen werden wegen Übersichtlichkeit offensichtliche Variablen, Getter/Setter und Funktionen wie z.B. getInstance bei Signleton nicht extra dazugeschrieben

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 'BusAccess' erweitern.

Alle Nachrichten sind Objekte von Klassen welche die abstrakte Klasse 'Message' erweitern und implementieren. Diese ist Serializable, also auch die Unterklassen. Hier wurden abstrakte Klassen gewählt da mehrere Funktionen zentral implementiert werden andere jedoch, wie z.B. Getter und Setter, von dem Nachrichtentyp abhängen.

Die Rohnachrichten kommen unter anderem von dem Blutooth-Modul, dass 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' das für die Verbindung zwischen Server und Client zuständig ist.
Die subscribten Daten werden vom Proxy über die PeerConnection an das Endgerät verschickt und wenn Daten von der PeerConnection, also vom Endgerät, ankommen werden diese auf den Bus gelegt.

Aggregierte Funktionen und virtuelle Sensoren sind Instanzen einer Klasse 'VirtualSensor'. Sie abonnieren die benötigten Signale, berechnen neue Werte sobald die Signale ankommen, und legen diese dann als neues Signal wieder auf den Bus.

Zentral gespeichert werden alle Daten in einer LevelDB-Datenbank. Ihr vorgeschaltet ist eine Klasse, die eine Schnittstelle zwischen Datenbank und Bus darstellt. In dieser werden alle Signale und Konfigurationsnachrichten abonniert und dann in der Datenbank gespeichert. Außerdem empfängt diese Klasse auch Befehle über den Bus und kann dann z.B. gewünschte Daten aus der Datenbank auf den Bus legen.

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

Dashes sind dann Subscriber des von ihnen angezeigten Signals. Sie erweitern die Klasse Widget

1.2 Chain of Responsibility

Die Werte der Sensoren und die Werte der aggregierten Funktionen können sich zu beliebigen Zeitpunkten ändern. Diese sind als subscribable Value gespeichert und so wie

es im Publish-Subscribe-Bus Entwurfsmuster vorgesehen ist, werden alle Objekte, die als Beobachter vorhanden sind bei Änderungen der abonnierten Werte benachrichtigt. Die Beobachter sind die Endgerät-Proxys. Wenn diese eine Nachricht empfangen (es sind neue Werte vorhanden), schicken Sie diese Information nach dem Chain of Responsibility Prinzip weiter an die PeerConnection. Da die Nachrichten alle Serializable sind kann sehr einfach eine Zeichenkette aus dem Objekt erzeugt werden. Die PeerConnection erhält dann diese Zeichenketten und leiten sie über die bestehende Netzwerkverbindung (sei es entweder LAN oder Internet) weiter. Dafür wird der RTC Data Channel genutzt. Die ankommende Zeichenketten auf dem Endgerät werden durch das Serializable wiederhergestellt und das Endgerät kann die Nachrichtenobjekte behandeln. Die werden genauso dargestellt, wie die losgeschickt worden (vom Endgerät-Proxy). Der Bus leitet dann alle Informationen an die passenden Dashes oder die Einstellungen weiter.

2 Klassendiagramme

2.1 Bus

In diesem Diagramm ist der Aufbau des Moduls ”Bus” zu sehen. Die Kommunikation aller Module 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 ”BusAccess” erbt. Der Bus funktioniert als eine Variation des Beobachter-Entwurfsmusters. Eine von ”BusAccess” erbende Klasse kann über subscribe() bzw. unsubscribe() auf dem ”MessageBus” Nachrichten eines bestimmten Topics (Sensortyp, Datenbanknachrichten, Konfigurationsnachrichten) abonnieren.
Diese Abbonements werden in der Klasse ”Broker” verwaltet. Sendet ein Objekt von ”BusAccess” 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 Oberklasse ”Message” besitzt eine Reihe von erbenden 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 Datum, und DBReqRange fordert alle Datenbankeinträge zwischen zwei Daten an.
Das Konstrukt der Klassen ”BusAccess” sowie erbende Klassen, ”Message” und die von ”Message” erbenden Klassen ergeben gemeinsam das Entwurfsmuster der Fabrikmethode.

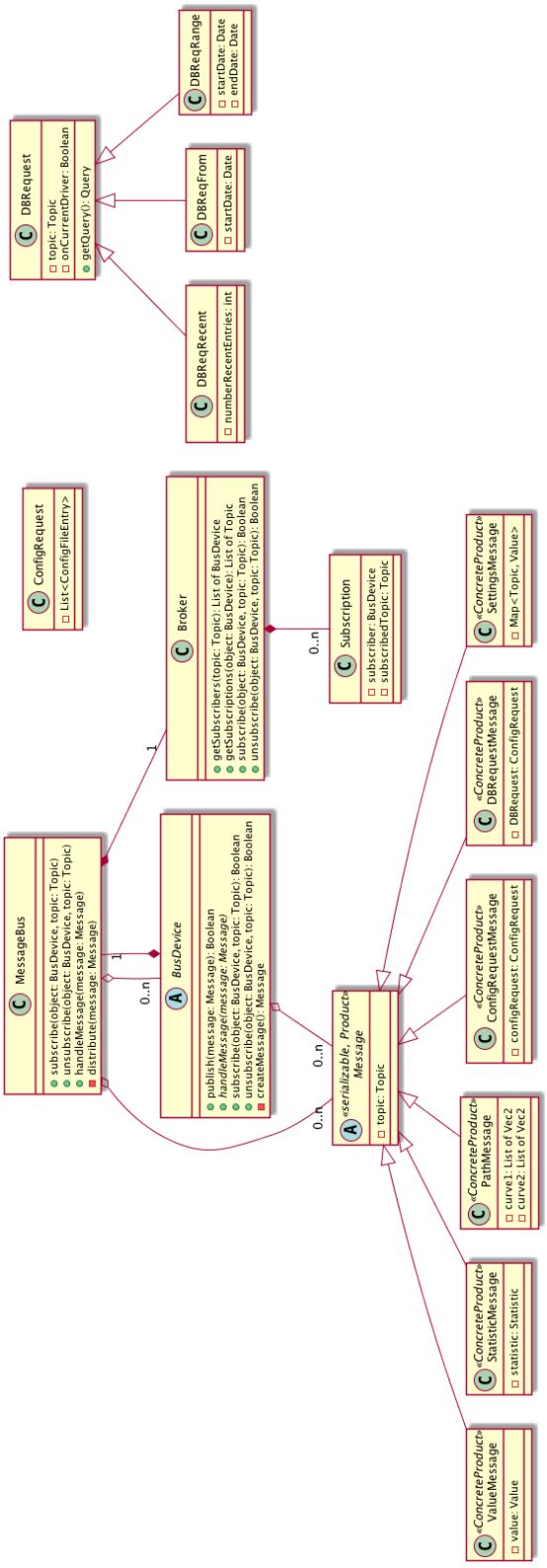

Abbildung 2.1: Bus Klassendiagramm

2.2 Virtual Sensors

Hier sieht man den Aufbau der Virtuellen Sensoren.

