Rheinisch Westfälische Technische Hochschule Aachen
Lehrstuhl für Software Engineering

Unterstützung der Fahrzeugdiagnose in Werkstätten durch Wearable Devices

Masterarbeit

von
Carlé, Marcel

1. Prüfer: Prof. Dr. Bernhard Rumpe

2. Prüfer: Prof. Dr. Jan Borchers

Betreuer: Klaus Müller, M.Sc.

Diese Arbeit wurde vorgelegt am Lehrstuhl für Software Engineering

Aachen, den 7. März 2017

Eidesstattliche Versicherung

Carlé, Marcel	320848
Name, Vorname	Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende ~~Arbeit/Bachelorarbeit/~~ Masterarbeit* mit dem Titel

Unterstützung der Fahrzeugdiagnose in Werkstätten durch Wearable Devices

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, 7. März 2017	
Ort, Datum	Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Aachen, 7. März 2017	
Ort, Datum	Unterschrift

Kurzfassung

Die vorliegende Masterarbeit bindet Wearable Devices zur Unterstützung der Werkstattmitarbeiter während einer Fahrzeugdiagnose in den Diagnoseprozess ein. Die Wearable Devices sollen die Diagnosedaten des Fahrzeugs in Echtzeit widerspiegeln und es den Werkstattmitarbeitern somit ermöglichen, in jeder Situation Zugriff auf diese Daten zu erhalten. Dazu wurden zwei bestehende Softwareprodukte der Firma DSA Daten- und Systemtechnik GmbH Aachen um eine Client-Server-Architektur erweitert und die Wearable Devices als Remote-Displays in die Diagnoseabläufe integriert. Besonderes Augenmerk wurde dabei auf eine einfache Anbindung der Wearable Devices in bereits bestehende Diagnoseabläufe gelegt. Mit Hilfe der Google Glass wurde das implementierte Konzept zum Schluss an zwei Beispieldiagnosen erprobt.

Abstract

The master thesis at hand focuses on the integration of wearable devices in vehicle diagnostics in order to support the mechanics. The wearable devices are setup in such a way that they display the diagnostic data of the vehicle in real time. Furthermore, they make the data accessible for the mechanic in any situation. To implement this process, two software products of the company DSA Daten- und Systemtechnik GmbH Aachen needed to be extended by a client-server-architecture. Additionally, the wearable devices had to be integrated into the diagnostic process as remote displays. For the implementation it was of great importance to design the integration process of wearable devices into the existing diagnostic procedures as easy as possible to make it easily extendable for future devices. The implemented concept was tested in two examples using the Google Glass.

Aufgabenstellung

In dieser Masterarbeit soll die bestehende Systemlandschaft rund um die Fahrzeugdiagnose der Firma DSA Daten- und Systemtechnik GmbH Aachen erweitert werden. Die Softwareprodukte der DSA GmbH richtet sich an Fahrzeughersteller. Diese setzen in den Werkstätten Notebooks als Diagnosegerät ein, auf denen die Diagnosesoftware PRODIS.WTS arbeitet. Mit PRODIS.WTS untersuchen Werkstattmitarbeiter, anhand von festgelegten Diagnoseabläufen, die Fahrzeuge auf Fehler. Die Diagnoseabläufe werden dabei im Vorfeld über die Autorensoftware PRODIS.Authoring vom Fahrzeughersteller erstellt.

Bei einer Fahrzeugdiagnose müssen die Werkstattmitarbeiter häufig zwischen dem Fahrzeug und dem Notebook wechseln. Dadurch wird der Diagnoseablauf kurz Unterbrochen was einen entsprechenden Zeitverlust zur Folge hat. Auch kann das Wechseln problematisch sein, wenn der Werkstattmitarbeiter gleichzeitig mit dem Diagnosegerät und dem Fahrzeug interagieren muss. Daher soll diese Arbeit eine Einbindung von Wearable Devices in die bestehenden Diagnoseabläufe ermöglichen. Somit soll sich das Wechseln von Fahrzeug zu Notebook weitestgehend erübrigen und es dem Werkstattmitarbeiter ermöglichen, in jeder Situation auf die Diagnosedaten zuzugreifen und mit ihnen interagieren zu können.

In dieser Arbeit soll PRODIS.WTS für die Einbindung der Wearable Devices erweitert und für bestehende Diagnoseabläufe in PRODIS.Authoring eine einfache Einbindung von Wearable Devices entwickelt werden. Die Wearable Devices sollen die Diagnosedaten vom PRODIS.WTS drahtlos erhalten und mit dem ausgeführten Diagnoseablauf interagieren können. Ein wichtiges Kriterium ist eine möglichst einfache Einbindung der Wearable Devices in bestehende Diagnoseabläufe, damit möglichst wenig Hürden für die Anbindung von Wearable Devices für die Fahrzeughersteller entstehen. Das erarbeitete Konzept soll anschließend in die bestehende Software implementiert werden und mit Hilfe einer Datenbrille, der Google Glass, sollen die Erweiterungen zum Schluss in einer Demoumgebung auf Funktionstüchtigkeit erprobt werden. Dazu muss eine Applikation für die Google Glass implementiert werden, welche die vom PRODIS.WTS bereitgestellten Daten verarbeiten, darstellen und mit dem Diagnoseablauf interagieren kann.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In dieser Arbeit wird die Einbindung von Wearable Devices in bestehende Diagnosesysteme und -abläufe betrachtet. Bisher müssen Werkstattmitarbeiter, im weiteren Verlauf der Arbeit *Werker* genannt, bei einer Fahrzeugdiagnose zwischen Fahrzeug und einem Diagnosegerät wechseln. Ein Diagnosegerät ist in vielen Fällen ein Notebook, welches auf einem Werkstattwagen platziert ist und die Diagnosedaten vom Fahrzeug auswertet. Bei schwierig zu erreichenden Stellen oder innerhalb des Fahrzeugs hat der Werker also meist keine Möglichkeit, mit dem Diagnosegerät zu interagieren. Dies wird in dieser Arbeit mit Hilfe von Wearable Devices, also am Körper tragbare elektronische Geräte, ermöglicht, welche dem Werker Ausschnitte der Diagnosedaten darstellen und es ihm ermöglichen, mit diesen zu interagieren.

Im Folgenden wird die Motivation der Arbeit anhand einer Beispieldiagnose genauer dargelegt und im Anschluss der Aufbau kurz beschrieben.

1.1 Motivation

Um die Motivation dieser Arbeit zu erläutern, wird im Folgenden eine beispielhafte klassische Fahrzeugdiagnose ohne den Einsatz von Wearable Devices gezeigt. Die dort vorgestellten Arbeitsabläufe sind teilweise recht umständlich und lassen sich mit den neuen Ansätzen dieser Arbeit deutlich vereinfachen. Das Beispiel wird in der Arbeit an einigen Stellen erneut aufgegriffen, um auf die auftretenden Probleme und deren Lösungen näher eingehen zu können.

Als Beispiel wird hier eine Überprüfung der Radsensorik herangezogen. In dieser soll überprüft werden, ob die Antiblockiersystem (ABS) Sensoren an den ihnen zugehörigen Rädern angebracht sind und sie die Drehgeschwindigkeit der Räder korrekt erfassen. Angenommen ein solcher ABS Sensor wurde falsch konfiguriert, falsch angebracht oder liefert falsche Daten, so würde das ABS System nicht korrekt arbeiten können. Im schlimmsten Fall würde es, statt für mehr Fahrsicherheit zu sorgen, falsche Schlüsse aus den Daten ziehen und entweder reagieren, obwohl keine Gefährdung vorliegt, oder nicht reagieren, obwohl eine Gefährdung vorliegt.

Nachdem der Werker sein Diagnosegerät, über welches er Sensordaten, Fehlercodes und andere Fahrzeugdaten dargestellt bekommt, mit dem Fahrzeug verbunden hat und sich in der darauf installierten Diagnosesoftware die Daten der ABS Sensoren ansehen kann, muss

er überprüfen, ob die Sensoren funktionieren. Dazu muss er sich zu jedem Rad begeben und es manuell drehen und auf seinem Diagnosegerät überprüfen, ob die Daten korrekt sind. Korrekt bedeutet zum Beispiel, dass sich für das Fahrzeug nicht irrtümlich ein anderes Rad dreht oder die Drehgeschwindigkeit falsch oder gar negativ ist, weil der Sensor falsch herum angebracht wurde.

Der Werker muss von jedem Rad aus sein Diagnosegerät einsehen können, um überprüfen zu können, ob alle Daten korrekt sind. Die eingesetzten Diagnosegeräte in Werkstätten ermöglichen eher stationäres Arbeiten, da Geräte wie Notebooks oder PCs Strom benötigen und somit häufig an eine Steckdose gebunden sind. Dadurch hat der Werker während einer Diagnose nicht immer freien Blick auf das Diagnosegerät, weil zum Beispiel das Fahrzeug den Blick versperrt. Auch ist es möglich, dass der Werker einige Meter von seinem Diagnosegerät entfernt ist und somit die Daten nicht mehr ablesen kann. Je nachdem wie die Software des Diagnosegeräts arbeitet, könnte es auch sein, dass der Werker zum Beispiel nach der Überprüfung jedes Rads mit der Oberfläche interagieren muss, damit aufgezeichnet werden kann, dass der Werker die Diagnose korrekt durchgeführt hat. Ein weiteres Szenario wäre, dass die Daten vom nächsten Rad erst nach der Bestätigung des aktuellen Rads angezeigt werden. Je nach Größe des Fahrzeugs, kann das Pendeln zwischen den Rädern und dem Diagnosegerät somit erheblich Zeit in Anspruch nehmen. Alternativ kann sich der Werker Unterstützung durch einen Kollegen holen, welcher entweder die Räder dreht oder die Daten auf dem Diagnosegerät kontrolliert. Dadurch steigen allerdings die Kosten der Diagnose für die Werkstatt, da nun zwei Werker mit dem Prozess beschäftigt sind.

Mit Hilfe von mobilen Diagnosegeräten kann die beschriebene Problematik abgeschwächt werden. Handelt es sich zum Beispiel um ein Tablet oder Notebook, welches entweder mit langen Kabeln ausgestattet ist oder drahtlos mit dem Fahrzeug kommuniziert, so kann der Werker das Gerät zu den Rädern mitnehmen. Bei der Verwendung von Kabeln kann es allerdings passieren, dass die Kabel trotzdem zu kurz sind oder sich zum Beispiel verknoten oder verhaken. Unter Verwendung einer Drahtlosverbindung hingegen, kann es, je nach genutztem Übertragungsmedium, zum Beispiel vorkommen, dass die Reichweite nicht ausreicht um bei einen LKW die Diagnose durchführen zu können. Des Weiteren muss das Diagnosegerät transportiert werden und könnte zum Beispiel durch Herunterfallen beschädigt werden oder das Fahrzeug beschädigen. Auch muss der Werker es bei jedem Rad so ablegen, dass er es gleichzeitig ablesen und das Rad drehen kann. Dies kann zum Beispiel auf dem Boden erfolgen, wodurch das Gerät schneller verschmutzt.

Die Lösung für einige dieser Probleme können Wearable Devices sein, welche die Werker immer tragen können. Wearable Devices arbeiten drahtlos, sind am Körper, also zum Beispiel am Handgelenk des Werkers befestigt, und müssen bei einer Diagnose somit nicht abgelegt werden. Über Knöpfe oder andere Interaktionsmöglichkeiten wie Touchscreens oder Sprachbefehle ist der Werker auch in der Lage aus der Entfernung mit dem Diagnosegerät zu interagieren. So könnte er in dem obigen Beispiel über sein Wearable Device die Daten der ABS Sensoren einsehen während er ein Rad dreht. Kurzum können Wearable Devices es zum Beispiel ermöglichen, dass ein einzelner Werker diese Arten von Diagnosen alleine durchführen kann und dabei ähnlich schnell ist, wie zwei Werker ohne die Verwendung von Wearable Devices.

Wie das Beispiel zeigt, können Wearable Devices die Arbeitsabläufe und somit die Durchführung von Fahrzeugdiagnosen, je nach Anwendungsfall, stark vereinfachen.

1.2 Aufbau der Arbeit

Die Arbeit ist in sechs Teile strukturiert. Im Anschluss dieser Einleitung, geht die Arbeit in die Beschreibung des aktuellen Standes der Technik über. Dabei wird zunächst kurz der Ablauf einer Diagnose und die eingesetzte Technik ohne Wearable Devices vorgestellt. Anschließend werden die zwei in dieser Arbeit genutzten Softwareprodukte PRODIS.Authoring und PRODIS.WTS der Firma DSA Daten- und Systemtechnik GmbH Aachen kurz vorgestellt und deren Einsatzgebiete beschrieben. Abschließend zum Stand der Technik werden Wearable Devices im Allgemeinen spezifiziert und die zwei in dieser Arbeit betrachteten Wearable Devices, die Google Glass und die Moto 360, kurz vorgestellt.

In Kapitel 3 wird das Konzept zur Anbindung von Wearable Devices an die bestehenden Diagnosesysteme und -abläufe beschrieben. Zu Beginn werden erst einmal die Anforderungen der DSA GmbH an das Konzept kurz vorgestellt und dann, mit Blick auf diese, ein Konzept Schritt für Schritt entwickelt. Dabei werden auch Ansätze für die Fahrzeughersteller vorgestellt, wie diese das Konzept in ihre bestehenden Diagnoseabläufe einbinden können.

Die Implementierung in Kapitel 4 zeigt abschließend, wie das Konzept in die beiden Softwaresysteme PRODIS.WTS und PRODIS.Authoring integriert wurde. Ebenso wird die Implementierung der Wearable Device-Applikation am Beispiel der Google Glass erläutert.

Kapitel 5 vergleicht, wie andere Arbeiten aus dem industriellen Kontext mit Wearable Devices umgehen. Dazu werden einige verwandte Arbeiten und deren Ergebnisse kurz vorgestellt und erläutert, wo Unterschiede und wo Gemeinsamkeiten zu dieser Arbeit vorliegen. Abschließend wird das Ergebnis der Arbeit in Kapitel 6 kurz zusammengefasst und ein Ausblick gegeben.

Kapitel 2

Stand der Technik

In diesem Kapitel wird zunächst analysiert, wie eine Fahrzeugdiagnose bisher ohne den Einsatz von Wearable Devices üblicherweise abläuft. Dazu wird in den folgenden Abschnitten die beispielhafte Fahrzeugdiagnose aus Abschnitt 1.1 genutzt. Mit dieser wird dargestellt, wie die Fahrzeughersteller Diagnoseabläufe mit dem aktuellen Softwarestand der DSA GmbH definieren und implementieren können. Des Weiteren wird gezeigt, wie die Werker auf Basis der Diagnoseabläufe beim Einsatz des aktuellen Softwarestands der DSA GmbH, die Fahrzeugdiagnose in den Werkstätten durchführen können. Dabei werden die zwei für diese Arbeit relevanten Produkte der DSA GmbH vorgestellt. Abschließend werden die zwei in dieser Arbeit genutzten Wearable Devices kurz eingeführt.

2.1 Fahrzeugdiagnose in Werkstätten

Typischerweise setzen die Werker in den Werkstätten sogenannte Werkstattwagen ein, auf denen häufig ein Notebook oder ein ähnliches Gerät als Diagnosegerät platziert ist. Dieses Diagnosegerät ist entweder über ein Kabel oder über Bluetooth mit einem Modul im zu diagnostizierenden Fahrzeug verbunden, welches im On-Board-Diagnose (OBD) Slot steckt [Sch15]. Dieses Modul stellt die Schnittstelle zwischen dem Fahrzeug und der Software auf dem Diagnosegerät dar. Über diese Schnittstelle können alle vom Fahrzeug bereitgestellten Daten, wie zum Beispiel Spannungswerte, Temperaturen und andere Sensordaten, abgefragt werden. Ebenso können Befehle an das Fahrzeug gesendet werden um zum Beispiel vom Diagnosegerät aus eine Aktion in einem Steuergerät auszulösen. Abbildung 2.1 zeigt wie ein OBD Modul aussehen kann, welches drahtlos mittels Bluetooth mit den Diagnosegeräten kommunizieren kann.

Abbildung 2.1: OBD Scanner [Los15]

Auf den Diagnosegeräten läuft üblicherweise eine Software mit vom Fahrzeughersteller vordefinierten Diagnoseabläufen. Dies hat zum Beispiel den Vorteil, dass die Fahrzeugdiagnose von verschiedenen Werkstätten des Fahrzeugherstellers sehr ähnlich durchgeführt werden kann und somit überall eine ähnliche Bearbeitungszeit und Qualität gewährleistet werden kann. Diese Diagnoseabläufe werden im Vorfeld vom Fahrzeughersteller definiert und im Verlauf des Einsatzes vor Ort, zum Beispiel durch Rückmeldungen der Werker, weiter optimiert.

Im Diagnose-Beispiel aus Abschnitt 1.1 könnten die Fahrzeughersteller zum Beispiel definieren, dass die Drehgeschwindigkeiten der Räder in einem Diagramm oder einer Tabelle dargestellt werden. Eventuell kombiniert ein Fahrzeughersteller die Diagnose aber auch mit weiteren Überprüfungen an den Rädern. So könnte es in diesem Fall sinnvoll sein, nur Daten zum aktuell zu überprüfenden Rad anzuzeigen. Es bleibt also dem Fahrzeughersteller überlassen, wie ein Diagnoseablauf aufgebaut ist und welche Daten dem Werker zur Verfügung stehen.

Um einen Diagnoseablauf zu starten, wählt der Werker auf dem Diagnosegerät den passenden Diagnoseablauf aus und folgt den dargestellten Anweisungen und kontrolliert die dargestellten Diagnosedaten. Je nach Diagnose muss er mit dem Diagnosegerät interagieren, um zum Beispiel Aktionen auszulösen oder Daten zu bestätigen. Im Beispiel aus Abschnitt 1.1 kann der Werker so zum Beispiel bestätigen, dass der aktuell überprüfte ABS Sensor korrekt arbeitet und den nächsten Sensor auswählen.

2.2 Softwarestand bei der DSA GmbH

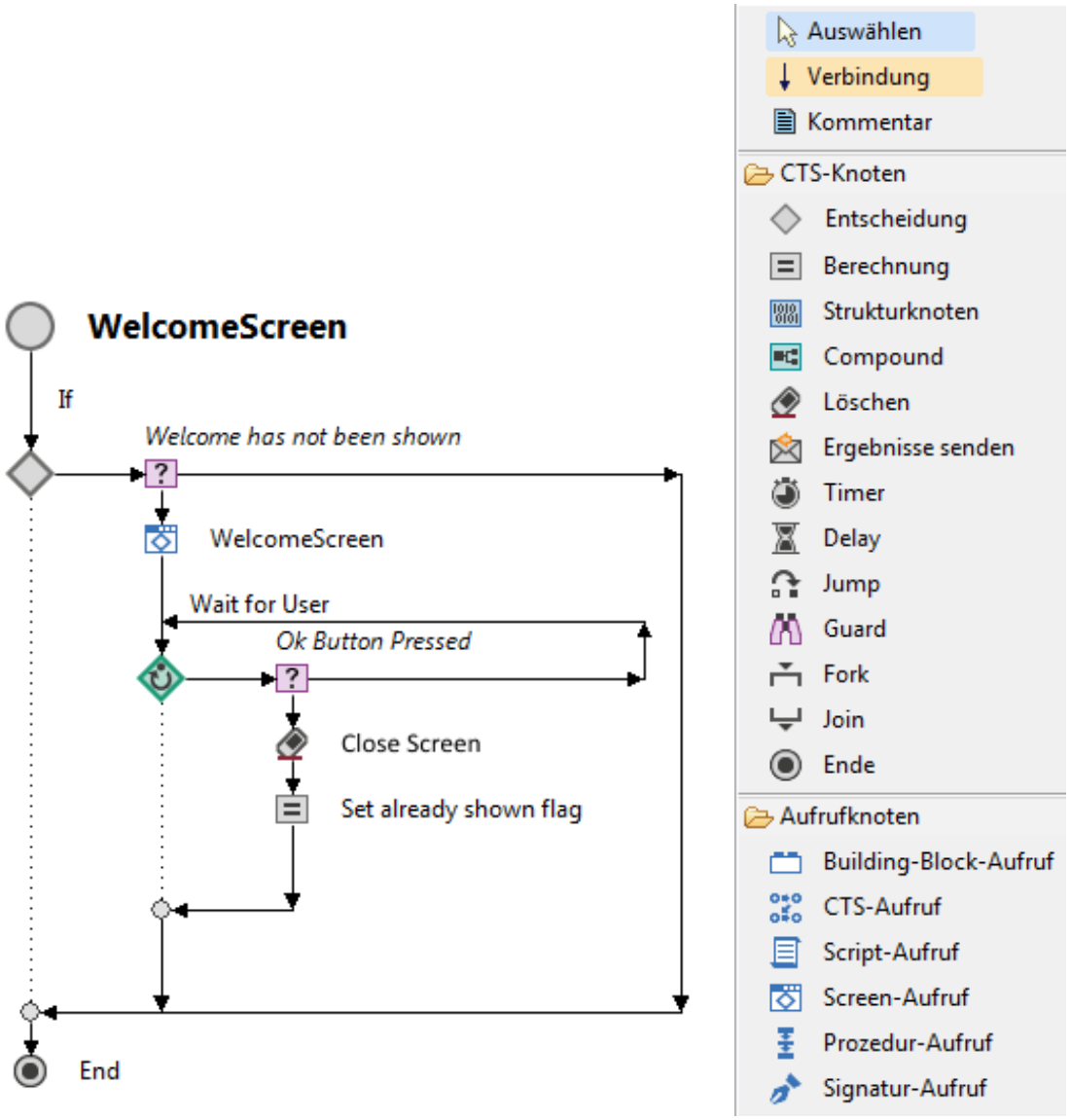
Die für diese Arbeit relevanten Softwareprodukte der DSA GmbH sind PRODIS.Authoring [DSA17a] und PRODIS.WTS [DSA17b]. Sie dienen unter anderem der Entwicklung und der Ausführung von Diagnoseabläufen in Werkstätten und richten sich an Fahrzeughersteller. Im Folgenden werden die beiden Produkte kurz vorgestellt.

2.2.1 PRODIS.Authoring

Mit Hilfe des Autorenwerkzeugs PRODIS.Authoring können unter anderem Diagnoseabläufe definiert und programmiert werden, welche später auf den Diagnosegeräten in den Werkstätten eingesetzt werden [DSA17a]. Ein Complex Test Sequence (CTS) stellt einen solchen Diagnoseablauf im PRODIS.Authoring dar und wird größtenteils von Diagnoseexperten bei den jeweiligen Kunden und nicht von Softwareentwicklern umgesetzt. Dazu steht dem Anwender eine grafische Programmierschnittstelle mit verschiedenen vordefinierten Bausteinen zur Verfügung, welche er beliebig kombinieren kann. Die grafische Programmierschnittstelle orientiert sich im Aufbau und Design an die aus Unified Modeling Language (UML) bekannten Aktivitätsdiagramme. Mittels verschiedenen elementaren Bausteinen können somit komplexe Diagnoseabläufe umgesetzt werden.

In Abbildung 2.2a ist ein beispielhaftes CTS dargestellt, welches eine Willkommensnachricht anzeigt. Mit Hilfe einer IF-Bedingung wird zu Beginn überprüft, ob die Willkommensnachricht bereits gezeigt wurde. Ist dies nicht der Fall, wird sie angezeigt und anschließend darauf gewartet, dass der OK Button geklickt wird. Sobald dies geschehen ist, wird die Willkommensnachricht geschlossen. Ebenfalls speichert sich das CTS zusätzlich noch, dass die Nachricht bereits sichtbar war.

In Abbildung 2.2b sind die verfügbaren Bausteine aufgeführt, mit welchen ein CTS aufgebaut werden kann. Dazu gehören *Entscheidungen*, welche unter anderem sowohl als IF-Bedingung, wie auch als WHILE-Schleife eingesetzt werden können. Zum Beispiel wird der Baustein in Abbildung 2.2a sowohl als Bedingung, als auch als Schleife genutzt. Mit Hilfe von *Berechnungen* kann eigener Java-Quellcode implementiert werden, welcher bei der Ausführung des CTS ausgeführt wird. Ein *Strukturknoten* oder ein *Compound* kann



(a) Willkommensnachricht CTS im PRODIS.Authoring

(b) Bausteine für die Entwicklung eines CTS im PRODIS.Authoring

Abbildung 2.2: Beispiel CTS und Übersicht der Bausteine

zum besseren Strukturieren eines CTS genutzt werden. Sie kapseln typischerweise zusammengehörige Bausteine und können als eine Art Mini-CTS gesehen werden. Mit Hilfe von *Timer*n können zum Beispiel wiederkehrende Aufgaben definiert werden und der *Delay* Baustein sorgt dafür, dass die weitere Ausführung des CTS um eine definierte Zeitdauer verzögert wird. Um auf Events, zum Beispiel auf einen Button-Klick-Event aus der Grafische Benutzeroberfläche (GUI), reagieren zu können, stellt PRODIS.Authoring dem Autor den *Guard*-Baustein zur Verfügung. Für parallele Aufgaben bietet PRODIS.Authoring die Bausteine *Fork* und *Join*, zwischen welchen beliebig viele Aktionen parallel ausgeführt werden können.

Alle Bausteine sind fest definiert und kapseln unter anderem Zugriffe auf den Programmquellcode von PRODIS.Authoring bzw. PRODIS.WTS. Neben den programmiertypischen

Bausteinen wie Bedingungen, Schleifen oder eventbasierten Triggern gibt es auch Möglichkeiten um den Bildschirminhalt des CTS zu definieren. In Abbildung 2.2a wird zum Beispiel mit dem *Screen-Aufruf* Baustein ein Screen, also ein Bildschirminhalt, referenziert. Zum Definieren des Screens steht im PRODIS.Authoring ebenfalls eine grafische Schnittstelle zur Verfügung, in welcher Bausteine wie Tabellen, Bilder, Texte oder Buttons genutzt werden können, um eine GUI zusammenzustellen. Abbildung 2.3 zeigt den in Abbildung 2.2a referenzierten Willkommensnachricht-Screen und einige Bausteine der GUI.

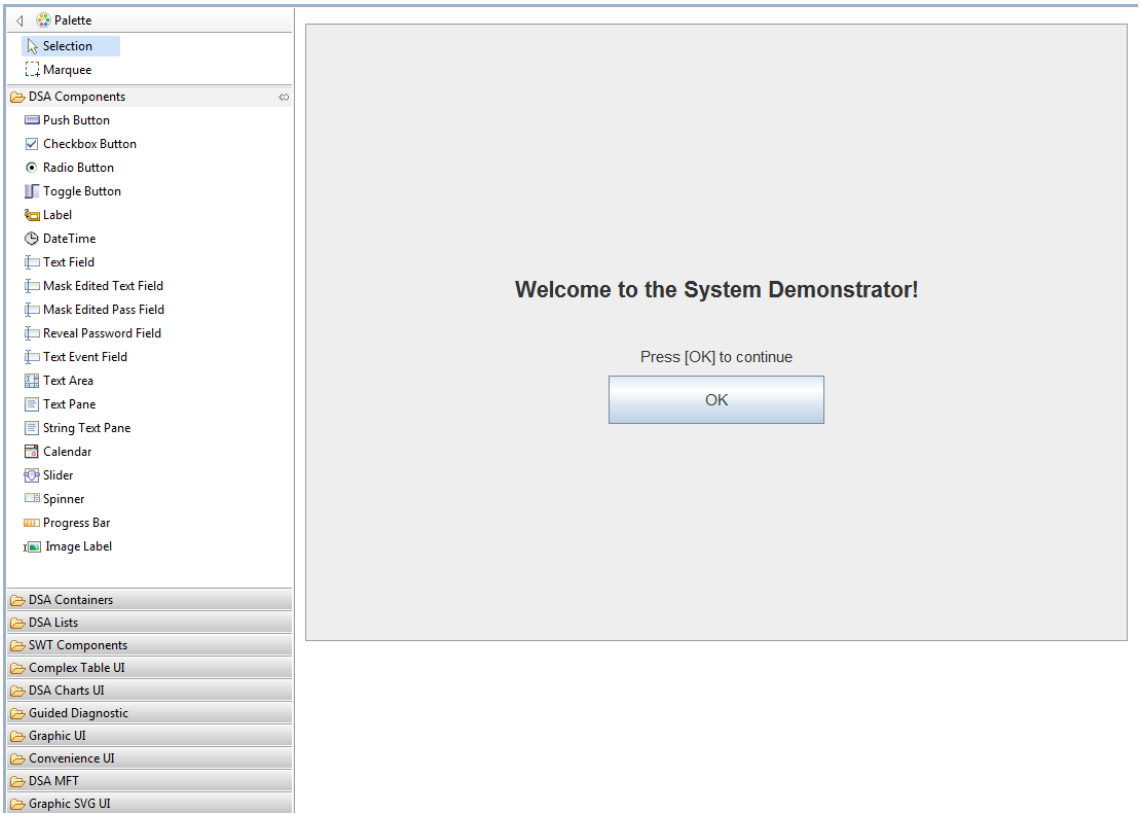


Abbildung 2.3: Beispiel eines CTS Screens im PRODIS.Authoring

Während ein CTS normalerweise den Hauptprogrammfluss widerspiegelt und sich eher mit Benutzerinteraktionen und der Darstellung der Diagnosedaten beschäftigt, verwendet es für die Kommunikation mit dem Fahrzeug Bausteine wie Skripte und Building Blocks. Mit Skripten werden üblicherweise wiederkehrende Aufgaben, wie das Auslesen von Diagnosedaten, geregelt. Dazu nutzen die Skripte häufig Building Blocks, welche unter anderem für die Kommunikation mit der Fahrzeugelektronik oder anderen Schnittstellen zuständig sind. Ein Building Block wird meistens eher für die *low-level* Kommunikation genutzt.

Ein Building Block kann mit einer Methode in einer Programmiersprache verglichen werden. Er kann bestimmte Parameter benötigen, die ihm bei einem Aufruf mit übergeben werden müssen. Einige Parameter dienen nur der Übergabe von Daten, andere kann der Building Block aber zum Beispiel verändern und somit etwa Ergebnisse zurückliefern. Mit Hilfe eines Building Blocks kann eigener Java-Quellcode implementiert werden. Der Java-Quellcode hat Zugriff auf die definierten Parameter, sowie auch auf ausgewählte Klassen des PRODIS.Authoring bzw. PRODIS.WTS Quellcodes. Er ähnelt somit sehr dem *Berechnungs*-Baustein, hat aber den Vorteil, dass er wiederverwendbar ist und in verschiedenen CTSs genutzt werden kann.

Ebenfalls möglich ist die Einbindung eines CTS in ein anderes CTS mit Hilfe des *CTS-Aufruf* Bausteins. Somit lassen sich wiederkehrende grafische Abläufe in eigene CTSs ausgliedern. Diese können dann von jedem CTS eingebunden werden und ermöglichen damit die Reduzierung von Duplikaten. So könnte als Beispiel ein CTS definiert werden, welches einen Dateibrowser öffnet, über welchen der Benutzer eine Datei vom Computer laden kann. Dieses CTS kann dann in alle CTSs eingebunden werden, in welchem Dateien vom Computer durch den Benutzer ausgewählt werden müssen.

Nachdem die Diagnoseabläufe definiert wurden, können diese in ein so genanntes Diagnosepaket exportiert werden. Dieses kann daraufhin in das Diagnoselaufzeitsystem PRODIS.WTS geladen werden, womit anschließend alle definierten Diagnoseabläufe dem PRODIS.WTS Anwender zur Verfügung stehen.

2.2.2 PRODIS.WTS

PRODIS.WTS ist das Diagnoselaufzeitsystem für den Einsatz im Servicebereich und erlaubt die Einbindung und Ausführung verschiedener Diagnosepakete. Es wird üblicherweise auf Diagnosegeräten wie Tablets oder Notebooks eingesetzt und wird von den Werkern einer Werkstatt genutzt [DSA17b]. Es werden also keine besonderen Fähigkeiten wie zum Beispiel Programmierkenntnisse oder ähnliches vorausgesetzt. Abbildung 2.4 zeigt das PRODIS.WTS mit einem Demo-Styling sowie einem Demo-Diagnosepaket, welches die verschiedenen Diagnoseabläufe in einer Ordnerstruktur darstellt.

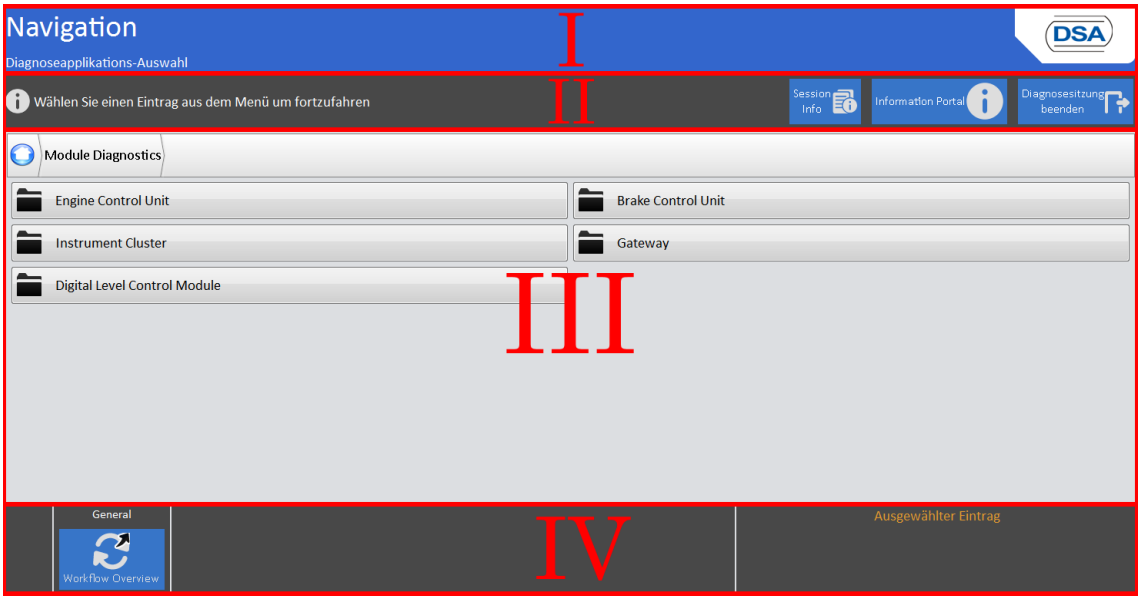


Abbildung 2.4: PRODIS.WTS und sein Aufbau

Der allgemeine grafische Aufbau des PRODIS.WTS ist in vier Teile strukturiert: Der oberste Teil, mit **I** markiert, stellt den Titel des aktuell ausgeführten CTS dar. Darunter folgen im dunkelgrauen Teil **II** die Diagnose-Anweisungen oder weitere Informationen zur Diagnose vom CTS sowie auf der rechten Seite die globalen Buttons. Diese werden unabhängig von dem ausgeführten CTS dargestellt und dienen zum Beispiel der Beendigung einer Diagnosesitzung. Der größte Teil, mit **III** markiert, wird für die Darstellung der Diagnosefunktionalität genutzt. In diesem werden die mit dem PRODIS.Authoring definierten Screens des aktuell ausgeführten CTS dargestellt.

Zuletzt gibt es noch den untersten Bereich, mit **IV** markiert, in welchem die CTSs ihre eigenen Buttons positionieren können. Dieser Bereich kann nochmals in verschiedene Bereiche aufgeteilt werden. Zum Beispiel könnte es einen Bereich allgemeiner Buttons geben, welche unabhängig von den dargestellten oder ausgewählten Diagnosedaten agieren, und einen Bereich für genau solche Buttons, die sich auf die ausgewählte Zeile einer Tabelle beziehen.

Zusammenfassend kann gesagt werden, dass PRODIS.WTS die zentrale Anwendung innerhalb der Werkstätten ist. Sie wird dazu genutzt, um Diagnoseabläufe, welche im Vorfeld über PRODIS.Authoring definiert wurden, auszuführen und die Werker bei der Fahrzeugdiagnose zu unterstützen.

