Entwicklung einer Radioapplikation für ein Embedded Fahrzeug Infotainment System

Zwölfwöchige Abschlussarbeit im Rahmen der Prüfung
im Bachelorstudiengang Elektromobilität
an der Berliner Hochschule für Technik

vorgelegt am: NICHT VERGESSEN

von: Daniel Kurniawan
Matrikelnummer: 936013
1. Betreuer: Prof. Dr. Ing. Sven Graupner
2. Betreuer: Prof. Dr. Ing. Detlef Heinemann

Berliner Hochschule für Technik

Vorwort

Diese Arbeit wäre ohne die Unterstützung einiger Arbeitskollegen sowie Professoren nicht möglich gewesen. Ich möchte mich bei der Firma IAV GmbH für die Gelegenheit und Unterstützung dieser Arbeit bedanken. Ich bedanke mich für die Möglichkeit, davor meine Praxisphase hier absolvieren zu können, woraus viele Ideen für diese Abschlussarbeit einfließen. Außerdem möchte ich mich bei den Arbeitskollegen im Fachbereich TT-U für die Bereitstellung der notwendigen Ressourcen sowie ihre Unterstützung bei Schwierigkeiten bedanken. Zum Schluss bedanke ich mich bei Professor Graupner für die gute Betreuung dieser Arbeit.

Diese Arbeit verwendet das generische Maskulinum, um die Lesbarkeit zu erhalten. Es sind dabei ausdrücklich alle Geschlechteridentitäten mitgemeint.

Inhaltsverzeichnis

Akronyme

ABS	Antilock Braking System
ARM	Advanced RISC Machine
CAN	Controlled Area Network
DAB	Digital Audio Broadcasting
ESP	Electronic Stability Program
FTDI	Future Technology Devices International
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HMI	Human Machine Interface
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
I2C	Inter-Integrated Circuit
IP	Internet Protocol
JSON	JavaScript Object Notation
PKW	Personenkraftwagen
QML	Qt Modelling Language
SDK	Software Development Kit
SPI	Serial Peripheral Interface
SSH	Secure Shell
TCP	Transmission Control Protocol
TMW	Tuner-Middleware
TTP/C	Timed Triggered Protocol
UI	User Interface
UKW	Ultrakurzwelle
UML	Unified Modelling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Loactor
USB	Universal Serial Bus
VHF	Very High Frequency
WWW	World Wide Web

Glossar

ABS	(Antilock Braking System) eine Antiblockliervorrichtung
Backend	Server-Teil der Client-Server Architektur
DAB	(Digital Audio Broadcasting) ein Übertragungsstandard für terrestrischen Empfang von Digitalradio
ESP	**(**Electronic Stability Program) ein elektronisches Fahrassistenzsystem, welches dem Ausbrechen des Fahrzeuges entgegenwirken soll
Flexray	serielles, determistisches und fehlertolerantes Feldbussystem im Automobil
Framework	eine strukturierte Plattform aus vordefinierten Klassen und Funktionen, die Entwicklern als Grundlage zur Erstellung von Softwareanwendungen dient.
Frontend	konsumierende Teile der Client-Server Architektur, z.B. eine graphische Benutzeroberfläche
Gemini	Die neue Infotainment-Plattform von IAV GmbH, welche unter anderem in Landwirtschaftsmaschinen eingesetzt wird.
GUI	(Graphical User Interface) eine Benutzeroberfläche, die es Nutzern ermöglicht, mit elektronischen Geräten wie Computern, Smartphones und Tablets über grafische Symbole und visuelle Indikatoren zu interagieren
HMI	(Human Maschine Interface) ein Dashboard, das Benutzern die Kommunikation mit Maschinen, Computerprogrammen oder Systemen ermöglicht
HTTP	**(**Hypertext Transfer Protocol) ein zustandsloses Protokoll zur Übertragung von Daten auf der Anwendungsschicht über ein Rechnernetz.
IP	(Internet Protocol) ein verbindungsloses Protokoll, das für die Adressierung und das Routing von Datenpaketen über Netzwerke hinweg verantwortlich ist
JSON	(Javascript Object Notation) ein Datenformat in lesbarer Textform für den Datenaustausch zwischen Anwendungen
QML	(Qt Modelling Language) eine deklarative Programmiersprache von QT, zur Erstellung von Benutzeroberflächen in C++
Systemd	eine Sammlung von Programmen, Hintergrundprogrammen und Bibliotheken für Linux-Betriebssysteme
TCP	(Transmission Control Protocol) ein verbindungsorientiertes Protokoll, das zuverlässige Datenübertragungen zwischen Computern in einem Netzwerk ermöglicht.
UI	(User Interface) die Benutzeroberfläche zum Bedienen und Steuern einer Software
URI	(Uniform Resource Identifier) eine Zeichenkette, die zur eindeutigen Identifizierung einer Ressource im Internet dient
WWW	(World Wide Web) ein globales Informationssystem, das über das Internet zugänglich ist

1 Einleitung

In der heutigen Zeit spielen elektronische Systeme eine zentrale Rolle in modernen Kraftfahrzeugen. Diese Systeme dienen nicht nur der Steuerung und Überwachung der Fahrzeugkomponenten, sondern auch der Bedienung durch den Benutzer. Ein wichtiger Bestandteil in modernen Kraftfahrzeugen ist die Mensch-Maschinen-Schnittstelle (HMI), denn die ist heutzutage die zentrale Bedienerschnittstelle zu deren technischen Komponenten.

Diese HMIs müssen daher Anforderungen wie Bedienerfreundlichkeit und Zuverlässigkeit erfüllen, um eine sichere und angenehme Nutzung während der Fahrt zu gewährleisten. Diese Abschlussarbeit geht um die Erstellung einer Radioapplikation für ein eingebettetes Fahrzeug Infotainmentsystem. Die Applikation soll hauptsächlich als Demonstrator auf Automobilmessen dienen.

Die Arbeit wurde bei der Firma IAV GmbH in einer Abteilung des Fachbereichs TT-U durchgeführt, welche unter anderem Instrumententafeln für Landwirtschaftsmaschinen entwickelt. Das eigentliche Tuner-Projekt stammt ursprünglich aus einer anderen Abteilung, welche sich auf Telemetrie in PKWs konzentriert. Dieses Tuner-Projekt besteht insgesamt aus einer Tuner-Hardware, einem Tuner-Service bzw. Middleware und einer Testapplikation für den Benutzer bzw. Entwickler. Am Ende dieser Abschlussarbeit soll eine Radioapplikation für die neue Infotainment-Plattform (IAV-Gemini), basierend auf dem vorhandenen Tuner-Projekt, realisiert werden. Es soll zusätzlich auch untersucht werden, ob eine Portierung der Tuner-Middleware (TMW) auf das IAV-Gemini möglich ist.

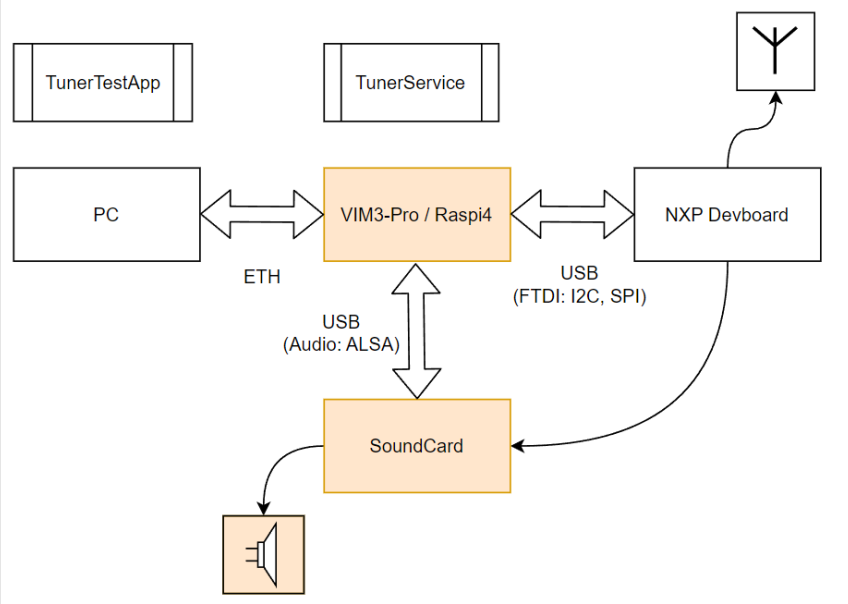

Abbildung 1: Struktur des bisherigen Tuner-Projekts. Firmeninterne Darstellung

Abbildung 1 zeigt die Struktur des existierenden Tuner-Projekts. Die Tuner-Middleware läuft auf einem ARM64 Dev-Board (z.B. Raspberry Pi), welches über USB mit den Peripherien verbunden ist. Zu den Peripherien gehören die Tuner-Hardware selbst und die Sound-Karte. Die Bedienerschnittstelle des Radios wurde bisher über die „Tuner Test App“, welche über Ethernet mit der Tuner-Middleware kommuniziert, realisiert.

1.1 Problemstellung

„Tuner Test App“ ist ein in C# geschriebenes Programm, welches als Schnittstelle zwischen dem Benutzer bzw. Entwickler und dem Tuner dient. Dieses Programm ist bisher nur für die Windows-Desktop Umgebung konzipiert und kommuniziert über HTTP mit der Tuner-Middleware. Das Programm besitzt verschiedene Fenster, welche detaillierte Informationen über das Radio enthalten.

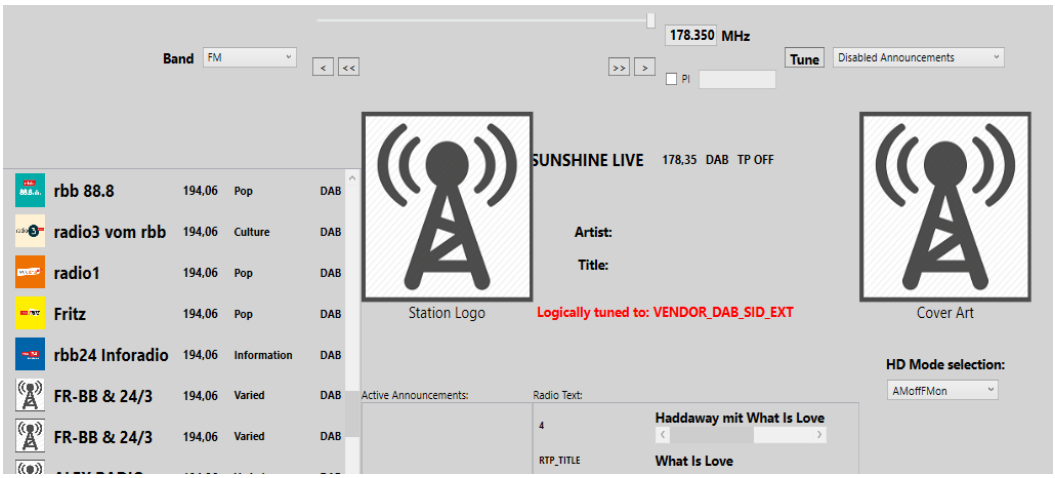

Abbildung 2: Hauptbenutzeroberfläche von „Tuner Test App". Firmeninterne Darstellung