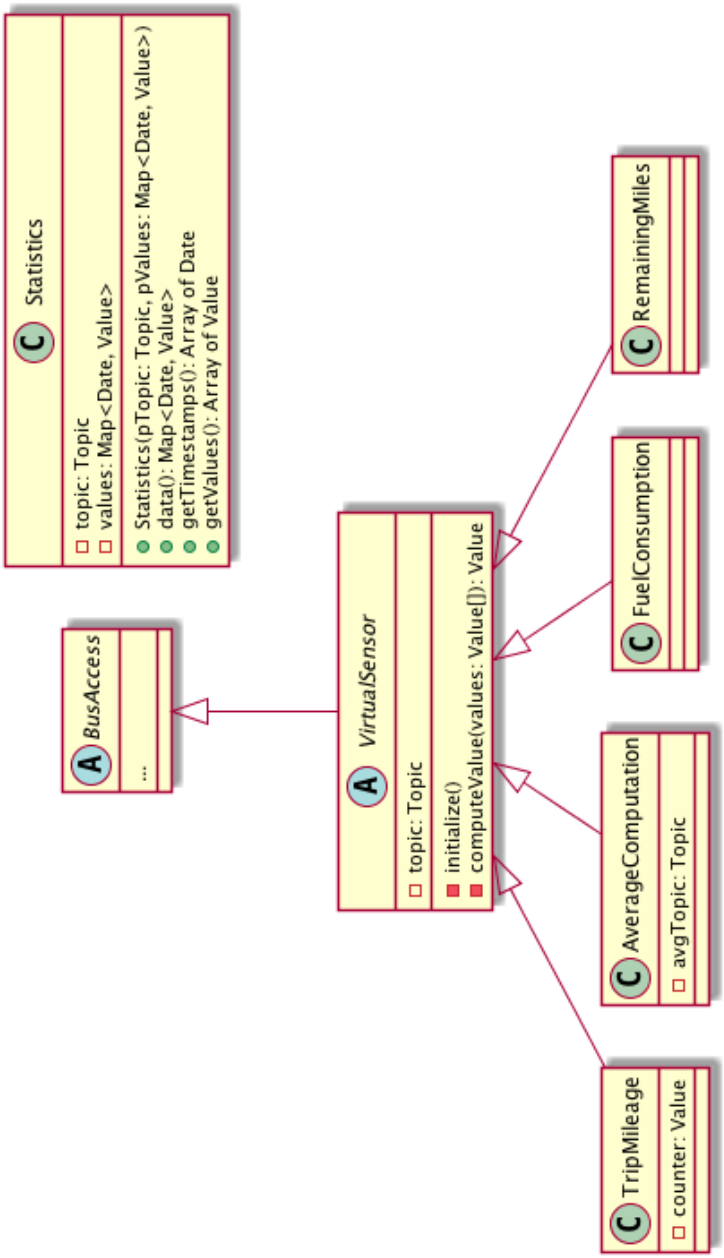

Abbildung 2.2: Virtuelle Sensoren Klassendiagramm

2.3 Database Access

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

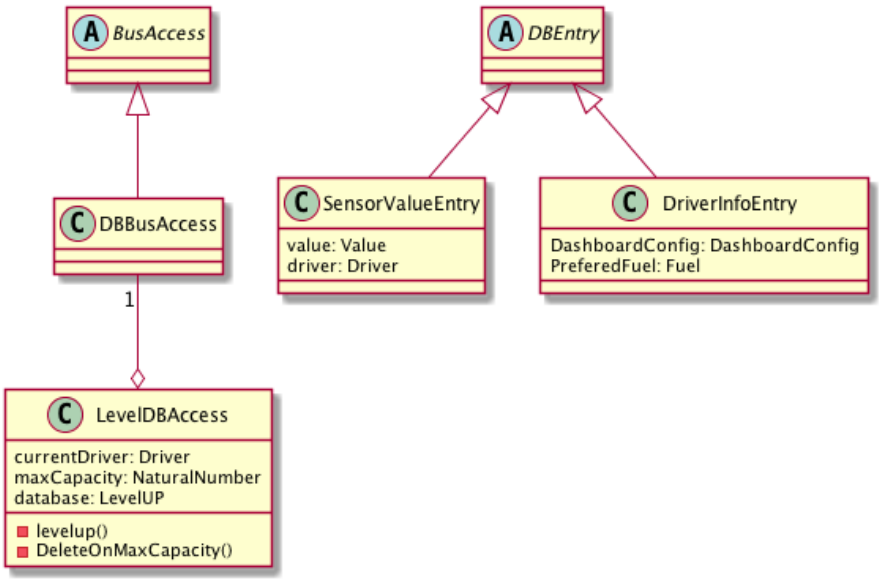

Abbildung 2.3: Virtuelle Sensoren Klassendiagramm

Als Datenbank wurde, wie im Pflichtenheft spezifiziert, LevelDB benutzt. Die Access-Klasse für LevelDB erbt von einer ”DBAccess”-Oberklasse, um leichte Austauschbarkeit zu gewährleisten. In der LevelDBAccess-Klasse wird außer der Referenz auf die LevelUp-Instanz auch der aktuelle Fahrer sowie die spezifizierte maximale Kapazität der Datenbank gehalten.
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 Fahrer gespeichert, zum anderen fahrerbezogene Einträge, die präferierten Kraftstoff und eine Referenz auf die Konfiguration des virtuellen Armaturenbretts mit der Identifikation des Fahrers als Key gespeichert werden. Zur Kommunikation mit anderen Modulen besitzt der Datenbankzugriff mit DBBusAccess eine Klasse, die den Zugang zum Bus möglich macht und Nachrichten vom Bus dekodiert.