2.3 Wearable Devices

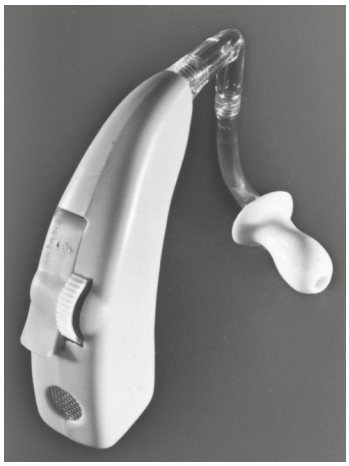
Der Begriff Wearable Device, häufig auch Wearable Computing oder einfach nur Wearable genannt, bezeichnet ein am Körper tragbares elektronisches Gerät, welches dem Träger bei seinen mobilen Tätigkeiten unterstützen soll, ohne ihn dabei zu beeinträchtigen [BR14]. Dabei sollen die Hände frei für jede Tätigkeiten bleiben und der Träger des Wearable Device sich frei bewegen können [BE05a]. Eine prägende Definition kommt von Steve Mann, der 1998 Wearable Devices als kleine am Körper getragene Computer, welche immer aktiv und immer verfügbar sind, bezeichnete:

> "Wearable computing facilitates a new form of human-computer interaction comprising a small body-worn computer (e.g. user-programmable device) that is always on and always ready and accessible. In this regard, the new computational framework differs from that of hand held devices, laptop computers and personal digital assistants (PDAs). The 'always ready' capability leads to a new form of synergy between human and computer, characterized by long-term adaptation through constancy of user-interface."[Man98]

Wearable Devices gibt es schon recht lange. Zu den ersten Wearable Devices zählen zum Beispiel Hörgeräte. Diese können die Träger heutzutage hinter dem Ohr tragen, wo sie die Geräusche für den Träger verstärken. Bereits im Jahr 1913 kam ein solches Gerät von der Firma Siemens, das Esha-Phonophor [Zen14], auf den Markt, welches von seiner Größe allerdings nicht mit der heutigen Technik mithalten kann. Das Esha-Phonophor hatte eine Stromquelle von der Größe einer damals beliebten Klappkamera und wurde, wie Abbildung 2.5a zeigt, wie heutige Kopfhörer auf den Ohren aufliegend getragen [Zen14]. Die ersten Hörgeräte, die den Definitionen der Wearable Devices tatsächlich gerecht werden, kamen Ende der 50er Jahren auf den Markt. So brachte Siemens 1959 das *Auriculette* auf dem Markt [Zen14], welches in Abbildung 2.5b dargestellt ist und den heute eingesetzten Hörgeräten sehr ähnelt und ebenfalls hinterm Ohr getragen werden konnte.

Auf dem Markt der Wearable Devices gab es in den letzten Jahren viel Bewegung. Ausgelöst durch immer kompakter und leistungsfähiger werdende Smartphones, welche überall hin mitgenommen und genutzt werden, erlebten kurze Zeit später die Wearable Devices einen regelrechten Boom [BSM+15]. Für Diabetes Patienten gibt es zum Beispiel mittlerweile ein etwa 2 Euro Stück großes Gerät der Firma Abbott, welches am Arm getragen wird und kontinuierlich den Blutzuckerwert des Trägers misst [Fre17]. In den letzten Jahren

(a) Das Phonophor in Kodakform aus dem Jahr 1914 [Zen14]



(b) Das *Auriculette* aus dem Jahr 1959 [Zen14]

Abbildung 2.5: Hörgeräte von Siemens

waren zudem Fitnessarmbänder, welche während des Sports am Arm getragen werden und dem Träger unter anderem Informationen über seinen Kalorienverbrauch oder seine Herzschlagfrequenz mitteilen, sehr beliebt [Sey15].

Die zwei in dieser Arbeit betrachteten Wearable Devices sind die Google Glass und die Moto 360 von Motorola, welche im Folgenden kurz vorgestellt werden.

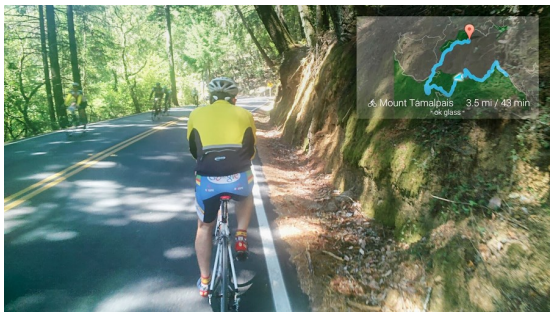
2.3.1 Google Glass

Die Google Glass, in den Abbildungen 2.6a und 2.6c dargestellt, ist ein Wearable Device aus dem Bereich der Datenbrillen [Tan14]. Sie wird wie eine Brille getragen und bietet unter anderem ein kleines optisches Display am rechten Auge, welches im Randgebiet des Sichtfeldes angebracht ist. Mit Hilfe eines Mini-Projektors und eines halbtransparentes Prismas wird das Bild dem Träger direkt auf die Netzhaut projiziert. Dabei erscheint das Display dem Betrachter wie ein, in etwa 240cm vor ihm stehender, 25 Zoll Bildschirm [Gla17c]. Die dargestellten Daten werden somit nicht im direkten Sichtfeld des Trägers angezeigt, wo sie ihn stören und andere Elemente überlagern könnten. Die Google Glass blendet dem Träger also Hinweise zusätzlich zu den realen Informationen ein und gehört somit zu den Geräten aus der *Augmented Reality (AR)* Kategorie.

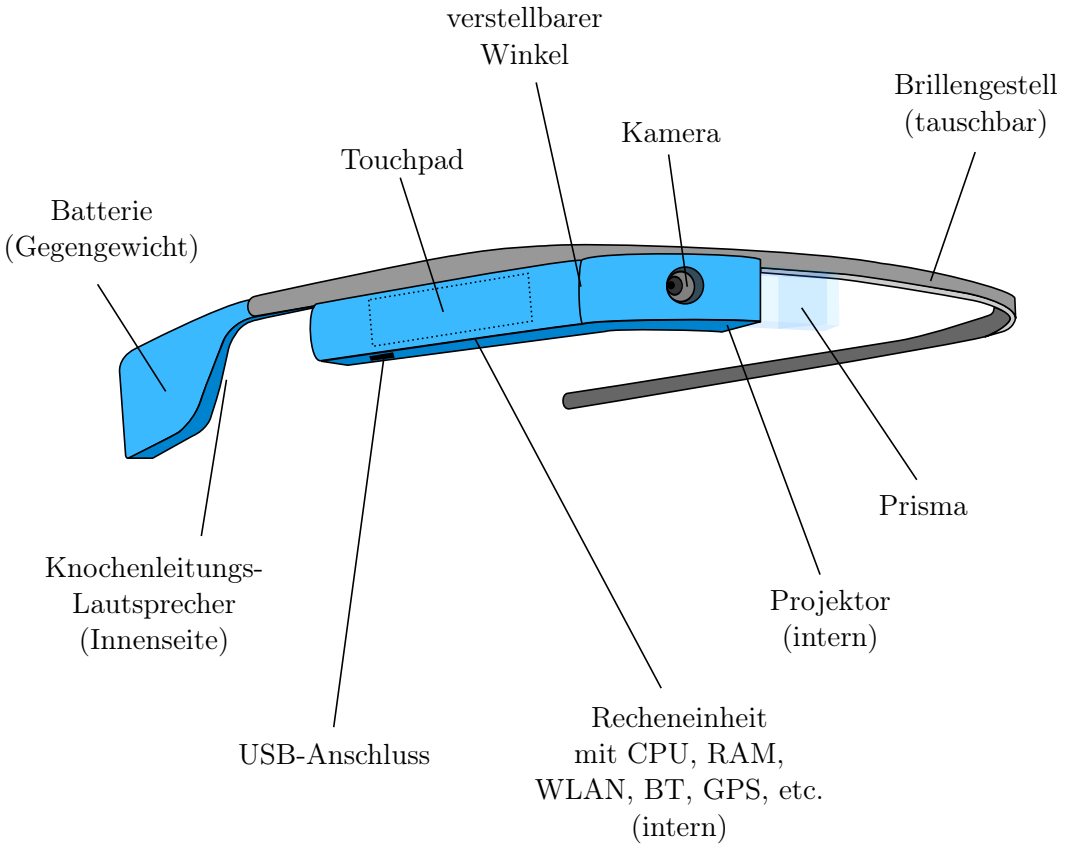
Zum Betrachten der Daten auf dem Display muss der Träger den Fokus seiner Augen bewusst auf das Display legen. Tut er dies nicht, erscheinen die Daten im Display am Rande des Sichtfeldes unscharf. Das Sichtfeld des Trägers sieht ähnlich aus, wie in Abbildung 2.6b dargestellt. In diesem Beispiel fährt der Träger Fahrrad und kann, wenn er den Blick auf die Google Glass fokussiert, Informationen über die Route einsehen. Solange er den Fokus nicht auf die Google Glass legt, verdecken die dargestellten Informationen dem Träger keine realen Informationen in seinem direkten Blickfeld, wie zum Beispiel den vor ihm fahrenden Fahrradfahrer.

Die Google Glass bietet des Weiteren eine Digitalkamera inklusive Mikrofon, mit welcher Bilder und Videos aufgezeichnet werden können. An der rechten Seite befindet sich ein

(a) Foto [Rec14]



(b) Sichtfeld [Gla17b]



(c) Aufbau [Mov13]

Abbildung 2.6: Foto, beispielhaftes Sichtfeld und schematische Darstellung der Google Glass

Touchpad, welches unter anderem zwischen einer einfachen Berührung und drei Touch-Gesten differenzieren kann [Gla17a]. Über dieses Touchpad und über Sprachbefehle kann die Google Glass gesteuert werden. Über einen Knochenleitungs-Lautsprecher kann die Google Glass Töne wiedergeben, welche nur der Träger hört. Somit kann die Google Glass mit dem Träger kommunizieren ohne das Umfeld des Trägers zu stören.

Mittels Bluetooth und WLAN kommuniziert die Google Glass mit anderen Geräten und bietet bei normaler Nutzung eine Akkulaufzeit von bis zu einem Tag [Gla17c]. Im Verlauf dieser Arbeit konnten allerdings lediglich Laufzeiten von einigen Stunden erreicht werden.

Das Betriebssystem der Google Glass ist ein auf Android 4.4 basiertes System [Gla17d], bei welchem einige Vereinfachungen und Anpassungen bei der Darstellung vorgenommen wurden, um dem Träger möglichst wenige, dafür aber relevante Daten anzuzeigen [Gla15b]. Durch viel Gegenwind in den Medien, vor allem aus Datenschutzgründen [Zin15], wurde die Google Glass von Google mittlerweile vom Consumer Markt entfernt. Sie wird weiterhin für *Glass Certified Partners*, also zertifizierte Unternehmen, unter dem Namen *Glass at Work* angeboten [Gla16].

2.3.2 Moto 360

Die Moto 360, in Abbildung 2.7 dargestellt, ist ein Wearable Device aus dem Bereich der Smartwatches. Sie wird wie eine Armbanduhr getragen und ermöglicht über einen runden, etwa 1,5 Zoll großen, kapazitiven Touchscreen eine Smartphone ähnliche Bedienung. Als Betriebssystem setzt die Moto 360 das auf Android basierende Android Wear ein, welches speziell für Wearable Devices ausgelegt ist [Mot17].

Abbildung 2.7: Moto 360 in der mitgelieferten Qi-Ladestation [Mot17]

An der Seite der Moto 360 befindet sich ihr einziger physischer Button, über welchen die Smartwatch eingeschaltet werden kann. Sie ist IP67 zertifiziert, schützt also für einen gewissen Zeitraum vor dem Eindringen von Staub und Wasser [Mot17]. Über zwei eingebaute Mikrofone kann die Smartwatch auch mittels Sprachbefehlen gesteuert werden. Einen Lautsprecher hat die Moto 360 nicht, allerdings kann sie den Träger mittels eines Vibrationsmotor auf sich aufmerksam machen oder Feedback geben.

Über Bluetooth und WLAN kann die Moto 360 mit anderen Geräten kommunizieren und bietet bei normaler Nutzung eine Akkulaufzeit von bis zu 24 Stunden. Ähnlich wie bei der Google Glass zeigten sich im Verlauf dieser Arbeit allerdings geringere Laufzeiten. Bei beiden Geräten ist dies vermutlich auch dem Anwendungsfall geschuldet, bei welchem das Display häufig deutlich länger, als bei einer normalen Nutzung, aktiv ist.

Ein Vorteil der Moto 360 gegenüber vielen anderen Wearable Devices, inklusive der Google Glass, ist die Möglichkeit die Smartwatch über den Qi-Standard kabellos laden zu können. Nach rund einer Stunde ist ihr 320 mAh Akku vollgeladen. Somit entfällt das Anschließen eines Kabels, was zum Beispiel mit Handschuhen sehr umständlich sein kann.

Kapitel 3

Konzept

3.1 Anforderungen

Damit die Wearable Devices sich sinnvoll bei Diagnoseabläufen einsetzen lassen und sich nahtlos in die bisherige Systemlandschaft integrieren, wurden im Vorfeld dieser Arbeit einige Anforderungen definiert, welche von dem Konzept und der Implementierung beachtet und umgesetzt werden müssen:

- **REQ 1**: *Daten aus PRODIS.WTS beziehen*
- **REQ 2**: *GUI generisch oder aus CTS generieren*
- **REQ 3**: *Minimale Anpassungen an bestehende CTSs*
- **REQ 4**: *Versionsunabhängigkeit*
- **REQ 5**: *Einfache Anbindung weiterer Geräte*

Da PRODIS.WTS die zentrale Diagnosesoftware in der Werkstatt ist, in der alle Diagnosedaten zusammenlaufen und neben den Wearable Devices keine weitere Hard- und Software in den Werkstätten eingesetzt werden soll, müssen die Wearable Devices die benötigten Daten über PRODIS.WTS beziehen (**REQ 1**).

Wie zuvor beschrieben, können die Hersteller im PRODIS.Authoring eigene CTS definieren. Dazu gehört auch die grafische Darstellung in Form von Screens im PRODIS.WTS. Viele Hersteller nutzen diese Funktionalität, weshalb es viele verschiedene GUIs gibt. Dementsprechend muss das Konzept so aufgebaut sein, dass die Implementierung entweder komplett generisch ist und es den Herstellern überlässt die bestehenden CTS zu erweitern, oder das Konzept muss beschreiben, wie die Implementierung aus einer bestehenden CTS GUI selbstständig eine Darstellung für Wearable Devices erstellen kann (**REQ 2**).

Damit möglichst wenige Hürden bei der Integration von Wearable Devices in bestehende Diagnoseabläufe bestehen, muss das Konzept mit minimalen Änderungen an den CTSs auskommen (**REQ 3**). Andernfalls sinkt die Wahrscheinlichkeit des Einsatzes von Wearable Devices, da die Hersteller zunächst ihre gesamten CTSs überarbeiten müssten.

In den Werkstätten sind häufig verschiedene Versionen von PRODIS.WTS im Einsatz. So kann in einer Werkstatt zum Beispiel Version X im Einsatz sein, bei einer anderen

Werkstatt aber noch Version X-1. Trotzdem können beide mit dem selben Diagnosepaket arbeiten, welches der Hersteller beispielsweise mit PRODIS.Authoring für Version X+1 entwickelt hat. Dies ist notwendig, da nicht sichergestellt werden kann, dass alle Werkstätten regelmäßig ihre PRODIS.WTSs aktualisieren. Ähnlich wird es auch mit den Wearable Devices sein, weshalb das Konzept möglichst versionsunabhängig sein soll und sowohl mit neueren als auch älteren PRODIS.WTS Versionen agieren können muss (**REQ 4**).

In späteren Ausbaustufen sollen gegebenenfalls auch weitere Geräte, wie zum Beispiel ein Smartphone oder Tablet, angebunden werden. Damit dies möglich ist, soll das Konzept den Einsatz von Standardtechnologien vorsehen, die unabhängig von Systemen und Programmiersprachen sind und die Anbindung neuer Geräte mit möglichst wenigen Anpassungen ermöglichen (**REQ 5**).

Es ergeben sich somit im Wesentlichen fünf Anforderungen, welche das Konzept bei der Lösung im Folgenden betrachten muss. Dazu werden zunächst verschiedene Ideen, wie die beschriebenen Anforderungen geschickt umgesetzt werden können, vorgestellt und anschließend verglichen und resümiert.

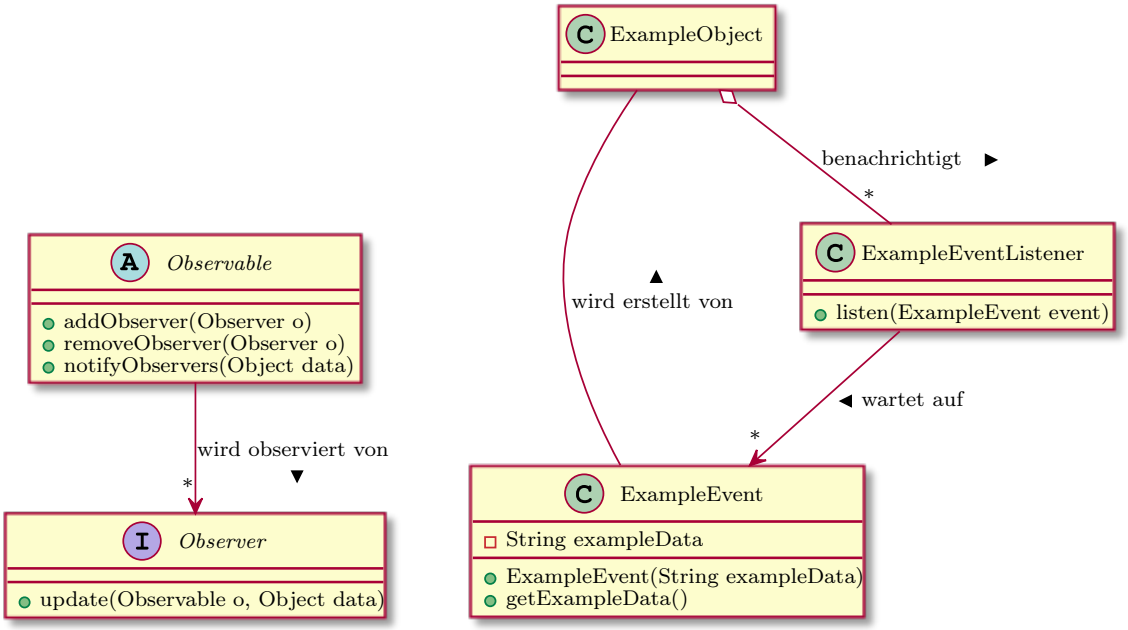
3.2 Daten aus PRODIS.WTS beziehen

Aufgrund von **REQ 1** müssen die Wearable Devices mit PRODIS.WTS kommunizieren und auf die relevanten Daten Zugriff bekommen. Dazu wurde schon vor Beginn dieser Arbeit PRODIS.WTS um Möglichkeiten erweitert, auf alle Aktionen zwischen den Screens und den CTSs zu reagieren, indem das Beobachter-Entwurfsmuster implementiert wurde. Dabei handelt es sich um ein Verhaltensmuster der Gang of Four (GoF), welches zum Beispiel bei Zustands- oder Datenänderungen die Informationen der Änderungen an andere Objekte weitergibt [GHJV94]. Die wohl bekanntesten Beobachter-Muster sind das Observer- und Listener-Pattern.

Observer Beim Observer-Pattern [GHJV94] werden an einer zu beobachtenden Klasse, *Observable* genannt, beliebig viele Beobachter, *Observer* genannt, hinzugefügt. Der Aufbau sieht üblicherweise wie im Klassendiagramm in Abbildung 3.1a dargestellt aus. Bei einer Statusänderung des *Observable*s benachrichtigt dieses selbständig alle registrierten *Observer*. Ein beispielhafter Ablauf wird im Sequenzdiagramm in Abbildung 3.2 gezeigt, in welchem die Methode *notifyObservers(Object data)* intern die *update(Observable o, Object data)* Methode aller registrierten *Observer* aufruft, sobald sich etwas am *Observable* ändert.

Listener Das Listener-Pattern [GHJV94] definiert, anders als beim Observer-Pattern, für jede mögliche Statusänderung des *Observable*s ein eigenes Event. Die *Observer*, in diesem Pattern *Listener* oder auch *EventListener* genannt, lauschen nur auf einen fest definierten Event-Typen und bekommen andere Statusänderungen des *Observable*s nicht mit. Dies hat den großen Vorteil, dass nicht erst überprüft werden muss, ob die Statusänderung relevant ist. Ein beispielhafter Aufbau ist im Klassendiagramm in Abbildung 3.1b dargestellt.

Im Java Umfeld gilt das Listener-Pattern als De-facto-Standard und wird auch von dem im PRODIS.WTS benutzten GUI Framework *Swing* eingesetzt. Dies macht die Integration deutlich einfacher und ermöglicht später auch einfachere Änderungen und Erweiterungen,



(a) Observer (b) EventListener

Abbildung 3.1: Beobachter-Entwurfsmuster: Observer vs EventListener

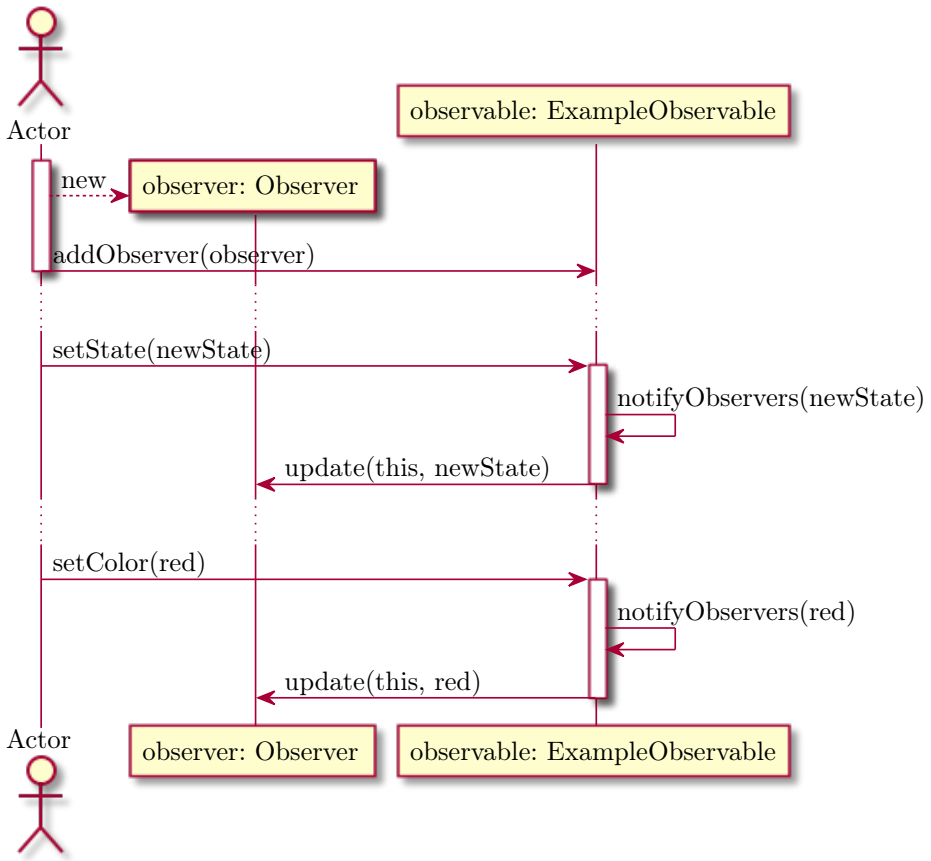
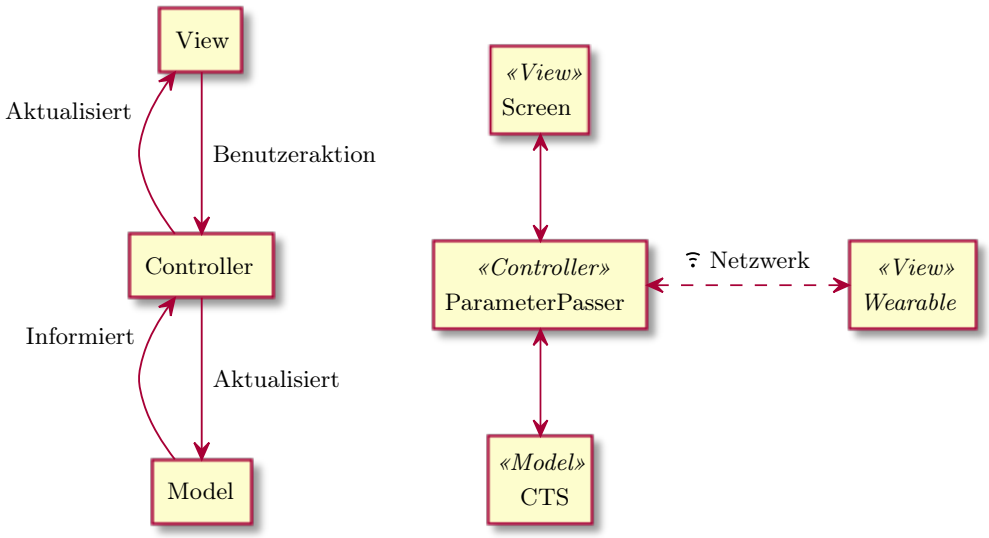


Abbildung 3.2: Sequenzdiagramm des Beobachter-Entwurfsmuster (Observer)

da nur die relevanten Events und Listener angepasst werden müssen. Wie bereits zu Beginn erwähnt, wurde PRODIS.WTS im Vorfeld dieser Arbeit bereits entsprechend angepasst. Dazu wurde der *ParameterPasser* eingeführt, welcher sich mit Hilfe des Listener-Pattern zwischen den CTSs und deren Screens positioniert.

Der *ParameterPasser* arbeitet mit vielen verschiedenen Listenern und spiegelt den vom Model–View–Controller (MVC) Konzept definierten *Controller* wider, während das CTS das *Model* und dessen Screens den *View* darstellen. Das MVC Konzept definiert drei Rollen: Das *Model*, welches im wesentlichen die Daten und Teile der Programmlogik beinhaltet. Den *View*, welcher eine Darstellung und Interaktion mit den bereitgestellten Daten ermöglicht. Zuletzt noch den *Controller*, welcher zwischen den anderen Komponenten arbeitet und die Eingaben und Ausgaben der Daten steuert [GHJ01]. Das Diagramm in Abbildung 3.3a zeigt, wie die drei Komponenten im MVC Konzept zusammen arbeiten. Abbildung 3.3b stellt darüber hinaus dar, wie das MVC Konzept im Umfeld vom PRODIS.WTS mit Hilfe des *ParameterPasser* benutzt wird.



(a) Zusammenspiel der Komponenten

(b) Umsetzung im PRODIS.WTS

Abbildung 3.3: MVC

Aufgrund von **REQ 2** muss die Darstellung auf den Wearable Devices, genau wie die Screens, über ein CTS definiert werden. Daraus ergibt sich, dass die Wearable Devices ebenfalls einen View im MVC Konzept darstellen und sich somit in das bestehende Konzept integrieren. Abbildung 3.3b zeigt daher das klassische MVC Konzept mit der Besonderheit, dass es nun weitere Views geben kann, die nicht auf dem selben Gerät dargestellt werden. Dies ist auch für **REQ 5** ein wichtiger Punkt, da so die Anbindung weiterer Geräte ermöglicht und vereinfacht wird. Die Geräte müssen nämlich weder selbst die CTSs interpretieren können, noch die Daten in irgendeiner Art und Weise mit dem PRODIS.WTS synchronisieren. Sie benötigen lediglich ein Display und eine Möglichkeit sich mit dem PRODIS.WTS verbinden zu können, den Rest übernimmt PRODIS.WTS mit Hilfe des *ParameterPasser*.

3.3 Client-Server Kommunikation

Wie zuvor erläutert, gibt es durch den *ParameterPasser* die Möglichkeit auf Aktionen des Benutzers und auf Änderungen der Daten zu reagieren. Damit später auf den jeweiligen Wearable Devices eine GUI definiert werden kann, müssen diese Daten aber zum Wearable Device gelangen. Dazu werden im Folgenden die Möglichkeiten einer Client-Server Verbindung aufgezeigt, bei der PRODIS.WTS als Server fungiert und die Wearable Devices als Clients. Als Grundlage dient das Open Systems Interconnection Model (OSI-Modell), welches die Kommunikation von verschiedenen Systemen untereinander in sieben Schichten, zu engl. Layer, regelt [ISO94]. Abbildung 3.4 zeigt, wie mit Hilfe des OSI-Modells eine Kommunikation zwischen zwei Systemen abläuft.

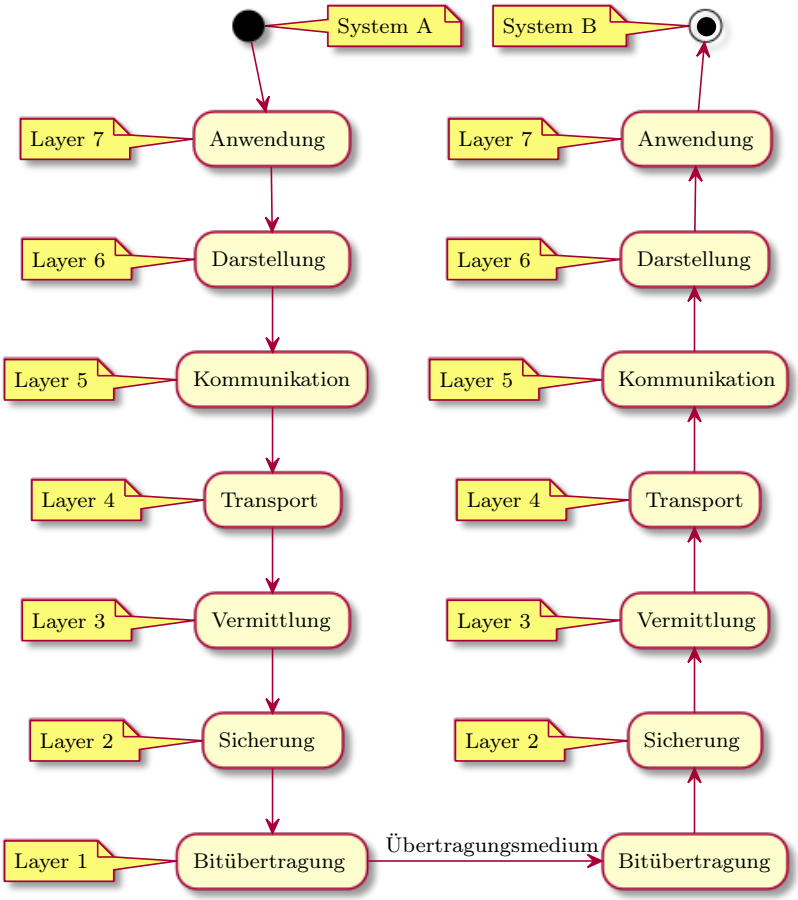


Abbildung 3.4: Aufbau des OSI-Modell

3.3.1 Netzzugriff (OSI-Modell Layer 1 und 2)

Die Wearable Devices mit denen diese Arbeit umgesetzt wurde, bieten beide WLAN und Bluetooth an, um Daten zu übertragen. PRODIS.WTS wird in den Werkstätten häufig auf Diagnosegeräten wie Notebooks oder Tablets eingesetzt, die mit einem werkstatteigenem WLAN Netz verbunden sind. Damit die Geräte untereinander kommunizieren können, müssen die Wearable Devices entweder in das werkstatteigene WLAN Netz eingebunden

werden oder die Diagnosegeräte müssen einen eigenen WLAN HotSpot oder ein Ad-Hoc Netz einrichten, mit welchem sich die Wearable Devices verbinden können.

Die Verbindung über Bluetooth ist eine weitere Möglichkeit, erfordert aber sowohl deutlich mehr Arbeit bei der Anbindung von Geräten als auch vom Werker selbst bei der Verwendung der Wearable Devices in den Werkstätten. Es ist zwar auch mit Bluetooth möglich eine Ad-Hoc Verbindung einzurichten, allerdings ist die Einrichtung deutlich aufwendiger. Auch werden in den Werkstätten die Diagnosedaten vom Auto zum Diagnosegerät ebenfalls häufig per Bluetooth übertragen und es ist unklar, ob all diese Geräte auch mit einer Ad-Hoc Verbindung funktionieren würden.

Eine direkte Verbindung über Bluetooth hat den Nachteil, dass die Wearable Devices an ein Diagnosegerät gebunden sind. Möchte der Werker mit den Wearable Devices an einem anderen Diagnosegerät arbeiten, müsste er diese zunächst per Bluetooth mit dem neuen Diagnosegerät verbinden. Der Aufwand ist, verglichen mit einem Ad-Hoc Netzwerk, etwas größer. Ein weiterer und entscheidender Nachteil von Bluetooth stellt die Datenrate dar, welche sich die Geräte sowohl bei Ad-Hoc als auch bei direkten Verbindungen teilen würden [IEE02]. So könnte es vorkommen, dass PRODIS.WTS so viele Diagnosedaten über Bluetooth erhält, dass die Kommunikation mit den Wearable Devices zeitverzögert stattfindet. Damit wäre eine verlässliche Diagnose nicht mehr sichergestellt, da die angezeigten Daten nicht den aktuellen Stand widerspiegeln.

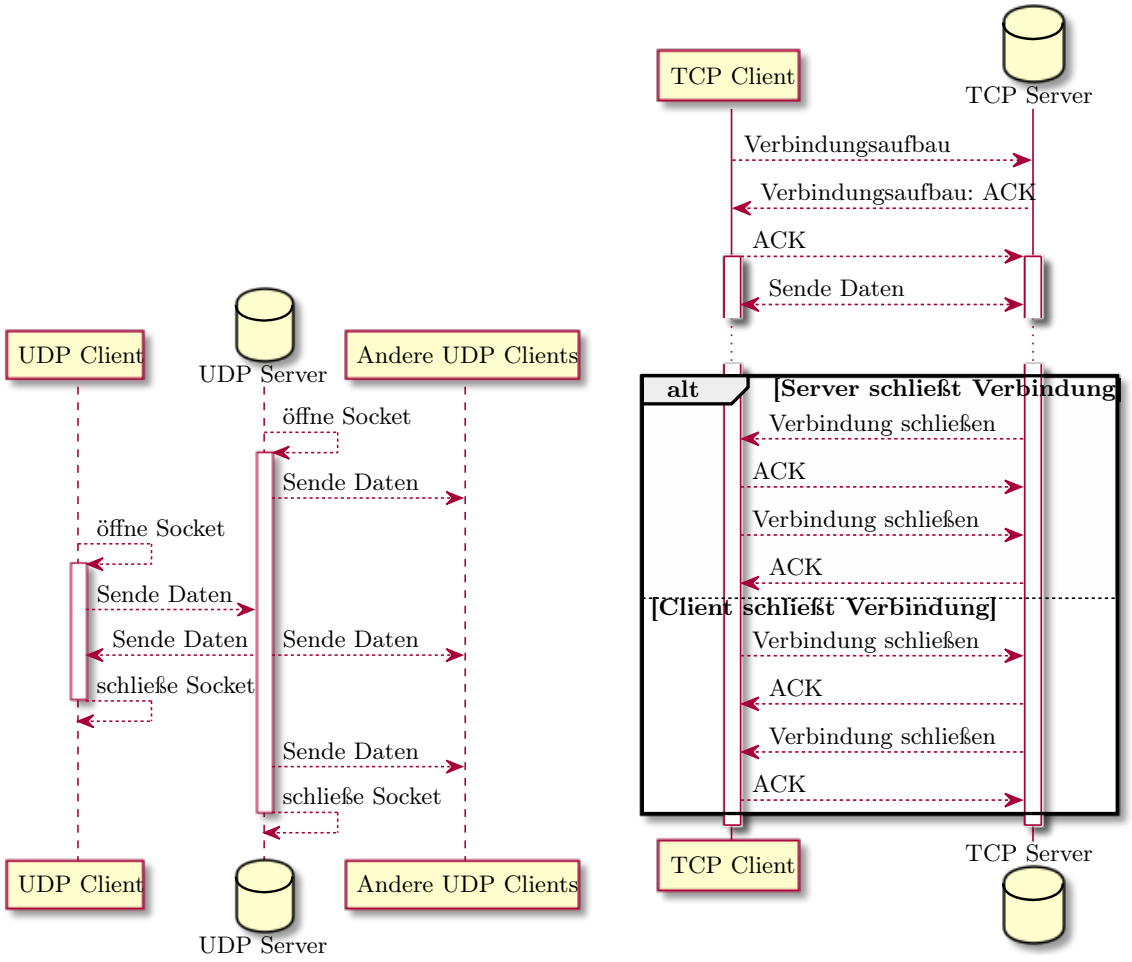
Aus diesen Gründen fällt die Wahl der Verbindung auf WLAN. Es ist sowohl einfacher in der Implementierung, da keine Sonderfälle beachtet werden müssen, als auch bei der Ausführung der Diagnose vor Ort in den Werkstätten, da das Konfigurieren ohne großen Aufwand umzusetzen ist.

3.3.2 Transport (OSI-Modell Layer 3 und 4)

Als Layer 3 Protokoll wird das Internet Protocol (IP) eingesetzt, da es fast alle Konkurrenzprotokolle verdrängt hat. Es ist die Grundlage des Internets und wird von nahezu allen Geräten, Betriebssystemen und Programmiersprachen unterstützt. Im Rahmen des OSI-Modell regelt es die Vermittlung der zu übertragenden Datenpakete mittels in einem Netzwerk eindeutigen IP-Adressen [Pos81a]. Gerade im Zeitalter des Internet of Things (IoT), welches ein Konzept bezeichnet, in welchem sich die virtuelle Welt der IT mehr und mehr in die reale Welt integriert [UHM11], und somit immer mehr Geräte über das Internet kommunizieren, ist es auch ein zukunftsorientiertes Protokoll.