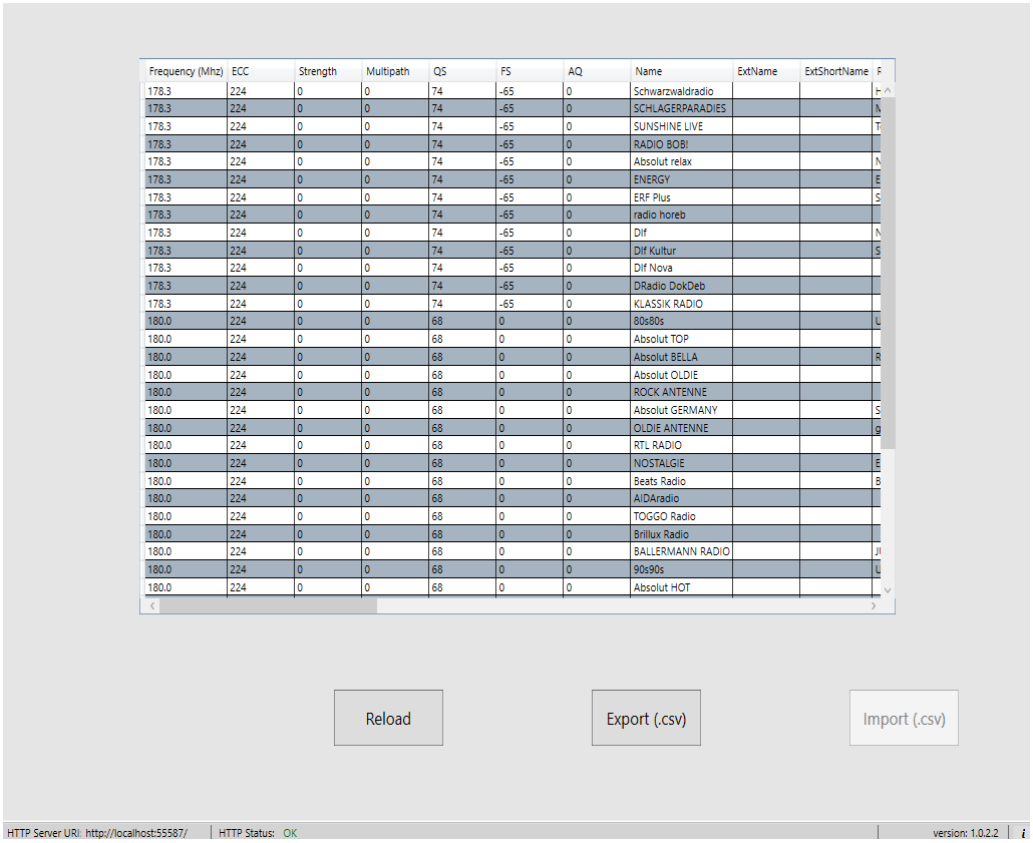

Abbildung 3: DAB-Senderliste auf „Tuner Test App“. Firmeninterne Darstellung

Aus der zuvor genannten Situation treten folgende Probleme hervor:

1. Zweck des Programms

   „Tuner Test App“ ist für die Entwicklung des Radios konzipiert, aus diesem Grund verfügt sie über umfangreiche Funktionen, die nur für Entwickler von Bedeutung sind. Diese umfangreichen Funktionen ermöglichen Entwicklern verschiedene Verhalten der Tuner-Hardware zu beobachten und detaillierte Anpassungen vorzunehmen. Die Benutzeroberfläche des Programms ist zudem mehr funktionsorientiert als schönheitsorientiert, was für den Einsatz auf Messegelände nicht geeignet ist.

2. Kompatibilität mit anderen Plattformen

   „Tuner Test App“ ist bisher nur für die Windows-Desktop Umgebung verfügbar. Der Einsatz dieses Programms ist ausschließlich auf Entwicklungs-Rechnern in Büroräumen gedacht. Dieses Programm lässt sich nur mit großem Aufwand auf ein Infotainmentsystem mit einem Linux-Betriebssystem portieren.

3. Benutzerfreundlichkeit

   Die umfangreichen Funktionen des Programms tragen zur Komplexität des Programms bei, und für den Einsatz auf Messegelände ist nur die Hauptseite des Radios relevant. Außerdem soll die Benutzeroberfläche ein attraktives Design besitzen, um den Besuchern einen positiven Eindruck zu hinterlassen.

4. Komplexität des Systems

   Das System ist bisher über mehrere Peripherien aufgebaut, dies erhöht am Ende die Produktionskosten. Es soll auch zusätzlich untersucht werden, ob eine Hardware-Einsparung für die Tuner-Middleware noch möglich ist.

1.2 Zielsetzung

Es soll nun eine Radio Applikation unter dem Namen „Tuner App“ erstellt werden, welche anschließend auf dem IAV-Gemini implementiert werden soll.

Die Arbeit hat folgende Ziele:

1. Realisierung der Kommunikationsschnittstelle zwischen „Tuner App“ und der Tuner-Middleware
2. Entwicklung einer Qt-Applikation für die Benutzeroberfläche des Radios
3. Implementierung der Applikation auf der Target-Hardware (IAV-Gemini)
4. Integration von „Tuner App“ mit existierenden Applikationen auf dem IAV-Gemini
5. Untersuchung der Portierbarkeit der Tuner-Middleware für das IAV-Gemini

Die Entwicklung von „Tuner App“ lässt sich in folgende Teil-Ziele unterteilen:

1. **Entwicklung des Backends**
   Die Verbindung und Kommunikation zur Tuner-Middleware sind die Grundlegende Funktion dieser Applikation. Die Applikation enthält zusätzlich auch eine Abstraktionsschicht, die sich zwischen dem Backend und dem UI befindet.

2. **Entwicklung des UIs**
   Um das Radio für den Benutzer bedienbar zu machen, soll eine benutzerfreundliche Oberfläche entwickelt werden. Diese Applikation soll als erstes für die Desktop-Umgebung entwickelt werden.

3. **Implementierung der Applikation auf dem Gemini**
   Die Funktionierende Applikation soll auf das Gemini, welches eine ARM64-Architektur besitzt, implementiert werden. In diesem Schritt läuft die Tuner-Middleware noch auf dem Raspberry Pi.

4. **Untersuchung der Portierbarkeit der Tuner-Middleware**
   Als letztes soll optional untersucht werden, ob eine Portierung der Tuner-Middleware auf das IAV-Gemini möglich ist.

1.3 Vorgehensweise

Die vorliegende Arbeit wird in mehrere Kapitel unterteilt:

- **Kapitel 1: Einleitung:** Die Einleitung beschreibt die Problemstellung, Ziele sowie die Motivation dieser Arbeit.

- **Kapitel 2: Grundlagen:** Bei den Grundlagen wird die Theorie über die zu realisierende Arbeit erklärt.

- **Kapitel 3: Anforderungsanalyse:** Eine Analyse über den aktuellen Stand findet im Kapitel 3 statt, daraus werden Anforderungen für die zu entwickelnde Applikation definiert.

- **Kapitel 4: Entwurf:** Im Entwurf wird die Architektur der Applikation sowie die Gestaltung der Benutzeroberfläche beschrieben.

- **Kapitel 5: Entwicklung:** Im Kapitel 5 wird sowohl die Projektstruktur als auch die Implementierung des Programms in Code beschrieben.

- **Kapitel 6: Validierung:** Die im Kapitel 3 definierten Anforderungen werden im Kapitel 6 mit der tatsächlichen Realisierung gegenübergestellt und verifiziert.

- **Kapitel 7: Fazit:** Das letzte Kapitel beinhaltet die Zusammenfassung aller Ergebnisse dieser Arbeit sowie die möglichen Schritte der Weiterentwicklung.

2 Grundlagen

2.1 Eingebettete Systeme in Fahrzeugen

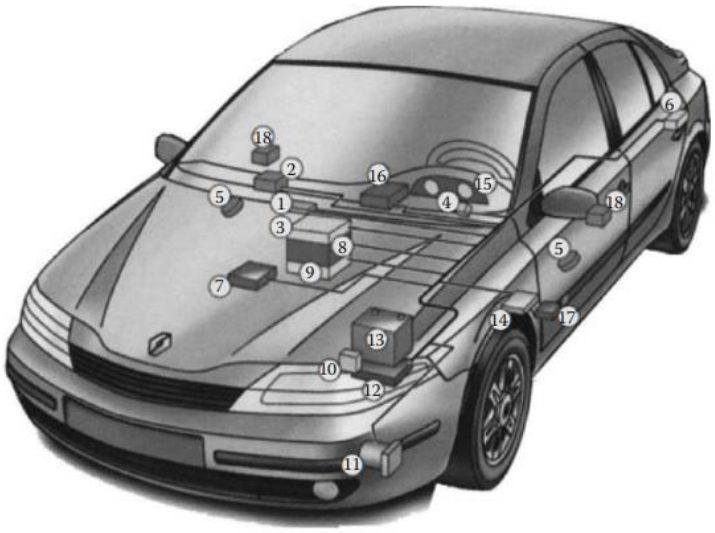

Abbildung 3: eingebettete Systeme im Fahrzeug ***[Nav 08]***

Die Elektroniktechnologie hat große Fortschritte gemacht, daher können sie heutzutage durch ihre Leistung, Robustheit und Zuverlässigkeit für den Einsatz in kritischen Systemen verwendet werden. Ein weiterer technischer Grund für den zunehmenden Einsatz von eingebetteten Systemen im Automobilbereich ist die Tatsache, dass eingebettete Systeme die Einführung von Funktionen, deren Realisierung mit rein mechanischer oder hydraulischer Technologie nicht machbar wäre, erleichtern. Bekannte Beispiele sind das Motorsteuergerät, ABS, ESP, aktives Fahrwerk, usw. In kürze können Automobilkunden dank dieser Technologie sichere, effiziente und personalisierte Fahrzeuge anschaffen. **[Nav08]**

Ein eingebettetes System (embedded system) ist ein binärwertiges digitales System (Computersystem), das in ein umgebendes technisches System eingebettet ist und mit diesem in Wechselwirkung steht. **[Sie12]**

Fahrzeughersteller unterscheiden üblicherweise verschiedene Bereiche für eingebettete Elektronik in deren Fahrzeugen. Es gibt in der Regel fünf Bereiche im Fahrzeug für diese Unterteilung nämlich Antriebsstrang, Fahrwerk, Karosserie, HMI und Telemetrie. Der Bereich Antriebsstrang befasst sich mit Komponenten wie Motor und Getriebe. Der Bereich Fahrwerk befasst sich z.B. mit der Lenkung und Bremsen. Der Bereich Karosserie schließt Komponenten, die nicht zu der Fahrzeugdynamik gehören, ein. Die sind z.B. Beleuchtung und Klimaanlage. HMI schließt die Ausrüstung zum Datenaustausch zwischen dem Fahrer und elektronischen Systemen im Fahrzeug ein. Letztlich ermöglicht Telemetrie den Datenaustausch zwischen dem Fahrzeug und der Außenwelt. Die Telemetrie-Komponenten sind z.B. Radio, Navi sowie Internet-Zugang. Die Anforderung an die eingebetteten Systeme in jedem Fahrzeugbereich ist

unterschiedlich. Die Antriebs- und Fahrwerkbereiche erfordern z.B. eine hohe Rechenleistung, um die Anforderung eines echtzeit-Systems zu erfüllen. **[Nav 08]**

Ein Echtzeitsystem ist ein Computersystem, das in der Lage ist, auf Eingaben oder Ereignisse innerhalb einer festgelegten Zeitspanne zu reagieren. Diese Systeme sind darauf ausgelegt, die Verantwortung für die Kontrolle von Prozessen zu übernehmen und potenziell destruktive Malfunktionen zu detektieren. **[Abb 06]**

Der Fahrzeugbereich Telemetrie andererseits erfordert eine große Menge an Datenaustausch. Aus diesem Standpunkt sind die technologischen Lösungen für jeden Fahrzeugbereich verschieden, weshalb die Methoden für den Entwurf sowie die Funktionsverifizierung dieser Komponenten sehr unterschiedlich sind. **[Nav 08]**

2.2 HMI

*Abbildung 4: Automobil HMI **[Ma 24]**.*

Human Maschine Interface (HMI) im Fahrzeug ist ein System, welches die Übertragung von dynamischen Informationen und Emotionen zwischen Menschen und Fahrzeug ermöglicht, mit Ausnahme der Hauptfahraufgabe **[Ma 24]**.

Die Übertragung von Information ist die grundlegende Funktion einer Automobil-HMI. Dieser Informationsaustausch schließt sowohl Eingabebefehle von dem Fahrer als auch Ausgabeinformationen wie Bild und Ton von dem Fahrzeug ein. Emotionsübertragung ist ein neuer Aspekt einer HMI, diese Interaktion überträgt in der Regel keine Information. Emotionen können in komplexen Formen ausgedrückt werden, die können z.B. das Gefühl von Technologie, Luxus und Komfort sein.