2.4 Website

Hier ist zu sehen wie die Seite allgemein aufgebaut ist. Die Website besteht aus zwei Frames. In einem wird die Statusbar angezeigt, im anderen werden können verschiedene Views sichtbar sein. Unter anderem z.B. das GridView auf dem die Dashes angezeigt werden. Für weiteres siehe: TODO Die Statusbar kann aus mehreren Buttons, Text und Icons bestehen.

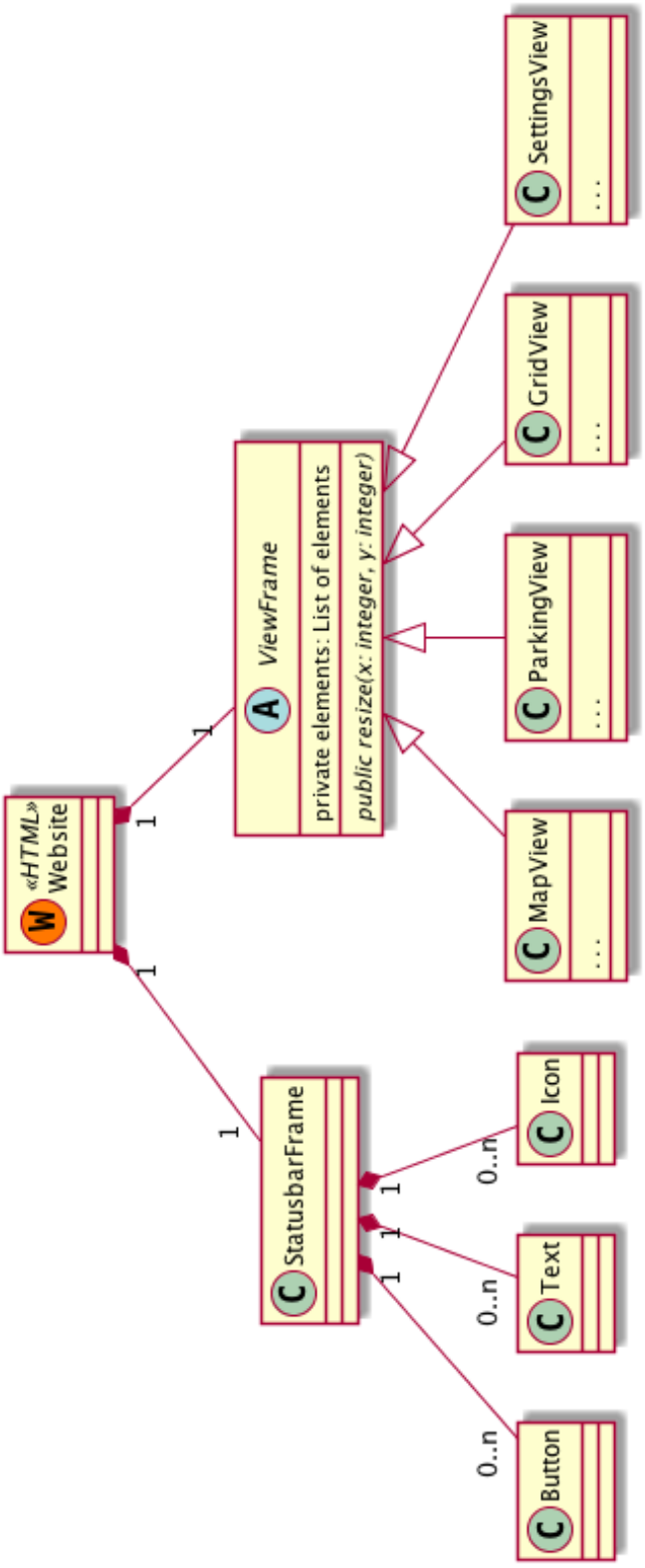

Abbildung 2.4: UI Klassendiagramm

2.5 User Interface

Wie in 2.4 zu sehen ist besteht ein Hauptteil des UserInterface aus dem ViewFrame der von verschiedenen Views vertreten werden kann. Hier sieht man den Aufbau des GridViews und des SettingViews. Das Grid nutzt eine Objekt der Klasse Gridster die aus der Bibliothek Gridster stammt. Das GridView besteht hauptsächlich aus diesem Objekt. Es ermöglicht verschiedene Widgets in einem Grid mit Elementen verschiedener Größen dynamisch anzeigen zu lassen.

Hierbei sind die Widgets selbst Abonnenten des angezeigten Signals, weshalb diese die Abstrakte Klasse BusAccess implementieren müssen.

Über die SettingsView ist es möglich verschiedene Widgets in dieses Grid hinzuzufügen oder zu ändern. Weshalb sich das SettingView immer die Instanz des Grids als Objekt speichert. Dies kommt zum Einsatz wenn z.B. ein früherer Zustand des Serializable Grids wiederhergestellt werden soll. Um früher gespeicherte Konfigurationen vom Bus zu empfangen und neue Einstellungen an den Server zu schicken implementiert auch die SettingsView den BusAccess.