Es gibt im wesentlichen zwei Layer 4 Protokolle, die hier benutzt werden können: User Datagram Protocol (UDP) und Transmission Control Protocol (TCP). Per UDP ist die Kopplung neuer Geräte vergleichsweise einfach, da es ein verbindungsloses Protokoll ist und somit ein neues Gerät sehr einfach angebunden werden kann [Pos80]. Abbildung 3.5a zeigt einen beispielhaften Kommunikationsablauf über UDP. Leider hat UDP auch einige entscheidende Nachteile. Für diesen Anwendungsfall ist es ein großer Nachteil, dass es nicht zuverlässig ist. Das bedeutet, dass weder die Reihenfolge der Nachrichten garantiert werden kann, noch dass die Nachrichten auch beim Empfänger ankommen. Dies könnte zu undefinierten Zuständen führen, was das Verhalten der Anwendung negativ beeinflussen würde und im schlimmsten Fall zum Absturz führen könnte.

Das TCP ist, anders als UDP, ein zuverlässiges und verbindungsorientiertes Protokoll. Ersteres garantiert, dass alle zu sendenden Daten in der gesendeten Reihenfolge beim Empfänger ankommen und das Protokoll selbstständig verlorene Datenpakete erneut sendet,



(a) UDP (b) TCP

Abbildung 3.5: Beispielhafte Kommunikationsabläufe von UDP und TCP

damit alle Inhalte erfolgreich übertragen werden [Pos81b]. Letzteres bedeutet allerdings, dass neue Geräte immer zunächst eine Verbindung herstellen müssen, bevor Daten empfangen werden können. Dies ist zum einen ein Nachteil, da somit der Verbindungsaufbau umständlicher ist. Andererseits ermöglicht es auch, dass die Wearable Devices keine Daten von anderen Systemen empfangen, da die Wearable Devices aktiv mit einem System verbunden werden müssen. Abbildung 3.5b zeigt einen beispielhaften Kommunikationsablauf über TCP.

Dadurch dass UDP kein zuverlässiges Protokoll ist und somit wichtige Diagnosedaten nicht bei den Wearable Devices ankommen könnten, disqualifiziert es sich für den geplanten Einsatz. Angenommen ein Werker würde an einem Rad drehen, um dessen Sensorik zu überprüfen und PRODIS.WTS würde ein Diagramm mit der aktuellen Drehgeschwindigkeit des Rades anzeigen. Kommt es dann zu Datenverlusten über UDP, würde auf den Wearable Devices das Diagramm unverändert bleiben. Im schlimmsten Fall würde der Werker mehrere Stunden mit der Fehlersuche verbringen und die Sensorik tauschen, obwohl diese eigentlich korrekt funktioniert. Daher kann die Wahl nur auf TCP fallen, da diese Probleme durch dessen Zuverlässigkeit unterbunden werden.

3.3.3 Anwendung (OSI-Modell Layer 5 bis 7)

Die Layer 5 bis 7 des OSI-Modell stehen für das Anwendungssystem. Die wohl bekanntesten Protokolle dieser Schichten sind Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP) oder auch die E-Mail Protokolle Simple Mail Transfer Protocol (SMTP) oder Post Office Protocol (POP). Jedes der Protokolle hat seine Vor- und Nachteile sowie bestimmte Einsatzgebiete. Um auswählen zu können, welche Protokolle in dieser Arbeit Sinn machen, muss geklärt werden, welche Funktionalitäten überhaupt benötigt werden.

Das Protokoll muss es dem Server, also PRODIS.WTS, ermöglichen, unabhängig von den Clients, also den Wearable Devices, Daten an diese senden zu können. Aber auch die Wearable Devices müssen PRODIS.WTS Nachrichten senden können, damit die Werker über die Wearable Devices mit den CTSs interagieren können. Weiterhin sollte das Protokoll den sonstigen Anforderungen an das Konzept nicht im Weg stehen, sondern, wenn möglich, die Erfüllung der Anforderungen vereinfachen. Es sollte also zum Beispiel auf vielen verschiedenen Geräten implementierbar sein (**REQ 5**) und unabhängig von Versionen agieren können (**REQ 4**).

Im Folgenden werden einige mögliche Protokolle kurz vorgestellt und mit Blick auf die Anforderungen ihre Vor- und Nachteile beleuchtet.

Ein eigenes Protokoll hat den Vorteil, dass alle benötigten Funktionalitäten selbst implementierbar sind. Es ist aber auch mit Abstand die aufwändigste Option, denn es erschwert nicht nur die Nutzung bestehender Bibliotheken, sondern erfordert für die Anbindung weiterer Geräte mehr Arbeit um das Protokoll zu implementieren. Dies widerspricht allerdings **REQ 5**, da für jedes weitere Gerät, welches nicht mit den erstellten Bibliotheken arbeiten kann, zum Beispiel weil es eine andere Programmiersprache benötigt, das Protokoll erneut implementiert, getestet und angebunden werden muss.

HTTP ist das Standard Protokoll im Internet, wenn es um Webseiten geht. Es ist zustandslos und dient zur Übertragung von Dateien zwischen zwei Rechnern innerhalb eines Netzwerkes. Zustandslos bedeutet, dass jede Anfrage unabhängig von vorherigen Anfragen ausgeführt und bearbeitet wird und jegliche Informationen aus anderen Anfragen nicht mehr vorhanden sind. Erst Layer 7 ermöglicht das Beibehalten einer Sitzung, mit Hilfe von Cookies oder anderen Header-Parametern, über mehrere Anfragen hinweg.

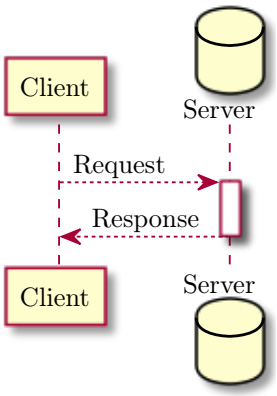


Abbildung 3.6: HTTP Anfragebeispiel

Beim HTTP geht die Aktion immer vom Client aus, welcher eine Anfragenachricht an den Server sendet. Dieser verarbeitet die Informationen aus der Nachricht, stellt eine Antwortnachricht zusammen und sendet diese an den Client zurück. Anschließend wird die Verbindung getrennt. Ein beispielhafter Ablauf ist in Abbildung 3.6 dargstellt. Eine Nachricht besteht aus einem Datenstrom und aus Header-Parametern, welche z.B. Informationen über die Kodierung der Daten beinhaltet [FGM+99].

Leider ist es mit HTTP nicht möglich, dass der Server den Client kontaktiert. Als Workaround gibt es das so genannte *polling* oder auch *long polling*. Beim *polling* fragt der Client

den Server regelmäßig nach neuen Daten [LSA11]. Sind keine vorhanden, wird die Verbindung wieder geschlossen. Beim *long polling* hingegen hält der Server die Verbindung solange wie die Timeouts es ermöglichen offen, oder bis neue Daten vorhanden sind und sendet dann erst seine Antwort [LSA11]. Beide Verfahren können zu hoher Netzwerk- und Gerätelast führen, je nachdem wie zeitnah der Client neue Informationen benötigt und wie häufig dementsprechend angefragt wird. Die beiden Sequenzdiagramme aus Abbildung 3.8a und Abbildung 3.8b stellen einen beispielhaften Verlauf von *polling* und *long polling* dar.

Beim geplanten Einsatz in einer Werkstatt dürfen die Diagnosedaten nicht zeitverzögert auf den Wearable Devices angezeigt werden. Dies würde die Diagnose eher erschweren, als vereinfachen. Es könnte ein *polling* mit möglichst wenig Zeitverzögerung definiert werden, was allerdings zu einer massiven Anzahl an Verbindungen führen würde. Auch ein *long polling* würde gerade bei Echtzeitdaten die selben Probleme verursachen.

HTTP 2 ist der Nachfolger von HTTP und komplett abwärtskompatibel. Eines der Hauptziele von HTTP 2 ist die Verbesserung der Übertragung der Daten im Internet. Eine Webseite besteht heutzutage nicht mehr nur aus einer Datei, sondern die Browser müssen häufig viele Skripte, Stylesheets und Bilder laden. Bei HTTP müsste für jede weitere Anfrage eine neue Verbindung aufgebaut werden, was die Performance verschlechtert und zu mehr Netzwerkverkehr führt, da jeder Verbindungsaufbau eine gewisse Zeit kostet.

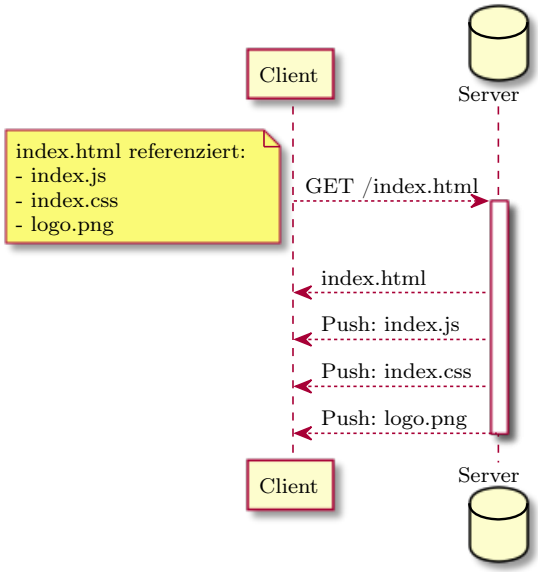


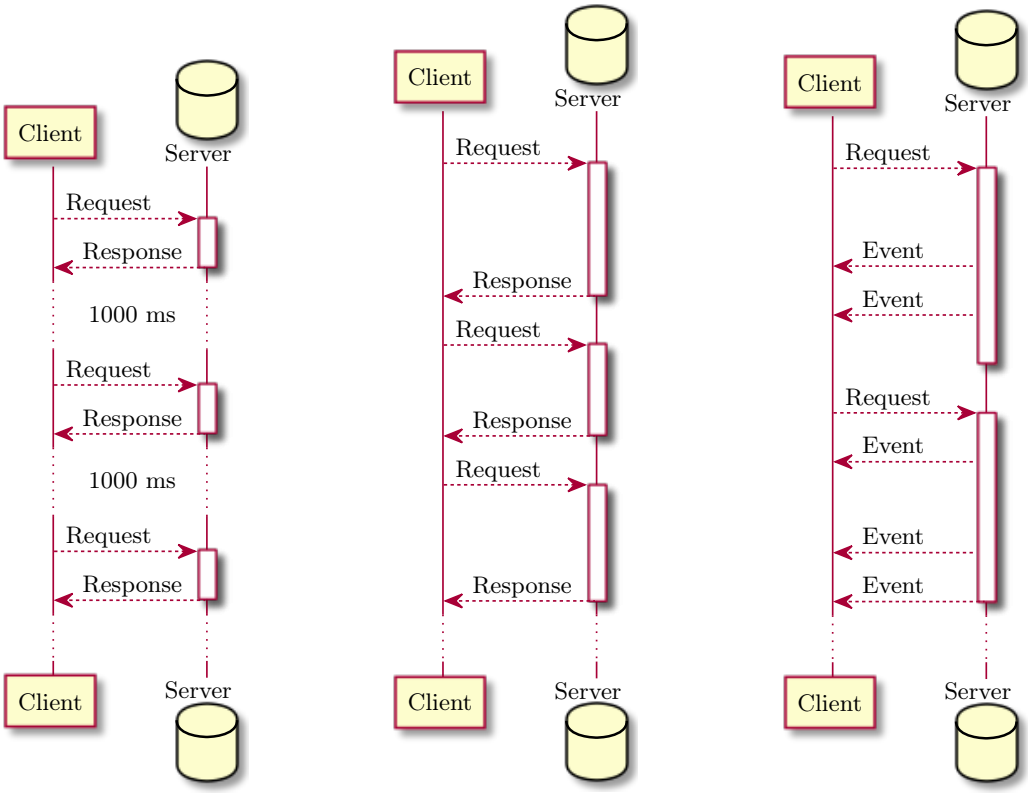
Abbildung 3.7: HTTP 2 - Server Push Beispiel

Dieses Problem wurde mit HTTP 2 gelöst, indem der Client mit einer Anfrage mehrere Dateien anfragen kann und der Server von sich aus auch sogenannte *Promises* an den Client senden kann. Fragt ein Client z.B. eine Datei an in welcher drei weitere Dateien referenziert sind, so kann der Server bei der Beantwortung der Anfrage auch direkt die drei referenzierten Dateien mitsenden [BPT15]. Dieser Ablauf ist in Abbildung 3.7 skizziert. Somit reduziert sich die benötigte Anzahl an Verbindungen für den Aufbau einer Webseite erheblich. Dies spiegelt sich dann auch in einer verbesserten Performance wieder.

Allerdings hat HTTP 2, genau wie sein Vorgänger, weiterhin den Nachteil, dass der Server nicht proaktiv, also außerhalb von Anfragen des Clients, Daten an den Client senden kann und dazu weiterhin ein *long polling* oder ähnliches Verfahren verwendet werden müsste.

Server Sent Events (SSE) werden benutzt, damit der Server Clients über neue Daten benachrichtigen kann. Dazu wird vom Client eine HTTP Verbindung geöffnet, über die der Server bei Bedarf Daten senden kann. Der Ansatz ähnelt dem des *long polling* sehr stark, allerdings mit dem großen Unterschied, dass bei SSE eine langlebige Verbindung genutzt wird. Diese wird nicht bei den ersten neuen Daten, die der Server erhält, geschlossen, sondern bleibt so lange wie möglich (häufig durch Timeouts limitiert) geöffnet. Sobald die

Verbindung geschlossen wird, beginnt der Ablauf wieder von vorne und es wird erneut eine langlebigen Verbindung geöffnet [LSA11]. Das Sequenzdiagramm in Abbildung 3.8c zeigt, wie ein solcher Aufbau funktionieren könnte.



(a) *polling* (b) *long polling* (c) SSE

Abbildung 3.8: *polling* vs *long polling* vs SSE

Es wurden bisher drei Möglichkeiten gezeigt, wie mit bestehenden Protokollen der Client neue Informationen vom Server mit möglichst wenig Verzögerung erhalten kann. Jedoch bietet keine der drei Möglichkeiten an, dass der Client dem Server über die selbe Schnittstelle Nachrichten senden kann. Im geplanten Anwendungsfall soll über die Wearable Devices die Interaktion mit den CTSs möglich sein. Dazu ist es erforderlich, dass die Wearable Devices PRODIS.WTS darüber benachrichtigen, sobald der Benutzer über die Wearable Devices eine Aktion ausgeführt hat. Dies könnte in allen drei Fällen mit einer vom PRODIS.WTS bereitgestellten HTTP Schnittstelle gelöst werden, an die die Wearable Devices ihre Daten senden können. Werden darüber jedoch sehr häufig Daten gesendet, gäbe es ähnliche Probleme wie beim *polling* oder *long polling*, denn pro Datenpaket wird eine eigene Verbindung benötigt. Da jedoch nicht zu erwarten ist, dass die User so viele Aktionen pro Sekunde durchführen müssen, dass sich die Anwendung merklich verlangsamt, wäre dieser Ansatz akzeptabel und widerspricht auch keiner der definierten Anforderungen.

Das WebSocket-Protokoll ermöglicht es bidirektional Daten zwischen Client und Server auszutauschen [FM11]. Im Vorfeld des Verbindungsaufbaus fragt der Client mittels HTTP an, ob der Server das Protokoll auf das WebSocket-Protokoll wechselt. Stimmt dieser dem Wechsel zu, sendet er seine Zustimmung zusammen mit einigen Verbindungsparametern an den Client. Dieser öffnet dann eine WebSocket Verbindung, über die im Fol-

genden Daten bidirektional ausgetauscht werden können [Wan13]. Das Sequenzdiagramm in Abbildung 3.9 zeigt einen solchen Ablauf.

Im Vergleich zu *polling*, *long polling* und SSE bietet WebSocket die Möglichkeit, dass der Client über dieselbe Verbindung, über die der Server Daten sendet, ebenfalls Daten an den Server senden kann. Dies erleichtert die Implementierung gegenüber SSE, da somit nur noch eine Schnittstelle benötigt wird, die eine bidirektionale Kommunikation über nur eine Verbindung ermöglicht. Da das WebSocket-Protokoll vergleichsweise neu ist und somit noch nicht von allen Geräten und Bibliotheken unterstützt wird, könnte dies der **REQ 5** widersprechen. Bei den weit verbreiteten Programmiersprachen, sowie bei den für mobile Geräte wichtigen Sprachen Java, Swift und Objective C, gibt es allerdings entsprechende Bibliotheken, wodurch unter anderem alle auf Android und iOS basierten Geräte das WebSocket-Protokoll ohne großen Aufwand integrieren können.

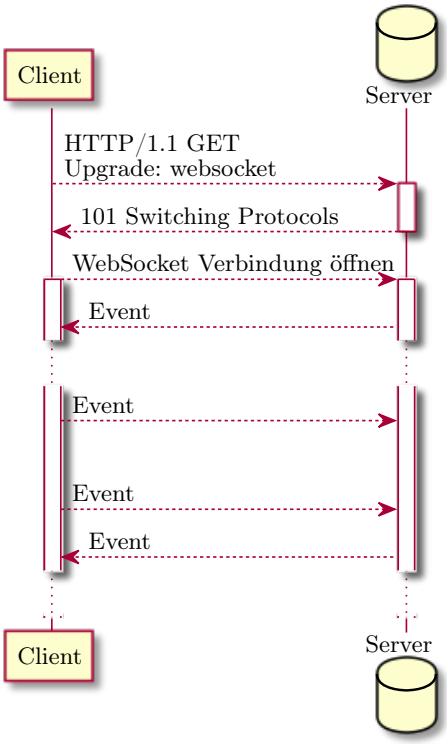


Abbildung 3.9: Beispielhafter Ablauf einer WebSocket Anwendung

Aufgrund der bidirektionalen Kommunikationsfähigkeit des WebSocket-Protokolls und den vorhandenen Bibliotheken bietet es sich an, eben jenes einzusetzen. Die später in der Implementierung eingesetzte Java Bibliothek *Atmosphere* bietet allerdings nicht nur WebSocket, sondern auch SSE und *long polling* an. Der Einsatz des WebSocket-Protokolls wird somit von dieser Arbeit empfohlen, allerdings lässt das Konzept und die Implementierung auch die Anbindung weiterer Geräte über SSE oder *long polling* zu.

3.4 GUI

Wie in **REQ 2** beschrieben, muss die GUI entweder komplett generisch sein, so dass die Hersteller eigene GUIs für die Wearable Devices erstellen können, oder die GUI muss

anhand des jeweiligen CTS automatisch generiert werden. Im Folgenden werden beide Ansätze inklusive ihrer Vor- und Nachteile vorgestellt.

3.4.1 Generischer Ansatz

Der generische Ansatz hat den Vorteil, dass lediglich die Bausteine zur Zusammenstellung einer GUI für Wearable Device bereitgestellt werden müssten, so dass die Hersteller die GUI damit aufbauen würden. Dieses Prinzip wird im PRODIS.Authoring bereits für die Erstellung der GUI eines CTS genutzt. Allerdings sind die Darstellungsmöglichkeiten von Wearable Devices naturgemäß sehr beschränkt, da die Bildschirme in der Regel nur wenige Zoll groß sind. Komplizierte Darstellungen mehrerer verschiedener, oder viel Raum einnehmender Bausteine, wie es im PRODIS.WTS durch deutlich größere Bildschirme möglich ist, wird es somit mit Wearable Devices nicht geben. Ein großer Nachteil dieses Ansatzes ist die, unter anderem aus dem Android Umfeld bekannte, Fragmentierung der Bildschirmgrößen und -formen [WLC16]. Schon die zwei in dieser Arbeit genutzten Wearable Devices haben nicht nur unterschiedliche Auflösungen, sondern auch eine unterschiedliche Bildschirmform. Zusätzlich muss das Konzept durch **REQ 5** weitere Geräte unterstützen. Die Schwierigkeit dabei ist, eine Darstellung zu finden, die auf möglichst vielen Geräten funktioniert. Der Hersteller müsste dafür für alle Wearable Devices eine eigene GUI mit den vorhandenen Bausteinen zusammenstellen, oder aber die eine vom Hersteller für die Wearable Devices konfigurierte GUI müsste sich selbstständig an die Umgebung der jeweiligen Wearable Devices anpassen können. Ersteres widerspricht **REQ 3** und wird somit nicht weiter betrachtet, da der Aufwand für die Hersteller viel zu groß ist. Letzteres fällt durch den Umsetzungsaufwand für solch eine Funktion weg, da PRODIS.Authoring um entsprechende Möglichkeiten für die Erstellung einer oder mehrerer GUIs erweitert werden müsste.

3.4.2 Generierung anhand vom CTS

Der zweite Ansatz, die Generierung der GUI anhand eines CTS, bedeutet, dass der Hersteller keine eigene GUI für die Wearable Devices erstellen muss und kann. Die GUI wird aus der bestehenden CTS GUI selbstständig erstellt. Der Hersteller muss neben der GUI für das CTS keine eigene GUI für die Wearable Devices definieren, kann dadurch allerdings auch deutlich weniger Anpassungen vornehmen. Ein Vorteil dieses Ansatzes ist, dass das Problem der Fragmentierung nun nicht mehr von jedem Hersteller, sondern einmalig von diesem Konzept betrachtet werden muss.

Im Folgenden werden daher verschiedene Ansätze vorgestellt, wie mit der Fragmentierung in diesem Ansatz umgegangen werden kann.

Größte GUI Komponente

Eine der einfachsten Varianten ist, dass nur die größte GUI Komponente auf den Wearable Devices dargestellt wird. Dies hat den Vorteil, dass der Hersteller nichts an den bestehenden CTS ändern muss. Allerdings gibt es einige Nachteile dieses Ansatzes. Ein komplexeres CTS kann zum Beispiel aus vielen verschiedenen ähnlich großen Elementen wie Listen, Tabellen oder Bilder bestehen. Ist nun die größte Komponente eine für die Diagnose eher unwichtige Komponente, werden die Wearable Devices nicht lang benutzt werden, da sie bei

den Fahrzeugdiagnosen keinen Mehrwert liefern. Auch kann es passieren, dass eine kleinere Komponente für eine mobile Darstellung auf einem Wearable Device deutlich interessanter im Kontext der Diagnose ist, als eine größere.

Platzhalter und vordefinierte Layouts

Mit Hilfe von vordefinierten Layouts und Platzhaltern kann dem Hersteller die Möglichkeit gegeben werden, sich die passenden Komponenten auszuwählen. So könnte die GUI der Wearable Devices mit Platzhaltern, wie in Abbildung 3.10 gezeigt, aufgebaut werden. Jedem dieser Platzhalter kann der Hersteller dann im CTS eine GUI Komponente mit Hilfe von Parametern zuordnen. Die Hersteller müssten dann bei jedem CTS, welches zusammen mit Wearable Devices genutzt werden soll, im PRODIS.Authoring entsprechende Parameter konfigurieren. Somit könnten die Hersteller die zur Diagnose relevanten GUI Elemente je CTS auswählen.

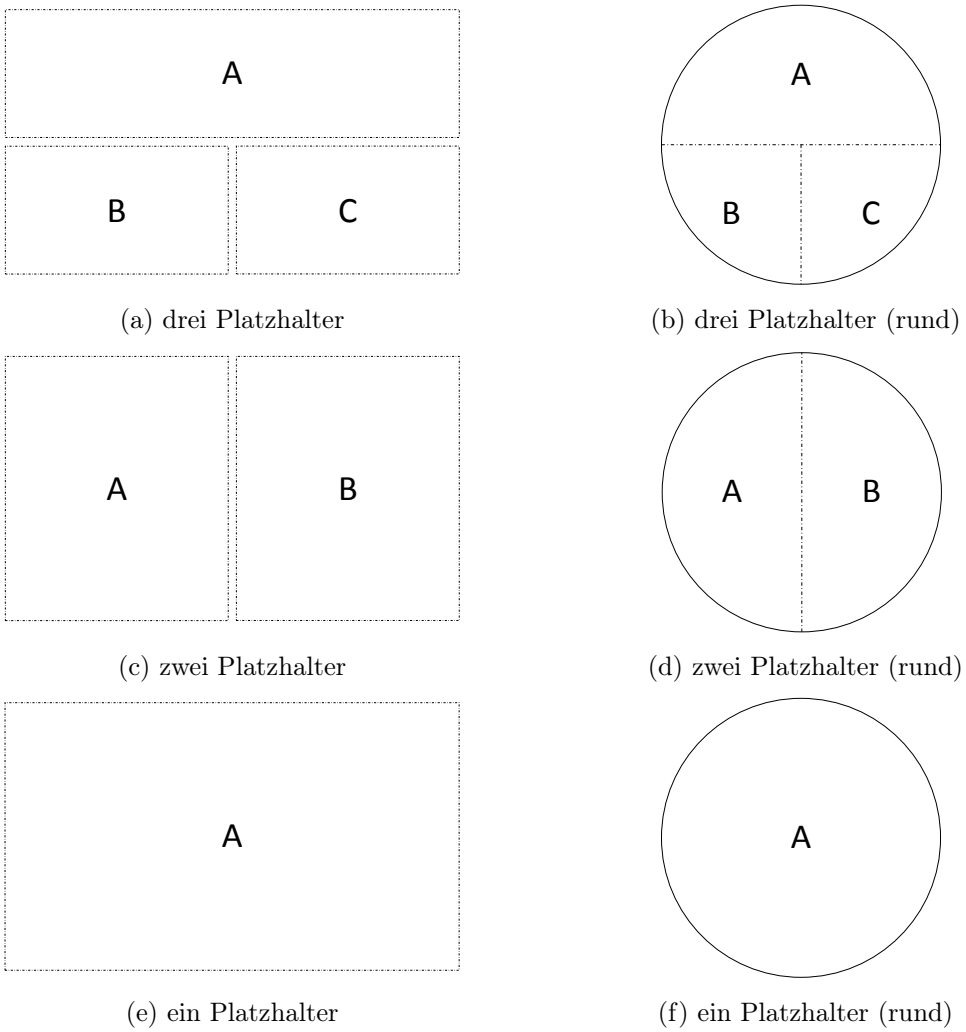
Können die Layouts vom Hersteller definiert werden, so müsste eine weitere Funktion im PRODIS.Authoring implementiert werden, die dies grafisch oder in einer anderen Form ermöglicht. Bei vordefinierten Layouts reicht es, wenn der Hersteller mit Hilfe eines Parameters im CTS das gewünschte Layout auswählt. Dies ist für die Hersteller etwas unflexibler und kann bei der Anbindung weiterer Geräte zu Problemen führen, wenn diese zum Beispiel nicht genügend Darstellungsraum bieten können. Andererseits wird dem Hersteller dadurch die Aufgabe abgenommen, sich mit der Problematik auseinander setzen zu müssen.

Weiterhin muss beachtet werden, dass es schon mit den zwei in dieser Arbeit genutzten Wearable Devices große Unterschiede in der Form des Bildschirms gibt. Dies führt zu Problemen, da zum Beispiel in den Abbildungen 3.10b und 3.10d deutlich weniger Raum zur Darstellung von Informationen vorhanden ist, als in den Abbildungen 3.10a und 3.10c. Des Weiteren bleiben die Bildschirme und Auflösungen der Wearable Devices sehr gering. Dadurch lassen sich die verschiedenen Layouts nicht sinnvoll nutzen, da die Informationen zu stark komprimiert werden müssten.

Wird dieser Ansatz auf ein einziges Layout mit nur einer einzigen Komponente limitiert, wie in den Abbildungen 3.10e und 3.10f, kann diese Problematik zumindest minimiert werden, da die Bildschirmgrößen der Wearable Devices optimal genutzt werden. Auch der Unterschied der Informationsdarstellung bei den verschiedenen Bildschirmformen ist somit am geringsten. Dabei verliert der Ansatz zwar einige Vorteile gegenüber mehreren Layouts, stellt aber sicher, dass die Anbindung weiterer Geräte sehr einfach ist, da diese immer nur eine Komponente auf dem gesamten Bildschirm darstellen müssen.

Durch **REQ 3** soll sichergestellt werden, dass die Hersteller so wenig Anpassungen wie möglich an ihren bestehenden CTS machen müssen. Trotzdem ist es wünschenswert den Herstellern die Möglichkeit zu gegeben, die Darstellung der GUI auf den Wearable Devices zu beeinflussen. Sonst könnten eventuell, wie zuvor beschrieben, für die Diagnose unnütze Komponenten auf den Wearable Devices dargestellt werden. Somit disqualifiziert sich die automatische Auswahl der größten GUI Komponente.

Nach **REQ 5** muss es möglich sein, dass später weitere Geräte angebunden werden können, ohne großen Aufwand zu investieren. Bei vielen verschiedenen Layouts wäre dies, wie zuvor beschrieben, ein Problem, da entweder die DSA GmbH oder die Hersteller bei neuen Geräten den Aufwand investieren müssten, die Layouts an die neuen Geräte anzupassen. Des



(a) drei Platzhalter

(b) drei Platzhalter (rund)

(c) zwei Platzhalter

(d) zwei Platzhalter (rund)

(e) ein Platzhalter

(f) ein Platzhalter (rund)

Abbildung 3.10: Beispielhafte Layouts

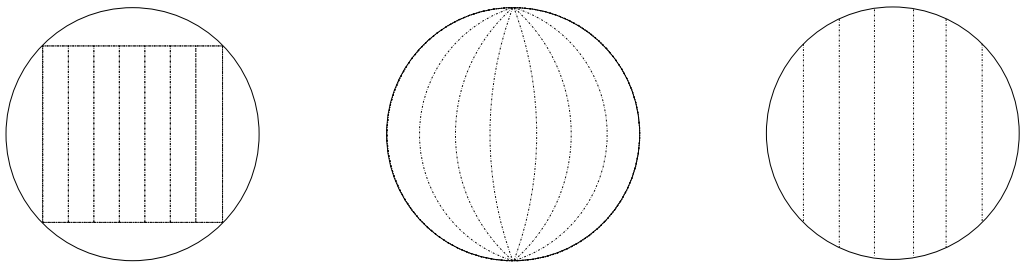
Weiteren funktioniert nicht jedes Layout auf jedem Gerät. Somit fällt auch die Nutzung mehrere Layouts weg.

Aufgrund von **REQ 3** und **REQ 5** arbeitet das Konzept somit mit einem einheitlichen Layout, in welchem die Hersteller genau eine darzustellende Komponente definieren können. Der Nachteil, dass die Möglichkeiten einiger Wearable Devices somit limitiert sind, ist zu verkraften. Es gibt sicherlich Diagnosen und Wearable Devices, bei denen die Darstellung mehrere GUI Komponenten wünschenswert ist. Für solche Fälle können die Hersteller zum Beispiel verschiedene GUI Komponenten eines CTS Screens auf verschiedene Zeitpunkte einer Fahrzeugdiagnose legen. Zu Beginn könnte zum Beispiel eine Tabelle hilfreich sein, im weiteren Verlauf der Diagnose ist aber beispielsweise ein Diagramm relevanter. Es wird zwar weiterhin immer nur eine GUI Komponente auf den Wearable Devices dargestellt, aber zumindest können verschiedene im Verlauf einer Diagnose von den Herstellern genutzt werden.

3.4.3 Darstellung in unterschiedlichen Bildschirmformen

Die zwei Wearable Devices in dieser Arbeit haben bereits unterschiedliche Bildschirmformen und -auflösungen. Für diese Unterschiede muss eine Lösung gefunden werden, wie sich die GUI Komponenten darstellen lassen. Als Beispiel haben Tabellen üblicherweise eine rechteckige Form, die Moto 360 allerdings ein rundes Layout. Das Design der Tabelle ist also von den Gegebenheiten der Wearable Devices abhängig und muss entsprechend angepasst werden. Diese Anpassung der Darstellung muss von den Apps der verschiedenen Wearable Devices vorgenommen werden, da nur diese die genauen Gegebenheiten berücksichtigen können.

Im Folgenden wird das Beispiel der Tabelle genutzt und gezeigt, wie dieses in einem runden Layout dargestellt werden kann. Es gibt zum Beispiel die Möglichkeit das Layout auf das größte Quadrat zu verkleinern wie in Abbildung 3.11a gezeigt, oder die Form anzupassen wie in Abbildung 3.11b. Auch die Minimierung des darzustellenden Inhaltes wäre eine Möglichkeit. So könnte bei einigen Wearable Devices der Bildschirm recht schmal sein und maximal 4-spaltige Tabellen anzeigen, während bei anderen auch 8-spaltige Tabellen möglich wären. Dies kann gelöst werden, indem zum Beispiel einige Spalten bei solchen Geräten entweder gar nicht angezeigt werden, oder, alternativ, mit Hilfe einiger Optionen in den Wearable Devices, die darzustellenden Spalten vom Werker ausgewählt werden können.



(a) im maximierten Quadrat (b) in gedehnten Spalten (c) in einem Kreis

Abbildung 3.11: Darstellung einer 7-spaltigen Tabelle

Die Darstellung im Quadrat hat den Nachteil, dass sehr viel Darstellungsraum ungenutzt bleibt und sich somit auch die darstellbaren Inhalte verringern. Werden hingegen rechteckige Formen an die runde Form angepasst, ergeben sich darstellerische Schwierigkeiten. Bei einer Tabelle wie in Abbildung 3.11b gezeigt, hätten die oberen und unteren Zeilen zum Beispiel deutlich weniger Platz zur Darstellung, da die Spalten dort deutlich enger sind. Soll der gesamte Darstellungsraum genutzt werden und möglichst gleich aufgeteilt sein, wäre auch eine Darstellung wie in Abbildung 3.11c möglich. Allerdings haben die äußeren Spalten deutlich weniger Darstellungsraum, als die Mittleren. Auch fällt der Vorteil einer tabellarischen Darstellung weg, bei der die Daten einer Zeile direkt nebeneinander auf der selben Höhe stehen und somit sehr einfach einander zuzuordnen sind.

Jede vorgestellte Variante hat Vor- und Nachteile, weshalb das Konzept hier keine feste Vorgabe macht. So kann es in manchen Szenario vorteilhaft sein, dass die Spalten und Zeilen gleichgroß sind und die Daten wie in einer normalen Tabelle dargestellt werden, während andere Szenarien so viel Spalten wie möglich benötigen und keinen Wert auf die Zeilen legen. Wie beschrieben, muss dieses Problem bei der Implementierung der Applikationen für die Wearable Devices näher betrachtet werden. Dieser Abschnitt gab dazu einige Denkanstöße, die für eine Entscheidungsfindung genutzt werden können, unterstrich

dabei aber nochmals, dass die Beschränkung auf eine einzelne GUI Komponente nicht alle Problematiken bei verschiedenen Wearable Devices löst.

3.4.4 Komponenten

Die Fahrzeughersteller haben mit dem gewählten Ansatz die Möglichkeit, jede Komponente auszuwählen, die auf den Wearable Devices dargestellt werden soll. Allerdings muss es für diese Komponenten auch eine Implementierung auf dem Wearable Devices geben, denn dort geschieht die Anpassung des Layouts auf die jeweiligen Gegebenheiten. Für diese Arbeit wurden zunächst zwei GUI Komponenten umgesetzt, um zu zeigen, dass das Konzept funktioniert:

- **Tabellen** werden im Umfeld vom PRODIS.WTS häufig eingesetzt. Sie haben den Vorteil, dass ähnlich strukturierte Daten zusammenhängend dargestellt werden können und es den Werkern somit erlauben, einen Überblick zu erhalten.
- **Scalable Vector Graphicss (SVGs)** werden häufig im PRODIS.WTS genutzt, um unter anderem die Werker bei der Diagnose oder Reparatur anzuleiten. Auch die technische Darstellung des zu diagnostizierenden Fahrzeuges ist eine häufig genutzte Variante, damit die Werker zum Beispiel einsehen können, wo sich welche Fahrzeugkomponenten befinden.

Die Auswahl dieser zwei Komponenten ist darin begründet, dass ein Großteil der weiteren Komponenten sich sehr ähnlich darstellen lassen. So unterscheidet sich zum Beispiel die Darstellung einer Liste oder eines Fließtextes in nur wenigen Punkten von der einer Tabelle und bei einem SVG-Bild ist die Darstellung sehr ähnlich zu der eines Diagramms. Eine Tabelle kann als eine Liste von Listen, in welchen sich Fließtext befinden kann, beschrieben werden. Ist also die Darstellung einer Tabelle möglich, könnten somit auch die anderen Komponenten dargestellt werden.

Zusätzlich werden die CTS Buttons, welche sich im PRODIS.WTS im Bereich **IV** befinden (siehe Abbildung 2.4), den Wearable Devices zur Verfügung gestellt. Mit diesen können die Werker über die Wearable Devices mit den CTSs interagieren. Diese Buttons werden, wenn das Wearable Device entsprechende Möglichkeiten besitzt, in einem Menü untergebracht. Dies verhindert, dass der GUI Komponente Darstellungsraum auf den kleinen Bildschirmen der Wearable Devices genommen wird. Bei den in dieser Arbeit genutzten Wearable Devices gibt es diese Möglichkeit, aber bei der Anbindung weiterer Geräte kann es vorkommen, dass diese Geräte keine Möglichkeit eines Menüs bieten. In diesen Fällen könnten die Buttons so klein wie möglich auf dem Bildschirm dargestellt oder mit Sprachbefehlen gearbeitet werden.