Über die letzten 100 Jahren wurde die Funktion einer HMI zunehmend erweitert. In dem letzten Jahrzehnt ab 2010 hat sich die Automobil-HMI schnell zu einem sehr wichtigen Bestandteil eines Automobils entwickelt. Im Jahr 2012 hat Tesla den Model S vorgestellt, welcher einen zentralen 17-Zoll Bildschirm besitzt. Durch dieses Display wurde viele analogen Knöpfe auf der Mittelkonsole ersetzt. Im Jahr 2016 hat SAIC-Motor den Roewe RX5 vorgestellt, das erste

Serienauto mit Internetverbindung, und somit auch drahtlose Softwareupdates ermöglicht. **[Ma 24]**

Die Entwicklung von Automobil-HMI ist eine Erweiterung von der menschlichen Sinneskanäle. Die Tasten erfordern minimale visuelle und taktile Sinne, und das zentrale Informationsdisplay erhöht das durch den visuellen Kanal übertragene Informationsvolumen, während die Sprachsteuerung den auditiven Kanal nutzt. Diese Entwicklung hat die Interaktionseffizienz verbessert, was das menschenzentrierte Designkonzept verkörpert. Heutzutage besteht eine Automobil-HMI in der Regel aus einem zentralen Display, einer Instrumententafel, Tasten auf der Mittelkonsole, Lenkradtasten und einer Sprachsteuerung. Diese Kombination an Instrumenten ermöglicht das Anzeigen von fast unbegrenzter Menge an Informationen auf einem begrenzten Bereich. **[Ma 24]**

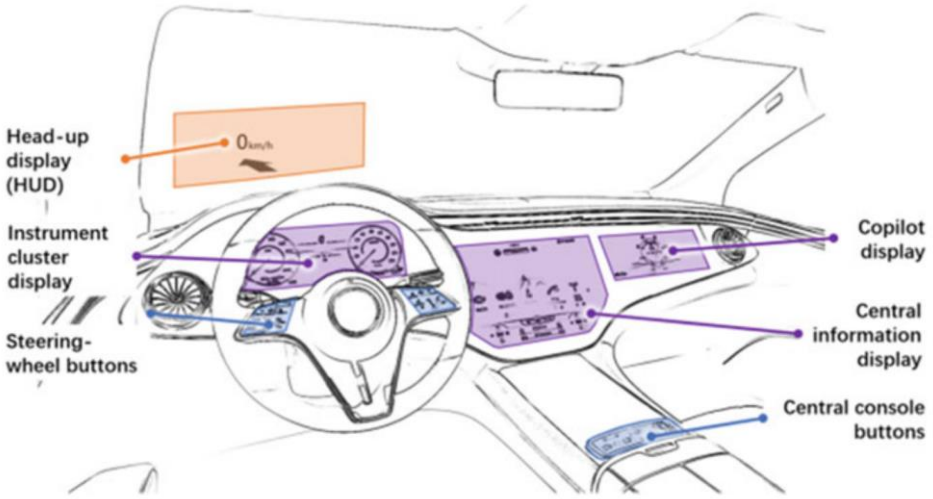

*Abbildung 5: Komponente einer Automobil-HMI **[Ma 24]***

Eine Herausforderung bei der Entwicklung von HMI-Systemen besteht darin, nicht nur die Qualität, Leistung und den Komfort des Systems zu berücksichtigen, sondern auch die Auswirkungen dieser Technologie auf die Sicherheit. In der Tat müssen HMI-Systeme eine einfache und intuitive Nutzung ermöglichen, um den Fahrer während der Fahrt nicht abzulenken. Eine Methode ist die Vermeidung von zu vielen Knöpfen und Tasten. Die Bedienungskommandos sollen so gruppiert werden, dass der Fahrer nur minimal abgelenkt wird. Außerdem soll die angezeigte Information klar und verständlich sein. Multimedia und Telemetrie Geräte werden in Zukunft erweiterbar, und für diesen Bereich ist ein Plug-and-Play-Ansatz vorzuziehen. Applikationen für HMIs sollen für verschiedene Target-Hardware mit unterschiedlichen Betriebssystemen portierbar sein. Eine große Herausforderung besteht darin, die Sicherheit der Informationen vom, zum oder innerhalb des Fahrzeugs zu gewährleisten. Der Fokus im Fahrzeugbereich HMI verschiebt sich von Frist Beschränkungen und Echtzeit auf flüssige Datenströme, Bandbreitenteilung und Multimedia-Qualität des Dienstes. **[Nav 08]**

2.3 Betriebssysteme für eine HMI

Ein Betriebssystem ist ein komplexes Programm, das die Verwaltung von Hardware und Peripheriegeräten, die Bereitstellung einer plattformunabhängigen Nutzung, eine Sammlung von Dienstprogrammen sowie Kommunikationsmechanismen ermöglicht. Es bietet dem Benutzer einen bequemen Zugriff auf die Hardware und dient als Schnittstelle für verschiedene Kommunikationsprotokolle und -dienste. **[Jae20]** Für die Anwendung in HMIs sind einige Betriebssysteme in der Automobilindustrie verbreitet.

2.3.1 Embedded Linux

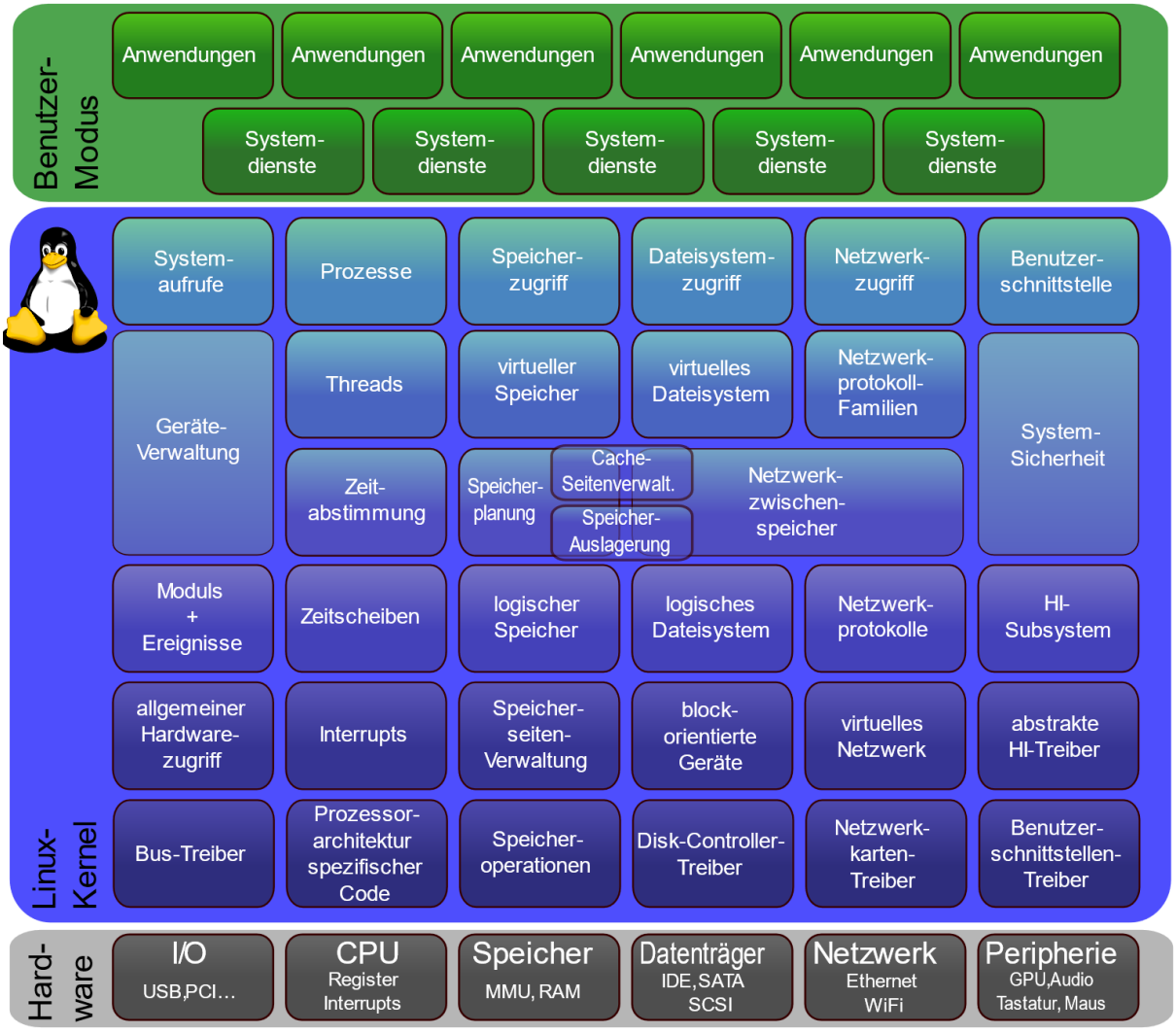

Abbildung 6: Linux Kernel ***[Wik1]***

Linux ist ein freies und Open-Source-Betriebssystem, das ursprünglich von Linus Torvalds im Jahr 1991 entwickelt wurde. Es basiert auf dem Unix-Betriebssystem und wurde ursprünglich als ein universell einsetzbares Betriebssystem entwickelt. Linux besitzt im Vergleich zu den vorherigen Generationen von Unix das open source Code Base. Open Source bedeutet, dass der Quellcode einer Software frei zugänglich ist und von jedem eingesehen, verändert und weiterverbreitet werden kann. Dies ermöglicht eine Anpassung des Betriebssystems für verschiedene Arten von Steuergeräten und Prozessoren mit unterschiedlichen Architekturen.

Die Flexibilität und Anpassungsfähigkeit von Linux, kombiniert mit der starken Unterstützung durch die Open-Source-Community, haben dazu beigetragen, dass es sich als eines der vielseitigsten und am weitesten verbreiteten Betriebssysteme etabliert hat, auch im Bereich von eingebetteten Systemen. **[Abb 06]** Embedded Linux verwendet im Gegensatz zu Linux angepasste Distributionen, die mithilfe von Buildsystemen wie Yocto oder Buildroot gebaut werden. Das Betriebssystem ist speziell für spezifische Hardware und Anwendungen entwickelt. Außerdem kann Embedded Linux mithilfe von Erweiterungen wie PREEMPT-RT Anforderungen eines Echtzeitsystems erfüllen.

2.3.2 Android Automotive

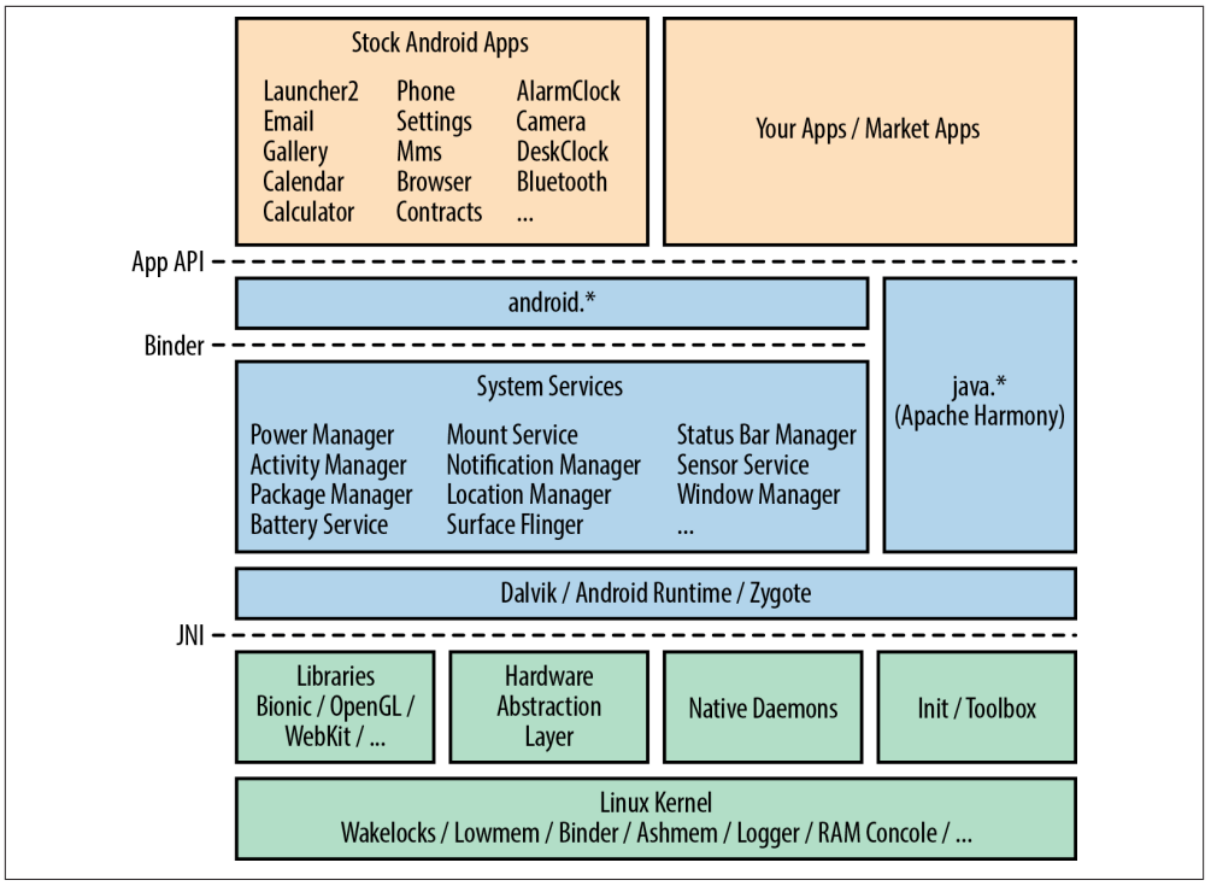

*Abbildung 7: Android Struktur **[Yag13]***

Android Automotive ist eine Variante von Googles Betriebssystem Android, die für die Anwendung in Instrumententafeln von Fahrzeugen angepasst ist. Android Automotive ist eine umfassende, Open-Source und hochgradig anpassbare Plattform, die das Infotainment-System unterstützt. Android Automotive besitzt den Vorteil von Googles APIs wie z.B. Maps und Play Store. Somit ist es möglich, vorhandene Applikationen von Drittanbietern auf dem Google Play Store in die Autos zu bringen. **[Wik2]** In Abbildung 7 lässt sich erkennen, dass Android einen Linux Basierten Kernel mit speziellen Erweiterungen besitzt.