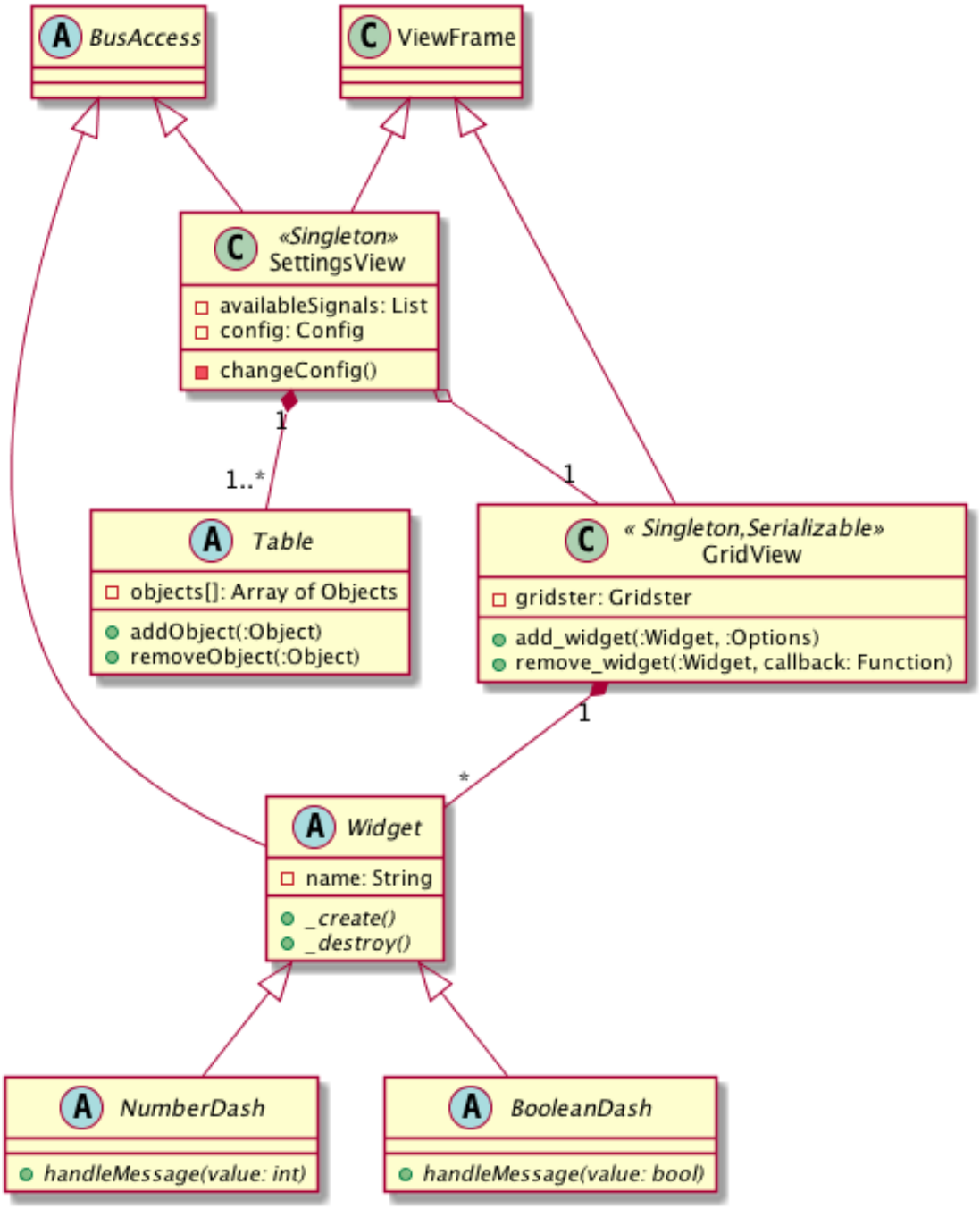

Abbildung 2.5: UI Klassendiagramm

2.6 Parking Sensor Serverside

Hier ist zu sehen wie das Rückfahrassistenzsystem aufgebaut ist. Siehe auch

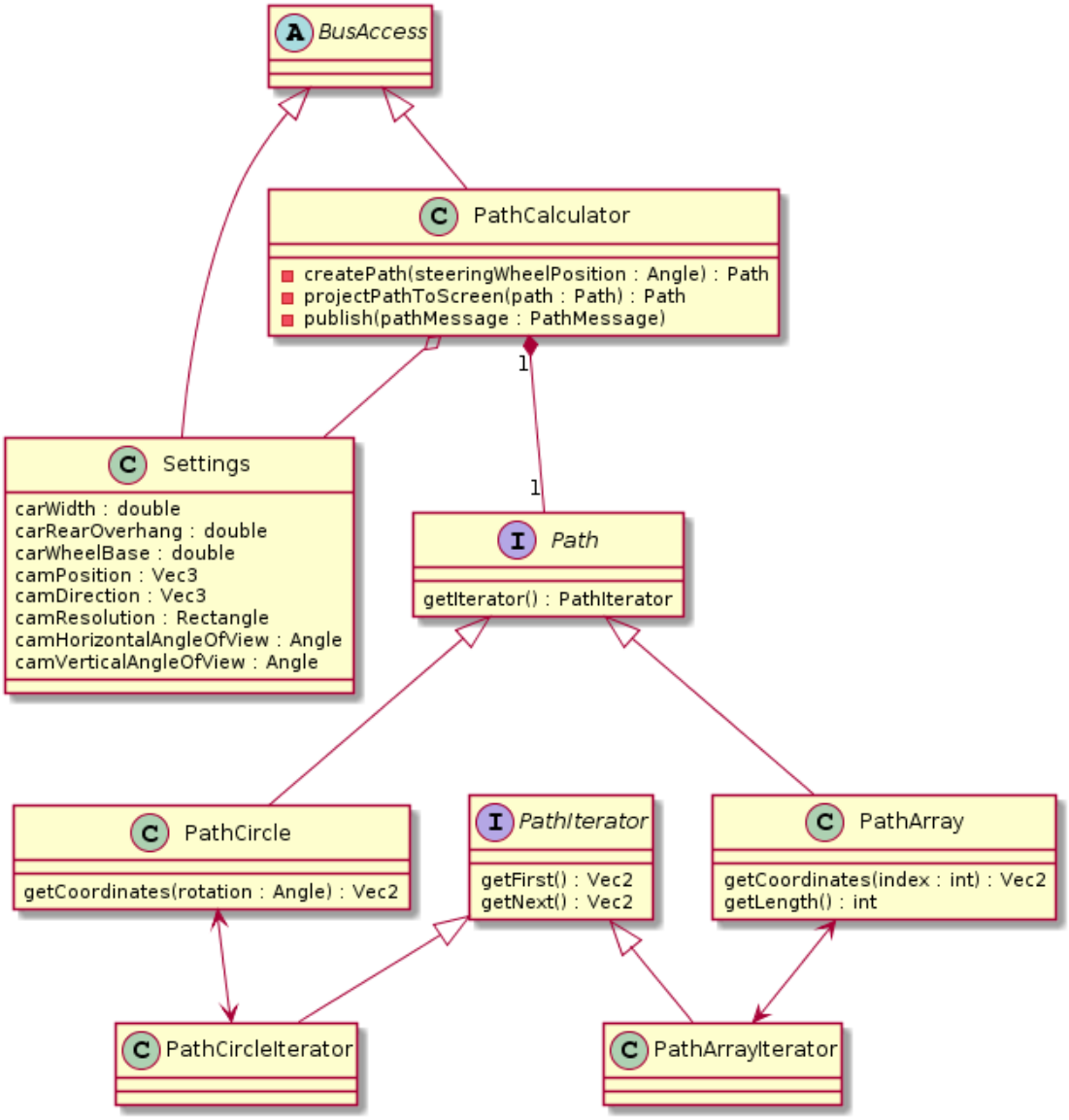

Abbildung 2.6: Rückfahrsystem Klassendiagramm

2.7 Parking Sensor Clientside

Hier ist zu sehen wie das Rückfahrassistenzsystem aufgebaut ist. Siehe auch

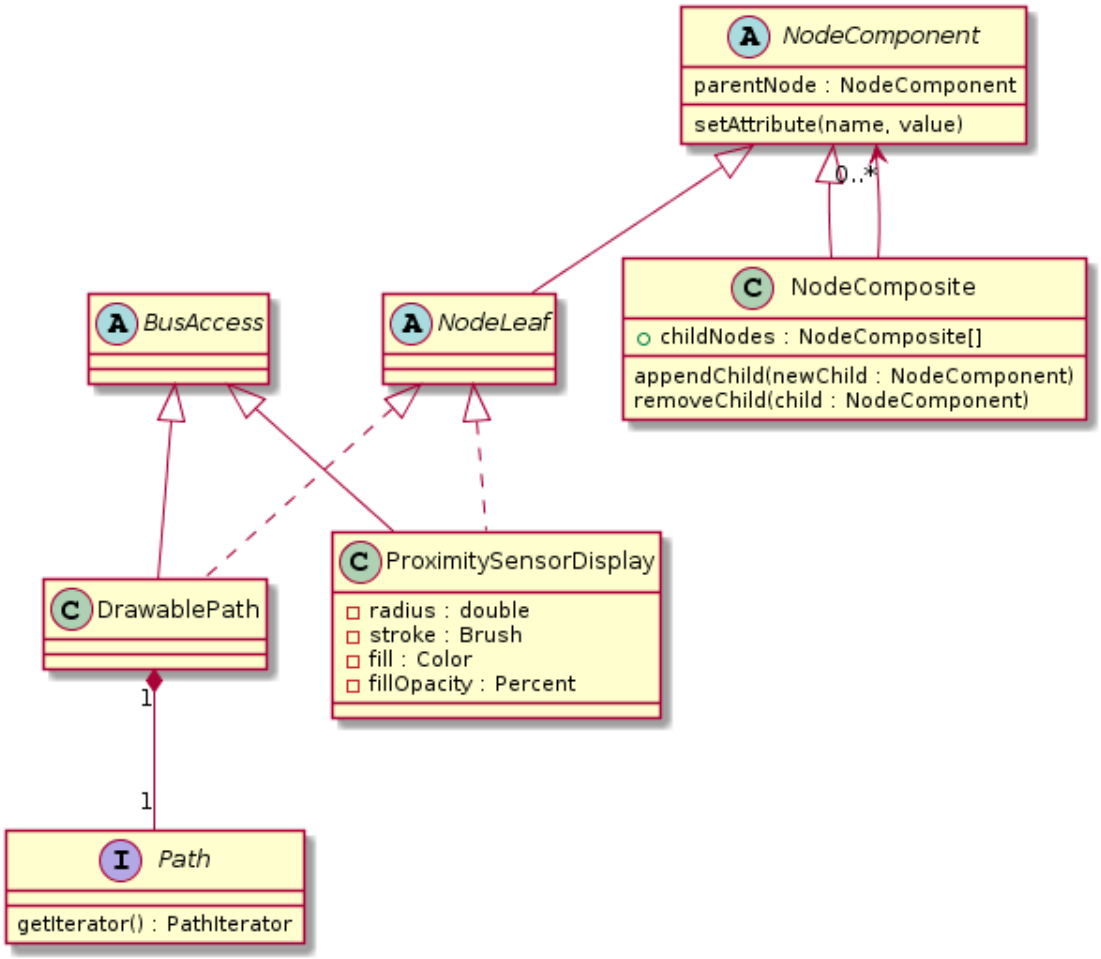

Abbildung 2.7: Rückfahrsystem Klassendiagramm

3 Klassen

3.1 Bus

3.1.1 BusAccess

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: BusAccess, 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: BusAccess, 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.

3.1.2 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.

3.1.3 MessageBus

Die Klasse, die die Funktionalität des Busses beinhaltet.

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

`public subscribe(object: BusAccess, topic: Topic): Boolean` Funktion, die für eine Instanz von BusAccess 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.

`public unsubscribe(object: BusAccess, topic: Topic): Boolean` Funktion, die für eine Instanz von BusAccess 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.

`handleMessage(message: Message): Void` Funktion, die asynchron eine Nachricht von einer Instanz von BusAccess 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) an Instanzen von BusAccess verteilt.
Parameter: Die versendete Nachricht

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

3.1.4 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: BusAccess): List of Topics` Funktion, die die Liste aller Abonnements einer bestimmten Instanz von BusAccess zurückgibt.
Parameter: Die Instanz von BusAccess, für die alle Abonnements zurückgegeben werden sollen.
Rückgabewert: Eine Liste aller Themen, für die die Instanz von BusAccess Abonnements hält.

`public subscribe(object: BusAccess, topic: Topic): Boolean` Funktion, die für eine Instanz von BusAccess 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: BusAccess, topic: Topic): Boolean` Funktion, die für eine Instanz von BusAccess 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.

3.1.5 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.

3.2 User Interface

3.2.1 Website

Da dies eine statische Webseite darstellt kann man es nicht als Objekt ansehen. Es besitzt zwei Frames die mit den gewünschten JS oder TS-Elementen gefüllt werden können.