3.4.5 Parametrisierung

Durch den gewählten Ansatz, dass die GUI für die Wearable Devices nur aus einer einzigen Komponente besteht, haben die Hersteller bisher keine Anpassungsmöglichkeiten. In Abschnitt 3.4 wurde allerdings bereits die Idee von Parametern beschrieben. Das Ziel dabei ist es, den Herstellern einen gewissen Freiraum für Anpassungen an der Darstellung

der Komponente zu gewähren. Dazu werden im Folgenden die Parameter definiert, welche im PRODIS.Authoring in den CTSs genutzt werden können, um entsprechende Werte festzulegen. Dabei steht der Platzhalter **X** am Ende der Parameternamen hierbei für eine beliebige Nummer.

Allgemeine Parameter

Zunächst benötigt die Anbindung von Wearable Devices allgemeine Parameter, mit welchen zum Beispiel definiert wird, welche GUI Komponente dargestellt werden soll.

- ComponentOid – Hiermit wird die Object Identifier (OID) der GUI Komponente definiert.
- ScreenOid – Hiermit wird die OID des Screens definiert, auf der sich die Komponente befindet.
- ButtonId**X** – Über diese Parameter können die Buttons des CTS definiert werden, für welche auf den Wearable Devices ein Menüeintrag erstellt werden soll.

Parameter bei Tabellen

Bei einer tabellarischen Darstellung ist es zum Beispiel wünschenswert die Spalten und Zeilen, welche auf den Wearable Devices dargestellt werden sollen, für die Hersteller konfigurierbar zu machen. Wie zuvor bereits erläutert, kann PRODIS.WTS deutlich mehr Informationen auf den Bildschirmen der Diagnosegeräte, also Notebooks und Tablets, darstellen als die Wearable Devices auf ihren deutlich kleineren Bildschirmen. Dementsprechend definiert dieses Konzept zunächst die folgenden Parameter. Durch **REQ 4** können später weitere Parameter definiert werden, ohne die auf älteren Versionen laufenden Wearable Devices zu beeinträchtigen.

- Row**X** – Hierüber können die Zeilen einer Tabelle definiert werden, welche immer auf den Wearable Devices dargestellt werden.
- ShowSelectedRow – Ein boolescher Wert, welcher definiert, ob die markierte Zeile einer Tabelle immer auf den Wearable Devices dargestellt wird.
- Column**X** – Hiermit können die Spalten einer Tabelle definiert werden, welche auf den Wearable Devices dargestellt werden.

Parameter bei SVGs

Für SVGs wird bisher lediglich ein Parameter vom Konzept definiert:

- CreateMenuItemsForElementsWithId – PRODIS.Authoring und PRODIS.WTS ermöglichen es auf Klick-Events von SVG Elementen mit IDs zu reagieren. Damit diese Möglichkeit auch bei Wearable Devices gegeben ist, kann mit diesem booleschen Parameter definiert werden, dass für diese Elemente des SVG ein Menüeintrag erstellt wird.

3.5 Verbindungsaufbau

Wie in Abschnitt 3.3 beschrieben, wird eine Client-Server-Kommunikation mit PRODIS.WTS als Server und den Wearable Devices als Client umgesetzt. Im Folgenden wird beschrieben, wie die Wearable Devices die nötigen Verbindungsinformationen, wie zum Beispiel die IP-Adresse des Rechners auf dem PRODIS.WTS läuft, erhalten.

Zuallererst muss sichergestellt sein, dass sich die Wearable Devices und das PRODIS.WTS im selben Netzwerk befinden und untereinander kommunizieren können. Ist dies gegeben, müssen die Wearable Devices, aufgrund des Einsatzes des WebSocket-Protokolls, eine Verbindung mit PRODIS.WTS aufbauen. Dazu benötigen sie die IP-Adresse und den Port auf dem PRODIS.WTS auf neue Verbindungen lauscht. Diese muss der Werker entweder manuell auf den Wearable Devices eingeben, oder die Daten könnten zum Beispiel durch akustische oder optische Signale vom PRODIS.WTS an die Wearable Devices übertragen werden.

3.5.1 Akustische Signale

Die beiden Wearable Devices Moto 360 und Google Glass verfügen zwar über ein Mikrofon, mit welchem akustische Signale aufgenommen werden können, jedoch hat dieser Ansatz drei große Nachteile:

- *Störgeräusche*
  In Werkstätten kann es recht laut werden, wodurch viele Störgeräusche eine solche Übertragung von Informationen stören können.
- *Aufwand*
  Zur Implementierung einer akustischen Datenübertragung gibt es vergleichsweise wenig Bibliotheken. Der Aufwand solch eine Schnittstelle zu implementieren ist somit für diese Arbeit deutlich zu hoch.
- *Fehlende Lautsprecher*
  Es ist nicht sichergestellt, dass die Systeme, auf denen PRODIS.WTS ausgeführt wird, entsprechende Schnittstellen, wie zum Beispiel Lautsprecher, haben, um akustische Signale senden zu können.

Diese Nachteile widersprechen dem Einsatz einer akustischen Datenübertragung in dieser Arbeit.

3.5.2 Optische Signale

Für die Google Glass ist eine optische Darstellung auf dem Monitor des PRODIS.WTS denkbar, da sie über eine Kamera verfügt, die die Daten erfassen kann. Die optische Darstellung ist zwar für die Moto 360 ungeeignet, da diese über keinen optischen Sensor verfügt, für die Google Glass und allen weiteren Geräten mit entsprechender Sensorik, ist dieser Ansatz jedoch sehr komfortabel. Um ihn umzusetzen, müsste PRODIS.WTS die Verbindungsinformationen mittels einer sogenannten optoelektronischen Schrift auf dem Bildschirm darstellen.

Es gibt mittlerweile viele verschiedene Arten dieser Schriften. Der klassische Strichcode, auch Barcode genannt, wie in Abbildung 3.12a dargestellt, wird zum Beispiel häufig im Einzelhandel eingesetzt. Dort identifiziert er den gekauften Artikel an der Kasse, damit diese den Preis ermitteln kann. In den letzten Jahren hat sich, gerade in Verbindung mit Smartphones, die Nutzung von Quick Response Codes (QR-Codes), wie in Abbildung 3.12b gezeigt, stark verbreitet [UH12]. Dies sind zweidimensionale Codes, welche 1994 von einer japanischen Firma veröffentlicht und frei zugänglich gemacht wurden [OKL11]. Ein QR-Code besteht aus weißen und schwarzen Quadraten mit welchen die enthaltenen Daten binär dargestellt werden, sowie aus, in drei Ecken angebrachten, Positionsmarkern und gegebenenfalls zwischen den Quadraten dargestellten Ausrichtungsmarkern, welche es den Scannern ermöglichen die Position und Orientierung des QR-Codes zu bestimmen [ISO15].

QR-Codes haben einige Vorteile gegenüber Barcodes. So bieten sie eine automatische Fehlerkorrektur, ermöglichen eine schnelle Erkennung und benötigen durch ihren zweidimensionalen Aufbau deutlich weniger Darstellungsraum als ein Barcode [OKL11]. Die Abbildung 3.12 zeigt einen Barcode im Code-128 Format neben einem QR-Code mit dem selben Inhalt. Es ist deutlich zu erkennen, dass die Länge des Barcodes bei weiteren Informationen die Breite eines DIN A4 Blattes überschreiten würde. Der QR-Code dagegen kann noch deutlich mehr Informationen darstellen, da er sich sowohl horizontal als auch vertikal ausdehnt.

(a) Code-128 (b) QR-Code

Abbildung 3.12: Darstellung von *255.255.255.255:12345* in zwei optoelektronischen Schriften

Mit QR-Codes lassen sich somit auch deutlich mehr und komplexere Daten abbilden. So ist es auch möglich die Anmeldeinformationen zum WLAN, in welchem PRODIS.WTS sich befindet, im QR-Code mit zu übergeben. Dies ist mit Code-128 und anderen eindimensionalen Barcodes auf begrenzten Darstellungsraum nur bedingt bis zu einer gewissen Menge und Komplexität an Daten möglich, weshalb diese nicht flexibel genug sind um in Zukunft komplexere und mehr Daten anzuzeigen. Neben den QR-Codes gibt es noch weitere zweidimensionale Codes, wie zum Beispiel PDF417, Data Matrix und MAXI Code. Allerdings ist die Erkennung, Verbreitung und Unterstützung von QR-Codes deutlich besser [LSO+15]. Außerdem vereint der QR-Code die Vorteile vieler anderer Codes. So bietet er ähnlich wie PDF417 eine hohe Datenkapazität, benötigt dabei wie Data Matrix wenig Darstellungsraum und kann wie der MAXI Code schnell gelesen werden [Soo08]. Durch die weite Verbreitung der QR-Codes gibt es auch entsprechende Unterstützung von Bibliotheken für viele Programmiersprachen. Somit bietet der QR-Code viele Vorteile gegenüber eindimensionalen und anderen zweidimensionalen Codes und wird im Folgenden dazu genutzt, die Verbindungsdaten im PRODIS.WTS darzustellen.

3.5.3 Manuelle Eingabe

Durch **REQ 5** soll ermöglicht werden, dass in Zukunft auch weitere Geräte angebunden werden können, welche andere Sensoren beinhalten. Es wird daher schwierig eine Möglichkeit zu finden, um die Verbindungsinformationen allen Geräten automatisiert zur Verfügung stellen zu können. Für Geräte mit einer Kamera kann ein QR-Code genutzt werden, bei allen anderen Geräten, inklusive der Moto 360, müssen die Daten in irgendeiner Form manuell eingegeben werden. Sie können zum Beispiel vorher auf den Wearable Devices konfiguriert werden, so dass der Werker sie einfach auswählen kann. Alternativ kann dem Werker auch eine Möglichkeit bereitgestellt werden, die Verbindungsinformationen manuell einzutragen, indem diese zusätzlich zum QR-Code, ähnlich wie in Abbildung 3.12a beim Code-128, im Klartext dargestellt werden. Letzteres wird in dieser Arbeit umgesetzt, kann aber später auch um andere Möglichkeiten erweitert werden.

3.6 Daten und Datenserialisierung

Nachdem bisher das Übertragungsmedium und die genutzten Protokolle definiert wurden, werden nachfolgend die zu übertragenden Daten behandelt. Wie bereits zuvor in Abschnitt 3.2 beschrieben, stellen die Wearable Devices lediglich weitere Views im MVC Konzept dar. Dementsprechend müssen zwischen PRODIS.WTS und den Wearable Devices zum Beispiel Informationen über das Aussehen, die Inhalte und Events der GUI ausgetauscht werden. Zunächst wird das Datenschema und anschließend das Übertragungsformat, welches beschreibt, wie die Daten serialisiert werden, definiert.

3.6.1 Datenschema

Das Datenschema wurde zusammen mit dem *ParameterPasser* definiert und wird im Umfeld von PRODIS.WTS an vielen Stellen genutzt. Damit nicht zwei verschiedene Schemata für die selbe Aufgabe genutzt werden, wird das Datenschema in dieser Arbeit nicht verändert. Es ist in drei voneinander abhängige Teile strukturiert, welche im Folgenden vorgestellt und ihre Funktionen erläutert werden. Dabei ist der Wert des Datenfelds VALUE abhängig vom Wert des Datenfelds KEY und dieses wiederum ist abhängig vom Wert des Datenfelds TYPE.

Datenfeld: TYPE

Mit dem Datenfeld TYPE wird beschrieben, welche Inhalte dieses Datenpaket enthält. Sind es zum Beispiel Daten die nur für Wearable Devices interessant sind, so kann dies über dieses Feld definiert werden. PRODIS.WTS unterscheidet unter anderem zwischen:

- **TITLE_DATA**: Beinhaltet Daten zu den Überschriften des CTS, also den obersten, mit **I** markierten Bereich in Abbildung 2.4 auf Seite 9.
- **INSTRUCTION_DATA**: Beinhaltet Daten zu den Diagnoseanweisungen des CTS. Dies ist der Bereich, welcher mit **II** in Abbildung 2.4 markiert ist.

- **SCREEN_DATA**: Beinhaltet Informationen über die darzustellenden GUI Komponenten. Dazu zählen unter anderem die Diagnosedaten und Events, wenn etwa eine Zeile in einer Tabelle selektiert wurde.
- **SCREEN_MANAGER_DATA**: Beinhaltet Informationen über die Screens des CTS.
- **BUTTON_DATA**: Beinhaltet Informationen über die globalen und lokalen Buttons des CTS. Erstere befinden sich im mit **II** markierten Bereich, letztere im untersten, mit **IV** markierten Bereich in Abbildung 2.4.
- **WEAR_DATA**: Beinhaltet nur für Wearable Devices relevanten Informationen. Dazu gehören zum Beispiel die in den Abschnitten 3.4.4 und 3.4.5 beschriebenen Parameter und GUI Komponenten.

Datenfeld: KEY

Mit dem Datenfeld KEY wird üblicherweise beschrieben, warum dieses Datenpaket übertragen wird. Häufig ist der Grund eine Änderung von Daten oder die Interaktion mit GUI Komponenten. Es wird mit dem Datenpaket also beschrieben, welche Aktion der Auslöser dieses Datenpakets ist. Einige mögliche Aktionen sind:

- **TITLE_CHANGED** / **INSTRUCTION_CHANGED**: Dieses Datenpaket wird übertragen, weil sich die Überschrift oder die Anweisungen des CTS geändert haben.
- **BUTTON_CREATED** / **BUTTON_CLICKED** / **BUTTON_REMOVE**: Dieses Datenpaket wird übertragen, weil ein Button hinzugefügt, geklickt oder entfernt wurde.
- **TABLE_INIT** / **SVG_INIT**: Das Datenpaket wird übertragen, weil eine Tabelle oder ein SVG initialisiert wurde.

An dieser Stelle ist die Aufteilung der Datenfelder nicht konsequent durchgeführt worden. Es gibt die folgenden zwei Ausnahmen:

- Bei TYPE = **SCREEN_MANAGER_DATA** beinhaltet der KEY die ID des Screens. In diesem Fall wurden der Inhalt von KEY und VALUE miteinander vertauscht.
- Bei allen Aktionen, die mit einer GUI Komponente des CTS zusammenhängen, beinhaltet der KEY neben dem Namen der veränderten Eigenschaft, zusätzlich noch die ID des Screens und der Komponente, getrennt durch ein Ausrufezeichen. Dadurch kann die Aktion direkt einer bestimmten Komponente zugeordnet werden. Ein Beispielwert für KEY ist somit **SCREEN_1!TABLE_2!propSelectedRow**, welcher beschreibt, dass sich die ausgewählte Zeile, in der Tabelle mit der ID 2 auf dem Screen mit der ID 1, verändert hat.

Bei der zweiten Ausnahme könnte ein optionales viertes Datenfeld, welches nur mit den entsprechenden IDs gefüllt wird, wenn sich das Datenpaket auf eine GUI Komponenten

bezieht, eine klarere Aufteilung ermöglichen. Da der Aufbau der Datenpakete im *ParameterPasser* jedoch bereits im Vorfeld dieser Arbeit so definiert ist, wird das Datenschema nicht verändert. Ansonsten könnte es bei der Wartung und Weiterentwicklungen zu Missverständnissen und Fehlentwicklungen kommen.

Datenfeld: VALUE

Das Datenfeld VALUE beinhaltet den eigentlichen Inhalt, welcher von den zwei vorherigen Datenfeldern TYPE und KEY abhängig ist. Dabei kann es sich um sehr einfache Daten handeln. Wurde eine Zeile in einer Tabelle selektiert, enthält dieses Feld die Nummer der selektierten Zeile. Allerdings gibt es auch komplexere Fälle. Wird zum Beispiel ein Button hinzugefügt, so beinhaltet dieses Datenfeld alle Informationen über den Button. Das sind unter anderem dessen ID, Aufschrift und eventuell ein Icon. Damit solche Fälle ebenfalls mit nur einem Datenfeld abgedeckt werden können, werden die Daten in ein entsprechendes Objekt konvertiert und dieses dem Datenfeld VALUE zugeordnet. Abbildung 3.13 zeigt, wie einige der obigen Datenfelder zusammenhängen.

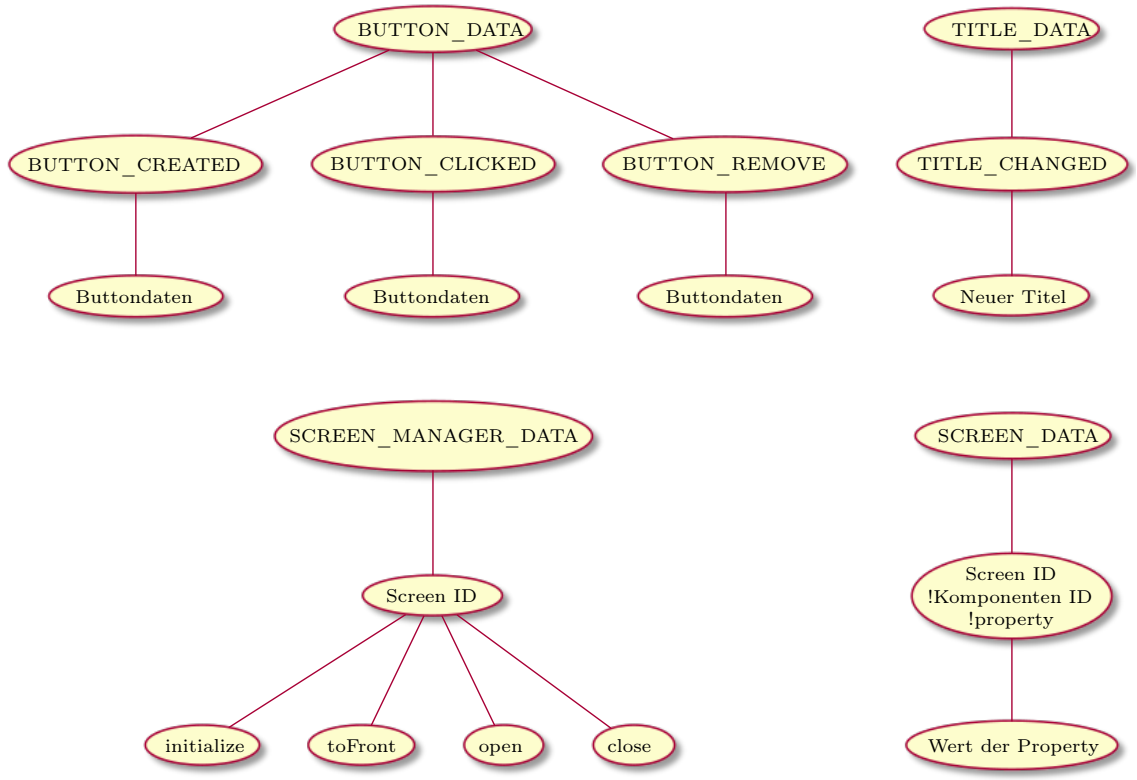


Abbildung 3.13: Ausschnitt möglicher Kombinationen von Datenfeld-Werten

3.6.2 Datenserialisierung

Die zuvor beschriebenen Daten müssen vom *ParameterPasser* an die Wearable Devices gesendet werden. Sie verlassen die Laufzeitumgebung des PRODIS.WTS, müssen also von einem Java Objekt so umgewandelt werden, dass sie übertragen und von den Wearable Devices gelesen werden können.

Mit dieser Umwandlung von strukturierten Daten in speicherbare Formate, wie Strings, Dateien oder Bytes, beschäftigt sich die Serialisierung von Daten. Mit Hilfe einer Deserialisierung können die Daten dann anschließend aus diesen Formaten wieder konstruiert werden. Datenserialisierungen werden häufig eingesetzt, um aus der Programmiersprache heraus Objekte von verschiedenen Klassen in eine Datenbank zu persistieren oder über ein Netzwerk zu übertragen.

Es gibt viele Möglichkeiten Daten zu serialisieren. Einige Verfahren erstellen dabei menschenlesbare Formate wie Extensible Markup Language (XML) [MYB+08] oder JavaScript Object Notation (JSON) [Bra14]. Andere Verfahren, wie zum Beispiel Googles *Protocol Buffers*, versuchen die CPU Last zu minimieren und erstellen kompakte und einfach zu gliedernde Formate, die jedoch wenig bis keinen Fokus auf die Menschenlesbarkeit legen [Pro16a].

Im Folgenden werden die vorher genannten Formate vorgestellt und, bezogen auf den geplanten Anwendungsfall, die Vor- und Nachteile dargelegt.

XML

XML ist eine Auszeichnungssprache, auch Markupsprache genannt, welche strukturierte Daten in Textform darstellt [MYB+08]. XML ist sowohl maschinen- als auch menschenlesbar und wird häufig zum Austausch von Daten zwischen verschiedenen Systemen eingesetzt. Viele Sprachen basieren auf XML, so zum Beispiel das in der Webentwicklung bekannte Extensible HyperText Markup Language (XHTML), welches vom World Wide Web Consortium (W3C) als Nachfolger von Hypertext Markup Language (HTML), mit welchem alle Webseiten aufgebaut sind, standardisiert wurde [Pem02].

Mit XML können durch seine Erweiterbarkeit viele verschiedene Informationen, so genannte Metadaten, neben den eigentlichen Daten übermittelt werden. So kann mit XML Schema Definition (XSD) der Datentyp eines jeden Wertes definiert [TMM+12], oder mit XML Linking Language (XLink) Verlinkungen zu anderen Ressourcen angegeben werden [ODM01]. Aber auch im Umfeld von XML gibt es viele Möglichkeiten. So können mittels XML Path Language (XPath) aus einem XML Dokument bestimmte Daten ausgelesen [DC99] oder mittels Extensible Stylesheet Language Transformations (XSLT) sogar das ganze Dokument komplett umstrukturiert oder neu aufgebaut werden [Cla99].

Die vorher genannte Erweiterbarkeit kann bei falscher Nutzung schnell zum Nachteil werden, denn sie kann somit zu einem Informationsüberschuss führen. Der Server reichert dabei das XML Dokument mit vielen Metadaten an, welche die Clients zum Großteil nie oder nur sehr selten benötigen. Dadurch vergrößert sich das Dokument und verlangsamt die Kommunikation. Auch ohne die Metadaten benötigt ein XML Dokument bereits vergleichsweise viel Speicherplatz.

Des Weiteren eignet sich XML nicht für Binärdaten, da es nur aus Text besteht. Mittels einer Kodierung wie Base64 können zwar Binärdaten in Text konvertiert werden, allerdings benötigt der Text 33% mehr Platz als die reinen Binärdaten [Jos06].

Listing 3.1 zeigt ein beispielhaftes Datenpaket, bestehend aus den in Abschnitt 3.6.1 definierten Datenfeldern.

```xml
<?xml version="1.0"?>
<datapackage>
  <type>INSTRUCTION_DATA</type>
  <key>INSTRUCTION_CHANGED</key>
  <value>Beispiel-Instruktion</value>
</datapackage>
```

Listing 3.1: Daten serialisiert als XML

JSON

JSON ist ein menschen- und maschinenlesbares Datenformat, welches in Textform dargestellt wird und seinen Ursprung in der Programmiersprache JavaScript hat [Bra14, Int13]. Im Vergleich zu XML ist es etwas kompakter, da unter anderem die XML-Deklaration, der Root-Tag und die schließenden Tags wegfallen. JSON wird häufig in der Webentwicklung eingesetzt, da es in JavaScript sehr gut integriert ist und wie native Objekte verwendet werden kann. Zur reinen Übertragung von Daten ist JSON häufig ein sehr guter Kandidat, da es sehr kompakt ist und trotzdem die Menschenlesbarkeit beibehält. Allerdings bietet JSON deutlich weniger Erweiterungsmöglichkeiten als XML. So kann den Werten mit JSON zum Beispiel keine zusätzlichen Informationen, wie Datentypen und andere Metadaten, zugeordnet werden.

Aber auch das Umfeld um JSON bietet einige Möglichkeiten mit dem JSON Dokument zu arbeiten. So gibt es mit JSON-Path ein ähnliches Tool für JSON Daten wie XPath für XML Daten [Fri16]. Damit können mit einer eigenen Syntax, Pfade entlang eines JSON Objektes definiert werden, über welche dann der Inhalt ausgelesen werden kann. Dies kann im PRODIS.WTS zum Beispiel dafür benutzt werden, um aus Nachrichten bestimmte Felder auszulesen und anhand derer zu entscheiden, ob dieses Datenpaket relevant für die Wearable Devices ist oder nicht. Somit lässt sich bereits serverseitig die Netzwerk- und Gerätelast bei den Wearable Devices reduzieren.

Jedoch eignet sich JSON genau so wenig für Binärdaten wie XML, da dazu ebenfalls eine Kodierung wie Base64 genutzt werden muss. Listing 3.2 stellt das selbe beispielhafte Datenpaket aus Listing 3.1 als JSON Dokument dar. Dabei ist gut zu erkennen, dass sich die benötigten Zeichen im Vergleich zu XML reduziert haben. Ohne die unsichtbaren Zeichen (u.a. Leerzeichen und Zeilenumbrüche) und die eigentlichen Werte der drei Datenfelder, besteht das XML Dokument in Listing 3.1 aus 87 Zeichen, während das JSON Dokument aus Listing 3.2 lediglich aus 32 Zeichen besteht. Selbst wenn im XML Dokument der Name des Root-Tags *datapackage* auf ein statt elf Zeichen verkürzt wird, ist dies eine Einsparung von über 50% gegenüber XML.

```json
{
  "type" : "INSTRUCTION_DATA",
  "key"  : "INSTRUCTION_CHANGED",
  "value": "Beispiel-Instruktion"
}
```

Listing 3.2: Daten serialisiert als JSON

Protocol Buffers

Protocol Buffers, auch Protobuf genannt, wurde von Google entwickelt und ist ein Datenformat zum effizienten Serialisieren von strukturierten Daten. Anders als XML und JSON,

ist Protocol Buffers ein binäres Datenformat, welches großen Wert auf die Performance und Kompaktheit legt und nicht auf die Menschenlesbarkeit [Pro16a].

Für Protocol Buffers muss zunächst ein Datenschema erstellt werden, welches dann mit Hilfe eines eigenen Compilers in verschiedene Programmiersprachen übersetzt wird. Der Compiler von Protocol Buffers unterstützt zur Zeit die Sprachen Java, C++, Python, Objective-C, JavaScript und einige weitere bekannte und häufig benutzte Programmiersprachen [Pro16a]. Listing 3.3 zeigt solch ein Schema für das in Abschnitt 3.6.1 beschriebene Datenpaket. Dabei wurde das Datenfeld VALUE der Einfachheit halber ebenfalls als String deklariert, mit Protocol Buffers ist aber auch eine Verschachtelung von verschiedenen Schemas möglich. Die Werte hinter den Gleichheitszeichen der Felder definieren die ID des jeweiligen Feldes, welche von Protocol Buffers dazu genutzt werden die Datenfelder bei der Serialisierung zu identifizieren.

```
1 syntax = "proto3";
2 message DataPackage {
3     string type = 1;
4     string key = 2;
5     string value = 3;
6 }
```

Listing 3.3: Schema Deklaration des Datenpakets mit Protocol Buffers

Im Folgenden wird mit Hilfe dieses Schemas und den Beispieldaten aus Listings 3.1 und 3.2 gezeigt, wie eine serialisierte Protocol Buffers-Nachricht im Vergleich zu XML oder JSON aufgebaut ist. Listing 3.4 zeigt in den ungeraden Zeilen eine solche Nachricht, während, zum besseren Verständnis, in den geraden Zeilen die Werte der drei Datenfelder unterhalb der zugehörigen Bytes dargestellt sind. Je zwei Zeichen der Nachricht stellen dabei ein Byte im Hexadezimalsystem dar.

Das erste Byte eines Datenfeldes, hier blau hervorgehoben, enthält die ID des Feldes, sowie seinen Datentypen. Zur Veranschaulichung wird das Byte in das Binärsystem umgewandelt, wodurch sich aus $0A_{16}$=00001010_2 ergibt. Der Datentyp ist in den hintersten drei Bits untergebracht, hier also 010_2=2_{10}. Es gibt aktuell sechs Datentypen in Protocol Buffers, wobei der Datentyp 2 für längenbegrenzte Werte steht, also zum Beispiel für Strings und Bytes [Pro16b]. Aus den restlichen fünf Bits ergibt sich dann die ID des Feldes, hier also 00001_2=1_{10}, welches die ID des Datenfelds TYPE ist.

Bei längenbegrenzten Werten folgt mindestens ein weiteres Byte zur Angabe der Länge des Wertes. Die genaue Anzahl benötigter Bytes ist abhängig von der Länge des Wertes und kann mit der Formel $f(x) = \lceil log(x+1)/7 * log(2) \rceil$ bestimmt werden. In Listing 3.4 sind diese Bytes in Rot hervorgehoben. So definiert zum Beispiel das Byte 10_{16}=16_{10}, dass die folgenden 16 Bytes den Wert des Datenfeldes definieren und somit ab dem siebzehnten Byte wieder ein neues Datenfeld beginnt.

```
1 0A 10 49 4E 53 54 52 55 43 54 49 4F 4E 5F 44 41 54 41
2       I  N  S  T  R  U  C  T  I  O  N  _  D  A  T  A
3 12 13 49 4E 53 54 52 55 43 54 49 4F 4E 5F 43 48 41 4E 47 45 44
4       I  N  S  T  R  U  C  T  I  O  N  _  C  H  A  N  G  E  D
5 1A 14 42 65 69 73 70 69 65 6C 2D 49 6E 73 74 72 75 6B 74 69 6F 6E
6       B  e  i  s  p  i  e  l  -  I  n  s  t  r  u  k  t  i  o  n
```

Listing 3.4: Beispielnachricht mit Protocol Buffers serialisiert

Strings werden im UTF-8 Encoding serialisiert und sind in Listing 3.4 in den geraden Zeilen unterhalb der entsprechenden Bytes in Grau dargestellt. Insgesamt benötigen die

serialisierten Beispieldaten 61 Bytes, wobei nur 6 Bytes von Protocol Buffers stammen. Verglichen mit den 32 Zeichen bei JSON und 87 Zeichen bei XML, spart Protocol Buffers somit nochmals deutlich Speicherplatz.

Ein Nachteil von Protocol Buffers ist die Unleserlichkeit der serialisierten Nachricht, wie das Beispiel eindeutig zeigt. Werden die Bytes direkt als UTF-8 kodierter String gelesen, können zwar die Werte der Datenfelder eingesehen werden, allerdings kann nicht erkannt werden, zu welchem Datenfeld sie gehören. Des Weiteren ist der Umgang mit Protocol Buffers verglichen mit JSON und XML deutlich komplexer. Zwar können etwa 26 Bytes gegenüber JSON mit Protocol Buffers pro serialisierter Nachricht eingespart werden, allerdings führt diese Einsparung im geplanten Anwendungsszenario nicht zu einer spürbaren Performancesteigerung.

3.6.3 Auswahl des Datenformats

Beim geplanten Einsatz des Datenformats in dieser Arbeit, ist ein reines Datenformat, welches möglichst kompakt ist und keinen der Anforderungen des Konzeptes widerspricht, wünschenswert. Protocol Buffers ist zwar das kompakteste Format, bietet bei dem geplanten Datenvolumen aber nur einen geringen Mehrwert. Dafür verkompliziert es die Implementierung, also die Anbindung weiterer Geräte, sowie die Fehlersuche, wodurch Protocol Buffers für diesen Anwendungsfall nicht geeignet ist.

Dem Konzept bieten die zusätzlichen Funktionen von XML keinen Mehrwert gegenüber von JSON. Somit fällt die Wahl auf JSON, welches eine gute Menschenlesbarkeit bei einer guten Kompaktheit vorweisen kann und fast alle Anforderungen dieses Anwendungsfalls erfüllt. Lediglich bei der Übertragung von Binärdaten hat JSON einige Nachteile. Allerdings werden diese nur selten übertragen, weshalb die Mehrdaten durch die Umwandlung in Base64 vertretbar sind.

3.7 Kommunikationsablauf

Im Folgenden wird der Ablauf der Kommunikation zwischen PRODIS.WTS und allen Views, also sowohl den Screens der CTSs, als auch den Wearable Devices, definiert. Wie in Abschnitt 3.2 beschrieben, sind die Informationen, welche die Views erhalten, größtenteils gleich.

Initialisierung der GUI

Nachdem sich die Wearable Devices mit dem PRODIS.WTS verbunden haben, sendet PRODIS.WTS über die Verbindung zunächst Informationen über die darzustellende GUI Komponente und deren Eigenschaften. Dazu wird ein Datenpaket übertragen, in welchem das Datenfeld Type, wie in Abschnitt 3.6.1 beschrieben, den Wert *WEAR_DATA* beinhaltet. Dieses Datenpaket enthält also nur für Wearable Devices relevante Informationen. Im Datenfeld Key wird die Art der GUI Komponente übertragen und kann entsprechend dem Abschnitt 3.4.4 zwei Werte annehmen: *TABLE_INIT* für Tabellen und *SVG_INIT* für SVGs. In späteren Entwicklungen können diese Werte für weitere mögliche Komponenten erweitert werden. Im Datenfeld Value wird ein JSON Objekt übertragen, welches die in Abschnitt 3.4.5 definierten Parameter der jeweiligen GUI Komponente, einschließlich der allgemeinen Parameter, beinhaltet.

Gestaltungsinformationen und Daten der GUI

Die Gestaltungsinformationen, also unter anderem die Positionen, Größen und Formen von GUI Komponenten werden bei der Erstellung des Screens im PRODIS.WTS zwischen CTS, *ParameterPasser* und Screen ausgetauscht. Als erstes wird jedoch der Screen initialisiert, weshalb zuerst die entsprechenden *SCREEN_MANAGER_DATA* Datenpakete vom *ParameterPasser* an die Views gesendet werden. Anschließend werden die Gestaltungsinformationen der einzelnen Elemente des Screens versendet. Bei Tabellen enthalten diese Informationen unter anderem die Anzahl der Spalten und deren Überschriften. Diese Informationen werden über ein *SCREEN_DATA*-Datenpaket an die Views versendet, bei denen das Datenfeld VALUE alle Gestaltungsinformationen der jeweiligen Komponente beinhaltet. Ändern sich Teile dieser Informationen, wozu auch die eigentlichen Daten gehören, wird ein weiteres Datenpaket versendet, in welchem sich nur die veränderten Daten der Komponente befinden.

Interaktion mit dem CTS

Im Folgenden wird die Interaktion zwischen den Views und dem *ParameterPasser* betrachtet. Klickt ein Benutzer im PRODIS.WTS auf einen Button des CTS, wird dies dem *ParameterPasser* mitgeteilt, welcher diese Events an das CTS weitergibt und dieses entsprechend darauf reagiert. Dabei wird ebenfalls ein Datenpaket übertragen, welches im Datenfeld TYPE den Wert *BUTTON_DATA* und in KEY den Wert *BUTTON_CLICKED* beinhaltet. Der Wert im Feld VALUE beinhaltet neben der ID des Buttons auch die boolesche Information, ob dieser global ist. Dies ist notwendig, da eine ID nur innerhalb eines CTS eindeutig ist, weshalb es passieren könnte, dass ein globaler Button dieselbe ID besitzt. Über diese Informationen identifiziert das CTS den Button und verarbeitet das Event. Abbildung 3.14 stellt solch ein Datenpaket grafisch dar. Dieses Datenpaket kann ebenso von einem Wearable Device gesendet werden, der *ParameterPasser* unterscheidet nicht von welchem View diese Events kommen.

Abbildung 3.14: Aufbau eines Interaktions-Datenpakets: TYPE –> KEY –> VALUE

3.8 Integration in PRODIS.Authoring und PRODIS.WTS

Abschließend wird nachfolgend betrachtet, wie die einzelnen Teile des Konzepts in die bestehenden Softwareprodukte PRODIS.Authoring und PRODIS.WTS integriert werden. Dabei ist unter anderem wichtig, dass an den eigentlichen Diagnoseabläufen so wenig wie möglich verändert werden muss und sich die notwendigen Änderungen einfach integrieren lassen. Abbildung 3.15 zeigt mit Hilfe eines Anwendungsfalldiagramms, welche neuen Möglichkeiten für die Werker und Diagnoseexperten hinzukommen. Dies ist zum einen der gesamte Teil der Wearable Devices und zum anderen die zwei Erweiterungsbeziehungen, welche deutlich zeigen, dass die Funktionen optional sind und nur bei Bedarf genutzt werden.