2.3.3 Gegenüberstellung

Embedded Linux besitzt den Vorteil des Opens Source Code Base. Dies ermöglicht eine genaue Anpassung des Quellcodes für die vorliegende Hardware. Da Embedded Linux eine hohe Flexibilität und Anpassungsfähigkeit bietet, erfordert es oft erhebliche Entwicklungsressourcen und Fachwissen, um das System optimal an die spezifische Hardware und die Anforderungen des Projekts anzupassen. Dies kann zu längeren Entwicklungszeiten und höheren Kosten führen. Zudem kann die Open-Source-Natur von Embedded Linux auch Sicherheitsrisiken mit sich bringen. Obwohl die Community kontinuierlich zur Verbesserung und Sicherheit des Systems beiträgt, besteht immer die Möglichkeit, dass Sicherheitslücken oder Schwachstellen übersehen werden. Im Vergleich dazu bietet Android Automotive, ein von Google entwickeltes Betriebssystem, eine spezialisierte und integrierte Lösung für Infotainment-Systeme in Fahrzeugen. Android Automotive basiert auf dem bekannten Android-Betriebssystem und bietet eine benutzerfreundliche Oberfläche, die speziell auf die Bedürfnisse von Fahrern und Passagieren zugeschnitten ist. Es unterstützt eine Vielzahl von Apps aus dem Google Play Store, die speziell für den Einsatz in Fahrzeugen entwickelt wurden, und ermöglicht die nahtlose Integration von Google-Diensten wie Google Maps, Google Assistant und Google Play Music. Dies führt jedoch zu einer Abhängigkeit der Fahrzeughersteller von Google, da sie sich an die Richtlinien und Anforderungen von Google halten müssen, Lizenzgebühren zahlen und auf regelmäßige Updates und Sicherheits-Patches von Google angewiesen sind. Diese Abhängigkeit kann die Flexibilität einschränken und Datenschutzbedenken aufwerfen. Die Wahl zwischen Embedded Linux und Android Automotive hängt daher von den spezifischen Anforderungen des Projekts, den verfügbaren Ressourcen und den gewünschten Funktionen ab. Die Aktuelle Infotainment-Plattform von IAV GmbH ist auf Embedded Linux basiert, währenddessen bleibt Android Automotive noch ein Forschungsthema für zukünftige Infotainment-Plattformen.

2.4 Netzwerkkommunikation im Fahrzeug

Spezielle Kommunikationsprotokolle und Netzwerk wurden für den Gebrauch in Fahrzeug-eingebetteten Systemen entwickelt. Im Jahr 1993 wurden drei Arten von Kommunikationsprotokollen im Fahrzeug definiert, nämlich Klasse A, Klasse B und Klasse C. Netzwerkklasse A bietet eine niedrige Datenrate von unter 10 Kbps und ist für Sensor- und Aktuator-Netzwerke gedacht. Klasse B besitzt eine mittlere Datenrate zwischen 10-500 Kbps und ist für den Datenaustausch im Infotainment, Fahrzeugdiagnose und Karosserieelektronik geeignet. Die Klasse C besitzt eine hohe Datenrate von unter 1 Mbps, diese Klasse ist für sicherheitsrelevante Funktionen mit einer hohen Zuverlässigkeit und Fehlertoleranz entwickelt. Beispiele für diese Kommunikationsklasse sind CAN-C, TTP/C und FlexRay. **[Nav 08]** Heutzutage wird die Verwendung von Internet im Fahrzeug für Telematik- und Infotainmentsysteme oft stattfinden. Auf dieser Basis soll die Kommunikation zwischen der Radioapplikation und der Tuner-Middleware funktionieren.

2.4.1 HTTP-Kommunikation

Hypertext Transfer Protocol (HTTP) ist das Protokoll, das von Webbrowsern, Servern und verwandten Webanwendungen verwendet wird, um miteinander zu kommunizieren. HTTP ist die gemeinsame Sprache des modernen globalen Internets und ermöglicht den Austausch von Informationen und Ressourcen über das Web. **[Gou 02]**

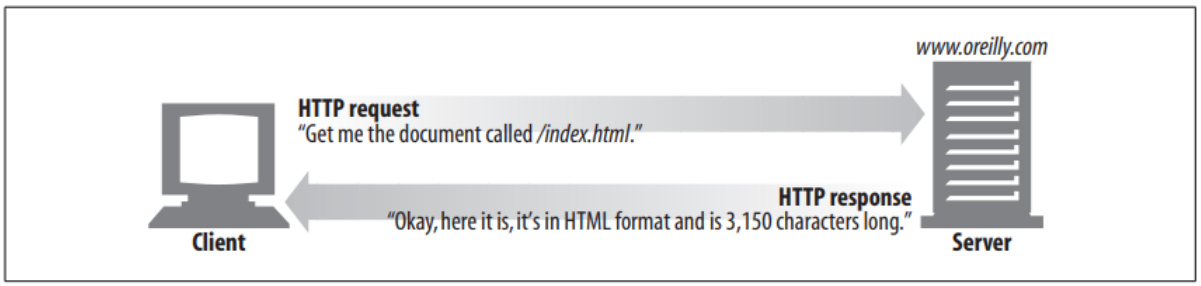

Abbildung 8: Aufbau einer HTTP-Verbindung ***[Gou 02]***

Der Aufbau einer HTTP-Verbindung besteht grundsätzlich aus zwei verschiedenen Teilnehmern, nämlich dem Server und dem Client. Der Server speichert Informationen, die dem Client bei einer Anfrage geteilt wird. Jeder Server besitzt einen Namen, damit jeder Client eine spezifische Anfrage an diesen Namen stellen könnte, dieser Name wird als Uniform Ressource Identifier (URI) bezeichnet. **[Gou 02]**

HTTP unterstützt verschiedene Arten von Kommandos, diese werden als Methoden definiert. Die Methode ermittelt dem Server die auszuführende Aktion.

HTTP method	Description
GET	Send named resource from the server to the client.
PUT	Store data from client into a named server resource.
DELETE	Delete the named resource from a server.
POST	Send client data into a server gateway application.
HEAD	Send just the HTTP headers from the response for the named resource.

Tabelle 1: Http-Methoden ***[Gou 02]***

Jede HTTP-Antwort besitzt wiederum einen Statuscode, dieser besteht aus drei Ziffern. Der Statuscode ermittelt dem Client, ob die Anfrage erfolgt.

HTTP status code	Description
200	OK. Document returned correctly.
302	Redirect. Go someplace else to get the resource.
404	Not Found. Can't find this resource.

Tabelle 2: HTTP-Status Code ***[Gou 02]***

HTTP ist eine Applikationsschicht-Protokoll, das auf dem TCP/IP-Protokollstapel aufbaut und für die Übertragung von Hypertext-Dokumenten im World Wide Web verwendet wird.

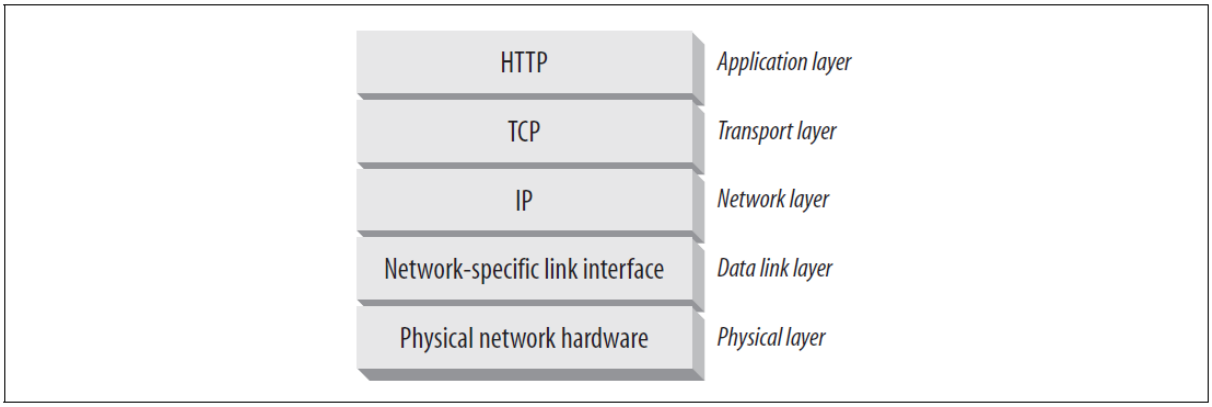

Abbildung 9: HTTP-Verbindungsschicht ***[Gou 02]***

TCP ermöglicht eine zuverlässige Datenübertragung, ihre Zustellung in richtiger Reihenfolge und einen unsegmentierten Datenstrom. Bevor Daten über HTTP übertragen werden können, muss ein TCP/IP Verbindung zwischen dem Server und dem Client über die richtige IP-Adresse und Portnummer hergestellt werden **[Gou 02]**.

2.5 DAB-Radio

Der Begriff Rundfunk wurde erstmals im Jahr 1919 von H. Bredow, dem damaligen Leiter der Abteilung für Funktelegraphie im Reichspostministerium in einem Vortrag geprägt. Die erste Sendung „Unterhaltungs-Rundfunk“ wurde erstmal in Berlin am 29. Oktober 1923 ausgestrahlt. Seit Anfang an richtete sich das Interesse der Techniker auf die Verbesserung der Rundfunkversorgung. Angesichts der physikalischen Eigenschaften der Funkwellenausbreitung und des FM-Übertragungsverfahrens des UKW-Hörfunks sind bei der intensiven Nutzung des Frequenzspektrums prinzipielle Grenzen der Übertragungsqualität erreicht. In den 80er Jahren entwickelte sich ein digitales Hörrundfunksystem, bei dem eine hohe Empfangsqualität auch bei Mehrwegempfang erhalten bleiben sollte **[Lau 04]**.

Digital Audio Broadcasting (DAB) nutzt in Deutschland hauptsächlich das VHF-Band III (174–230 MHz), das einst für Fernsehsender verwendet wurde. Diese Frequenzen wurden in TV-Kanäle 5 bis 12 unterteilt und enthalten Blöcke wie 5A bis 12D. Ferner gibt es das selten genutzte L-Band (1452–1492 MHz) für lokale Übertragungen, aufgrund seiner geringen Reichweite und hohen Sendeleistung **[Wik3]**.