3.2.2 ViewFrame

Abstrakte Superklasse für alle Views. 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, welches automatisch auch alle darin angezeigten Elemente anpasst.

3.2.3 StatusBarFrame

Zweiter Anzeigeframe. Eine 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 funktioniert hat.

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

3.2.4 GridView

Das Dashboard. Die gesamte Anzeige besteht aus einem Grid. Als Serializable 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 das angibt ob das Hinzufügen ein Erfolg war.

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

3.2.5 SettingsView

Die Einstellungsanzeige. Die Anzeige besteht aus mehreren Tabellen die bei Auswahl eines Elements eine Tiefergelegene öffnet.

`private availableSignals: List` Liste aller erhältlichen 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.

3.2.6 Table

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

3.2.7 Widget

Entspricht den Vorgaben von JQuery ein neues Widget zu erstellen. Diese werden für das GridView benötigt. Es existieren außer die beschriebenen Funktionen weitere Mögliche 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.

3.2.8 NumberDash

Überklasse für alle Dashes die einen Zahlenwert anzeigen.

3.2.9 BooleanDash

Überklasse für alle Dashes die einen Wahrheitswert anzeigen.

3.3 Konfiguration und erhältliche Signale

3.3.1 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 Einstellung. **Rückgabewert:** Der Wert der Einstellung oder null falls die Einstellung nicht existiert.

3.3.2 SignalCollector

Sammelt die Information welche Signale erhältlich sind. Reagiert auf bestimmte Request-Messages welche entweder das Hinzufügen oder das Auslesen dieser Signale ermöglicht.

`signalsList` Die Liste der erhältlichen Signale.

4 Sequenzdiagramme

4.1 Verbindung zwischen Client und Server herstellen

In dem folgenden Diagramm sieht man den Ablauf vom Verbinden eines Endgeräts mit dem Server bis zur Live-Übertragung der Signale und die Änderung der Anzeigekonfiguration auf dem Endgerät.

Hierbei handelt es sich beim Terminal um das Endgerät auf dem über einen Browser die Vinjab-Seite aufgerufen wird. Der Befehl openWebpage() vom Terminal zum Webserver nutzt das HTT-Protokoll.

Das Hypertext Transfer Protocol (HTTP, englisch für Hypertext-Übertragungsprotokoll) ist ein Protokoll zur Übertragung von Daten auf der Anwendungsschicht über ein Rechnernetz und gehört dementsprechend zur Internetprotokollfamilie. Es wird hauptsächlich eingesetzt, um Webseiten (Hypertext-Dokumente) aus dem World Wide Web (WWW) in einen Webbrowser zu laden. Es ist jedoch nicht prinzipiell darauf beschränkt und auch als allgemeines Dateiübertragungsprotokoll sehr verbreitet.

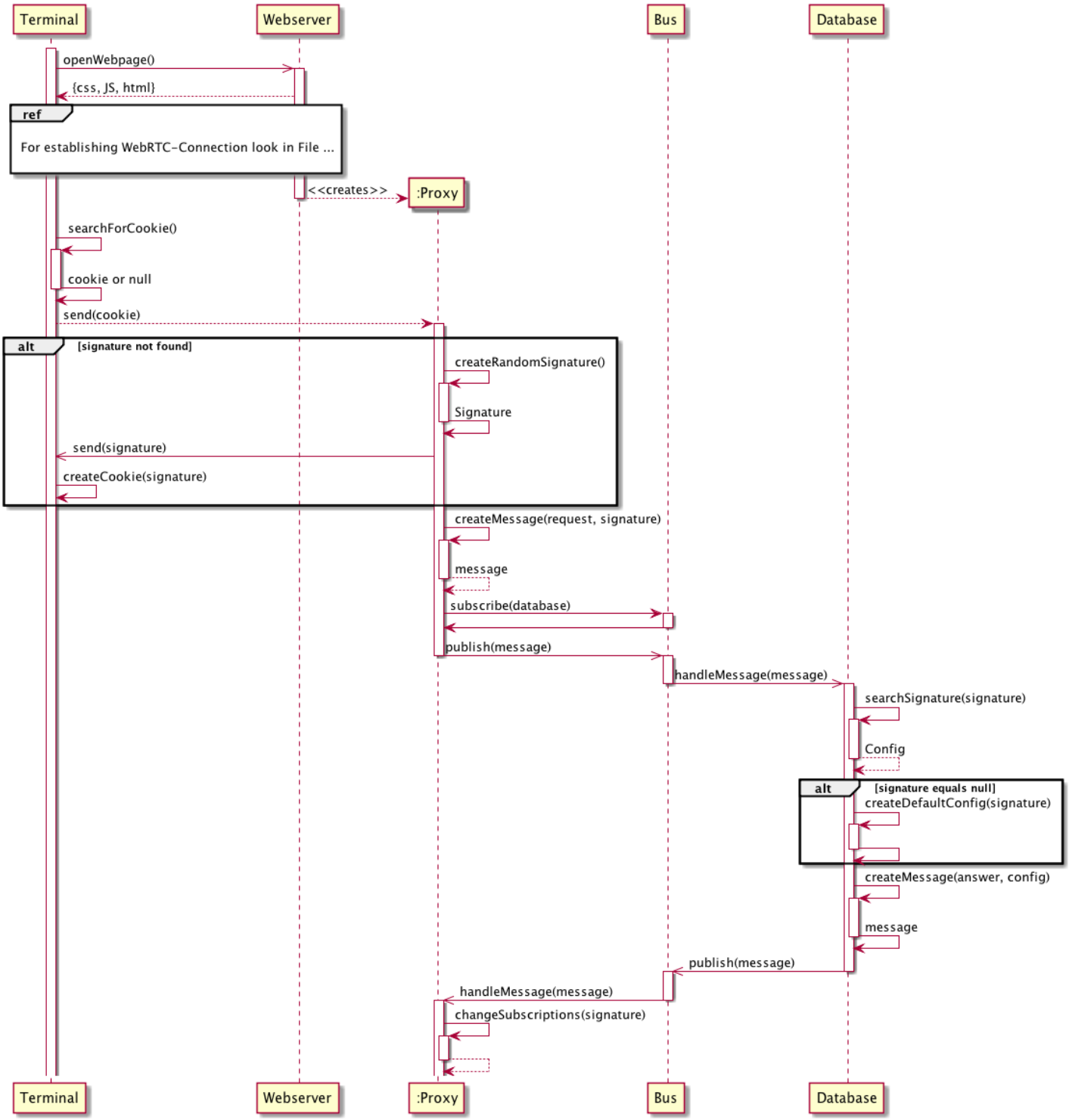

Abbildung 4.1: Verschiedene Dash-Anzeigeelemente.