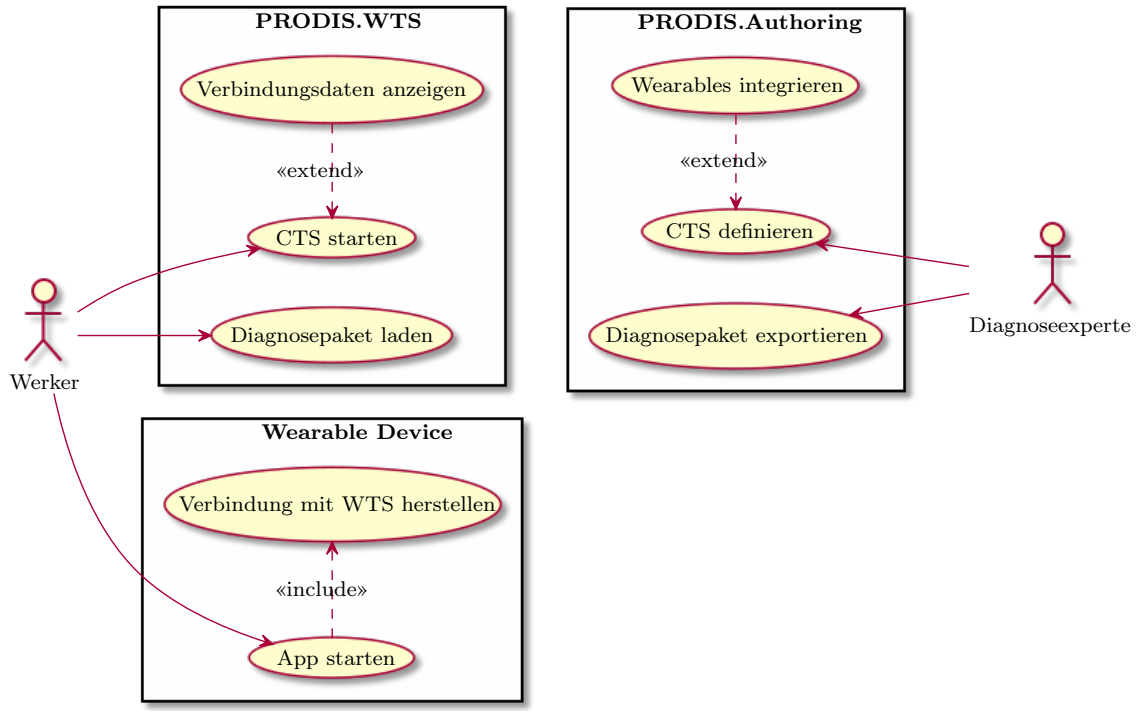


Abbildung 3.15: Anwendungsfalldiagramm mit neuen Möglichkeiten durch Wearable Devices bei der Fahrzeugdiagnose

3.8.1 Verbindungsaufbau und Kommunikation

Für den Verbindungsaufbau und die Kommunikation zwischen Wearable Device und PRODIS.WTS startet PRODIS.WTS im Hintergrund einen Server, welcher die Kommunikation, wie in Abschnitt 3.3.3 beschrieben, zwischen PRODIS.WTS und den Wearable Device übernimmt. Zusätzlich wird ein eigenes CTS definiert, welches auf einem eigenen Screen die Verbindungsdaten darstellt. Auf diesem Screen liegen die Verbindungsdaten, wie in Abschnitt 3.5 beschrieben, sowohl als QR-Code, als auch als Text vor. Dieses CTS stellt keinen Diagnoseablauf dar, sondern dient der Wiederverwendbarkeit der Verbindungsherstellung. Es wird im Folgenden, zur klareren Unterscheidung zum eigentlichen CTS, Hilfs-CTS genannt.

Diese Hilfs-CTSs müssen von den Herstellern in die zu erweiternden CTSs eingebunden werden. Das Hilfs-CTS kann an jeder beliebigen Stelle im CTS hinzugefügt werden und ermöglicht somit verschiedene Szenarien. So kann es Diagnosen geben, bei welchen der Einsatz von Wearable Devices nur zu bestimmten Zeitpunkten sinnvoll ist. Bei solchen kann das Hilfs-CTS an diesen Stellen eingebunden werden. Wenn der Werker die Wahl haben soll, ob und wenn ja, an welchen Stellen oder ab welchem Zeitpunkt in der Diagnose er mit einem Wearable Device arbeiten möchte, kann das Hilfs-CTS auch zum Beispiel auf einen Button gelegt werden und wird dementsprechend nach einem Button-Klick-Event ausgeführt. Dabei entsteht allerdings das Problem, dass die Wearable Devices die wichtigen Initialisierungsinformationen der GUI nicht erhalten, da diese, wie in Abschnitt 3.7 erläutert, bei der Erstellung des Screens übertragen werden. Ohne diese Informationen haben die Wearable Devices bei einer Tabelle zum Beispiel keine Informationen über dessen Spaltenüberschriften.

Um dieses Problem zu lösen, kann zu Beginn des CTS eine *Berechnung* oder ein Building Block hinzugefügt werden, welcher PRODIS.WTS auffordert, alle Datenpakete in einer Queue zwischenzuspeichern, bis das CTS beendet wird. Haben zwei Datenpakete die selben Werte für TYPE und KEY, so wird das ältere Datenpaket aus der Queue entfernt, da dessen Informationen veraltet sind. Nach einem erfolgreichen Verbindungsaufbau eines Wearable Device mit PRODIS.WTS, werden die zwischengespeicherten Datenpakete an das Wearable Device übertragen. Somit ist sichergestellt, dass alle nötigen Informationen zur Darstellung der GUI Komponente auf dem Wearable Device vorhanden sind.

Sobald sich ein Wearable Device erfolgreich mit PRODIS.WTS verbunden hat, wird das Hilfs-CTS inklusive des Screens mit den Verbindungsdaten automatisch geschlossen und die Ausführung des vorherigen CTS und dessen Screens fortgesetzt. Der Verbindungsdaten Screen wird nur angezeigt, wenn beim Start des Hilfs-CTS kein Wearable Device mit PRODIS.WTS verbunden ist. Ansonsten beendet sich das Hilfs-CTS direkt wieder, womit die Ausführung des eigentlichen CTS nicht unterbrochen wird.

3.8.2 GUI Komponente initialisieren

Für jede GUI Komponente, die an die Wearable Devices übertragen werden soll, wird ein eigener Building Block erstellt. Dies hat den Grund, dass jede GUI Komponente, wie in Abschnitt 3.4.5 beschrieben, ihre eigenen Parameter hat. Durch die Modularisierung in verschiedene Building Blocks, bleibt PRODIS.Authoring aufgeräumt und sauber strukturiert.

Es wird zunächst ein Building Block für Tabellen und einer für SVGs erstellt. Soll nun ein CTS um eine tabellarische Darstellung auf einem Wearable Device erweitert werden, muss dessen Building Block in das CTS integriert werden. Dabei muss darauf geachtet werden, dass der Building Block immer nach dem Hilfs-CTS aus Abschnitt 3.8.1 ausgeführt wird. Bei der Ausführung des Building Blocks, sendet dieser die in Abschnitt 3.7 spezifizierte Nachricht an alle Wearable Devices mit den Initialisierungsinformationen der GUI Komponente. Die Wearable Devices verfügen somit anschließend über alle nötigen Informationen zur Darstellung der GUI Komponente.

3.8.3 Datenaustausch

Nachdem ein Wearable Device mit PRODIS.WTS verbunden ist und die Initialisierungsinformationen für die GUI Komponente erhalten hat, benötigt es die darzustellenden Inhalte. Wie in Abschnitt 3.2 beschrieben, platziert sich der *ParameterPasser* zwischen Screen und CTS und erhält somit alle Informationen über die Daten und Werte der Komponenten. Damit diese ebenso an die Wearable Devices gesendet werden, muss PRODIS.WTS wie in Abschnitt 3.3 erläutert, um eine Client-Server Kommunikation erweitert werden. Alle Informationen, die der *ParameterPasser* zwischen CTS und Screens austauscht, müssen allen verbundenen Clients zur Verfügung gestellt werden.

Das Sequenzdiagramm in Abbildung 3.16 stellt den in Abschnitt 3.8.1 beschriebenen Verbindungsaufbau, das Initialisieren der GUI Komponente aus Abschnitt 3.8.2 und den soeben beschriebenen Datenaustausch beispielhaft dar. Dabei startet der Werker zunächst die Fahrzeugdiagnose ohne den Einsatz eines Wearable Device und startet erst nach einer gewissen Zeit die Anbindung des Wearable Device. Dazu klickt er zum Beispiel auf einen

Button im CTS, damit die Verbindungsinformationen auf einem neuen Screen dargestellt werden. Anschließend stellt er mit dem Wearable Device die Verbindung her, wodurch der Screen mit den Verbindungsinformationen geschlossen wird und das normale CTS weiterläuft. Durch einen entsprechenden Building Block im CTS werden anschließend die Initialisierungsinformationen an das Wearable Device übertragen und nachfolgend sendet PRODIS.WTS die vergangenen Datenpakete an das Wearable Device. Ab diesem Moment sendet PRODIS.WTS dem Wearable Device genau die selben Informationen, welche über den *ParameterPasser* zwischen Screen und CTS ausgetauscht werden.

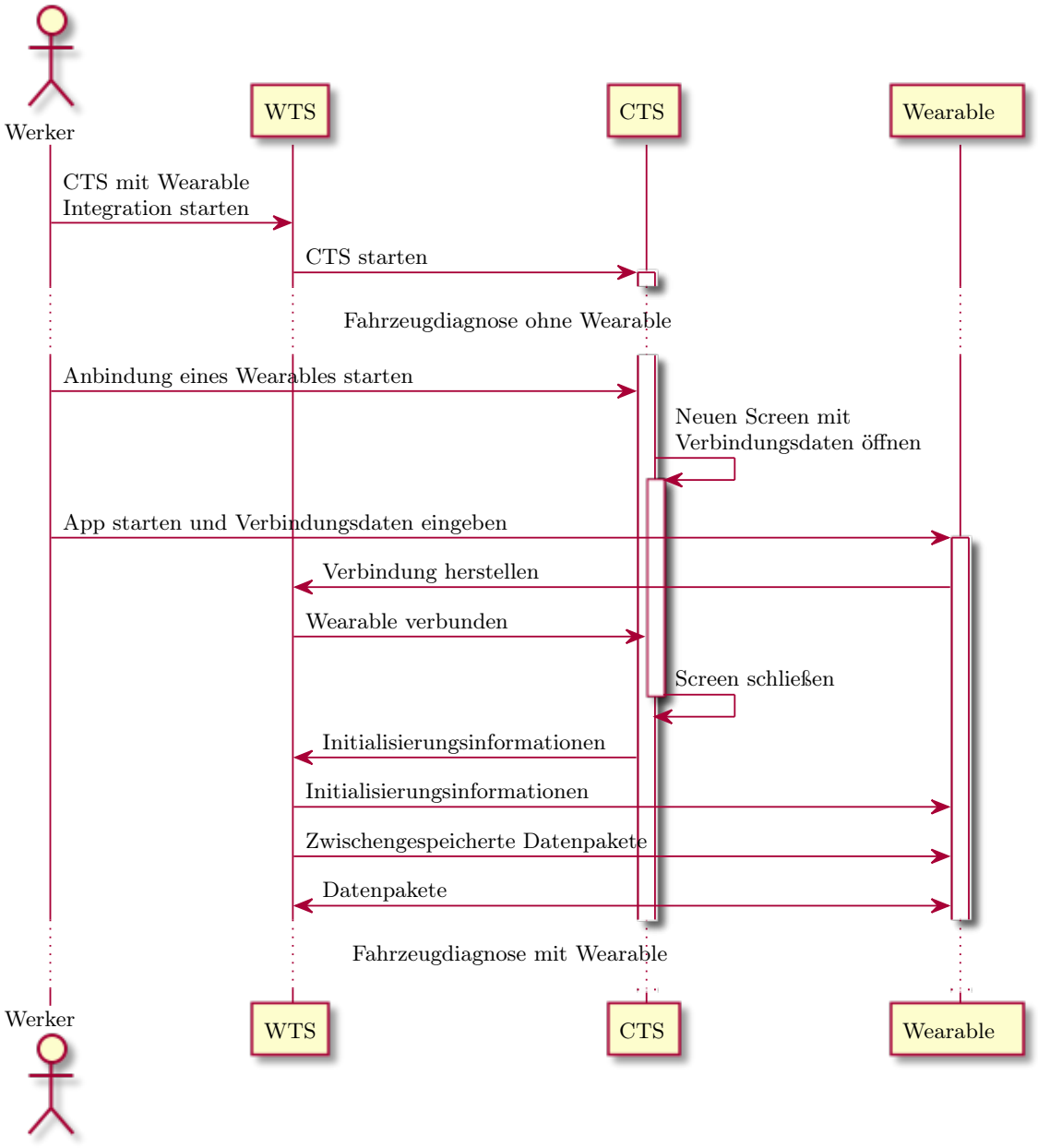


Abbildung 3.16: Beispielhafter Ablauf eines Verbindungsaufbaus zwischen Wearable Device und PRODIS.WTS

3.8.4 CTS beenden

Sobald ein CTS beendet wird, beenden die Wearable Devices die GUI ebenfalls und zeigen stattdessen einen Warte-Screen an. Die Verbindung wird allerdings nicht getrennt, sondern wartet auf neue Initialisierungsinformationen vom PRODIS.WTS. Wurden diese übertragen, zeigt der Bildschirm automatisch den neuen Inhalt an. Somit kann zwischen verschiedenen CTSs gewechselt werden, ohne das Wearable Device erneut verbinden zu müssen.

Kapitel 4

Implementierung

Nachdem die Anforderungen dargestellt und ein Konzept entwickelt wurde, wird dieses abschließend in die bestehende Softwarelandschaft der DSA GmbH integriert. Die Implementierung ist in zwei Phasen aufgeteilt. Zum einen müssen einmalige Anpassungen an PRODIS.WTS durchgeführt werden, um Wearable Devices in die Systemlandschaft integrieren zu können. Zum Anderen müssen bestehende CTSs erweitert werden, damit die Wearable Devices im Diagnoseablauf genutzt werden können. Da diese Anpassungen von jedem Hersteller individuell in sein PRODIS.Authoring System eingepflegt werden muss, wird dies anhand eines Demosystems von PRODIS.Authoring demonstriert. Abschließend wird für die Google Glass eine Applikationen implementiert, wobei ein Großteil der Kommunikationsaufgaben in eine Bibliothek ausgegliedert wird.

4.1 PRODIS.WTS

In Abschnitt 3.3 wurde erläutert, dass der *ParameterPasser* die Informationen zwischen CTS und den Screens ebenfalls an die Wearable Devices über ein WLAN Netzwerk übertragen soll. Dazu wurden Protokolle und Schnittstellen für eine Client-Server Anwendung betrachtet, um welche PRODIS.WTS im Folgenden erweitert wird.

4.1.1 Einleitung Atmosphere

Unter Abschnitt 3.3.3 wurde bereits dargelegt, dass das Atmosphere Framework zur Umsetzung der Client-Server-Architektur in dieser Arbeit genutzt wird. Atmosphere unterstützt neben WebSocket unter anderem auch SSE und *long polling*. Es ist mit allen Servlet-basierten Servern, wie zum Beispiel Tomcat, Jetty oder GlassFish, verwendbar, kann aber, zum Beispiel mit Hilfe der Anwendungsframeworks Netty oder Vert.x, auch außerhalb eines Servlet Containers eingesetzt werden [Atm16a]. Abbildung 4.1 zeigt die grundlegenden Bestandteile des Atmosphere Frameworks.

Um Atmosphere verstehen zu können, werden einige für diese Arbeit relevanten und von dem Atmosphere Projekt genutzten Konzepte im Folgenden kurz vorgestellt und erläutert.

- **AtmosphereResource**
  Eine AtmosphereResource repräsentiert eine Fernverbindung zu einem Client. Sie

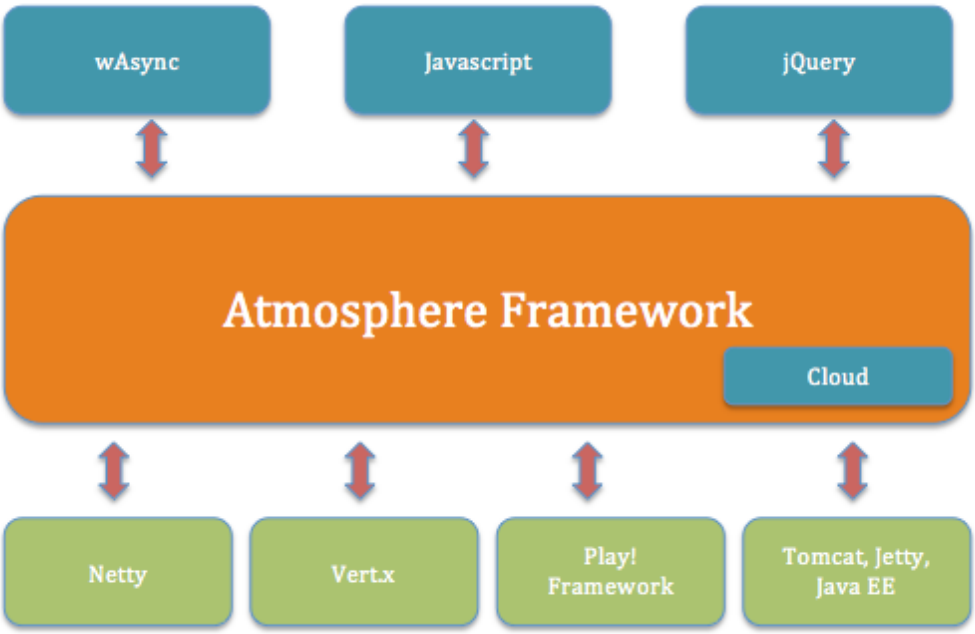


Abbildung 4.1: Bestandteile des Atmosphere Framework Stacks [Atm16a]

wird von Atmosphere als Kommunikationskanal zwischen Client und Server, sowie zum Verwalten der Lebensdauer genutzt [Atm15a].

- **AtmosphereHandler**
  Jeder AtmosphereHandler wird an einer Uniform Resource Locator (URL) registriert und verarbeitet anschließend alle Client-Anfragen an diese URL.

- **Broadcaster**
  Ein Broadcaster ist Teil des *publish-subscribe* Pattern von Atmosphere. Dieses ist ein Beobachter-Entwurfsmuster wie das Observer- und Listener-Pattern, welche bereits in Abschnitt 3.2 vorgestellt wurden. Im Wesentlichen ähnelt es dem Observer-Pattern. Ein Broadcaster kann als ein Channel betrachtet werden, an welchem sich beliebig viele *Publisher* und *Subscriber* registrieren können. Erstere veröffentlichen ihre Nachrichten an diesen Channel, letztere bekommen Nachrichten aus dem Channel, sobald neue Nachrichten an den Channel veröffentlicht wurden [Atm15b].
  Eine AtmosphereResource kann sowohl *Publisher* als auch *Subscriber* sein. Als *Publisher* veröffentlichen diese Nachrichten an die Broadcaster, als *Subscriber* empfangen sie von anderen veröffentlichte Nachrichten.

- **BroadcastFilter**
  Mit Hilfe von BroadcastFiltern können Nachrichten gefiltert oder manipuliert werden, bevor sie an die *Subscriber* veröffentlicht werden.

Als kleines Beispiel, zum besseren Verständnis der Komponenten, könnte eine Twitter ähnliche Anwendung entwickelt werden. Dazu erhält jeder Client (AtmosphereResource)

seinen eigenen Channel (Broadcaster), in welchen er seine Nachrichten veröffentlichen kann. Des Weiteren können Clients (AtmosphereResourcen), welche an Nachrichten von anderen interessiert sind, die interessanten Channels (Broadcaster) abonnieren [Atm15b]. Mehr Schritte sind nicht notwendig, denn nun können die Clients sowohl in ihren Channels Nachrichten veröffentlichen, als auch bei anderen die veröffentlichten Nachrichten mitlesen. Mit Hilfe von BroadcastFiltern kann das Beispiel noch um die Funktion der Filterung von Schimpfwörtern oder dem Kürzen von URLs erweitert werden.

4.1.2 Integration von Atmosphere

Wie in Abschnitt 3.2 bereits beschrieben, gibt es im PRODIS.WTS den *ParameterPasser*, welcher zwischen der GUI und der Programmlogik arbeitet. Dieser wird nachfolgend um eine Möglichkeit erweitert, jegliche Kommunikation zwischen GUI und Programmlogik ebenfalls über ein Netzwerk abzubilden. Dazu werden die in Abschnitt 3.3 vorgestellten Protokolle in einem Client-Server-Modell umgesetzt.

Der Aufbau des PRODIS.WTS wird, wie in Abbildung 4.2 dargestellt, unter anderem um einen CloudClient und CloudServer erweitert. Der CloudClient erhält vom *ParameterPasser* alle Events, welche dieser zwischen den CTSs und den Screens vermittelt. Er fungiert als WebSocket-Client und verbindet sich mit dem WebSocket-Server, dem CloudServer. Alle für die Wearable Devices relevanten Events leitet er an den Server weiter, welcher diese dann an alle anderen verbundenen WebSocket-Clients verteilt. Ist ein Wearable Device mit dem CloudServer verbunden, erhält dieses auf diesem Weg alle relevanten Daten zur Darstellung der GUI. Ebenso können die verbundenen Wearable Devices dem CloudServer Nachrichten senden, wenn über sie zum Beispiel auf einen Button des CTS geklickt wurde (siehe Abschnitt 3.7). Diese Events sendet der CloudServer dann dem CloudClient weiter, welcher den *ParameterPasser* benachrichtigt, welcher seinerseits wiederum das Event verarbeitet. Dabei unterscheidet der *ParameterPasser* nicht, ob das Event von einem Screen oder dem CloudClient kommt.

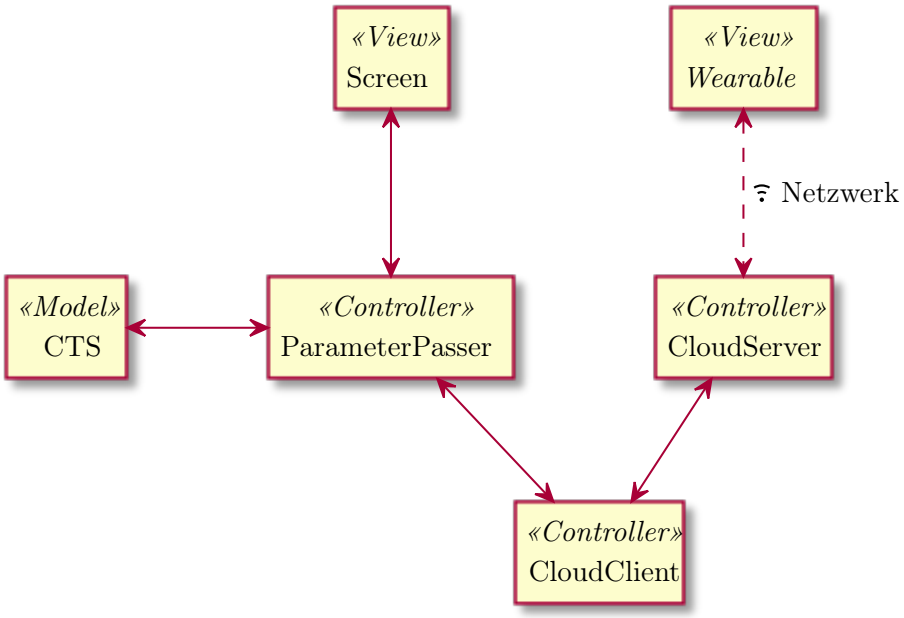


Abbildung 4.2: Erweiterter Aufbau des PRODIS.WTS

Der CloudServer unterscheidet also zwischen Wearable Devices und dem CloudClient, damit bei einer Aktion über ein Wearable Device nicht die anderen Wearable Devices über diese Aktion informiert werden, sondern lediglich der *ParameterPasser*. Solche Events sind für die Wearable Devices nicht von Bedeutung, da die gesamte Logik nur im PRODIS.WTS bzw. im jeweiligen CTS vorhanden ist. Dieses reagiert auf das Event, wodurch sich zum Beispiel Datenänderungen oder ähnliches ergeben. Diese werden anschließend über den normalen Ablauf an alle Wearable Devices verteilt. Abbildung 4.3 stellt diesen Ablauf grafisch dar. Aus Platz- und Übersichtlichkeitsgründen wurden nicht mehrere Wearable Devices, sondern nur eine Gruppe als *Wearables* dargestellt. Die oberste Aktion geht natürlich nur von einem einzigen Wearable Device aus, während die unterste Nachricht bei allen Wearable Devices ankommt.

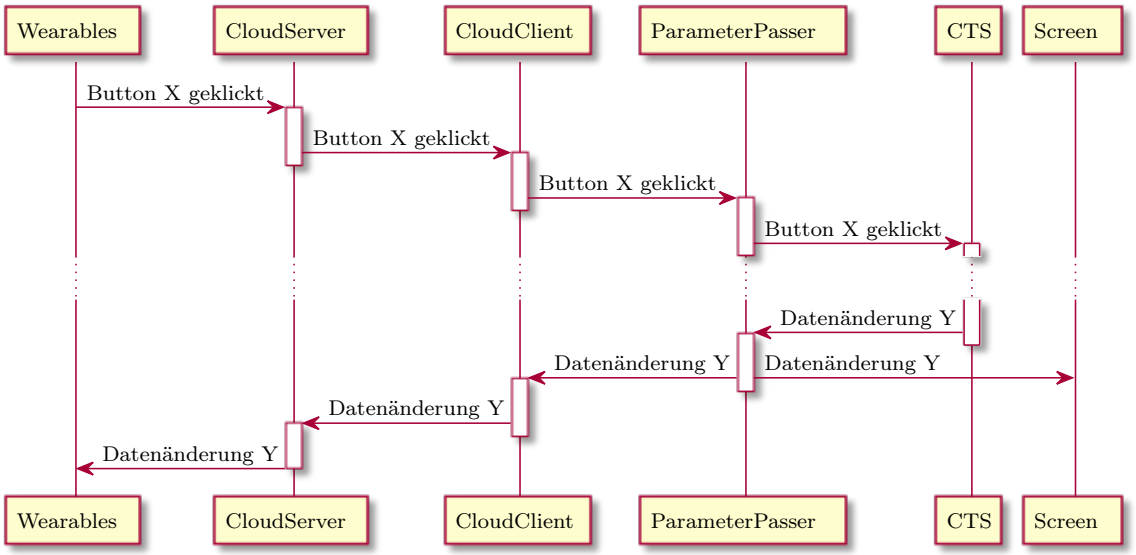


Abbildung 4.3: Beispielhafter Kommunikationsablauf bei einer Interaktion mit der GUI auf einem Wearable Device

CloudServer

PRODIS.WTS ist eine internationalisierte Software, weshalb auch alle Beschriftungen und Anweisungen an eindeutige Identifikatoren gebunden sind. Diese können im PRODIS.Authoring für ein CTS verwendet und um eigene Anweisungen erweitert werden. Das CTS gibt diese Identifikatoren an die Screens weiter, welche diese vor der Darstellung in die übersetzten Texte umwandelt. Der *ParameterPasser* erhält somit nicht den eigentlichen Text, sondern sehr häufig dessen Identifikator. Damit der *ParameterPasser* oder CloudClient die Nachrichten nicht nach Identifikatoren durchsuchen und diese durch den eigentlichen Text ersetzen müssen, wurde bereits im Vorfeld der Arbeit eine Webschnittstelle auf Basis von Jetty entwickelt. Der CloudServer startet dazu einen eingebetteten Jetty und registriert dazu entsprechende Servlets und andere Ressourcen. Er stellt zum Beispiel eine Schnittstelle bereit, welcher ein Identifikator mitgeteilt werden kann und als Ergebnis der internationalisierte Text zurückgeliefert wird. Auch für lokal auf dem Diagnosegerät vorhandene Dateien gibt es entsprechend eine Schnittstelle, über die diese Dateien angefragt werden können.

Der CloudServer wird für die Nutzung von WebSocket um eine weitere Schnittstelle

erweitert, welche für die WebSocket Verbindungen über Atmosphere genutzt wird. Dazu registriert der CloudServer am Jetty ein neues Servlet an einer URL, welches alle Anfragen an diese Schnittstelle verarbeitet. Atmosphere bietet für diese Zwecke das AtmosphereServlet an, welches mit einigen Parametern konfiguriert werden kann. So können zum Beispiel die maximale Datengröße oder auch Interceptoren und Handler definiert werden.

CloudAtmosphereHandler

Der CloudAtmosphereHandler ist ein AtmosphereHandler und wurde am AtmosphereServlet registriert. Er verwaltet alle eingehenden Anfragen und registriert zum Beispiel neue AtmosphereResources als Empfänger und Sender an die entsprechenden Broadcaster. Sendet eine AtmosphereResource eine neue Nachricht, leitet er diese an die Broadcaster weiter, an denen er als Sender registriert ist.

Sollten nur Wearable Devices angebunden werden, würden zwei Broadcaster genügen: Einer für alle Events, welche von den Wearable Devices gesendet werden und von PRODIS.WTS verarbeitet werden müssen, und ein zweiter für alle Nachrichten an die Wearable Devices. Da für die Wearable Devices aber einige Erweiterungen eingeführt wurden, welche nur für Wearable Devices interessant sind, aber zum Beispiel nicht für andere Diagnosegeräte, bietet es sich von Anfang an einen dritten Broadcaster vorzusehen, welcher die speziellen Nachrichten für Wearable Devices nicht erhält. Somit können auch später die für Wearable Devices irrelevanten Nachrichten mit einem BroadcastFilter gefiltert oder manipuliert werden.

Die folgenden drei Broadcaster werden genutzt:

- WearableBroadcaster
  Hier registrieren sich alle Wearable Devices als Empfänger und der CloudClient des Servers als Sender. Hierüber wird PRODIS.WTS alle für Wearable Devices relevanten Datenpakete an die Wearable Devices senden.

- CloudClientBroadcaster
  Hier registrieren sich alle Wearable Devices und andere Geräte als Sender und der CloudClient des Servers als Empfänger. Wurde auf einem Gerät zum Beispiel ein Button geklickt, sendet das Gerät ein entsprechendes Button-Klick-Event an diesen Broadcaster und PRODIS.WTS verarbeitet das Event (siehe Abbildung 4.3).

- OtherBroadcaster
  An diesem Broadcaster können sich alle Geräte als Empfänger registrieren, welche alle Nachrichten ungefiltert oder keine Extradaten, die nur für Wearable Devices relevant sind, empfangen wollen. Der CloudClient des Servers registriert sich wie beim WearableBroadcaster als Sender.

In Abbildung 4.4 werden die soeben beschriebenen Verbindungen zwischen den Broadcastern, den Wearable Devices, den anderen Geräten und dem CloudClient des Servers grafisch dargestellt. Ein Pfeil symbolisiert dabei die Richtung in der die Daten gesendet werden.

Mit diesem Aufbau könnten sich zum Beispiel zwei PRODIS.WTS verbinden. Dabei würde der CloudClient des Clients sich als Empfänger am OtherBroadcaster registrieren und

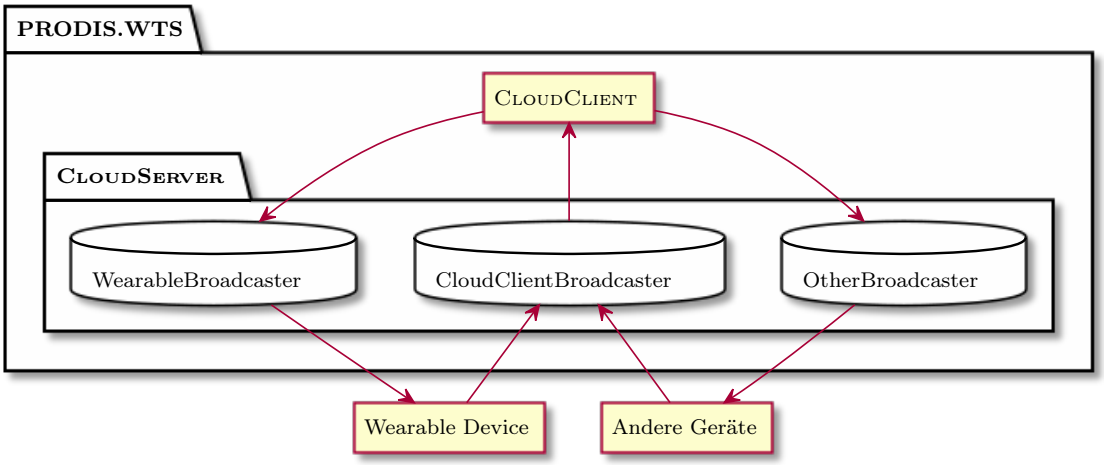


Abbildung 4.4: Unterteilung des CloudServer in Broadcaster und der Datenfluss der Clients

der CloudClient des Servers als Sender am OtherBroadcaster. Durch diesen Aufbau ermöglicht die Implementierung die Anbindung weiterer Geräte.

Listing 4.1 zeigt, wie der CloudAtmosphereHandler die Anfragen verarbeitet. Bei GET-Anfragen handelt es sich hier um das Herstellen einer WebSocket Verbindung und bei POST-Anfragen um das Senden von Datenpaketen. Mit Hilfe von Zeile 4 wird Atmosphere mitgeteilt, dass die Verbindung geöffnet bleiben soll und somit die WebSocket Verbindung akzeptiert wird. Anschließend wird in den Zeilen 7 und 8 der Typ des Clients aus einem Header-Parameter ausgelesen und, entsprechend den zuvor beschriebenen Zuordnungen, in Zeile 11 bis Zeile 30 an die Broadcaster als *Publisher* und *Subscriber* registriert. Handelt es sich um das Senden von Datenpaketen wird in Zeile 37 zunächst die zuvor bei der Verbindungsherstellung konfigurierte Zuordnung geladen und dann, in Zeile 40, mit einer Schleife das Datenpaket an alle Broadcaster, an denen der Client als *Publisher* registriert ist, verteilt.

```
1  public void onRequest(AtmosphereResource resource) throws
       IOException {
2    if ("GET".equalsIgnoreCase(resource.getRequest().getMethod())) {
3      // Halte die Websocket-Verbindung offen
4      resource.suspend();
5
6      // Ermittle den Typen des Clients
7      String senderType = resource.getRequest()
8          .getHeader(SENDER_HEADER_NAME);
9
10     // Ordne Resource entsprechenden Broadcastern zu:
11     if (CLOUDCLIENT_SENDER_TYPE.equalsIgnoreCase(senderType)) {
12       // Empfängt Nachrichten vom CloudClientBroadcaster
13       cloudClientBroadcaster.addAtmosphereResource(resource);
14
15       // Sendet Nachrichten an Other- & WearableBroadcaster
16       resource.broadcasters().forEach(resource::removeBroadcaster);
17       resource.addBroadcaster(otherBroadcaster)
18           .addBroadcaster(wearableBroadcaster);
```

```
19      } else {
20        if (WEAR_SENDER_TYPE.equalsIgnoreCase(senderType)) {
21          // Empfängt Nachrichten vom WearableBroadcaster
22          wearableBroadcaster.addAtmosphereResource(resource);
23        } else {
24          // Empfängt Nachrichten vom OtherBroadcaster
25          otherBroadcaster.addAtmosphereResource(resource);
26        }
27        // Sendet Nachrichten an CloudClientBroadcaster
28        resource.broadcasters().forEach(resource::removeBroadcaster);
29        resource.addBroadcaster(cloudClientBroadcaster);
30      }
31    }
32
33    if ("POST".equalsIgnoreCase(resource.getRequest().getMethod())) {
34      StringBuilder msg = IOUtils.readEntirelyAsString(resource);
35      if (msg.length() > 0) {
36        // Ursprüngliche AtmosphereResource laden
37        resource = resource.getAtmosphereConfig().framework()
38            .atmosphereFactory().find(resource.uuid());
39        for (Broadcaster b : resource.broadcasters()) {
40          b.broadcast(msg.toString());
41        }
42      }
43    }
44  }
```

Listing 4.1: Zuordnen der AtmosphereResources zu den entsprechenden Broadcastern und Verteilung der Nachrichten an diese

4.1.3 BroadcastFilter und allgemeine Filter

Da einige Datenpakete für die Wearable Devices irrelevant sind, wurden zur Reduzierung unnötiger Datenpakete einige Filter implementiert:

- **WearBroadcastFilter**
  Dieser Filter ist der einzige BroadcastFilter. Er ist immer am WearableBroadcaster registriert und überwacht die an die Wearable Devices gesendeten Datenpakete. Der WearBroadcastFilter bietet die Möglichkeit, weitere Filter zu registrieren. Dies ist notwendig, da die CTSs keinen Zugriff auf den WearableBroadcaster haben, um dort weitere BroadcastFilter zu registrieren. Beim Verarbeiten eines Datenpakets leitet WearBroadcastFilter diese an alle, an ihm registrierten, Filter nacheinander weiter, welche die Daten kontrollieren, filtern und, wenn nötig, auch verändern können.