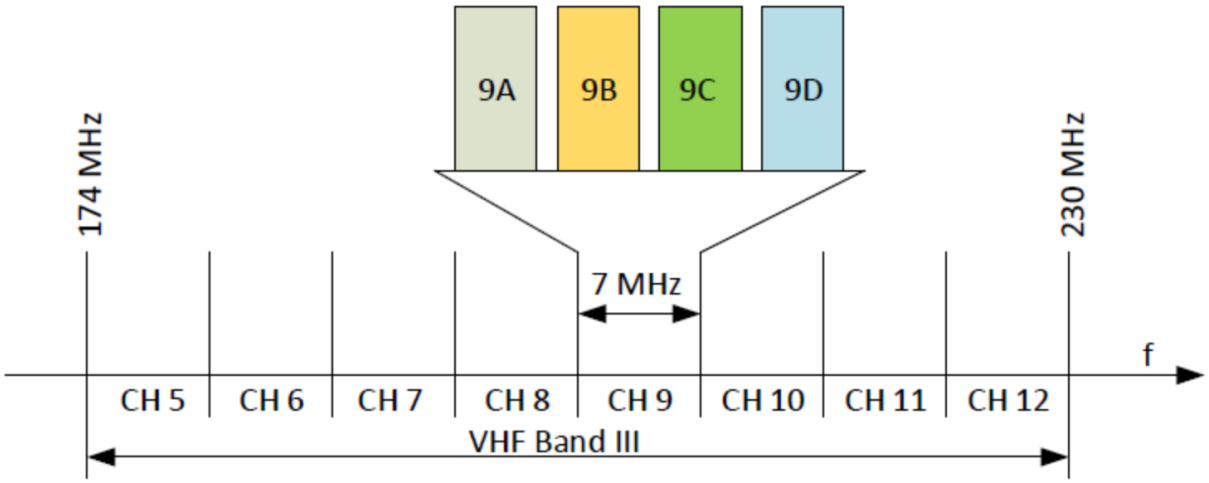

Abbildung 10: DAB-Frequenzbereich (VHF-Band 3) ***[Wiki 1]***

Frequenz [MHz]	Kanal	Standorte
178,352	5C	Berlin-Mitte, Alexanderplatz, Berlin-Charlottenburg, Scholzplatz
180,064	5D	Berlin-Mitte, Alexanderplatz, Berlin-Charlottenburg, Scholzplatz
190,640	7B	Berlin-Mitte, Alexanderplatz
194,064	7D	Berlin-Mitte, Alexanderplatz, Berlin-Charlottenburg, Scholzplatz
211,648	10B	Berlin-Charlottenburg, Scholzplatz, Berlin-Rudow, Zwickauer Damm
229,072	12D	Berlin-Mitte, Alexanderplatz Berlin-Wannsee, Schäferberg

Tabelle 3: DAB-Radiosender Berlin ***[Ber 1]***

2.6 HMI-Framework

Ein Framework ist ein Programmiergerüst, welches in der Softwaretechnik, insbesondere bei der objektorientierten Softwareentwicklung sowie bei komponentenbasierten Entwicklungsansätzen, verwendet wird. **[Wik4]** Frameworks, die sich auf die Erstellung von grafischen Benutzeroberflächen konzentrieren, sind sogenannte GUI-Toolkits. Ein oft verwendeter GUI-Toolkit zur Erstellung von HMI-Benutzeroberflächen ist Qt.

2.6.1 Qt

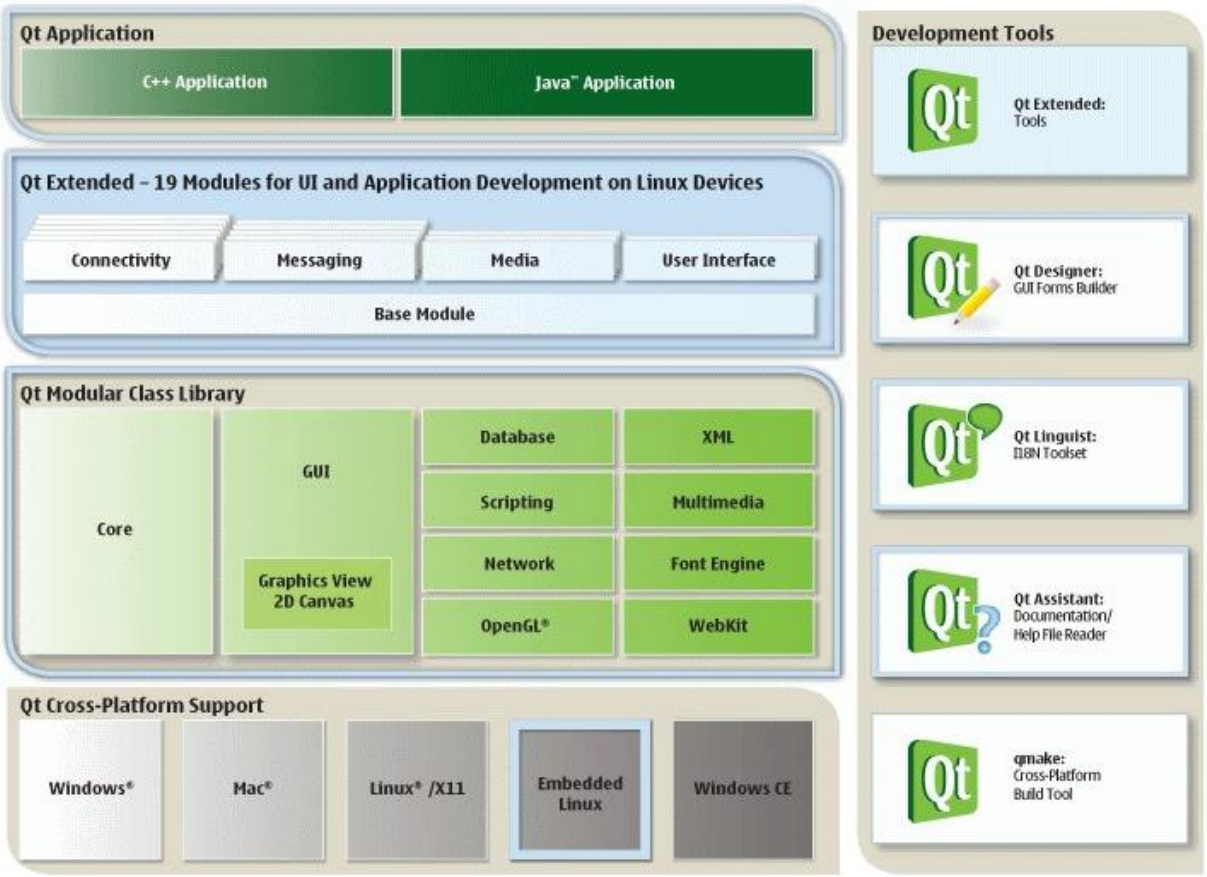

Abbildung 11: QT-Struktur ***[Lin 1]***

Qt ist ein Anwendungsframework und GUI-Toolkit zur plattformübergreifenden Entwicklung von Programmen und grafischen Benutzeroberflächen. Es ermöglicht Entwicklern, Anwendungen zu erstellen, die auf verschiedenen Betriebssystemen wie Windows, MacOS, Linux, Android und Embedded-Systemen laufen, ohne dass der Quellcode wesentlich geändert werden muss. Qt bietet eine umfangreiche Bibliothek von Widgets und Tools, die die Erstellung intuitiver und ansprechender Benutzeroberflächen erleichtern. Darüber hinaus unterstützt es eine Vielzahl von Funktionen wie Netzwerkkommunikation, Datenbankzugriff und Multithreading, was es zu einer vielseitigen Lösung für die Softwareentwicklung macht. **[Wik5]**

Abbildung 12: eine auf Qt erstellte Instrumententafel. Eigene Darstellung

2.6.1.1 QML

Qt Modelling Language (QML) ist eine deklarative Programmiersprache, die als Bestandteil von Qt, einer Bibliothek zur Erstellung von Benutzeroberflächen in C++, entwickelt wurde. Die Sprache dient der Entwicklung von Benutzeroberflächen für Desktop und Mobile Anwendungen. Durch die deklarative Grundstruktur und nahtlose Einbindung von JavaScript vereint sie deklarative und imperative Ansätze in einer Programmiersprache **[Wik6].** Zur Erläuterung dieser Sprache wird folgendes Beispielprogramm für einen Zähler verwendet:

Abbildung 13: Einfaches Zählerprogramm. Eigene Darstellung

Die Benutzeroberfläche des einfachen Zählerprogramms besteht aus einem Text und zwei Knöpfen. Die Definition dieser Benutzeroberfläche in QML wird in Code Beispiel 1 erläutert.

```
Window {
    width: 400 //Breite des Fensters
    height: 280 //Höhe des Fensters
    visible: true //Einblendung des Fensters
    title: qsTr("Simple Counter") //Titel des Fensters

    Text {
        id: count //ID des Elements
        anchors.centerIn: parent //Dieses Textfeld wird Mittig im Window Positioniert
        anchors.verticalCenterOffset: -90 //Dieses Textfeld wird um 90 Pixel nach oben von der Mitte
verschoben
        text: counterInstance.count
        font.pixelSize: 70 //Schriftgröße
    }
    Button {
        id: upButton
        text: "+"
        font.pixelSize: 40
        anchors.centerIn: parent //Der Knopf wird Mittig im Window Positioniert
        anchors.horizontalCenterOffset: 50 //Der Knopf wird um 50 Pixel nach rechts verschoben
        onClicked: {
            counterInstance.setCount(
                        counterInstance.count + 1) //Inkrementierung des Zählerstandes
        }
    }
    Button {
        id: downButton
        text: "-"
        font.pixelSize: 40
        anchors.centerIn: parent //Der Knopf wird Mittig im Window Positioniert
        anchors.horizontalCenterOffset: -50 //Der Knopf wird um 50 Pixel nach Links verschoben
        onClicked: {
            counterInstance.setCount(
                        counterInstance.count - 1) //Dekrementierung des Zählerstandes
        }
    }
}
```

Code Beispiel 1: Einfaches QML-Programm

Zu Beginn jedes QML-Elements steht der Typ dieses Elements, gefolgt von geschweiften Klammern. In den geschweiften Klammern werden die Eigenschaften des Elements in der Form „Name der Eigenschaft: Wert“ beschrieben. Jede QML-Applikation ist auf einem *Window*-Item basiert, dieses dient als das Grundelement des GUIs. Auf diesem *Window* können Objekt-Elemente wie Texte, Knöpfe und Listen frei platziert werden. Diese Elemente können allerdings auch durch *anchors* an das *Parent* Objekt z.B. mittig festgebunden werden. Es kann jedem Element ein Identifier hinzugefügt werden, dieser wird dann verwendet, um eine Verbindung bzw. Beziehung zwischen Elementen herzustellen.

2.6.1.2 Signale und Slots

Die Elemente auf der Benutzeroberfläche lassen sich mit dem Backend Programm über sogenannte Signale und Slots verbinden **[Her 02]**. Das Signal-Slot System in Qt ermöglicht die Kommunikation zwischen Objekten. Es handelt sich um ein ereignisgesteuerter Mechanismus, der es Objekten erlaubt, auf Ereignisse zu reagieren und miteinander zu interagieren, ohne dass sie direkt voneinander wissen müssen.

Signale sind Nachrichten, die von einem Objekt gesendet werden, wenn ein bestimmtes Ereignis eintritt. Sie dienen dazu, andere Objekte über das Ereignis zu informieren.

Slots sind Funktionen, die auf Signale reagieren. Wenn ein Signal emittiert wird, werden alle verbundenen Slots aufgerufen. Slots können als Reaktionen auf Ereignisse dienen und ermöglichen die Ausführung von spezifischem Code, wenn ein Signal empfangen wird.