4.2 WebRTC Verbindung herstellen

In dem folgenden Diagramm wird dargestellt wie eine WebRTC-Verbindung zwischen dem Endgerät und dem Webserver hergestellt wird.

WebRTC ist eine Sammlung von Kommunikationsprotokollen und Programmierschnittstellen (API) für die Implementierung in Webbrowsern, die diesen Echtzeitkommunikation über Rechner-Rechner-Verbindungen ermöglichen. Damit können Browser nicht mehr nur Datenressourcen von Backend-Servern abrufen, sondern auch (Echtzeitinformationen) von Browsern anderer Benutzer.

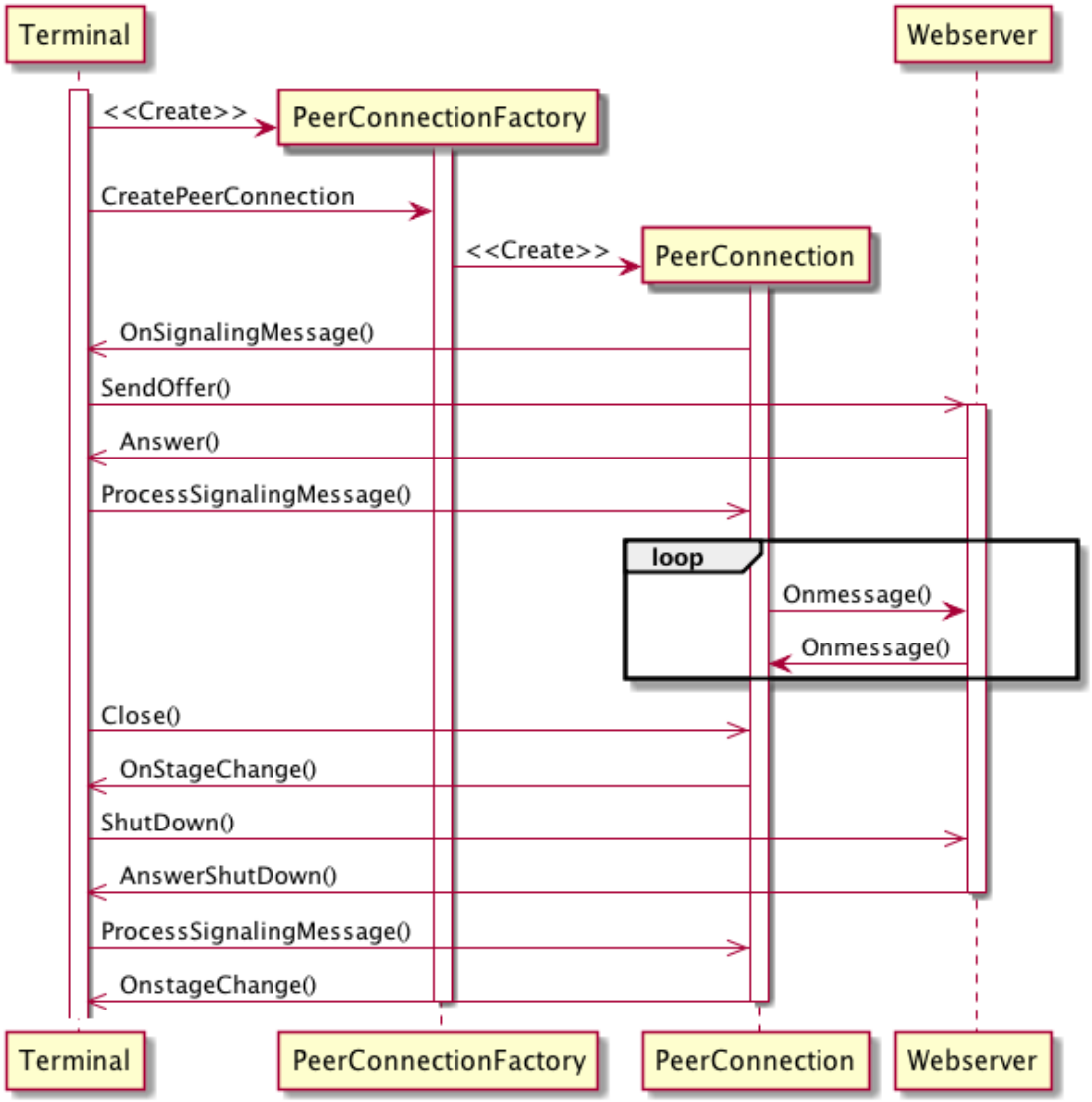

Abbildung 4.2: Verschiedene Dash-Anzeigeelemente.

4.3 Changed config

Nach hergestellter Verbindung befindet sich das System in einer Schleife in der eigentlich nur noch Live-Daten vom Server zum Client geschickt wird. Nur wenn die Anzeigeeinstellungen der Dashes geändert wurden