  Abbildung 4.5 zeigt anhand von drei einfachen Beispielnachrichten, welche Aktionen der WearBroadcastFilter durchführen kann. Er kann eine Nachricht filtern (Nachricht A), verändern (Nachricht C/Z), oder unverändert passieren lassen (Nachricht B).

- **WearButtonFilter & WearScreenFilter**
  Mit Hilfe dieser Filter werden alle Datenpakete über Buttons bzw. Screens, welche für die Wearable Devices nicht mittels der in Abschnitt 3.4.5 beschriebenen Parameter konfiguriert wurden, herausgefiltert.

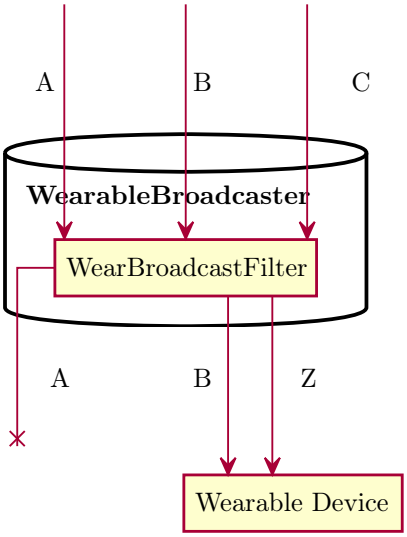


Abbildung 4.5: Beispielfilterung des WearBroadcastFilter

- **WearTableFilter**
  Dieser Filter betrachtet alle Datenpakete, welche Daten über die konfigurierte Tabelle enthalten. Wird zum Beispiel ein Wert einer Zeile verändert, welche auf dem Wearable Devices nicht angezeigt wird, so wird diese vom WearTableFilter herausgefiltert.

- **WearSVGFilter**
  Da Bilder, und somit auch SVGs, im PRODIS.WTS aus einer Datei geladen werden können, kann es passieren, dass das Datenpaket nicht die Daten des SVGs enthält, sondern den Pfad zur SVG Datei. Die Wearable Devices haben allerdings keinen Zugriff auf diese Datei, weshalb sie diese Informationen nicht verarbeiten können. Dieser Filter lädt in solchen Fällen die Datei und manipuliert das Datenpaket. Dabei ersetzt der WearSVGFilter den Dateipfad im Datenpaket durch den Inhalt der SVG Datei.

CloudClient und Wearable Devices

Das in Abbildung 4.1 dargestellte wAsync ist eine Client-Bibliothek für asynchrone Kommunikation mit einem WebSocket- oder HTTP-Webserver [Atm16b]. wAsync unterstützt neben WebSocket ebenfalls SSE und *long polling*. Es kann sowohl im PRODIS.WTS selbst genutzt werden, wie auch bei den beiden Wearable Devices.

Der CloudClient kümmert sich intern mit Hilfe von wAsync um die Abwicklung jeglicher Kommunikation mit dem CloudServer. Nach außen bietet er im wesentlichen vier Methoden, welche sich um den Verbindungsaufbau, dessen Abbau, dem Senden von Nachrichten an den Server und dem registrieren von Listenern kümmern. Sobald eine neue Nachricht empfangen wurde, werden alle registrierten Listener über diese benachrichtigt. Listing 4.2 zeigt, wie der CloudClient und die Wearable Devices die WebSocket Verbindung zum Server herstellen.

```
1  // wAsync Client für eine Verbindung zu einem Atmosphere Server
2  // erstellen
3  AtmosphereClient client =
       ClientFactory.getDefault().newClient(AtmosphereClient.class);
4
5  // Initialer Request zum Herstellen einer WebSocket Verbindung
6  // konfigurieren
7  AtmosphereRequestBuilder request = client.newRequestBuilder();
8  request.method(Request.METHOD.GET);
9  request.uri(mServerUrl);
10 request.encoder(new CustomEncoder());
11 request.decoder(new CustomDecoder());
12 // Wenn möglich: WebSocket, zur Not SSE oder long polling
13 request.transport(Request.TRANSPORT.WEBSOCKET);
14 request.transport(Request.TRANSPORT.SSE);
15 request.transport(Request.TRANSPORT.LONG_POLLING);
16 // Identifizierung: Entweder ein Wearable oder ein anderes Gerät
17 request.header(SENDER_HEADER_NAME, "wear/other");
18
19 // Socket erstellen und u.a. Listener für eingehende Nachrichten
20 // registrieren
21 Socket socket = client.create();
22 socket.on(Event.MESSAGE.name(), new OnMessageReceived());
23 socket.on(Event.OPEN.name(), new OnConnectionOpened());
24 socket.on(Event.CLOSE.name(), new OnConnectionClosed());
25 socket.on(new OnExceptionThrown());
26
27 // Verbindung zum Server öffnen
28 socket.open(request.build());
```

Listing 4.2: Aufbau einer WebSocket-Verbindung mit Hilfe von wAsync

Zu Beginn wird ein Client passend zum Atmosphere Server erstellt und die initiale Anfrage zum Herstellen der Verbindung konfiguriert. Wichtig sind die Zeilen 13 bis 15, in denen dem Server mitgeteilt wird, welche Transportarten der Client unterstützt. In Zeile 17 wird ein Header zur Identifizierung des Clients gesetzt, damit der Server die AtmosphereResource den korrekten Broadcastern zuordnen kann. Für alle eingehenden Nachrichten wird in Zeile 22 ein Listener registriert, welcher alle am CloudClient registrierten Listener über die neue Nachricht informiert. Sobald die Verbindung hergestellt wurde, kann, wie in Listing 4.3 gezeigt, ein Datenpaket an den Server gesendet werden.

```
1  socket.fire(packageToSend);
```

Listing 4.3: Senden eines Datenpakets über wAsync

4.2 PRODIS.Authoring und bestehende CTSs

Am eigentlichen PRODIS.Authoring Quellcode mussten keine Anpassungen vorgenommen werden, da die Funktionalität von PRODIS.Authoring nicht erweitert wurde. Alle neuen Funktionalitäten wurden im PRODIS.WTS implementiert, welches PRODIS.Authoring nutzt, um zum Beispiel den Java-Quellcode eines Building Blocks zu validieren.

In einer Demoumgebung von PRODIS.Authoring wurden allerdings, die für diese Arbeit in Abschnitt 3.8.1 beschriebenen, notwendigen Anpassungen der Hersteller implementiert.

Dazu zählen das Hilfs-CTS zum Darstellen der Verbindungsinformationen, die Building Blocks zum Konfigurieren der GUI Komponente für die Wearable Devices und die Anpassung zwei bestehender CTSs.

4.2.1 Hilfs-CTS

Das Hilfs-CTS dient dazu, dem Werker die Verbindungsinformationen für die Wearable Devices darzustellen. Dazu werden diese, wie in Abschnitt 3.8.1 beschrieben, in einem QR-Code, sowie im Klartext dem Werker angezeigt.

Für das Hilfs-CTS wird ein Screen erstellt, in welchem ein QR-Code in zentraler Position dargestellt wird. Ebenfalls werden die Verbindungsinformationen im Klartext unterhalb des QR-Codes angezeigt. Da es keine GUI Komponente für die Darstellung eines QR-Codes gibt, wird stattdessen die Komponente zur Darstellung einer Bilddatei benutzt und der generierte QR-Code in einer temporären Bilddatei gespeichert und von dort geladen. Sobald der Screen geöffnet und der QR-Code dargestellt wird, geht das Hilfs-CTS in eine Warteschleife über, bis sich ein Wearable Device verbunden hat.

In Abbildung 4.6 ist das Hilfs-CTS dargestellt. Mit Hilfe des *Guard* Bausteins wird ein weiterer Ablauf im Hilfs-CTS angestoßen, welcher unabhängig vom Hauptprogrammfluss agiert. In diesem Teil wird jede Sekunde mit Hilfe einer *Berechnung* kontrolliert, ob ein Wearable Device mit dem CloudServer verbunden ist. Sobald dies geschehen ist, oder der Werker auf den Überspringen-Button geklickt hat, wird eine Eigenschaft des Hilfs-CTS auf einen neuen Status gesetzt, wodurch im Hauptprogrammfluss die WHILE-Schleife beendet wird. Dabei wird der Screen geschlossen, der Überspringen-Button entfernt und anschließend, sobald das Hilfs-CTS beendet wurde, wird die Ausführung des ursprünglichen CTS, welches das Hilfs-CTS aufgerufen hat, fortgesetzt.

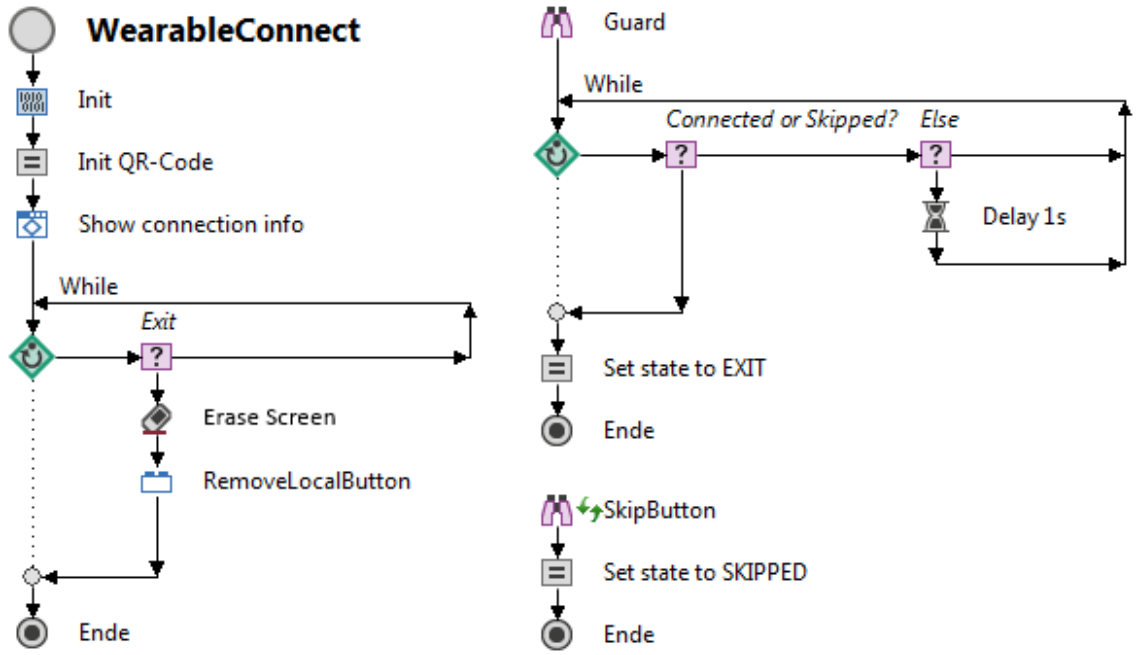


Abbildung 4.6: Hilfs-CTS zur Darstellung der Verbindungsinformationen

Im QR-Code sind die Verbindungsinformationen im JSON-Format hinterlegt. Diese enthalten die IP-Adresse bzw. den Hostnamen des Diagnosegerät auf welchem PRODIS.WTS ausgeführt wird, sowie den Port und den Pfad, auf welchem der CloudServer auf eingehende WebSocket-Verbindungen lauscht. Diese Informationen können in Zukunft zum Beispiel um Informationen über das verbundene WLAN und den eingesetzten Proxy erweitert werden, wodurch sich die Wearable Devices selbstständig in das richtige WLAN Netz einwählen und den entsprechenden Proxy konfigurieren können. Zum Generieren des QR-Codes wird die Java-Bibliothek Zebra Crossing (ZXing) von Google verwendet. ZXing ist ein ein Open Source Projekt mit dem Ziel eine Bibliothek zum Lesen und Generieren von verschiedenen ein- und zweidimensionalen Barcode-Formaten bereitzustellen [ZXi16b].

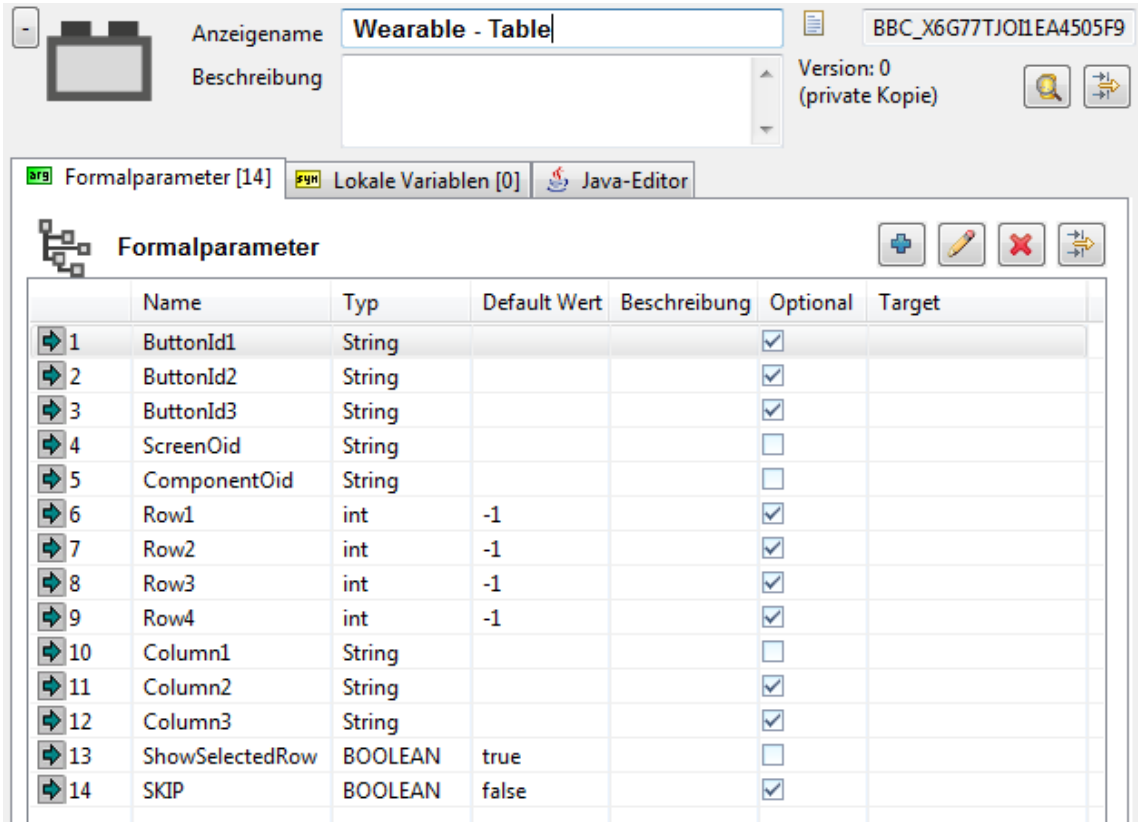
Abbildung 4.7 zeigt, wie das Hilfs-CTS mit beispielhaften Verbindungsinformationen aussieht.

Abbildung 4.7: Screen des Hilfs-CTS

4.2.2 Building Blocks

In Abschnitt 3.8.2 wurde bereits beschrieben, dass für jede GUI Komponente, welche auf den Wearable Devices dargestellt werden soll, ein eigener Building Block entwickelt werden muss. Diese verarbeiten die in Abschnitt 3.4.5 beschriebenen Parameter und übermitteln den Wearable Devices die initial benötigten Informationen für die jeweilige GUI Komponente. Da diese Initialisierungsinformationen unabhängig von der eigentlichen Darstellung auf dem Screen sind, werden diese Informationen nicht über den *ParameterPasser* übertragen. Stattdessen wird direkt mit Hilfe des CloudClient die Nachricht an den CloudServer übermittelt.

In Abbildung 4.8 ist ein beispielhafter Building Block für die Tabellen Komponente dargestellt. Auf der linken Seite sind die Parameter, wie zum Beispiel die Spalten und Zeilen der



	Name	Typ	Default Wert	Beschreibung	Optional	Target
1	ButtonId1	String			☑	
2	ButtonId2	String			☑	
3	ButtonId3	String			☑	
4	ScreenOid	String			☐	
5	ComponentOid	String			☐	
6	Row1	int	-1		☑	
7	Row2	int	-1		☑	
8	Row3	int	-1		☑	
9	Row4	int	-1		☑	
10	Column1	String			☐	
11	Column2	String			☑	
12	Column3	String			☑	
13	ShowSelectedRow	BOOLEAN	true		☐	
14	SKIP	BOOLEAN	false		☑	

Abbildung 4.8: Building Block Parameter für die Initialisierung einer Tabelle

Tabelle, aufgelistet. Rechts daneben ist der Java Quellcode des Building Blocks dargestellt. Dieser reicht die Parameter an die Klasse WearSetup weiter, welches eine neu eingeführte Klasse im PRODIS.WTS ist. Im Komponentendiagramm in Abbildung 4.9 sind die Schnittstellen und Komponenten dargestellt, welche WearSetup nutzt und bereitstellt. WearSetup stellt für die beiden GUI Komponenten je eine Schnittstelle bereit, der die Parameter übergeben werden. Sobald eine davon aufgerufen wird, aktiviert WearSetup die übergebenen Filter am CloudServer und wandelt die übergebenen Parameter in ein Datenpaket, wie in Abschnitt 3.6 beschrieben, um und übergibt dieses dem CloudClient. Im Datenfeld Value enthält das Datenpaket ein JSON serialisiertes Objekt mit den übergebenen Parametern. Abschließend holt sich WearSetup alle zwischengespeicherten Datenpakete vom WearMessageStorage und leitet diese zum CloudClient weiter, welcher diese an die Wearable Devices überträgt.

In der Klasse WearMessageStorage können alle Nachrichten, welche vom CloudServer an die Wearable Devices gesendet werden, zwischengespeichert werden. Dazu muss ein Building Block oder eine *Berechnung* zu Beginn des CTS eingefügt werden, welches das Zwischenspeichern aller Datenpakete am WearBroadcastFilter startet. Der WearBroadcastFilter wiederum ist ein BroadcastFilter, welcher beim Start des CloudServers am WearableBroadcaster vom PRODIS.WTS registriert wird und somit alle Datenpakete, welche über den WearableBroadcaster an die Wearable Devices gesendet werden, analysieren kann.

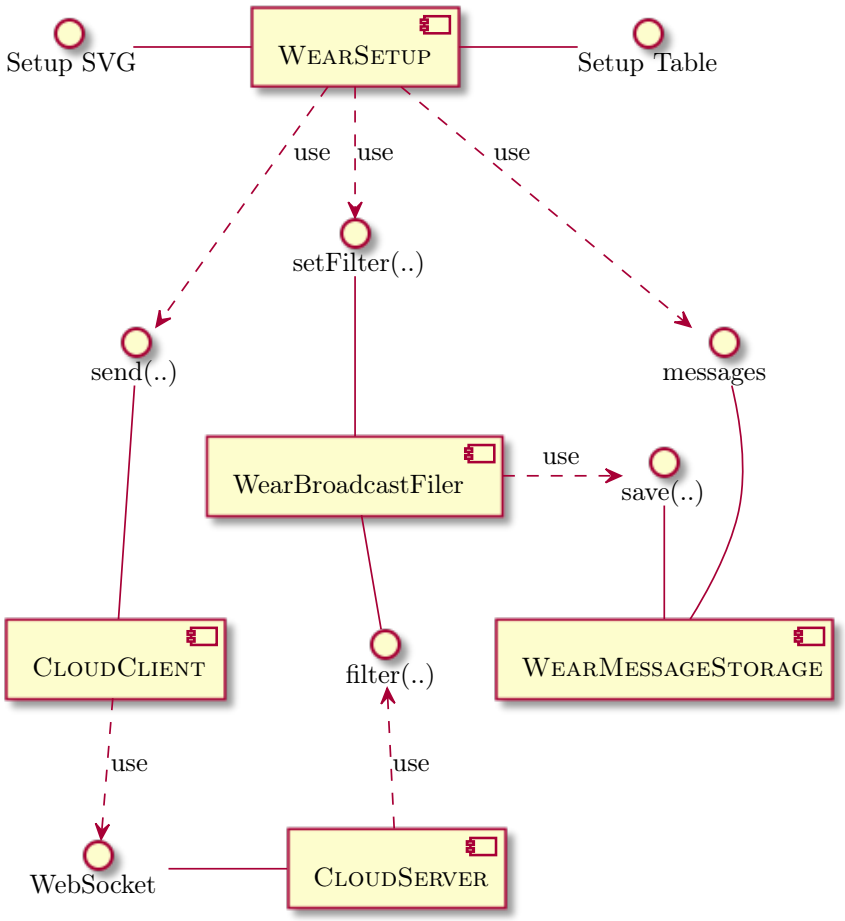


Abbildung 4.9: Genutzte und bereitgestellte Schnittstellen von WearSetup

4.2.3 Anpassung bestehender CTSs

Im Folgenden wird die Beispieldiagnose aus Abschnitt 1.1 und das dazugehörige CTS aus Abbildung 4.10 genutzt, um zu zeigen, wie die Hersteller mit den bereitgestellten Building Blocks und dem Hilfs-CTS ihre CTSs für die Nutzung von Wearable Devices anpassen können.

Zunächst wird das bisherige CTS kurz erläutert, damit im Anschluss die Anpassungen besser nachvollzogen werden können. Das CTS stellt eine Tabelle mit allen Rädern des Fahrzeugs und deren Drehgeschwindigkeiten dar.

Im Strukturknoten *Init* werden zunächst einige Parameter, wie zum Beispiel die darzustellenden Spalten der Tabelle, definiert und die Buttons des CTS erstellt. Anschließend wird ein Screen, in welchem sich lediglich eine Tabelle befindet, geöffnet. Die Berechnung *Fill Table* dient zum Befüllen der Tabelle mit Beispieldaten und der nebenläufige Programmfluss dem aktualisieren der Daten in der selektierten Zeile der Tabelle. In einem echten CTS würde ein nebenläufiger Programmfluss existieren, welcher die Werte, zum Beispiel über die OBD Schnittstelle, vom Fahrzeug auslesen und die Daten in der Tabelle aktualisieren würde. Die WHILE Schleife überprüft, ob der *Nächstes Rad* oder der *Zurück* Button des CTS gedrückt wurde. Im ersten Fall wird das nächste Rad, also die nächste Zeile in der Tabelle, selektiert. Im Letzterem räumt das CTS auf, schließt also den Screen, entfernt die

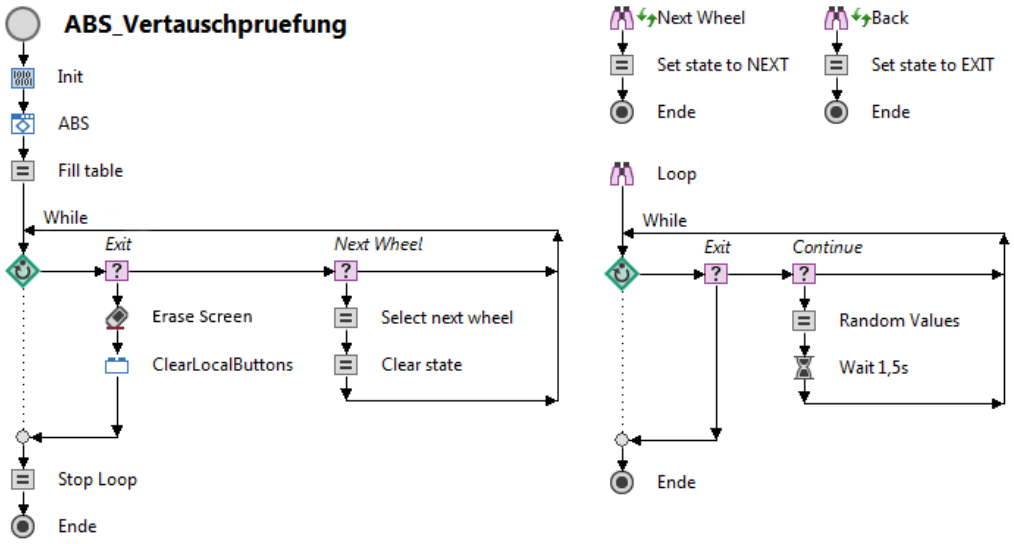


Abbildung 4.10: CTS zur Überprüfung der ABS Sensoren (ohne Anpassungen für Wearable Devices)

Buttons und beendet sich anschließend.

Es gibt im wesentlichen zwei verschiedene Arten die Wearable Devices an das CTS anzubinden: Eine einfache Anpassung, bei der direkt beim Start des CTS die Anbindung der Wearable Devices erfolgt, oder eine dynamische und gegebenenfalls kontextbezogene Anbindung, bei der zu bestimmten Diagnoseabschnitten oder auf Wunsch des Werkers die Wearable Devices verwendet werden. Im Folgenden werden die beiden Varianten kurz dargelegt.

Anbindung zum Start des CTS

In Abbildung 4.11 ist das CTS mit der einfachsten Form der Anbindung von Wearable Devices dargestellt. Zu Beginn des CTS wird direkt das Hilfs-CTS zum Herstellen der Verbindung gestartet und anschließend der Building Block zur Übertragung der Initialisierungsinformationen der Tabellen Komponente. Dem Building Block werden dabei mittels den in Abschnitt 3.4.5 definierten Parametern, die gewünschte Darstellung der Tabelle übergeben. Der Building Block *Wearable - Cleanup* am Ende des CTS entfernt alle vom Building Block *Wearable - Table* eingestellten Filter am WearBroadcastFilter.

Anbindung zu einem beliebigen Zeitpunkt

In Abbildung 4.12 ist das CTS mit einer dynamischeren Form der Anbindung von Wearable Devices dargestellt. Statt das Wearable Device direkt beim Start des CTS zu verbinden, ermöglicht dieses angepasste CTS, dass der Werker bei einem Klick auf den *Cast*-Button die Anbindung zu jedem beliebigen Zeitpunkt während des Diagnoseablaufs starten kann. Dazu wird ganz zu Beginn des CTS der Building Block *Wearable - Start recording* platziert, welcher über den WearBroadcastFilter die Zwischenspeicherung der Datenpakete im WearMessageStorage aktiviert. Sobald der Werker den *Cast*-Button betätigt, werden im Hauptprogrammfluss innerhalb der WHILE-Schleife, die Schritte zur Anbindung des

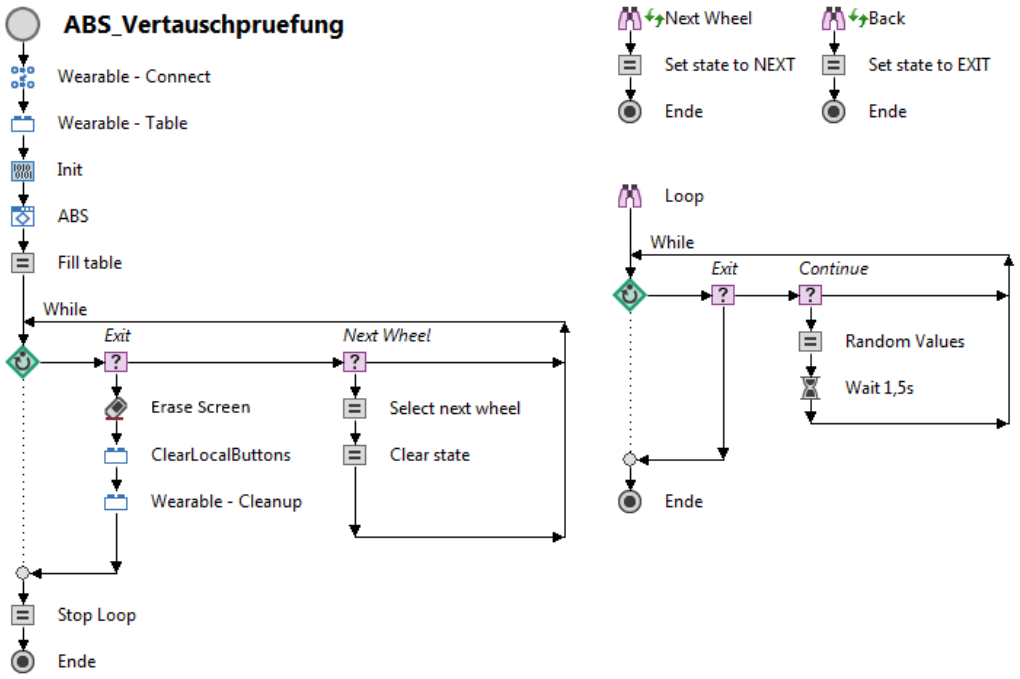


Abbildung 4.11: CTS zur Überprüfung der ABS Sensoren (mit Anpassungen für Wearable Devices)

Wearable Device gestartet. Dies geschieht wie in Abschnitt 4.2.3 über das Hilfs-CTS und den Building Block *Wearable - Table*. Ebenfalls wird am Ende des CTS der Building Block *Wearable - Cleanup* platziert, welcher in diesem Fall neben der Zurücksetzung der Filter ebenfalls das Zwischenspeichern der Datenpakete deaktiviert und vorhandene Datenpakete im WearMessageStorage entfernt.

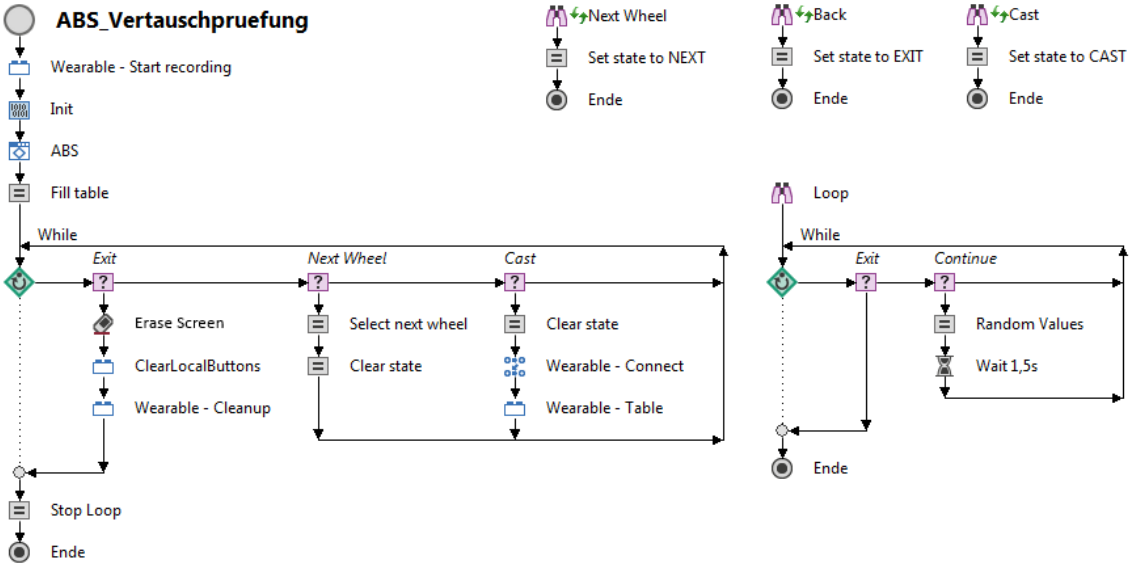


Abbildung 4.12: CTS zur Überprüfung der ABS Sensoren (mit Anpassungen für Wearable Devices)

4.3 Wearable Devices

Für die beiden Wearable Devices Google Glass und Moto 360, dessen Betriebssysteme beide auf Android basieren, wurde eine Bibliothek für die WebSocket Verbindung und den Datenaustausch geschrieben. Diese kann von beiden Geräten genutzt werden und kann in Zukunft auch von anderen Geräten, welche Java-Bibliotheken einbinden können, verwendet werden. Für die Wearable Devices muss somit nur noch die GUI Darstellungen implementiert werden. Im Folgenden wird zunächst die Bibliothek beschrieben und anschließend kurz auf einige Konzepte von Android eingegangen. Abschließend wird beschrieben, wie die erstellte Bibliothek bei der Erstellung der Applikation für die Google Glass genutzt wurde.

4.3.1 Bibliothek

Wie auch für den CloudClient, wird wAsync zur WebSocket Kommunikation mit dem CloudServer genutzt. Dazu besitzt die Bibliothek ebenfalls eine Klasse CloudClient, welche zu großen Teilen identisch mit dem CloudClient des PRODIS.WTS ist. Anders als beim PRODIS.WTS benachrichtigt der CloudClient der Bibliothek nicht den *ParameterPasser*, sondern gibt alle eingehenden Nachrichten, mit Hilfe des Listener-Patterns, an die registrierten MessageReceivedListener weiter.

Insgesamt gibt es in der Bibliothek fünf vordefinierte MessageReceivedListener, welche Manager genannt werden. Diese verwalten die Daten teilweise ähnlich wie ein *Registry* oder geben die Daten an spezifischere Listener weiter. Eine *Registry*-Klasse stellt seine Daten für alle anderen Klassen und Objekten innerhalb eines Softwaresystems zur Verfügung und ist Teil des Registry-Entwurfsmusters [Fow02]. Dazu wurden die fünf Manager als *Singleton*s [GHJV94] implementiert, wodurch immer genau eine Instanz der jeweiligen Klasse existiert auf die alle anderen Klassen und Objekte Zugriff haben. Im Folgenden werden die Aufgaben der fünf Manager beschrieben:

- Der **MenuManager** verarbeitet alle Datenpakete in denen das Datenfeld Type den Wert *BUTTON_DATA* enthält und verwaltet die verfügbaren Buttons. Enthält das erhaltene Datenpaket also zum Beispiel Daten über eine Erstellung eines Buttons, wird dieser in eine Liste gespeichert und allen anderen Klassen über eine *getButtons()*-Methode zur Verfügung gestellt.
- Der **SetupManager** reagiert auf die neu eingeführten Datenpakete, welche die Initialisierungsinformationen für die Wearable Devices beinhalten. Er wandelt die JSON Nachricht im Datenfeld Value in ein Java-Objekt um und benachrichtigt mit diesem den, an ihm registrierten, InitializationListener.
- Der **ScreenManager** analysiert die Datenpakete mit dem Wert *SCREEN_MANAGER_DATA* im Datenfeld Type. Er verwaltet die geöffneten Screens in einer Liste und benachrichtigt bei der Öffnung und Schließung eines Screens alle, an ihm registrierten, ScreenChangedListener. Über die *getCurrentScreenId*-Methode stellt er ebenfalls die ID des aktuell geöffneten Screens zur Verfügung.
- Der **SVGManager** kommt nur zum Einsatz, wenn der SetupManager zuvor das Datenpaket zur Initialisierung der SVG Komponente erhalten hat. Er überprüft

die Datenpakete mit dem TYPE *SCREEN_DATA* auf Veränderungen der *image*-Eigenschaft der SVG Komponente. Wurde das SVG verändert oder das erste mal übertragen, benachrichtigt er den, an ihm registrierten, ImageListener mit dem aktuellen Bild.

- Der **TABLEMANAGER** ist nur dann aktiv, wenn das Wearable Device eine Tabellen-Komponente darstellen soll. Er verwaltet unter anderem die Daten der dargestellten Spalten und Zeilen, sowie die Zeilennummer der aktuell ausgewählten Zeile. Ändert sich etwas an den Daten oder die ausgewählte Zeile, so übergibt er die neuen Daten an den TableChangedListener.

Durch diesen Aufbau und durch den Einsatz des Listener-Patterns kann die Bibliothek sehr einfach um weitere Manager erweitert werden. Soll zum Beispiel der Titel oder die Instruktion des CTS auf dem Wearable Device verfügbar sein, so kann ein TITLEANDINSTRUCTIONMANAGER implementiert und am CLOUDCLIENT registriert werden, welcher die entsprechenden Datenpakete auswertet. In Abbildung 4.13 ist der soeben beschriebene Aufbau, inklusive des TITLEANDINSTRUCTIONMANAGER, zur Veranschaulichung in einem Klassendiagramm dargestellt.