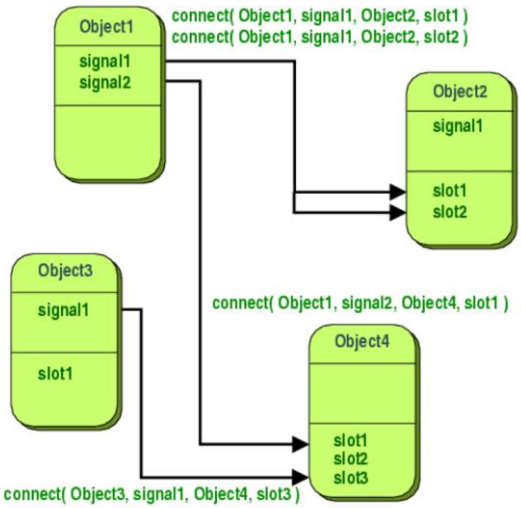

*Abbildung 14:Darstellung von Signalen und Slots **[Qt1]***

Slots können als das Ende von Signalen dargestellt werden, durch das Emittieren des entsprechenden Signals wird die vordefinierte Funktion bzw. Slot automatisch ausgeführt **[Qt1]**. Qt Widgets bzw. Elemente besitzen standardmäßig vordefinierte Signale. Eins davon ist z.B. die `onClicked` Methode eines Knopfes, die bestimmte Funktionen beim Einklicken ausführen soll.

```
onClicked: {
            counterInstance.setCount(
                        console.log("Button Up Pressed")
                        counterInstance.count + 1) //Inkrementierung des Zählerstandes
        }
```

```
Application Output                                    Filter
appSimple_Explanation
QML debugging is enabled. Only use this in a safe environment.
05:51:19: /home/iav/Qt_Projects/build-Simple_Explanation-Desktop-Debug/appSimple_Explanation crashed.

08:36:21: Starting /home/iav/Qt_Projects/build-Simple_Explanation-Desktop-Debug/appSimple_Explanation...
QML debugging is enabled. Only use this in a safe environment.
qml: Button Up Pressed
```

Code Beispiel 2: vordefinierte Signale

3 Anforderungsanalyse

Das vorliegende Kapitel analysiert den bestehenden Prozess für Projektdemonstrationen auf Messen bzw. Akquisen. Basierend auf dieser Analyse wird der Soll-Zustand des Projekts definiert, indem genaue Anforderungen an das System gestellt werden.

3.1 Analyse der aktuellen Prozesse

Nach Beobachtung von mehreren Messe- und Akquise- Vorbereitungen im Fachbereich TT-U ist dem Autor dieser Arbeit der Vorbereitungsprozess bekannt. Um diesen Prozess zu verdeutlichen, wurden folgende UML-Diagramme erstellt.

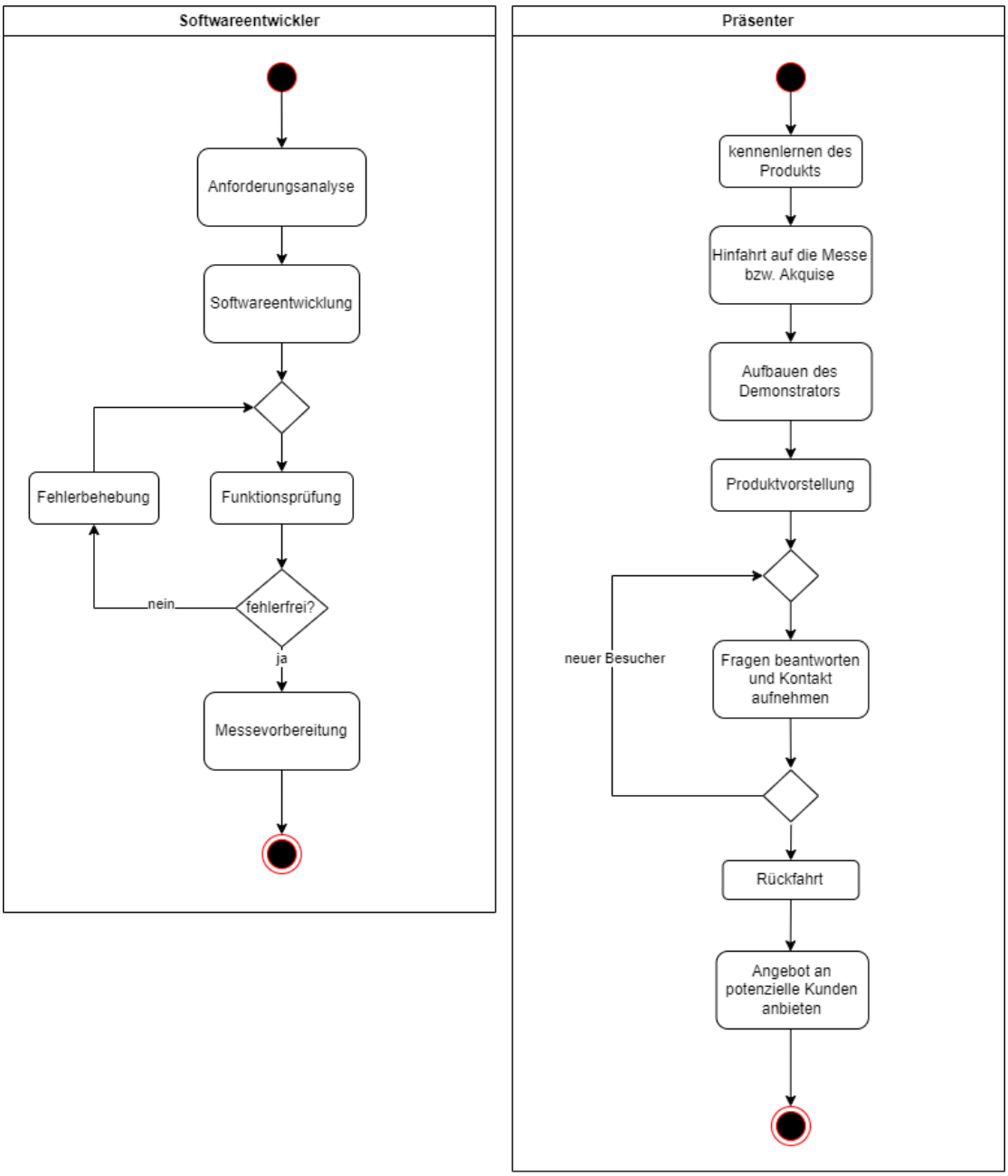

Abbildung 15: UML-Aktivitätsdiagramm für eine Messevorbereitung

Aus Abbildung 14 gehen zwei verschiedene Rollen hervor. Der Softwareentwickler (links) ist für die Bereitstellung der auszustellenden Software zuständig. Seine Aufgabe fängt bei der Analyse der Produktanforderungen an, anschließend soll die eigentliche Software entwickelt werden. Vor der Aufbereitung des Produkts für die Messe soll die Software auf ihre Funktion und nach möglichen Fehlern geprüft werden. Da auf Messen üblicherweise Prototypen vorgestellt werden, soll die Funktions- und Fehlerprüfung sicherstellen, dass das Produkt innerhalb der Messe fehlerfrei vorgestellt werden kann. Der Präsenter auf der Messe (rechts) hingegen stammt üblicherweise aus einer anderen Abteilung. Aus diesem Grund soll der Präsenter alle Funktionen des Produkts erst kennenlernen, um diese vorstellen zu können. Bei einem Produkt aus mehreren Bestandteilen soll der Präsenter den gesamten Aufbau vorbereiten können. Bei der Produktvorstellung auf Messen und Akquisen ist der Präsenter dafür verantwortlich, Fragen von den Besuchern zu beantworten sowie Kontakt mit potenziellen Kunden aufzunehmen. Der Prozess zum Demonstrieren des aktuellen Tuner-Projekts wird mithilfe folgenden UML-Diagramms erläutert.

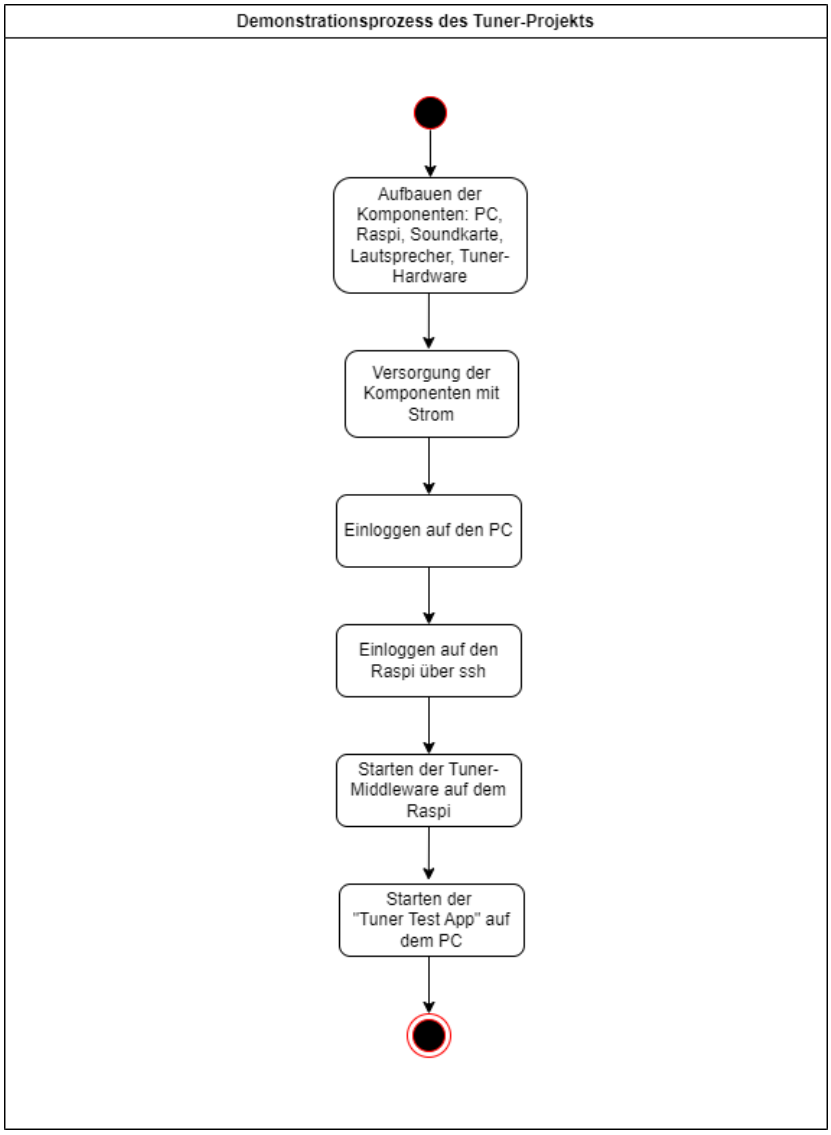

Abbildung 16: UML-Aktivitätsdiagramm zur Demonstration von dem Tuner-Projekt

Der in Abbildung 15 dargestellte Prozess zeigt die Schritte zur aktuellen Projektdemonstration. Als erstes sollen die Komponenten zusammengebracht werden. An das Raspberry Pi werden die Soundkarte und die Tuner-Hardware über USB angeschlossen. Der Lautsprecher wird über ein Klinkenkabel mit der Soundkarte verbunden, anschließend sollen die Komponenten mit Strom versorgt werden. Die Tuner-Hardware wird über ein 12 Volt Netzteil versorgt. Die Tuner-Middleware auf dem Raspberry Pi muss vor „Tuner Test App“ gestartet werden. Zum Starten der Middleware ist das Einloggen über SSH erforderlich. Anschließend kann „Tuner Test App“ über ein Ethernet-Kabel mit dem Raspberry Pi verbunden werden. Dieser Prozess ist bei dem Einsatz auf Messen mit mehreren Auf- und Abbauvorgängen ziemlich aufwendig.

Aus den zuvor genannten Prozessen ergeben sich die nachfolgenden Rollen:

1. **Softwareentwickler**

   Die Rolle der Person, die die auszustellende Software entwickelt und testet.

2. **Präsenter**

   Die Rolle der Person, die das Produkt auf Messen vorstellt, Besucherfragen beantwortet sowie Kundenkontakte aufnimmt.

Sobald das Produkt sich auf dem Messegelände befindet, ergibt sich eine weitere Rolle des Besuchers.

3. **Besucher**

   Die Rolle der Person, die fachlich interessiert ist und potenzielles Interesse am vorgestellten Produkt hat.

Aus den oben genannten Rollen können anschließend Use Cases und User Stories definiert werden, die spezifischen Interaktionen und Anforderungen der jeweiligen Rollen detailliert beschreiben. Aus diesen Use Cases und User Stories kann die Entwicklung und Präsentation des Produkts zielerichtet erfolgen, dass alle relevanten Szenarien abgedeckt sind. Letztlich werden aus den analysierten Use Cases und User Stories alle funktionalen- und nicht-funktionalen- Anforderungen definiert.

3.2 Use Cases

In den folgenden Abbildungen werden die bereits definierten Rollen als Akteure dargestellt. Die Use Cases ergeben sich aus der Prozessanalyse sowie Interviews.