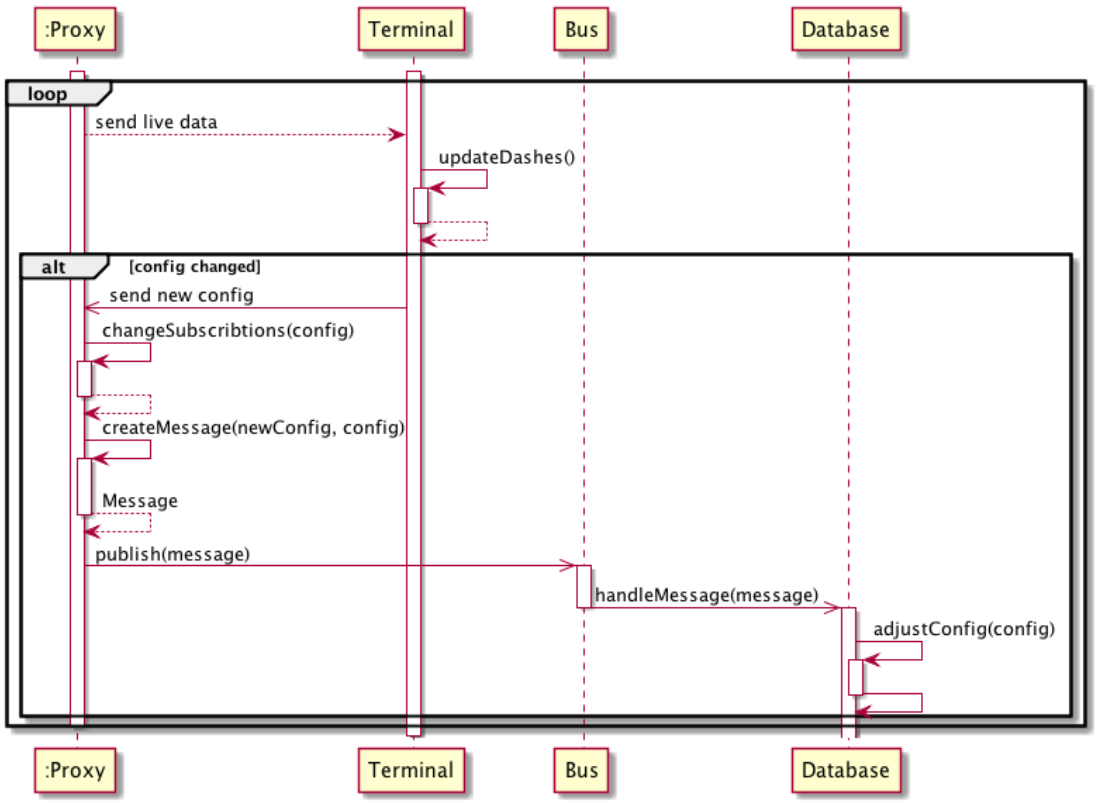

Abbildung 4.3: Verschiedene Dash-Anzeigeelemente.

4.4 Statistik übertragen

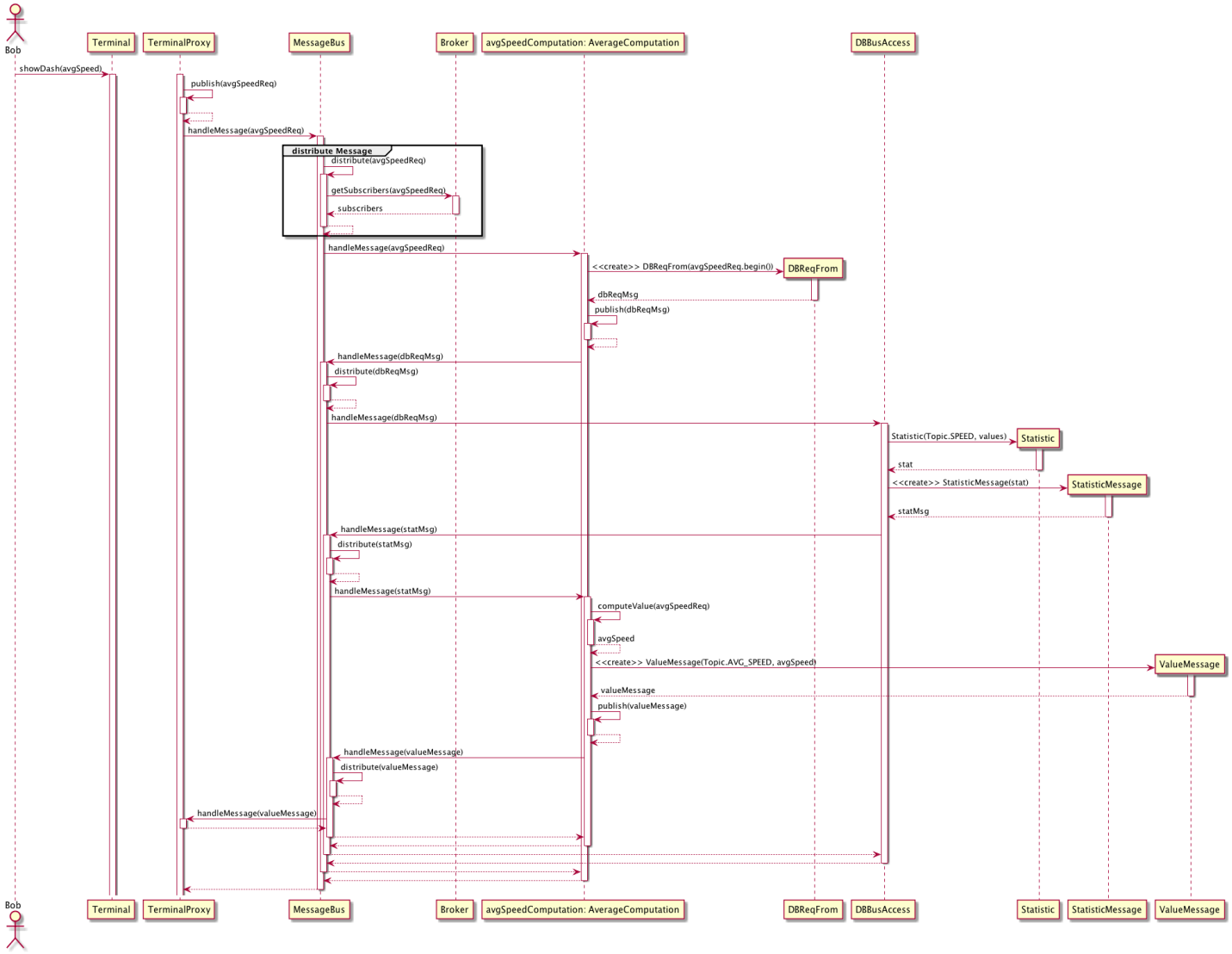

Abbildung 4.4: Verschiedene Dash-Anzeigeelemente.