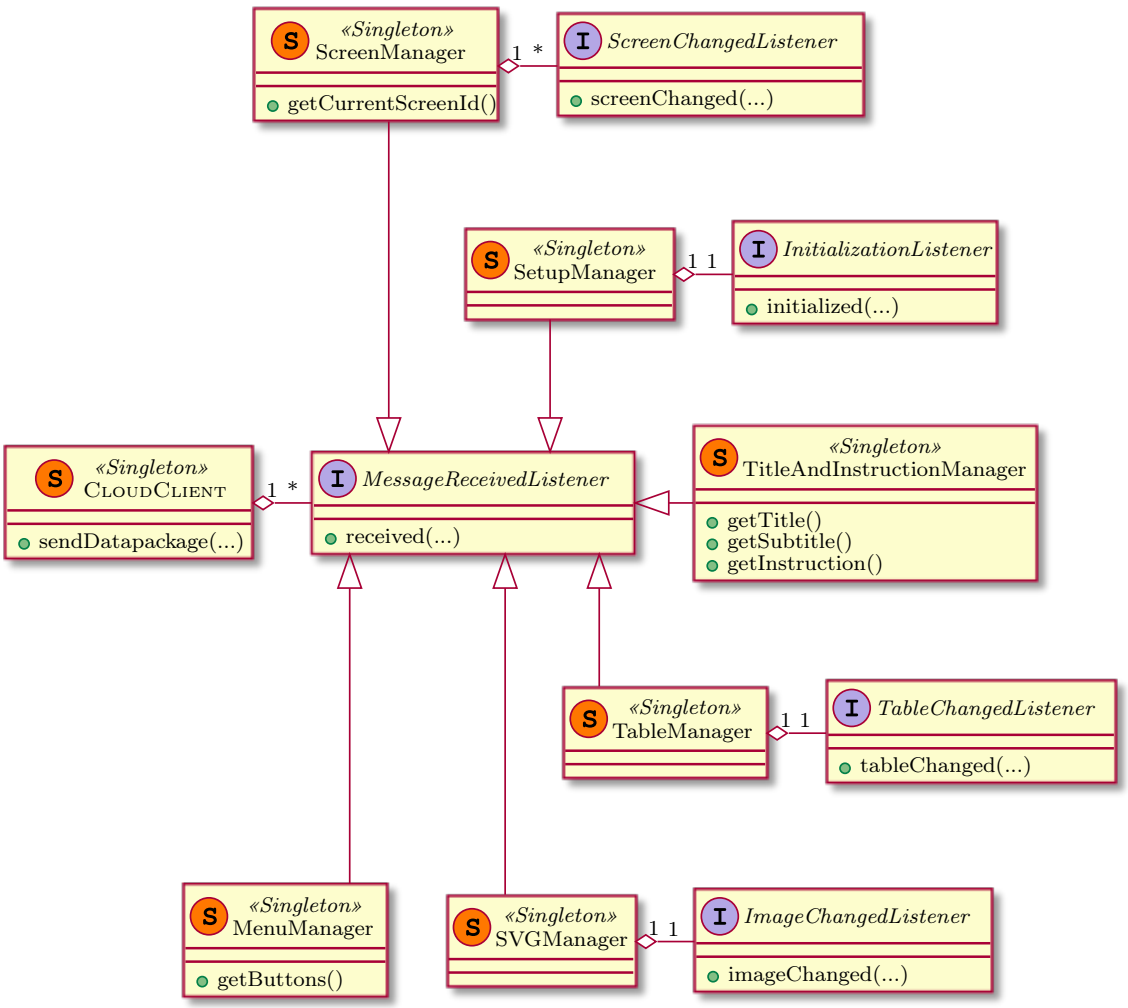


Abbildung 4.13: Klassendiagramm der verschiedenen Manager

Es ist möglich, dass sich in Zukunft der Aufbau der Datenpakete oder der eigentlichen Daten ändert. Damit die Bibliothek weitgehend versionsunabhängig ist, werden die übertragenen Datenpakete nicht in feste Strukturen geparsed. Stattdessen werden lediglich die benötigten Daten mit Hilfe von JSON-Path ausgewertet. Dies hat vor allem den Vorteil, dass die Manager den Wert im Datenfeld VALUE nicht erneut, abhängig von den enthaltenen Daten, parsen müssen. Mit JSON-Path wird das Datenpaket lediglich einmal geparsed und anschließend, vergleichsweise lose gekuppelt, über JSON-Path Ausdrücke auf die Daten zugegriffen.

In Listing 4.4 ist ein Ausschnitt aus dem TABLEMANAGER dargestellt. Die dargestellte Methode wird vom CLOUDCLIENT bei jeder erhaltenen Nachricht aufgerufen und erhält die Werte der Datenfelder TYPE und KEY, sowie das gelesene JSON Dokument als DocumentContext. Über letzteres werden die JSON-Path Ausdrücke ausgeführt, um die benötigten Daten aus dem Datenfeld VALUE auszulesen.

Der TABLEMANAGER überprüft zunächst in den Zeilen 2, 3, 5 und 18, ob das Datenpaket für ihn relevant ist. Die Datenpakete mit dem TYPE *SCREEN_DATA* und den in diesen Zeilen dargestellten KEY Werten, werden vom TABLEMANAGER ausgewertet. Wird die Zeile 4 ausgeführt, enthält das Datenpaket alle Überschriften der Tabellenspalten, welche in einer Variablen gespeichert werden. Im Block nach Zeile 5 enthält das Datenpaket alle Spalten-IDs der Tabelle. Zusammen mit den Initialisierungsinformationen, welche dem TABLEMANAGER zuvor per Setter-Methode mitgeteilt wurden, werden die ausgewählten Spalten und Spaltenüberschriften ermittelt und in eine Variable gespeichert. Trifft die Bedingung in Zeile 18 zu, so enthält das Datenpaket die Nummer der aktuell ausgewählten Zeile, welche ebenfalls in einer Variablen gespeichert wird. Des Weiteren wird in Zeile 20 der registrierte TableChangedListener mit den aktuellen Daten benachrichtigt.

```
1  public void received(String type, String key, DocumentContext doc) {
2     if (DataPackage.Type.SCREEN_DATA.equals(type)) {
3        if (key.endsWith("propColumnLabels")) {
4           allColumnLabels = doc.read("$.value[1]", String[].class);
5        } else if (key.endsWith("propColumnProperties")) {
6           allColumnProperties = doc.read("$.value[1]",
7              String[].class);
8           List<String> columnPropertyList = Arrays
9              .asList(allColumnProperties);
10          selectedColumnLabels = new String[
11             initData.getColumns().length
12          ];
13          for (int i = 0; i < selectedColumnLabels.length; i++) {
14              int index = columnPropertyList
15                 .indexOf(initData.getColumns()[i]);
16              selectedColumnLabels[i] = allColumnLabels[index];
17          }
18       } else if (key.endsWith("propSelectedRow")) {
19          selectedRow = doc.read("$.value", Integer.class);
20          listener.tableChanged(selectedRow,
21             selectedColumnLabels,
22             tableCellValues);
23       } // Weitere else-Blöcke ...
24    }
25 }
```

Listing 4.4: Ausschnitt aus dem TABLEMANAGER

Es gibt noch weitere ELSE-Blöcke für weitere KEY Werte. In diesen werden unter anderem die Werte der Zellen extrahiert und in der Variablen *tableCellValues* gespeichert. Die in Listing 4.4 genutzten JSON-Path Ausdrücke sind noch recht einfach. Die Ausdrücke zum Auslesen von Zellenwerten sind, wie Listing 4.5 zeigt, deutlich komplexer. Dieser Ausdruck wird benutzt, wenn das Datenpaket neue Werte für eine Zeile der Tabelle übermittelt. In diesem Beispiel wird der Wert der ersten Spalte ausgelesen. Die Ermittlung des Wertes ohne den Einsatz von JSON-Path ist schwierig, da jedes CTS seine eigenen Datenstrukturen definiert und somit die Werte unterschiedlich verschachtelt sein können. Somit ist ein Parsen in eine festgelegte Datenstruktur kaum möglich und würde bei Strukturänderungen große Probleme bereiten.

```
1 doc.read("$.value[1].f" +
    initData.getColumns()[0].replaceAll("\\.", "[1].").
    Object.class);
```

Listing 4.5: Auslesen des Wertes einer Zelle

Der SVGMANAGER, sowie die anderen Manager, arbeiten alle sehr ähnlich wie der in Listing 4.4 dargestellte TABLEMANAGER. Jeder Manager liest die für ihn relevanten Daten aus, speichert sie gegebenenfalls oder gibt sie an die registrierten Listener weiter.

4.3.2 Android

Damit im nachfolgenden Abschnitt die Anbindung an PRODIS.WTS besser verstanden wird, wird hier kurz auf einige Konzepte von Android eingegangen.

Activities

Nahezu jede Android Applikation benötigt zumindest eine Activity. Diese stellt dem Nutzer grafisch Daten in einer GUI zur Verfügung, mit denen er interagieren kann, und reagiert auf die Interaktionen. Eine Activity kann zum Beispiel eine Liste von Datensätzen darstellen und bei einem Klick auf ein Element eine weitere Activity starten, welche genauere Daten zu dem ausgewählten Datensatz darstellt.

Activity Lifecycle

Auf einem Android Smartphone ist üblicherweise immer nur eine Applikation mit einer Activity für den Nutzer sichtbar. Jedoch kann er zwischen verschiedenen Applikation und Activities wechseln, wobei es immer nur eine aktive Activity geben kann. Die Activities wechseln zwischen den in Abbildung 4.14 dargestellten Status [Act17]. Bevor eine Activity zum Beispiel pausiert wird, wird die Methode *onPause()* der Activity aufgerufen, in welcher diese die Möglichkeit hat, den aktuellen Status der Applikation zwischenzuspeichern. Wird die Activity wieder reaktiviert, kann sie in der *onResume()* Methode den zuletzt dargestellten Status der Applikation wiederherstellen, so dass der Benutzer dort weiter arbeiten kann, wo er die Activity verlassen hat.

Menü

Um dem Nutzer ein Menü darzustellen, gibt es in Android verschiedene Möglichkeiten. Mit Hilfe einer XML Datei können zum Beispiel statische Menüs definiert werden. Da es von

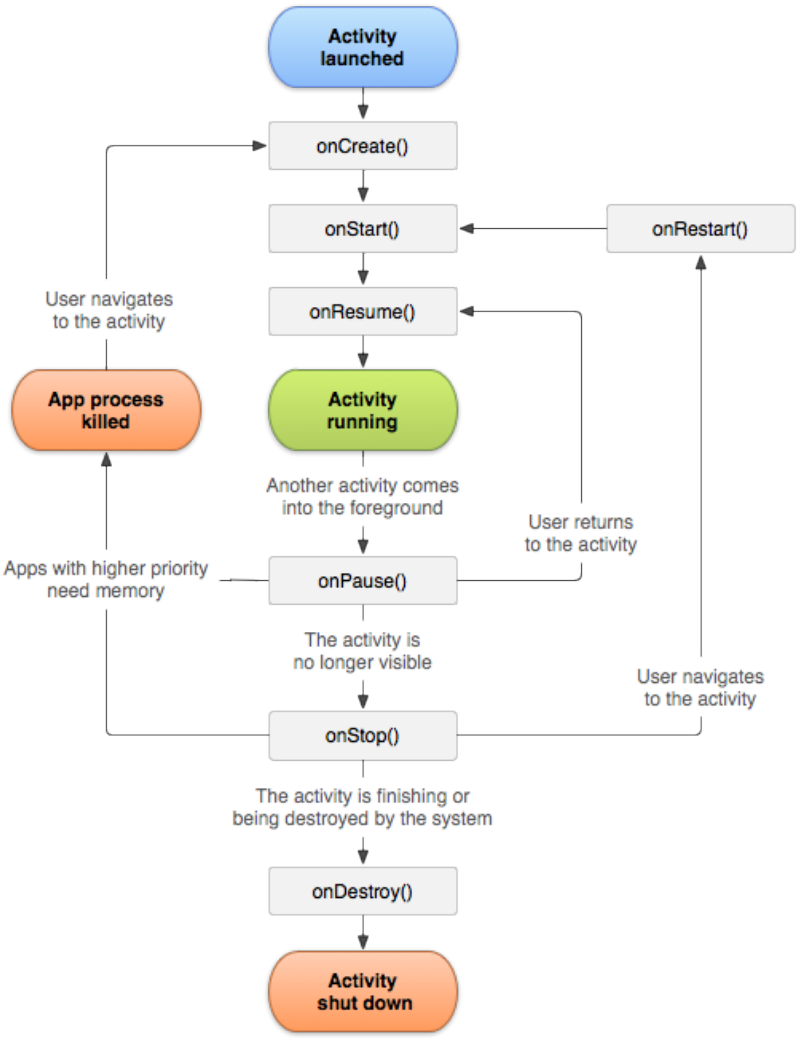


Abbildung 4.14: Vereinfachte Darstellung des Android Activity Lifecycle [Act17]

CTS zu CTS jedoch unterschiedliche Buttons geben wird, müssen die Menüs dynamisch zusammengestellt werden. Dazu bietet Android die Methode *onCreateOptionsMenu(...)* an einer Activity. Dieser wird ein Menu-Objekt übergeben, an welches die Activity dynamisch Menüeinträge hinzufügen kann. Jeder Eintrag erhält seine eigene ID, damit er in der *onOptionsItemSelected(...)* Methode der Activity identifiziert werden kann. Diese Methode wird aufgerufen, sobald der Nutzer auf einen Menüeintrag geklickt hat.

Aufruf anderer Activities

Eine Activity kann andere Activities aufrufen, um zum Beispiel weitere Informationen zu einem ausgewählten Datensatz anzuzeigen. Dieser können Daten übergeben werden, mit denen diese Activity zum Beispiel den ausgewählten Datensatz identifizieren und darstellen kann. Ebenfalls können Activities auch Ergebnisse zurückliefern. Hat eine Activity, mittels der *startActivityForResult()* Methode, eine andere Activity gestartet, so erhält sie die Ergebnisse der gestarteten Activity über die *onActivityResult(...)* Methode, sobald diese beendet wurde.

4.3.3 Google Glass

Die Implementierung einer Applikation für die Google Glass, auch Glassware genannt, unterscheidet sich kaum von der Implementierung einer Android-Applikation. Beide werden mit Hilfe des Android Software Development Kit (SDK) implementiert und anschließend in eine .apk-Datei paketiert. Google Glass erweitert das Android SDK um ein eigenes Glass Development Kit (GDK) [Gla15a], welches Google Glass spezifische Funktionalitäten hinzufügt.

Für die Anbindung an PRODIS.WTS wird die zuvor beschriebene Bibliothek eingesetzt. Die Aufgabe der eigentlichen Anwendung ist dadurch das Darstellen der empfangenen Daten. Allerdings müssen vor dem Aufbau der Verbindung mit PRODIS.WTS noch dessen Verbindungsdaten an die Google Glass mit einem QR-Code übermittelt werden. Dazu wird ein Fork von ZXing eingesetzt [ZXi16a], welcher eine Bibliothek zur Einbindung eines Barcode Scanners in eine Android Applikation bereitstellt. Zum Starten des Scanners genügt der Aufruf aus Listing 4.6 innerhalb einer Activity, wodurch eine neue Activity gestartet wird, welche von der Bibliothek bereitgestellt wird. Innerhalb des *IntentIntegrator*s wird dazu die zuvor beschriebene *startActivityForResult()* Methode aufgerufen. Die gestartete Activity stellt auf dem Bildschirm ein Live-Bild der Google Glass Kamera, wie in Abbildung 4.15 gezeigt, dar und analysiert die Videodaten.

```
1 new IntentIntegrator(this)
2   .setPrompt("Please scan the QR-Code")
3   .initiateScan(IntentIntegrator.QR_CODE_TYPES);
```

Listing 4.6: Starten des Barcode Scanners

Abbildung 4.15: Barcode Scanner des ZXing Forks

Sobald der QR-Code erkannt wurde, wird der aufrufenden Activity das Ergebnis zurückgeliefert. Diese entnimmt den JSON Daten, wie in Listing 4.7 gezeigt, die Verbindungsinformationen und startet anschließend, über die in Abschnitt 4.3.1 erstellte Bibliothek, den Verbindungsaufbau mit PRODIS.WTS. Zuvor registriert die Anwendung noch einen InitializationListener am SetupManager, welcher benachrichtigt wird, sobald die Initialisierungsinformationen, welche in Abschnitt 3.7 erläutert sind, empfangen wurden. Beim Starten des CloudClient in Zeile 17 wird neben den Verbindungsdaten ein Connection-Listener übergeben. Dieser reagiert auf die Herstellung und Trennung der Verbindung und zeigt dem Träger der Google Glass zum Beispiel nach der Herstellung an, dass auf die Initialisierungsinformationen vom PRODIS.WTS gewartet wird.

```
protected void onActivityResult(int requestCode, int resultCode,
      Intent data) {
   IntentResult result = IntentIntegrator
      .parseActivityResult(requestCode, resultCode, data);
   if (result != null && result.getContents() != null) {
      String contents = result.getContents();
      DocumentContext doc = JsonPath.parse(contents);
      connectToServer(doc.read("$.cloudServerHost"),
                      doc.read("$.cloudServerPort"),
                      doc.read("$.cloudServerPath"));
   }
   // ...
}
private void connectToServer(String host, int port, String path) {
   CloudClient.INSTANCE.stop();
   SetupManager.INSTANCE.setListener(this);
   CloudClient.INSTANCE.start(host, port, path, connectionListener);
}
```

Listing 4.7: Auswerten des QR-Codes und Starten der Verbindungsherstellung

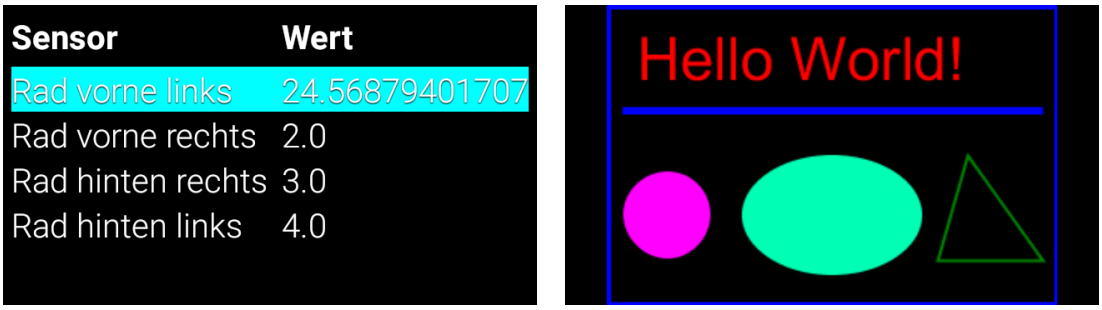
Anhand der erhaltenen Initialisierungsinformationen wird die, zur darzustellenden GUI Komponente passende, Activity gestartet. Es gibt somit ebenfalls eine Activity je GUI Komponente. Diese Activity fügt am CloudClient den TableManager oder SVGManager als MessageReceivedListener hinzu. Listing 4.8 zeigt dies in Zeile 4 anhand der TableActivity. In Zeile 5 wird anschließend am TableManager ein TableChangedListener gesetzt, welcher bei den Datenänderungen der Tabelle über die neuen Daten benachrichtigt wird. Handelt es sich um die Darstellung eines SVG, so fügt die SVGActivity am SVGManager einen ImageChangeListener hinzu, welcher über Änderungen am SVG Bild benachrichtigt wird.

Wird die Activity beendet, entfernt sie in Zeile 10 den TableManager, beziehungsweise den SVGManager bei der Darstellung eines SVG, als MessageReceivedListener vom CloudClient.

```
protected void onCreate(Bundle savedInstanceState) {
   // ...
   CloudClient.INSTANCE
      .addMessageReceivedListener(TableManager.INSTANCE);
   TableManager.INSTANCE.setListener(this);
}
protected void onStop() {
   // ...
   CloudClient.INSTANCE
      .removeMessageReceivedListener(TableManager.INSTANCE);
}
```

Listing 4.8: Ausschnitte des Lifecycle der TableActivity

In Abbildung 4.16 wird der Bildschirminhalt der zwei Activities beispielhaft dargestellt. Die Tabelle in Abbildung 4.16a stellt zwei Spalten, sowie alle vier Räder des ABS CTS aus Abschnitt 4.2.3 dar. Die aktuell markierte Zeile in der Tabelle auf dem Screen des CTS im PRODIS.WTS, wird auch auf der Google Glass farblich von den anderen Zeilen hervorgehoben. Abbildung 4.16b zeigt ein, im PRODIS.WTS von einer Datei geladenes, SVG auf der Google Glass.



(a) Darstellung der Tabelle aus dem ABS CTS

(b) Darstellung eines Beispiel SVG (SVG-Quelle: [SVG17])

Abbildung 4.16: Beispielhafter Bildschirminhalt einer Tabelle und eines SVG auf der Google Glass

Für die Darstellung der ausgewählten CTS Buttons tippt der Werker auf das Touchpad der Google Glass. Daraufhin wird, wie in Listing 4.9 gezeigt, ein Menü aus den übertragenen Buttons zusammengestellt. Dazu werden die Buttons in Zeile 2 vom MenuManager ermittelt und in Zeile 5 dem Menü hinzugefügt.

```
1  public boolean onCreateOptionsMenu(Menu menu) {
2    List<CloudButton> buttons = MenuManager.INSTANCE.getButtons();
3    for (int i = 0; i < buttons.size(); i++) {
4      CloudButton cloudButton = buttons.get(i);
5      menu.add(Menu.NONE, i, Menu.NONE, cloudButton.getLabel());
6    }
7    return true;
8  }
9  public boolean onOptionsItemSelected(MenuItem item) {
10   List<CloudButton> buttons = MenuManager.INSTANCE.getButtons();
11   CloudButton clickedButton = buttons.get(item.getItemId());
12   CloudClient.INSTANCE.fire(clickedButton.createClickEvent());
13   return true;
14 }
```

Listing 4.9: Erstellung und Auswertung des Menüs

In Abbildung 4.17a wird dabei der Button *Nächstes Rad* als Menüpunkt dargestellt. Über Touchgesten kann der Werker zwischen allen vorhandenen Menüpunkten wechseln. Durch tippen auf das Touchpad wird der aktuelle Menüpunkt ausgewählt und der Activity mitgeteilt. Die beiden Activities ermitteln aus dem ausgewählten Eintrag die ID des Buttons und übertragen in Zeile 12 ein Klick-Event-Datenpaket, wie in Abschnitt 3.7 beschrieben, mittels des CloudClient an den CloudServer.

Wird das CTS im PRODIS.WTS beendet, bleibt die WebSocket Verbindung weiterhin bestehen und das Wearable Device wartet auf neue Initialisierungsinformationen. Dazu beendet die Google Glass die aktuell ausgeführte Activity und stellt stattdessen eine Warte-Nachricht, wie in Abbildung 4.17b, dar.

(a) Menüdarstellung beim ABS CTS auf der Google Glass

(b) Warte-Screen auf der Google Glass

Abbildung 4.17: Darstellung eines Menüs und eines Wartebildschirms

Kapitel 5

Verwandte Arbeiten

In der Industrie wurden schon früh die Möglichkeiten von AR Systemen untersucht, zu denen auch die Google Glass gehört. So untersuchte [FMS93] zu Beginn der 1990er den Einsatz von kontextbezogenen AR Systemen anhand der Wartung von Laserdruckern. Dabei wurde das Reflection Technology Private Eye Head-Mounted Display (HMD) genutzt, welches ein Auge verdeckt und dort die Wartungsschritte dargestellt. Da das Gehirn die zwei Bilder vereint, erscheint das HMD durchsichtig, wodurch die Realität um die dargestellten Informationen erweitert wird.

Ende der 1990er untersuchten [SB97] und [BHBS99] wie HMDs in der Montage und Wartung gegen klassische Papier-Handbücher abschneiden. [SB97] stellte fest, dass die Dauer einer Wartung mit einem HMD, auf welchem die Wartungsschritte dargestellt wurden, etwas mehr Zeit in Anspruch nahm, als mit den Papier-Handbüchern. Allerdings wurde durch die Schritt-für-Schritt Anleitung der Vorgang deutlich sorgfältiger erledigt. [BHBS99] untersuchte einen etwas anderen Ansatz. Dabei übten die Testpersonen im Vorfeld die Montage einer Wasserpumpe mittels AR, Virtual Reality (VR) und Papier-Handbüchern. Anschließend sollten sie die Montage ohne jegliche Hilfe durchführen, wobei festgestellt wurde, dass durch den Einsatz von HMDs der Wissenstransfer deutlich schneller ist als mit klassischen Papier-Handbüchern. HMDs, oder auch allgemein AR, werden von vielen Arbeiten genutzt. Sie entsprechen zwar nicht immer der Definition von Wearable Devices, jedoch werden die Begriffe im Folgenden miteinander vermischt, da die Geräte aus den Arbeiten für einen Einsatz in der Industrie meist auf Wearable Devices verkleinert werden müssten.

[Bür02] erstellte 2002 das Interaction Constraints Model für Wearable Devices, bei welchem im Vorfeld Arbeitssituationen modelliert werden. Dabei werden die Arbeitsabläufe zusammen mit einigen Anforderungen an die Umgebung in einer Datenbank gespeichert. Diese Anforderungen umfassen zum Beispiel die Geräte, die Personen, die Arbeitsumgebung oder die genutzte Anwendung. Vor der Erstellung von zukünftigen Arbeitsabläufen wird anhand der zuvor erstellten Anforderungen die bestehende Datenbank durchsucht und überprüft, ob gegebenenfalls bereits ähnliche Szenarien wiederverwendet werden können. Dies ähnelt dem Autorensystem PRODIS.Authoring, da auch dort die Diagnoseabläufe im Vorfeld definiert werden und einige Teile wiederverwendet werden können.

In den folgenden Jahren untersuchten einige Arbeiten wie effektiv der Einsatz von AR und Wearable Devices, wie zum Beispiel HMDs, bei Wartungs- und Montagetätigkeiten sind. [ZZA04] nutzte ein HMD, welches Objekte erkennen und dem Träger Wartungsanleitungen

per Text, Bild oder Video anzeigen konnte. [PC05] evaluierte, wie solche AR und HMD Systeme den Wissenstransfer beeinflussen und stellte fest, dass sich die Montagezeit, sowie die benötigten Arbeitsschritte verringerten.

[RBW05] nutzte 2005 HMDs zusammen mit klassischen Handhelds bei der Montage und Wartung von Autos, Flugzeugen und Straßenbahnen. Ein Handheld ist ein kleines elektronisches Gerät, welches mit einer Hand gehalten werden kann. Bekannte Handhelds zu der Zeit waren zum Beispiel Personal Digital Assistants (PDAs), heute fallen unter anderem Smartphones in diese Kategorie. Anhand von Markern wurden Objekte erkannt und zugehörige statische Daten, wie zum Beispiel der Bauplan eines Garage Information Systems, vom Wearable Device abgefragt und dargestellt. Die eigentliche Diagnose geschah aber weiterhin über das Handheld. Auf dem Wearable Device wurden dem Träger Texte, Bilder oder Videos passend zur Diagnose vom eingesetzten System dargestellt. Anders als in der vorliegenden Arbeit, stellte das Wearable Device allerdings keine Echtzeitdaten dar, sondern unterstützte den Träger mit Schritt-für-Schritt Anweisungen oder erweiterte die Realität mittels AR zum Beispiel um eine Luftströmungs-Simulation in der Flugzeugkabine.

[Wil06] nutzte HMD Wearable Devices zur Schritt-für-Schritt Anleitung bei der Montage und Wartung in Flugzeugen. Mit Hilfe von AR sollten dabei die Anweisungen grafisch über die realen Objekte gelegt werden. Dabei wurde festgestellt, dass die automatische Erkennung von Objekten schlicht durch das ähnliche Aussehen und, gerade bei Kleinteilen, durch schlechte Lichtverhältnisse nicht zuverlässig funktioniert. Durch den Einsatz von Markern, wie zum Beispiel QR-Codes, konnte man dem Problem entgegenwirken.

[BE05b] untersuchte ebenfalls den Einsatz von Wearable Devices bei der Wartung von Flugzeugen. Die Wearable Devices erhielten dabei Zugriff auf alle Dokumentationen und Nachschlagewerke auf einem Server, wodurch die Träger diese direkt vor Ort nutzen können. Auch konnten die durchgeführten Aktionen direkt ohne Medienbruch protokolliert und auf dem Server eingepflegt werden. Ein Vorteil der Wearable Devices ist, dass sie auch weniger qualifizierten Personal ermöglichen die Aufgaben durchzuführen. Bei Bedarf könnten diese eine Ferndiagnose von Experten über das Wearable Device durchführen lassen. Da bei vielen Tätigkeiten beide Hände benötigt werden, war dies ein weiterer Vorteil der Wearable Devices. Auch [WNK06] und [NSW+06] sehen darin einen großen Vorteil. Der Fokus dort lag auf dem Design der GUI für *hands-free* Wartungen bei Flugzeugen. Es wurde festgestellt, dass die Steuerung mittels Gesten je nach Kontext gut funktioniert. Auch das Design des UI der Wearable Devices sei sehr wichtig, da zu viele oder schlecht dargestellte Informationen dem Träger bei HMD zum Beispiel eher verwirren als unterstützen würden. Anders als wie in der vorliegenden Arbeit, wurde in [WNK06] ein zweispaltiges Layout gewählt. Dabei beinhaltete die größere Spalte den darzustellenden Inhalt und die kleinere die Menüeinträge.

Die EU startete 2004 das wearIT@work Projekt, welches sich mit der Integration von Wearable Devices in der Industrie beschäftigte. Über 35 Unternehmen aus 14 europäischen Ländern nahmen an dem Projekt teil und untersuchten den Einsatz von Wearable Devices in vielen verschiedenen Industriesektoren. Unter anderem untersuchten [BLCB06], [MM08] und [LHW07], im Rahmen dieses Projekts, den Einsatz von Wearable Devices in der Automobil- und Luftfahrt-Industrie. [BLCB06] und [LHW07] sehen die geführte Inspektion, Montage und Fehlersuche als sinnvollen Einsatz für Wearable Devices. [MM08] untersuchte die Unterschiede in der Dauer einer Montage am Auto und kam zum Schluss, dass sich Wearable Devices vor allem zum Wissenstransfer und für Schulungen eignen.

Durch die ständige Verfügbarkeit von Informationen und Nachschlagewerken, konnte die Montage mit Wearable Devices schneller als mit entsprechenden Informationen auf Papier durchgeführt werden.

Die Dissertation [Teg07] untersuchte AR im Kontext der Automobilindustrie. Sie ähnelt der vorliegenden Arbeit, da dort ebenfalls zwischen einem Laufzeitsystem und einem Autorensystem unterschieden wurde. Anders als in dieser Arbeit, wurde das Laufzeitsystem allerdings speziell für HMD Geräte entwickelt und nicht in ein bestehendes System integriert. In der Dissertation wurden dazu die bisherigen Montageabläufe untersucht und anschließend Abläufe mit Hilfe von AR optimiert. Mittels eines so genannten Interaktionsstifts konnten die bestehenden Abläufe um Leitwege erweitert werden. Dazu wurde ein Ablauf aufgezeichnet und mit Hilfe des Interaktionsstifts dreidimensionale Informationen wie Pfeile, Zeichnungen oder ähnliches hinzugefügt. Diese Informationen, sowie die Zuordnung von Markern zu Abläufen und Objekten, wurden in das Autorensystem eingepflegt. Das Laufzeitsystem erhält ebenfalls, wie in vielen zuvor vorgestellten Abhandlungen, Zugriff auf die Protokolle, Baupläne und Handbücher passend zur durchgeführten Montage. Mit Hilfe einer Montage eines LEGO Modells wurde das System evaluiert und festgestellt, dass die Montage mit AR etwas langsamer als die klassische Montage mit Papier war. Als Grund dafür sieht der Autor einige technische Schwierigkeiten wie die Ladezeiten der Modelle und durch Lichtverhältnisse instabile Erkennung und Tracking von Objekten. Als Anwendungsfälle der HMDs werden unter anderem Schulungen und Weiterbildungen genannt, aber auch bei unregelmäßigen Aufgaben oder zur Unterstützung von Nicht-Fachpersonal können HMDs eingesetzt werden.

[SSL+08] untersuchte die Effizienz von HMDs im Vergleich zu Papier. Dabei wurde festgestellt, dass HMD eine geringere Fehlerquote bei einer Montage von Traktor-Bauteilen aufwiesen.

In [RRP+09] wurde der Einsatz von Wearable Devices in der Logistik bei Automobilterminals überprüft. Auf Automobilterminals werden Fahrzeuge für den Transport zwischengelagert und umgeschlagen. Zur Unterstützung der Arbeiter soll eine *Smart Jacket* dem Träger bei der Positions- und Zielbestimmung, sowie Identifizierung des Fahrzeugs unterstützen.

[WSWL14] untersuchte den Einsatz der Google Glass gegenüber eines Tablets bei der Montage eines LEGO Automodells. Dabei stellte sich heraus, dass die Montage mittels Google Glass, durch die ständige Refokussierung zwischen den Anweisungen auf dem Display und den Bauteilen, länger dauerte als mit den auf dem Tablet dargestellten Anweisungen.

Mit Hilfe eines AR Tablets wurden einem Servicetechniker in [KGW14] zusätzliche Informationen zu erkannten Objekten dargestellt. Mit Blick auf Wearable Devices wurde festgestellt, dass gerade in der Industrie häufig *hands-free* Interaktionen nötig sind und daher ein Tablet für einen solchen Einsatz ungeeignet ist.

Die Google Glass wurde in [RZT+15] bei der Produktion eines Audi A8 eingesetzt. Dabei wurde festgestellt, dass mit der Google Glass unabhängig vom Wissenstand der Arbeiter, Kalibrierungen am Fahrzeug durchgeführt werden konnten. Dabei war eine konsistente Interaktion und eine aufgeräumte GUI sehr wichtig, damit der Wissenstransfer funktionieren kann.

[ZSF+15] setzte die Google Glass vor allem für Checklisten und Reporting bei der Wartung von Industriemaschinen ein. Mit Hilfe von Fotos, Audio- und Videoaufnahmen, sowie einer Zeiterfassung kann die Wartung direkt digital dokumentiert und ausgewertet werden.

[SSK+16] nutzte dazu die Datenbrille Vuzix M100 und stellte dabei fest, dass die Steuerung mittels Sprache und Gesten deutlich langsamer als die Steuerung über Tasten ist.

[ZHU15] nutzte eine Smartwatch, welche von einem Server die benötigten Wartungsprozesse, Anleitungen und weitere Informationen erhält und dem Träger mittels Benachrichtigungen, zum Beispiel über einen Druckverlust, auf Probleme aufmerksam macht und ihn bei der Wartung unterstützt. Als großen Vorteil sah die Arbeit, dass dem Träger ein zusätzliches Display direkt vor Ort zur Verfügung steht, über welches er bei Bedarf Informationen einsehen und seine durchgeführten Aktionen, zum Beispiel durch eine Checkliste, protokollieren kann.

[ZFS+15] verglich vier verschiedene Hilfsmittel einer Fahrzeugwartung: Papier, Tablet und zwei Datenbrillen: Epson Moverio und Google Glass. Dabei zeigt die Epson Moverio dem Träger, anders als die Google Glass, die Informationen direkt im Sichtfeld an und nicht am Rand des Sichtfelds. In einer Evaluation wurde festgestellt, dass die Wartung mit Hilfe von Papier und Tablet schneller als mit den zwei Wearable Devices war. Des Weiteren wurde festgestellt, dass das zentrale Anzeigen der Informationen im Sichtfeld schneller war als die periphere Visualisierung der Google Glass. Ein Grund dafür ist die notwendige Neufokussierung bei der Google Glass, da in der Zeit nicht weitergearbeitet wurde. Anders als bei [BE05b, WNK06, NSW+06] wurde hier kein Vorteil in *hands-free* Interaktionen gesehen, da die Probanden beim Einsatz von Papier und Tablet immer schnell eine Ablage im Fahrzeug gefunden haben. Bei der Evaluation stellten die Autoren bei den Probanden mit Wearable Devices des Weiteren einen *over-reliance* Effekt fest. Die Probanden vertrauten den Wearable Devices teilweise mehr als sich selbst und befolgten die Anweisungen quasi blind, ohne darüber nachzudenken. Dieser Effekt ist nicht neu und wird häufig nach der Einführung von neuen technischen Hilfsmitteln erkannt. So zeigte eine Studie, dass durch die Nutzung der Auto-Korrektur beim Verfassen einer Nachricht über ein Smartphone, die Menschen nicht mehr kontrollieren, was sie geschrieben haben [Pat12].

Insgesamt lässt sich feststellen, dass es in den letzten Jahren vermehrt Untersuchungen von Wearable Device im industriellen Kontext gegeben hat. Einige Arbeiten zeigten dabei gewisse Ähnlichkeiten im Vergleich zu dieser Arbeit. So untersuchten viele Arbeiten den Einsatz von Wearable Devices oder AR im Kontext von Montage- oder Wartungsarbeiten, welche sehr ähnlich zu Diagnosearbeiten sind. Teilweise bezogen die Geräte dabei ihre Daten von einem Server, allerdings gab es in keiner Arbeit eine Echtzeitkommunikation. Am nächsten dazu kam [ZHU15], wo die Smartwatch vom Server proaktiv benachrichtigt werden konnte. Der getrennte Aufbau eines Autoren- und eines Laufzeitsystems wurde in [Teg07] betrachtet, allerdings wurde dort keine bestehende Software erweitert. Interessant sind die deutlichen Unterschiede bei den Ergebnissen der Evaluationen. So war der Einsatz von Wearable Devices mal schneller und mal langsamer als mit einem Tablet oder mit Hilfe von Anweisungen auf Papier. Allerdings waren sich fast alle Arbeiten einig, dass sich Wearable Devices, und vor allem HMDs, sehr gut zum Wissenstransfer und für Schulungen und Weiterbildungen einsetzen lassen. Dieser Effekt wurde in [BHBS99] und [PC05] genauer untersucht und bestätigt.

Kapitel 6

Zusammenfassung und Ausblick

Im Folgenden wird die vorliegende Arbeit kurz zusammengefasst und anschließend ein Ausblick auf mögliche Erweiterungen und weitere Anwendungsfälle gegeben.