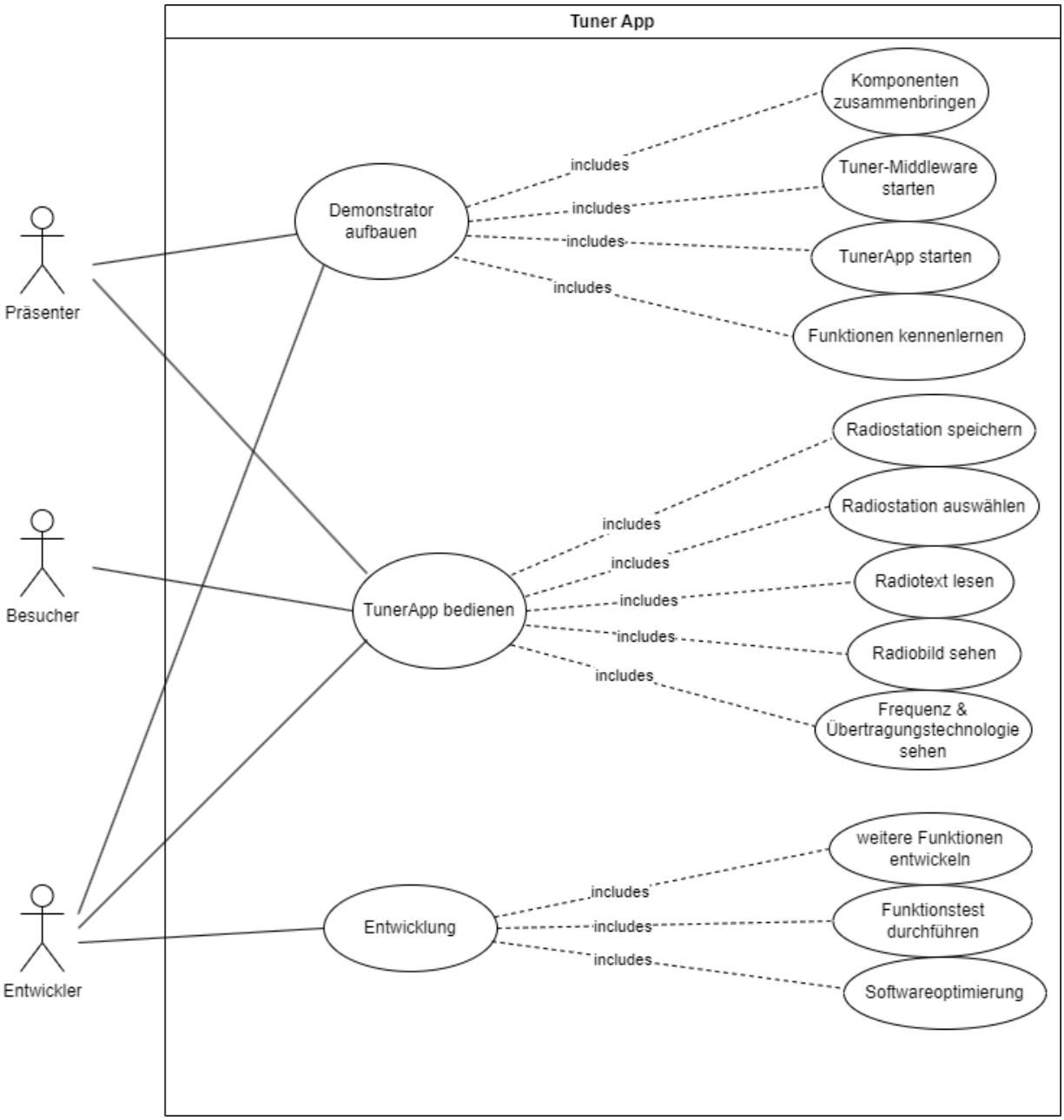

Abbildung 17: UML-Use-Case-Diagramm

3.3 User Stories

Die folgenden User Stories ergeben sich aus den zuvor definierten Use Cases.

US#1. *Als Präsenter möchte ich den Demonstrator mit wenig Aufwand aufbauen können.*

US#2. *Als Präsenter möchte ich die Software mit wenig Aufwand starten können.*

US#3. *Als Präsenter möchte ich das Produkt ohne Probleme vorstellen können.*

US#4. *Als Präsenter möchte ich umfangreiche Fragen von den Besuchern über das Produkt beantworten können.*

US#5. *Als Besucher möchte ich den Ton aus dem Radio hören können.*

US#6. *Als Besucher möchte ich die Senderliste sehen können.*

US#7. *Als Besucher möchte ich die Radiostation wählen können.*

US#8. *Als Besucher möchte ich auf die nächste bzw. vorherige Radiostation wechseln können.*

US#9. *Als Besucher möchte ich den Radiotext lesen können.*

US#10. *Als Besucher möchte ich das Radiobild sehen können.*

US#11. *Als Besucher möchte ich Radiostationen speichern können.*

US#12. *Als Besucher möchte ich die Radiofrequenz sehen können.*

US#13. *Als Besucher möchte ich die Übertragungstechnologie wie DAB/FM sehen können.*

US#14. *Als Besucher möchte ich von dem Präsenter erklärende Antworten auf meine Fragen bekommen.*

US#15. *Als Softwareentwickler möchte ich, dass die Software einfach zu bedienen ist.*

US#16. *Als Softwareentwickler möchte ich, dass die Software für die Messe zuverlässig funktioniert.*

US#17. *Als Softwareentwickler möchte ich, dass der Präsenter die Software gut bedienen und vorstellen kann.*

US#18. *Als Softwareentwickler möchte ich, dass ich den bereits existierenden Code verstehen kann.*

US#19. *Als Softwareentwickler möchte ich, dass ich mit wenig Aufwand weitere Funktionen entwickeln kann.*

US#20. *Als Softwareentwickler möchte ich, dass das Produkt den Besuchern einen positiven Eindruck hinterlässt.*

US#21. *Als Softwareentwickler möchte ich, dass die Software ressourcenschönend funktioniert*

3.4 Funktionale Anforderungen

Um die Funktionen der im Umfang dieser Arbeit entwickelten Software validieren zu können, müssen Anforderungen definiert werden. Die funktionalen Anforderungen werden in zwei folgenden Hauptkategorien unterteilt:

3.4.1 Muss-Kriterien

Die Kriterien, die „Tuner App“ zwingend erfüllen muss:

FA#1. „Tuner App“ muss die HTTP-Verbindung zur Tuner-Middleware herstellen können. **US#5-13**

FA#2. „Tuner App“ muss bei unterbrochener Verbindung die Verbindung wieder aufbauen können. **US#16**

FA#3. „Tuner App“ muss von der Tuner-Middleware HTTP-Anfragen in Form von JSON-Nachrichten schicken können. **US#5-13**

FA#4. „Tuner App“ muss HTTP-Responses in Form von JSON-Nachrichten abfangen können. **US#5-13**

FA#5. „Tuner App“ muss Parameter von der Tuner-Middleware abfragen können. **US#5-13**

FA#6. „Tuner App“ muss Parameter von der Tuner-Middleware setzen können. **US#5-13**

FA#7. „Tuner App“ muss die Information der Radiosender abfragen können. **US#5-13**

FA#8. „Tuner App“ muss die Senderliste anzeigen können. **US#6**

FA#9. „Tuner App“ muss die Senderliste aktualisieren können. **US#6**

FA#10. „Tuner App“ muss nicht-empfangbare Radiosender aus der Liste entfernen können. **US#6**

FA#11. „Tuner App“ muss die Frequenzen der Radiosender anzeigen können. **US#12**

FA#12. „Tuner App“ muss die Übertragungstechnologie des Funks anzeigen können. **US#13**

FA#13. „Tuner App“ muss den aktuellen Text der Radiosender anzeigen können **US#9**

FA#14. „Tuner App“ muss das Stationslogo der Radiosender anzeigen können. **US#10**

FA#15. „Tuner App“ muss das Stationslogo der Radiosender aktualisieren können. **US#10**

FA#16. „Tuner App“ muss das aktuelle Sendungsbild des Radiosenders anzeigen können. **US#10**

FA#17. „Tuner App“ muss das aktuelle Sendungsbild des Radiosenders abrufen können. **US#10**

FA#18. „Tuner App“ muss den Benutzern ermöglichen, eine Radiostation zu wählen **US#7**

FA#19. „Tuner App“ muss den Benutzern ermöglichen, auf das nächste bzw. vorherige Radiostation umzustellen. **US#7**

FA#20. „Tuner App“ muss den Benutzern ermöglichen, Radiostationen zu speichern. **US#11**

FA#21. „Tuner App“ muss den Benutzern ermöglichen, Radiostationen aus der Speicherliste zu laden. **US#11**

FA#22. „Tuner App“ muss auf dem IAV Gemini laufen können. **US#1-2**

3.4.2 Soll-Kriterien

Die Kriterien, die „Tuner App“ noch zusätzlich haben kann.

FA#23. „Tuner App“ soll in die Micro-Service Architektur des IAV-Gemini integriert werden. **US#1-2 & US#20**

FA#24. Die Tuner-Middleware soll für das Gemini portiert werden. **US#1-2**

FA#25. Das Starten von „Tuner App“ und der Tuner-Middleware soll zusammen in den Bootvorgang des Gemini integriert werden. **US#1-2**

3.5 Nicht-funktionale Anforderungen

Es ergeben sich folgende nicht-funktionale Anforderungen aus den Interviews und den User Stories:

NFA#1. Das UI von „Tuner App“ soll ein einheitliches Design mit den anderen Gemini-Applikationen besitzen. **US#20**

NFA#2. Der Demonstrator soll anderen Abteilungen für Messen und Akquisen zur Verfügung gestellt werden. **US#18-19**

NFA#3. Die Entwicklungsumgebung von „Tuner App“ auf dem Dienstrechner soll für andere Entwickler zur Verfügung gestellt werden. **US#19**

NFA#4. Die Schritte zur Einrichtung der Entwicklungsumgebung soll dokumentiert werden. **US#1-2**

NFA#5. „Tuner App“ darf einen Ressourcenverbrauch von 100MB RAM nicht überschreiten. **US#21**

4 Entwurf

In diesem Kapitel soll der Gestaltungsprozess der Radio-Benutzeroberfläche erläutert werden, anschließend wird die Architektur des Backend-Programms genauer beschrieben.

4.1 Entwurf der Benutzeroberfläche

Im folgenden Abschnitt wird das Grunddesign des Radios erläutert. Anschließend werden die einzelnen Radiokomponenten auf dem GUI beschrieben.

4.1.1 Grunddesign

Abbildung 18: Hauptmenü des IAV-Gemini. Firmeninterne Darstellung

Abbildung 18 stellt das Haupt-UI des IAV-Gemini dar. Das Produkt kann aus mehreren Bildschirmen bestehen, die auf dem Armaturenbrett eines Fahrzeugs montiert werden sollen. Die Radioapplikation soll neben der Hauptapplikation auf dem zweiten Display angezeigt werden.

Um das in **NFA#1** angesprochene einheitliche Design umzusetzen, wurde ein Fenster mit einer Auflösung von 1920 x 1080 Pixel (Voll-HD) für das Radio erstellt. Der Hintergrund dieses Fensters wird durch ein einheitliches IAV-Hintergrundbild sowie ein IAV-Logo gefüllt. In diesem Fenster wurde eine durchschichtige Swipe-View erstellt, dieses soll zukünftig die Integration von weiteren Media-Applikationen ermöglichen.