6.1 Zusammenfassung

Zu Beginn der Arbeit wurden zunächst anhand einer beispielhaften Fahrzeugdiagnose, einige Schwierigkeiten bei der Nutzung bestehender Hilfsmittel aufgezeigt. So ist das Transportieren und Interagieren mit den Diagnosegeräten nicht immer während einer Diagnose von einem einzelnen Werker durchführbar. Um diese Probleme zu reduzieren, wurden zunächst die bestehenden Systeme der DSA GmbH vorgestellt um anschließend ein Konzept zur Einbindung von Wearable Devices in diese System zu erstellen. Dabei gab es fünf wichtige Anforderungen, die das Konzept erfüllen sollte. So sollen, neben den Wearable Devices, keine zusätzlichen Geräte zur Übertragung der Daten oder ähnliches benötigt werden. Dazu sollen die Wearable Devices die benötigtet Daten über PRODIS.WTS beziehen. Des Weiteren sollen die Wearable Devices mit möglichst wenig Aufwand in bestehende Diagnoseabläufe integriert werden können., welches den Diese sollen des Weiteren Zugriff auf die Daten aus PRODIS.WTS erhalten und mit möglichst wenigen Anpassungen an den bestehenden CTSs auskommen.

PRODIS.WTS, das Laufzeitsystem für die Fahrzeugdiagnose der DSA GmbH, wurde bereits im Vorfeld der Arbeit um die Möglichkeit erweitert auf die Daten zwischen CTS und dessen Screens zuzugreifen. Damit die Wearable Devices Zugriff auf diese Daten erhalten und selbst mit dem CTS interagieren können, wurde ein Client-Server-Aufbau genutzt, bei dem PRODIS.WTS als Server agiert und die Wearable Devices mittels WebSocket mit diesem kommunizieren. Zur schnellen und einfachen Verbindungsherstellung wurde definiert, dass ein QR-Code zur Übertragung der Verbindungsinformationen an die Google Glass genutzt wird. Für Wearable Devices ohne eine Kamera aber mit einem Mikrofon wurde alternativ eine akustische Übertragung der Verbindungsinformationen vorgestellt. Als Fallback-Möglichkeit, wenn das Wearable Device keine Kamera und kein Mikrofon besitzt, wurde festgelegt, dass die Verbindungsinformationen von den Werkern manuell in die Wearable Devices eingegeben werden können.

Durch die eingeschränkten Bildschirmauflösungen und große Unterschieden in Form und Größe der Displays von Wearable Devices wurden Überlegungen erörtert, wie eine gewisse

Flexibilität bei den darzustellenden GUI Komponenten gewährleistet werden kann, ohne größere Anpassungen seitens der Hersteller an den bestehenden CTSs zu erforderlichen. Eine Möglichkeit war die Erweiterung des PRODIS.Authoring, damit eigene GUIs für die Wearable Devices konstruiert werden können. Auch verschiedene Layouts, denen mittels Platzhalter eine GUI Komponente zugewiesen werden konnte, wurden vorgestellt. Schlussendlich wurde aber festgelegt, dass nur eine, dafür aber frei definierbare, GUI Komponente eines CTS Screens auf den Wearable Devices dargestellt wird. Aus diesem Grund wurden in dieser Arbeit nur zwei GUI Komponenten näher betrachtet und implementiert: Tabellen und SVGs. Damit die Hersteller das Aussehen auf den Wearable Devices mitbestimmen können, wurden Parameter definiert, welche in einem CTS genutzt werden können, um beispielsweise die darzustellenden Spalten einer Tabelle zu definieren.

Nachdem der Datenaufbau der bestehenden Systeme kurz vorgestellt wurde, wurden drei Datenformate gegenübergestellt und, mit Hinblick auf den Anwendungsfall, nämlich der Übertragung der Datenpakete an die Wearable Devices, analysiert. JSON platzierte sich bei vielen Anforderungen vor XML und Protocol Buffers und wurde vor allem wegen seiner guten Kompaktheit, weiten Verbreitung und Einfachheit ausgewählt.

Abschließend wurde erläutert, wie diese Anpassungen und Neuerungen in die bestehenden Systeme integriert wurden und wie die Hersteller ihre bestehenden CTSs erweitern müssten, um die neuen Funktionalitäten nutzen zu können. Dabei wurde die Atmosphere Bibliothek genutzt, um im PRODIS.WTS einen WebSocket Server bereitzustellen. Für die Wearable Devices wurde das Unterprojekt von Atmosphere, wAsync, genutzt um die Client Anbindung zu implementiert. Mit Hilfe von ZXing wurde im PRODIS.WTS die Möglichkeit zum Generieren von QR-Codes hinzugefügt und auf der Google Glass wurde ein Fork von ZXing zum Lesen dieser Codes genutzt. Anhand der zu Beginn eingeführten Beispieldiagnose von ABS Sensoren wurde ein CTS entwickelt, welches zunächst gänzlich ohne den Einsatz von Wearable Devices arbeitet. Dieses wurde zur Veranschaulichung der durchzuführenden Schritte um die Nutzung von Wearable Devices erweitert. Zum Schluss wurde die Implementierung des Konzepts auf den Wearable Devices betrachtet. Dabei wurde die Kommunikation und Auswertung der Datenpakete in eine eigene Bibliothek ausgegliedert, damit diese wiederverwendet werden können. Für die Google Glass wurde abschließend eine Glassware, also eine um Google Glass spezifische Funktionalitäten erweiterte Android Applikation, implementiert. Durch die Nutzung der zuvor erstellten Bibliothek musste für die Google Glass lediglich die Darstellung und Visualisierung der bereitgestellten Daten implementiert werden.

Abschließend kann gesagt werden, dass das Ziel dieser Arbeit, die Einbindung von Wearable Devices in eine bestehende Systemlandschaft, erreicht wurde. An den bestehenden CTSs müssen für die Wearable Devices nur wenig Änderungen vorgenommen werden und mit Hilfe der Google Glass wurde die Funktionstüchtigkeit des Konzepts und der Implementierung bestätigt.

6.2 Ausblick

Während der Arbeit wurde eine erste Evaluierung mit einem Diagnoseexperten der DSA GmbH durchgeführt. Dieser sieht großes Potenzial, weshalb eine wissenschaftlichen Evaluierung und Auswertung der Ergebnisse in produktiven Diagnoseabläufen geplant ist, um den Nutzen und die Möglichkeiten, welche das Konzept den Werkern in der Diagnose ermöglicht, objektiver und praxisbezogener beurteilen zu können.

In Kapitel 5 wurden Anwendungsgebiete und Szenarien aus anderen Arbeiten vorgestellt. Einige davon können ohne großen Aufwand in dieses Konzept integriert werden. In [ZZA04, BE05b, NSW+06, ZHU15] waren auf den Geräten unter anderem Handbücher oder technischen Zeichnungen während der Montage und Wartung zugänglich. Eine solche Möglichkeit ist auch während eines Diagnoseablaufs denkbar. Dazu könnte entweder PRODIS.WTS zur Bereitstellung solcher Daten erweitert, oder die Daten könnten innerhalb des Netzwerk der Werkstätten auf einem Filesharing-Server zur Verfügung gestellt werden. Bisher stellen die Wearable Devices die Diagnosedaten von PRODIS.WTS dar. In [ZSF+15] wurden Wearable Devices aber auch zur Protokollierung der durchgeführten Schritte benutzt. Eine solche Erweiterung wäre auch für diesen Anwendungsfall denkbar. Durch die Protokollierung können später zum Beispiel besondere Diagnosefälle im Anschluss analysiert oder Schulungen für neue Mitarbeiter gegeben werden. Zur Protokollierung können auch Bilder, Videos oder Notizen des Werkers über das Wearable Device gesammelt werden. Werden diese in einer zentralen Datenbank hinterlegt, könnten später unter anderem unerfahrene Werker davon profitieren, indem sie anhand dieser Dokumentation lernen können.

Der Einsatz einer Checkliste, wie in [ZSF+15, SSK+16] vorgestellt, hätte bei der Fahrzeugdiagnose den Vorteil, dass alle Werker die Arbeitsschritte auf der Checkliste in jedem Fall durchführen und somit eine gewisse Qualität der Diagnose gewährleistet werden kann. Statt eine GUI Komponente, würde stattdessen eine Checkliste auf dem Wearable Device dargestellt werden. Die Schritte könnten über PRODIS.Authoring von den Fahrzeugherstellern für jedes CTS individuell festgelegt werden. Eine Zeitverfolgung der einzelnen Arbeitsschritte, wie in [ZSF+15] vorgestellt, ist ebenfalls eine mögliche Erweiterung des Konzepts. Dadurch könnten die Diagnoseabläufe im Anschluss ausgewertet und optimiert werden. Zur Protokollierung können auch Bilder, Videos oder Notizen des Werkers über das Wearable Device gesammelt werden. Werden diese in einer zentralen Datenbank hinterlegt, könnten später unter anderem unerfahrene Werker davon profitieren, indem sie anhand dieser Dokumentation lernen können.

Während diese Arbeit den Live-Einsatz von Wearable Devices während der Diagnose mit Hilfe von Diagnosedaten behandelt, zeigen fast alle Arbeiten in Kapitel 5, dass der Einsatz von Wearable Devices für Schritt-für-Schritt Anleitungen, Schulungen und Weiterbildungen einen großen positiven Effekt erzielte. Vor allem bei komplexen oder selten durchgeführten Tätigkeiten könnten Wearable Devices den Werkern in solchen Situationen deutlich mehr Vorteile bringen.

Bisher können Tabellen und SVGs auf den Wearable Devices dargestellt werden. Im Anschluss der Arbeit sollen weitere GUI Komponenten folgen um zum Beispiel Diagramme oder komplexere Darstellungen auf den Wearable Devices zu ermöglichen. Auch die Erweiterung um Medien, wie Audio und Video, könnten den Werkern je nach Diagnose behilflich sein. Dabei könnten auf den Wearable Devices zum Beispiel Videoanleitungen oder Audioanweisungen abgespielt werden. So wäre es zum Beispiel denkbar, dass die Google Glass ein Warnsignal zusätzlich zur grafischen Darstellung wiedergibt. Ein interessanter Ansatz wäre die Nutzung von kontextbezogener AR. Beobachtet der Werker mit der Google Glass oder einem anderem HMD ein Fahrzeugrad, könnten Daten wie die Drehgeschwindigkeit oder der Reifendruck zu diesem Rad dargestellt werden. Auch interessant wäre beispielsweise eine farbliche Hervorhebung der Hitzeentwicklung innerhalb des Motorraums mit Hilfe von AR.

Allgemein sollte in Zukunft evaluiert werden, welche weiteren Wearable Devices für den Einsatz in Werkstätten sinnvoll sind. Das *Cicret Bracelet* scheint zur Unterstützung von

Diagnoseabläufen vielversprechend. Dabei handelt es sich um einen Armreif, welcher den Bildschirm eines Smartphones auf die Haut des Arms projiziert [Cic17]. Des Weiteren soll der Träger auf dem projizierten Display das Smartphone nahezu normal bedienen können. Mittels einer einfachen Geste kann dabei das Display bei Bedarf aktiviert und deaktiviert werden. *Cicret Bracelet* bietet somit mehr Möglichkeiten als eine Smartwatch und vor allem ein größeres Display zur Darstellung von Inhalten.

Sobald die Wearable Devices von den Fahrzeugherstellern positiv angenommen wurden und diese sich mehr Freiraum und Mitspracherecht bei den dargestellten Inhalten wünschen, können einige Entscheidungen dieser Arbeit überdacht werden. So würde die Anforderung **REQ 3**, bei der es um minimale Änderungen der CTSs geht, vermutlich entfallen können, da die Hersteller nicht erst überzeugt werden müssten und für komplexere Anwendungsfälle auch mehr Anpassungen in Kauf nehmen. Auch könnte die Beschränkung auf die Darstellung einer GUI Komponente und eines einzigen Layouts aufgehoben, oder den Herstellern die Möglichkeit gegeben werden, die GUI der Wearable Devices, ähnlich wie die Screens der CTSs, in PRODIS.Authoring frei zu konstruieren.

Literaturverzeichnis

[Act17] *The Activity Lifecycle | Android Developers.* https://developer.android.com/guide/components/activities/activity-lifecycle.html. Version: Februar 2017

[Atm15a] *Understanding AtmosphereResource · Atmosphere/atmosphere Wiki · GitHub.* https://github.com/Atmosphere/atmosphere/wiki/Understanding-AtmosphereResource/aa80808e975dff72bb060caa8c88952af14ce83b. Version: Juni 2015

[Atm15b] *Understanding Broadcaster · Atmosphere/atmosphere Wiki · GitHub.* https://github.com/Atmosphere/atmosphere/wiki/Understanding-Broadcaster/5def570d2aca5a9877da789223c811d33b8ca8de. Version: Mai 2015

[Atm16a] *GitHub - Atmosphere/atmosphere: Realtime Client Server Framework for the JVM, supporting WebSockets with Cross-Browser Fallbacks.* https://github.com/Atmosphere/atmosphere/tree/atmosphere-project-2.4.9. Version: November 2016

[Atm16b] *GitHub - Atmosphere/wasync: WebSockets with fallback transports client library for Node.js, Android and Java.* https://github.com/Atmosphere/wasync/tree/wasync-project-2.1.3. Version: November 2016

[BE05a] Bublitz, Siegfried ; Eikerling, Heinz-Josef: Optimierung von Wartungs- und Instandhaltungsprozessen durch Wearable Computing. In: *C-LAB Report* 4 (2005), Nr. 2

[BE05b] Bublitz, Siegfried ; Eikerling, Heinz-Josef: Optimierung von Wartungs- und Instandhaltungsprozessen durch Wearable Computing. In: *C-LAB Report* 4 (2005), Nr. 2

[BHBS99] Boud, A. C. ; Haniff, D. J. ; Baber, C. ; Steiner, S. J.: Virtual reality and augmented reality as a training tool for assembly tasks. In: *1999 IEEE International Conference on Information Visualization (Cat. No. PR00210)*, 1999. – ISSN 1093–9547, S. 32–36

[BLCB06] Bo, G. ; Lorenzon, A. ; Chevassus, N. ; Blondel, V.: Wearable Computing and Mobile Workers: The Aeronautic Maintenance Showcase in the WearIT@Work Project. In: *3rd International Forum on Applied Wearable Computing 2006*, 2006, S. 1–12

[BPT15] Belshe, M. ; Peon, R. ; Thomson, M.: Hypertext Transfer Protocol Version 2 (HTTP/2) / RFC Editor. Version: May 2015. http://www.rfc-editor.org/rfc/rfc7540.txt. RFC Editor, May 2015 (7540). – RFC. – ISSN 2070–1721

[BR14] Bliem-Ritz, Daniela: *Wearable Computing. Benutzerschnittstellen zum Anziehen.* 1. disserta Verlag, 2014. – ISBN 9783954257942

[Bra14] Bray, T.: The JavaScript Object Notation (JSON) Data Interchange Format / RFC Editor. Version: March 2014. http://www.rfc-editor.org/rfc/rfc7159.txt. RFC Editor, March 2014 (7159). – RFC. – ISSN 2070–1721

[BSM+15] Ballhaus, Werner ; Song, Bin ; Meyer, Friedrich-Alexander ; Ohrtmann, Jan-Peter ; Dressel, Christian: *Media Trend Outlook - Wearables: Die tragbare Zukunft kommt näher.* https://www.pwc.at/images/tmt-studie-3.pdf. Version: März 2015

[Bür02] Bürgy, Christian: *An interaction constraints model for mobile and wearable computer-aided engineering systems in industrial applications*, Carnegie Mellon University Pittsburgh, Pennsylvania, USA, Diss., 2002

[Cic17] *Cicret Bracelet - Your skin is your new touchscreen.* http://www.cicret.com/. Version: Februar 2017

[Cla99] Clark, James: XSL Transformations (XSLT) Version 1.0 / W3C. Version: November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116. 1999. – W3C Recommendation

[DC99] DeRose, Steven ; Clark, James: XML Path Language (XPath) Version 1.0 / W3C. Version: November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116. 1999. – W3C Recommendation

[DSA17a] *Autorenwerkzeug PRODIS.Authoring - DSA GmbH.* http://www.dsa.de/de/loesungen/produkte/autorenwerkzeug-prodisauthoring/. Version: Februar 2017

[DSA17b] *Laufzeitsystem PRODIS.WTS - DSA GmbH.* http://www.dsa.de/de/loesungen/produkte/laufzeitsystem-prodiswts/. Version: Februar 2017

[FGM+99] Fielding, Roy T. ; Gettys, James ; Mogul, Jeffrey C. ; Nielsen, Henrik F. ; Masinter, Larry ; Leach, Paul J. ; Berners-Lee, Tim: Hypertext Transfer Protocol – HTTP/1.1 / RFC Editor. Version: June 1999. http://www.rfc-editor.org/rfc/rfc2616.txt. RFC Editor, June 1999 (2616). – RFC. – ISSN 2070–1721

[FM11] Fette, I. ; Melnikov, A.: The WebSocket Protocol / RFC Editor. Version: December 2011. http://www.rfc-editor.org/rfc/rfc6455.txt. RFC Editor, December 2011 (6455). – RFC. – ISSN 2070–1721

[FMS93] Feiner, Steven ; Macintyre, Blair ; Seligmann, Dorée: Knowledge-based Augmented Reality. In: *Commun. ACM* 36 (1993), Juli, Nr. 7, S. 53–62. http://dx.doi.org/10.1145/159544.159587. – DOI 10.1145/159544.159587. – ISSN 0001–0782

[Fow02] Fowler, Martin: *Patterns of Enterprise Application Architecture*. 1. Addison-Wesley Professional, 2002. – ISBN 9780321127426

[Fre17] *FreeStyle Libre – Blutzucker messen ohne Stechhilfe*. http://www.freestylelibre.de/. Version: Februar 2017

[Fri16] Friesen, Jeff: Extracting JSON Values with JsonPath. Apress, 2016. – ISBN 978–1–4842–1916–4, S. 223–239

[GHJ01] Gamma, Erich ; Helm, Richard ; Johnson, Ralph: *Entwurfsmuster . Elemente wiederverwendbarer objektorientierter Software (Programmer's Choice)*. 2. Aufl. Addison-Wesley, 2001. – ISBN 9783827318626

[GHJV94] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. – ISBN 9780201633610

[Gla15a] *Glass Development Kit | Glass | Google Developers*. https://developers.google.com/glass/develop/gdk/. Version: Mai 2015, Abruf: 17.02.2017

[Gla15b] *Principles | Glass | Google Developers*. https://developers.google.com/glass/design/principles. Version: Mai 2015, Abruf: 17.02.2017

[Gla16] *Glass at Work | Glass | Google Developers*. https://developers.google.com/glass/distribute/glass-at-work. Version: Mai 2016, Abruf: 13.02.2017

[Gla17a] *Glass gestures - Google Glass Help*. https://support.google.com/glass/answer/3064184. Version: Februar 2017

[Gla17b] *Make vignette - Google Glass Help*. https://support.google.com/glass/answer/3405215. Version: Februar 2017, Abruf: 13.02.2017

[Gla17c] *Tech specs - Google Glass Help*. https://support.google.com/glass/answer/3064128. Version: Februar 2017

[Gla17d] *XE16 release notes - Google Glass Help*. https://support.google.com/glass/answer/4578099. Version: Februar 2017

[IEE02] IEEE Standard for Telecommunications and Information Exchange Between Systems - LAN/MAN - Specific Requirements - Part 15: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks (WPANs). In: *IEEE Std 802.15.1-2002* (2002), June, S. 1–473. http://dx.doi.org/10.1109/IEEESTD.2002.93621. – DOI 10.1109/IEEESTD.2002.93621

[Int13] International, ECMA: *The JSON Data Interchange Format*. https://www.ecma-international.org/publications/standards/Ecma-404.htm. Version: Oktober 2013, Abruf: 19.02.2017

[ISO94] International Organization for Standardization: Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model. 1994. – Standard

[ISO15] International Organization for Standardization: Information technology – Automatic identification and data capture techniques – QR Code bar code symbology specification. 2015. – Standard

[Jos06] Josefsson, S.: The Base16, Base32, and Base64 Data Encodings / RFC Editor. Version: October 2006. http://www.rfc-editor.org/rfc/rfc4648.txt. RFC Editor, October 2006 (4648). – RFC. – ISSN 2070–1721

[KGW14] Krugmann, Martina ; Groenefeld, Jan ; Willmann, Stephan: Augmented Reality-Vom Spielzeug zum Arbeitswerkzeug. In: *UP14-Vorträge* (2014)

[LHW07] Lawo, Michael ; Herzog, Otthein ; Witt, Hendrik: An Industrial Case Study on Wearable Computing Applications. In: *Proceedings of the 9th International Conference on Human Computer Interaction with Mobile Devices and Services*. New York, NY, USA : ACM, 2007 (MobileHCI '07). – ISBN 978–1–59593–862–6, S. 448–451

[Los15] Losch: *File:Bluetooth ELM327 OBD2-Scanner IMG 6322.jpg - Wikimedia Commons*. https://de.wikipedia.org/wiki/Datei:Bluetooth_ELM327_OBD2-Scanner_IMG_6322.jpg. Version: Februar 2015, Abruf: 17.02.2017

[LSA11] Lubbers, Peter ; Salim, Frank ; Albers, Brian: *Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development (Expert's Voice in Web Development)*. 2nd ed. Apress, 2011. – ISBN 9781430238645

[LSO+15] Lerner, Adam ; Saxena, Alisha ; Ouimet, Kirk ; Turley, Ben ; Vance, Anthony ; Kohno, Tadayoshi ; Roesner, Franziska: Analyzing the Use of Quick Response Codes in the Wild. In: *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. New York, NY, USA : ACM, 2015 (MobiSys '15). – ISBN 978–1–4503–3494–5, S. 359–374

[Man98] Mann, Steve: Wearable Computing as Means for Personal Empowerment. Fairfax, VA : IEEE Computer Society Press, Mai 1998

[MM08] Matysczok, C. ; Maurtua, I.: Supporting mobile workers in car production by wearable computing – Applied context detection. In: *2008 IEEE International Technology Management Conference (ICE)*, 2008, S. 1–8

[Mot17] *Moto 360 - Smartwatch powered by Android Wear - Motorola*. https://www.motorola.com/us/products/moto-360. Version: Februar 2017

[Mov13] MovGP0: *Datei:Google Glass Main.jpg – Wikipedia*. https://de.wikipedia.org/w/index.php?title=Datei:Google_Glass.svg. Version: April 2013, Abruf: 17.02.2017

[MYB+08] Maler, Eve ; Yergeau, François ; Bray, Tim ; Sperberg-McQueen, Michael ; Paoli, Jean: Extensible Markup Language (XML) 1.0 (Fifth Edition) / W3C. Version: November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/. 2008. – W3C Recommendation

[NSW+06] Nicolai, T. ; Sindt, T. ; Witt, H. ; Reimerdes, J. ; Kenn, H.: Wearable Computing for Aircraft Maintenance: Simplifying the User Interface. In: *3rd International Forum on Applied Wearable Computing 2006*, 2006, S. 1–12

[ODM01] Orchard, David ; DeRose, Steven ; Maler, Eve: XML Linking Language (XLink) Version 1.0 / W3C. Version: Juni 2001. http://www.w3.org/TR/2001/REC-xlink-20010627/. 2001. – W3C Recommendation

[OKL11] Oh, Dong-Sik ; Kim, Bong-Han ; Lee, Jae-Kwang: A study on authentication system using QR code for mobile cloud computing environment. In: *Future Information Technology: 6th International Conference, FutureTech 2011, Loutraki, Greece, June 28-30, 2011, Proceedings, Part I*. Springer, 2011. – ISBN 978–3–642–22333–4, S. 500–507

[Pat12] Paton, Graeme: *Over-reliance on technology is undermining spelling skills*. http://www.telegraph.co.uk/education/educationnews/9280203/Over-reliance-on-technology-is-undermining-spelling-skills.html. Version: Mai 2012

[PC05] Pathomaree, N. ; Charoenseang, S.: Augmented reality for skill transfer in assembly task. In: *ROMAN 2005. IEEE International Workshop on Robot and Human Interactive Communication, 2005.*, 2005. – ISSN 1944–9445, S. 500–504

[Pem02] Pemberton, Steven: XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition) / W3C. Version: August 2002. http://www.w3.org/TR/2002/REC-xhtml1-20020801. 2002. – W3C Recommendation

[Pos80] Postel, J.: User Datagram Protocol / RFC Editor. Version: August 1980. http://www.rfc-editor.org/rfc/rfc768.txt. RFC Editor, August 1980 (6). – STD. – ISSN 2070–1721

[Pos81a] Postel, Jon: Internet Protocol / RFC Editor. Version: September 1981. http://www.rfc-editor.org/rfc/rfc791.txt. RFC Editor, September 1981 (5). – STD. – ISSN 2070–1721

[Pos81b] Postel, Jon: Transmission Control Protocol / RFC Editor. Version: September 1981. http://www.rfc-editor.org/rfc/rfc793.txt. RFC Editor, September 1981 (7). – STD. – ISSN 2070–1721

[Pro16a] *Developer Guide | Protocol Buffers | Google Developers*. https://developers.google.com/protocol-buffers/docs/overview. Version: August 2016, Abruf: 13.02.2017

[Pro16b] *Encoding | Protocol Buffers | Google Developers*. https://developers.google.com/protocol-buffers/docs/encoding. Version: April 2016, Abruf: 13.02.2017

[RBW05] Regenbrecht, H. ; Baratoff, G. ; Wilke, W.: Augmented reality projects in the automotive and aerospace industries. In: *IEEE Computer Graphics and Applications* 25 (2005), Nov, Nr. 6, S. 48–56. http://dx.doi.org/10.1109/MCG.2005.124. – DOI 10.1109/MCG.2005.124. – ISSN 0272–1716

[Rec14] Reckmann, Tim: *Datei:Google Glass Main.jpg – Wikipedia*. https://de.wikipedia.org/wiki/Datei:Google_Glass_Main.jpg. Version: Mai 2014, Abruf: 17.02.2017

[RRP+09] Rügge, Ingrid ; Ruthenbeck, Carmen ; Piotrowski, Jakub ; Meinecke, Christian ; Böse, Felix: Supporting Mobile Work Processes in Logistics with Wearable Computing. In: *Proceedings of the 11th International Conference on Human-Computer Interaction with Mobile Devices and Services*. New York, NY, USA : ACM, 2009 (MobileHCI '09). – ISBN 978–1–60558–281–8, S. 77:1–77:2

[RZT+15] Rauh, S. ; Zsebedits, D. ; Tamplon, E. ; Bolch, S. ; Meixner, G.: Using Google Glass for mobile maintenance and calibration tasks in the AUDI A8 production line. In: *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, 2015. – ISSN 1946–0740, S. 1–4

[SB97] Siegel, J. ; Bauer, M.: A field usability evaluation of a wearable system. In: *Digest of Papers. First International Symposium on Wearable Computers*, 1997, S. 18–22

[Sch15] Schäffer, Florian: *Fahrzeugdiagnose mit OBD: OBD I, OBD II sowie KW 1281*. Elektor, 2015. – ISBN 9783895763038

[Sey15] Seyrkammer, Sabine: *Wearable Computing Technology: Potenzielle Einsatzmöglichkeiten in der Industrie*. Diplomica Verlag Gmbh, 2015. – ISBN 9783959348195

[Soo08] Soon, Tan J.: QR code. In: *Synthesis Journal* 2008 (2008), S. 59–78

[SSK+16] Stocker, Alexander ; Spitzer, Michael ; Kaiser, Christian ; Rosenberger, Manfred ; Fellmann, Michael: Datenbrillengestützte Checklisten in der Fahrzeugmontage. In: *Informatik-Spektrum* (2016), S. 1–9

[SSL+08] Sääski, Juha ; Salonen, Tapio ; Liinasuo, Marja ; Pakkanen, Jarkko ; Vanhatalo, Mikko ; Riitahuhta, Asko: Augmented reality efficiency in manufacturing industry: a case study. In: *DS 50: Proceedings of NordDesign 2008 Conference, Tallinn, Estonia, 21.-23.08. 2008*, 2008

[SVG17] *Introduction to Computer Graphics, Section 2.7 – SVG: A Scene Description Language*. http://math.hws.edu/graphicsbook/c2/s7.html. Version: Februar 2017

[Tan14] Tang, Jeff: *Beginning Google Glass Development*. 1st ed. Apress, 2014. – ISBN 9781430267881

[Teg07] Tegtmeier, André: *Augmented Reality als Anwendungstechnologie in der Automobilindustrie*, Otto-von-Guericke-Universität Magdeburg, Universitätsbibliothek, Diss., 2007

[TMM+12] Thompson, Henry ; Mendelsohn, Noah ; Maloney, Murray ; Beech, David ; Sperberg-McQueen, Michael ; Gao, Sandy: W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures / W3C. Version: April 2012. http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/. 2012. – W3C Recommendation

[UH12] Uitz, Iris ; Harnisch, Michael: Der QR-Code – aktuelle Entwicklungen und Anwendungsbereiche. In: *Informatik-Spektrum* 35 (2012), Nr. 5, S. 339–347. http://dx.doi.org/10.1007/s00287-012-0608-5. – DOI 10.1007/s00287–012–0608–5. – ISSN 1432–122X

[UHM11] Uckelmann, Dieter (Hrsg.) ; Harrison, Mark (Hrsg.) ; Michahelles, Florian (Hrsg.): *Architecting the Internet of Things*. 2011. Springer, 2011. – ISBN 9783642191565

[Wan13] Wang, Vanessa: *The Definitive Guide to HTML5 WebSocket (Definitive Guide Apress)*. 1st ed. Apress, 2013. – ISBN 9781430247401

[Wil06] Willers, Dominik: Augmented reality at Airbus. In: *International Symposium on Mixed & Augmented Reality*, 2006

[WLC16] Wei, L. ; Liu, Y. ; Cheung, S. C.: Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, S. 226–237

[WNK06] Witt, H. ; Nicolai, T. ; Kenn, H.: Designing a wearable user interface for hands-free interaction in maintenance applications. In: *Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW'06)*, 2006, S. 4 pp.–655

[WSWL14] Wille, M. ; Scholl, P. M. ; Wischniewski, S. ; Laerhoven, K. V.: Comparing Google Glass with Tablet-PC as Guidance System for Assembling Tasks. In: *2014 11th International Conference on Wearable and Implantable Body Sensor Networks Workshops*, 2014, S. 38–41

[Zen14] Zenger, Ingo: *Die Geschichte der Siemens-Hörsysteme - Ein Rückblick*. https://www.siemens.com/history/pool/newsarchiv/downloads/2014_cc_audiologie_broschuere_dd_final_deu.pdf. Version: 2014

[ZFS+15] Zheng, Xianjun S. ; Foucault, Cedric ; Silva, Patrik Matos d. ; Dasari, Siddharth ; Yang, Tao ; Goose, Stuart: Eye-Wearable Technology for Machine Maintenance: Effects of Display Position and Hands-free Operation. In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. New York, NY, USA : ACM, 2015 (CHI '15). – ISBN 978–1–4503–3145–6, S. 2125–2134

[ZHU15] Ziegler, J. ; Heinze, S. ; Urbas, L.: The potential of smartwatches to support mobile industrial maintenance tasks. In: *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, 2015. – ISSN 1946–0740, S. 1–7

[Zin15] Zinser, Rebecka: Privatautonomie und Datenschutz–Google Glass, Big Data, Web 3.0: Neue Herausforderungen in einer Augmented Reality. In: *Privatautonomie* Nomos Verlagsgesellschaft mbH & Co. KG, 2015, S. 235–244

[ZSF+15] Zheng, Xianjun S. ; Silva, Patrik Matos d. ; Foucault, Cedric ; Dasari, Siddharth ; Yuan, Meng ; Goose, Stuart: Wearable Solution for Industrial

Maintenance. In: *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*. New York, NY, USA : ACM, 2015 (CHI EA '15). – ISBN 978–1–4503–3146–3, S. 311–314

[ZXi16a] *GitHub - journeyapps/zxing-android-embedded: Port of the ZXing Android application as an Android library project, for embedding in an Android application.* `https://github.com/journeyapps/zxing-android-embedded/tree/v3.4.0`. Version: Oktober 2016

[ZXi16b] *GitHub - zxing/zxing: Official ZXing ("Zebra Crossing") project home.* `https://github.com/zxing/zxing/tree/zxing-3.3.0`. Version: September 2016

[ZZA04] Zenati, N. ; Zerhouni, N. ; Achour, K.: Assistance to maintenance in industrial process using an augmented reality system. In: *Industrial Technology, 2004. IEEE ICIT '04. 2004 IEEE International Conference on* Bd. 2, 2004, S. 848–852 Vol. 2

Glossar

Building Block

Ein Programmierbaustein, welcher im PRODIS.Authoring genutzt werden kann, um wiederkehrenden Quellcode zu kapseln. x, 8, 43, 44, 55–61

Gang of Four (GoF)

sind die vier Autoren Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides, welche 1994 ein Buch über wiederverwendbare Entwurfsmuster (engl. Originaltitel: Design Patterns. Elements of Reusable Object-Oriented Software) herausbrachten, welches heute zu den Standardwerken in der Softwareentwicklung zählt [GHJV94]. 16, 89

On-Board-Diagnose (OBD)

ist eine Schnittstelle von einem Fahrzeug, über welche alle Daten zur Fahrzeugdiagnose übertragen werden [Sch15]. 5, 90

PRODIS.Authoring

ist das Autorensystem, mit welchem die Hersteller eigene Diagnoseabläufe mit Hilfe einer grafischen Programmierschnittstelle erstellen können [DSA17a]. vii, ix, x, 3, 6–10, 15, 16, 26, 27, 31, 41–43, 47, 50, 55, 71, 76–78, 87

PRODIS.WTS

ist das Laufzeitsystem, welches im Servicebereich eingesetzt wird und Diagnoseabläufe flexibel ausführen kann [DSA17b]. vii, ix, x, 3, 6–10, 15, 16, 18–22, 24, 26, 30–34, 36, 38, 40–45, 47, 49–52, 54, 55, 57, 58, 62, 65, 67–69, 75–77

Wearable Devices

sind kleine Computer, welche zum Beispiel in Kleidungsstücke integriert sind, oder auch als Accessoire (z.B. als Armbanduhr) getragen werden können [BR14]. Bekannteste Beispiele sind Hörgeräte, Fitnessarmbänder und Smartwatches. v, vii, ix, x, 1–3, 5, 10, 11, 13, 15, 16, 18–32, 34–36, 38, 40–45, 47, 49–54, 56–63, 69, 71–78

Akronyme

ABS

Antiblockiersystem 1, 2, 6, 60, 61, 68–70, 76

AR

Augmented Reality 11, 71–74, 77

CTS

Complex Test Sequence ix, x, 6–10, 15, 16, 18, 22, 24, 26–28, 30, 31, 34, 35, 40–45, 47, 49, 50, 53, 55–61, 63, 65, 66, 68–70, 75–78

FTP

File Transfer Protocol 22

GDK

Glass Development Kit 67

GoF

Gang of Four 16, 89, *Glossar:* GoF

GUI

Grafische Benutzeroberfläche (vom engl. graphical user interface) ix, x, 7, 8, 15, 16, 19, 25–31, 34, 35, 40–43, 45, 49, 50, 56–58, 62, 65, 68, 72, 73, 76–78

HMD

Head-Mounted Display 71–74, 77

HTML

Hypertext Markup Language 37

HTTP

Hypertext Transfer Protocol 22–24, 54

IoT

Internet of Things 20

IP

Internet Protocol 20, 32, 57

JSON

JavaScript Object Notation 37–40, 57, 58, 62, 64, 65, 67, 76

MVC

Model–View–Controller 18, 34

OBD

On-Board-Diagnose 5, 59, 90, *Glossar:* OBD

OID

Object Identifier 31

OSI-Modell

Open Systems Interconnection Model ix, 19, 20, 22

PDA

Personal Digital Assistant 72

POP

Post Office Protocol 22

QR-Code

Quick Response Code 33, 34, 42, 56, 57, 67, 68, 72, 75, 76

SDK

Software Development Kit 67

SMTP

Simple Mail Transfer Protocol 22

SSE

Server Sent Events 23–25, 47, 54

SVG

Scalable Vector Graphics 30, 31, 40, 43, 54, 62, 63, 68, 69, 76, 77

TCP

Transmission Control Protocol 20, 21

UDP

User Datagram Protocol 20, 21

UML

Unified Modeling Language 6

URL

Uniform Resource Locator 48, 49, 51

VR

Virtual Reality 71

W3C

World Wide Web Consortium 37

XHTML

Extensible HyperText Markup Language 37

XLink

XML Linking Language 37

XML

Extensible Markup Language 37–40, 65, 76

XPath

XML Path Language 37, 38

XSD

XML Schema Definition 37

XSLT

Extensible Stylesheet Language Transformations 37

ZXing

Zebra Crossing 57, 67, 76