4.1.2 Radio UI

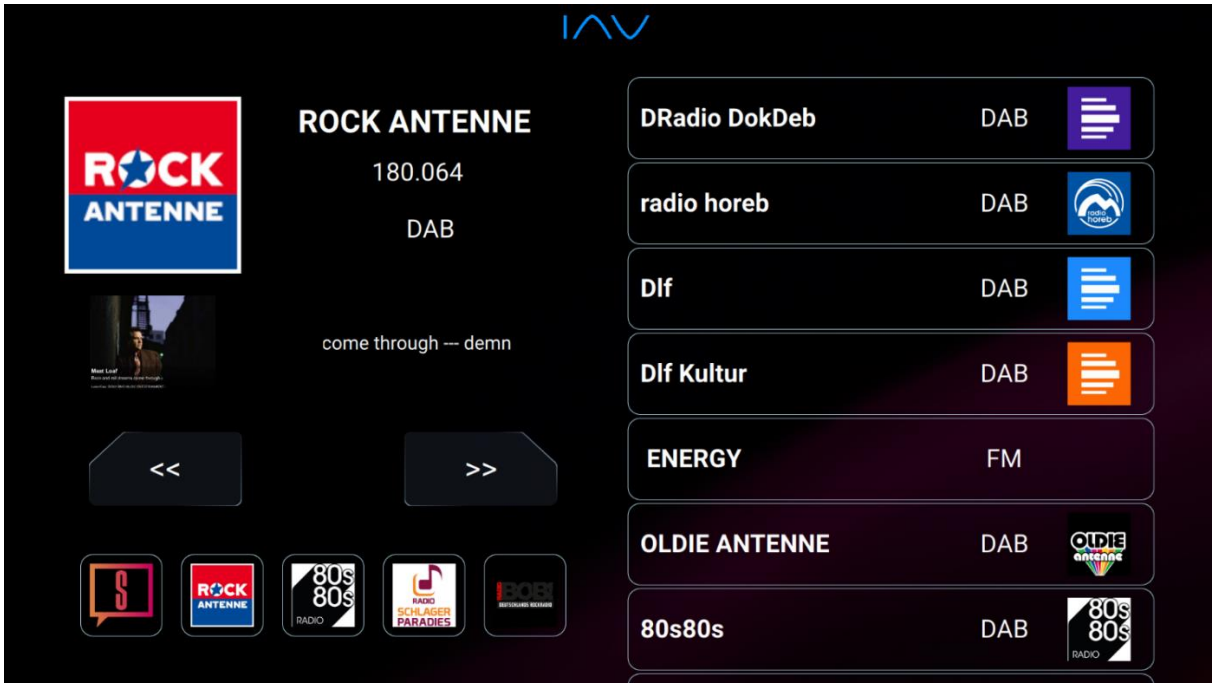

Abbildung 19: UI der Radioapplikation. Eigene Darstellung

Abbildung 19 stellt das Enddesign der Radioapplikation dar. Um die Bedienerfreundlichkeit nach **US#15** zu realisieren, sind die Bedienerschnittstellen wie Knöpfe, Listen und Texte so gestaltet, dass der Fahrer möglichst wenig abgelenkt wird. Die Elemente sind möglichst groß und gruppiert gestaltet.

4.1.3 Radio Senderliste

Abbildung 20: Radiostationsliste. Eigene Darstellung

Die Radiostationsliste soll aktuell alle empfangenen Radiosendungen mit ihrem Stationslogo anzeigen. Es kann passieren, dass das Stationslogo verspätet empfangen wird. In diesem Fall wird das Stationsbild aktualisiert, sobald dieses empfangen wurde. Die Liste zeigt zusätzlich die Übertragungstechnologie jeder Station an. Die Tuner-Middleware unterstützt für manche Radiostationen einen nahtlosen Wechsel zwischen DAB und FM-Übertragung. Diese Funktion bietet den Vorteil, Radio ohne Unterbrechungen zu hören, selbst wenn das Signal von einer der beiden Übertragungstechnologien schwächer wird.

4.1.4 Aktuelle Radioinformation

Abbildung 21: Aktuelle Radioinformation. Eigene Darstellung

Der Bereich für den aktuellen Radiosender stellt ein paar schriftliche und optische Informationen über die aktuelle Radiosendung dar. Es werden Titel und Übertragungsmethode sowie Übertragungsfrequenz angezeigt. Es werden in diesem Bereich zwei Hauptinformationen über die aktuelle Sendung angezeigt. Es wird zum einen der aktuelle Radiotext angezeigt, da dieser manchmal in der Länge wachsen kann, wird dieser Text seitlich automatisch bewegt. Zum anderen wird das aktuelle Sendungsbild angezeigt. Das Textfeld kann zusätzlich Informationen anzeigen, ob die Radiosendung Verkehrsfunk überträgt.

4.1.5 Shortcuts

Abbildung 22: Radio-Shortcuts. Eigene Darstellung

Die Shortcut-Funktionen sollen die Bedienung des Radios während der Fahrt erleichtern. Die zwei Step-Knöpfe wechselt die aktuelle Radiostation auf die nächste bzw. vorherige

vorhandene Station auf der Liste. Die Radioapplikation verfügt zusätzlich über fünf Memory-Knöpfe, um die bevorzugten Radiostationen des Benutzers zu speichern. Durch das Gedruckt-Halten des Memory-Knopfes lässt sich die aktuelle Radiostation speichern. Durch ein kurzes Drücken des Memory-Knopfes lässt sich die Radiostation aus dem Memory laden.

4.2 Entwurf der Softwarearchitektur

Im folgenden Kapitel wird die Architektur der Applikationssoftware sowie die Kommunikation mit der Tuner-Middleware detailliert beschrieben. Im Anschluss wird die Verbindung zwischen dem Backend und UI näher erläutert.

4.2.1 Gesamtsystemarchitektur

Das Endsystem mit Integration der Tuner-Middleware auf dem Gemini ist im Vergleich zum Anfangsstand in der Einleitung um einiges vereinfacht. Es wurden Hardware-Komponenten für die Tuner-Middleware sowie die Sound-Karte gespart. Die aktuelle Systemarchitektur wird durch Abbildung 23 erläutert:

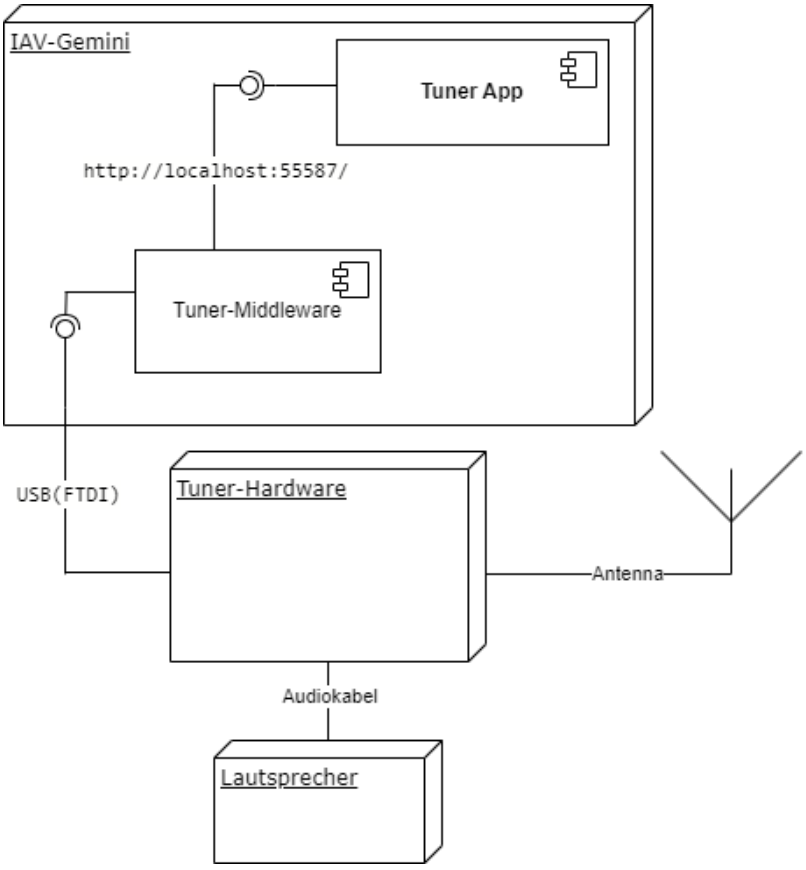

Abbildung 23: UML-Komponentendiagramm zur Systemarchitektur

In Abbildung 23 sind alle Hardware-Komponenten visuell dargestellt. Die Entwicklung in dieser Arbeit schließt die Applikation „Tuner App“ und auch die Portierung der Tuner-Middleware auf das Gemini ein. Durch diese Modifikationen wurde die Systemarchitektur vereinfacht. Dadurch dass der Ton direkt aus der Tuner-Hardware kommt, kann auf die Sound-Karte verzichtet werden. Es wird nun durch die Portierung keine zusätzliche Recheneinheit für die Tuner-Middleware benötigt. Die Netzwerkkommunikation zwischen der Tuner-Middleware und „Tuner App“ erfolgt nun über `Localhost` im Port `55587`.

4.2.2 Tuner App

Die Entwicklung von „Tuner App“ gilt als Hauptteil dieser Abschlussarbeit. Da „Tuner App“ mit der Tuner-Middleware über den vorhandenen Kommunikationsprotokoll kommunizieren soll, muss die auf eine ähnliche Art und Weise wie die „Tuner Test App“ funktionieren. Der Aufbau von „Tuner App“ lässt sich mithilfe folgender Abbildung verdeutlichen.

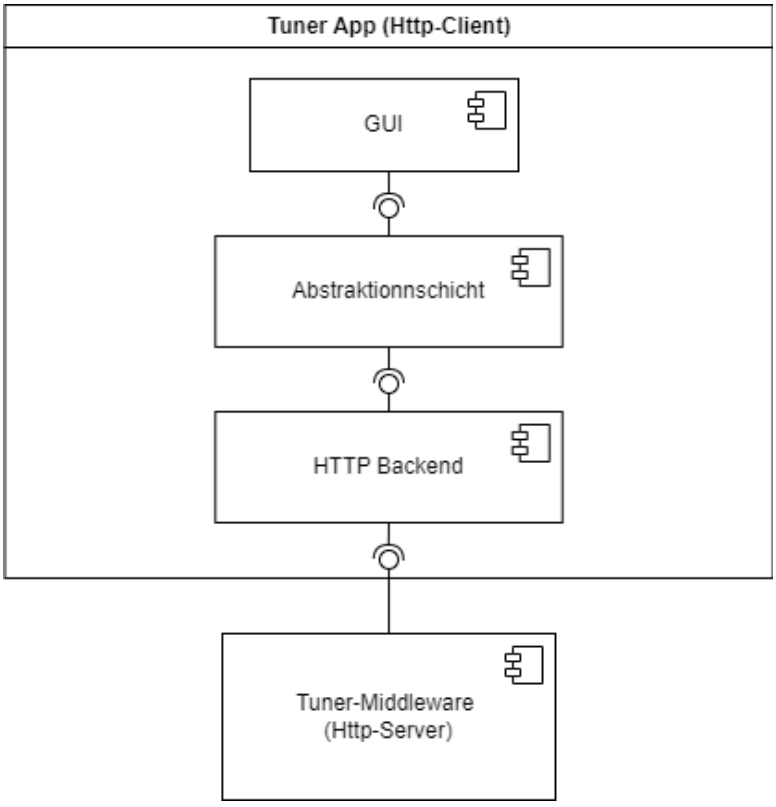

Abbildung 24: UML-Komponentendiagramm zur Applikation-Struktur

In Abbildung 25 sind die Hauptkomponenten von „Tuner App“ visuell dargestellt. „Tuner App“ beinhaltet folgende Komponenten:

1. **GUI**

   Die grafische Benutzeroberfläche, mit der der Benutzer interagieren kann. Die im Abschnitt 4.1 beschriebene Benutzeroberfläche wird durch die Sprache QML realisiert.

2. **Abstraktionsschicht**

   Die Abstraktionsschicht dient als Datenbank für die aktuelle Radiosendung sowie Radiostationen. Diese dient zusätzlich als Zwischenspeicher für die letzt-empfangenen Informationen zwischen HTTP-Nachrichten und dem GUI.

3. **HTTP-Kommunikation**

   Die HTTP-Kommunikation gilt als unterste Schicht dieser Applikation und dient dem Informationsaustausch zwischen „Tuner App“ und der Tuner-Middleware.

4.2.3 Systemablauf von „Tuner App“

4.2.3.1 Initialisierung

Beim Starten des IAV-Gemini wird die Tuner-Middleware sowie „Tuner App“ automatisch gestartet. Beide Programme werden als Systemd Service auf dem IAV-Gemini nacheinander gestartet. Die Tuner-Middleware wird als erstes gestartet, da diese zum Starten ca. 10 Sekunden länger braucht als die „Tuner App“, muss „Tuner App“ warten. Die Initialisierung von „Tuner App“ lässt sich mithilfe folgender Abbildung visuell darstellen:

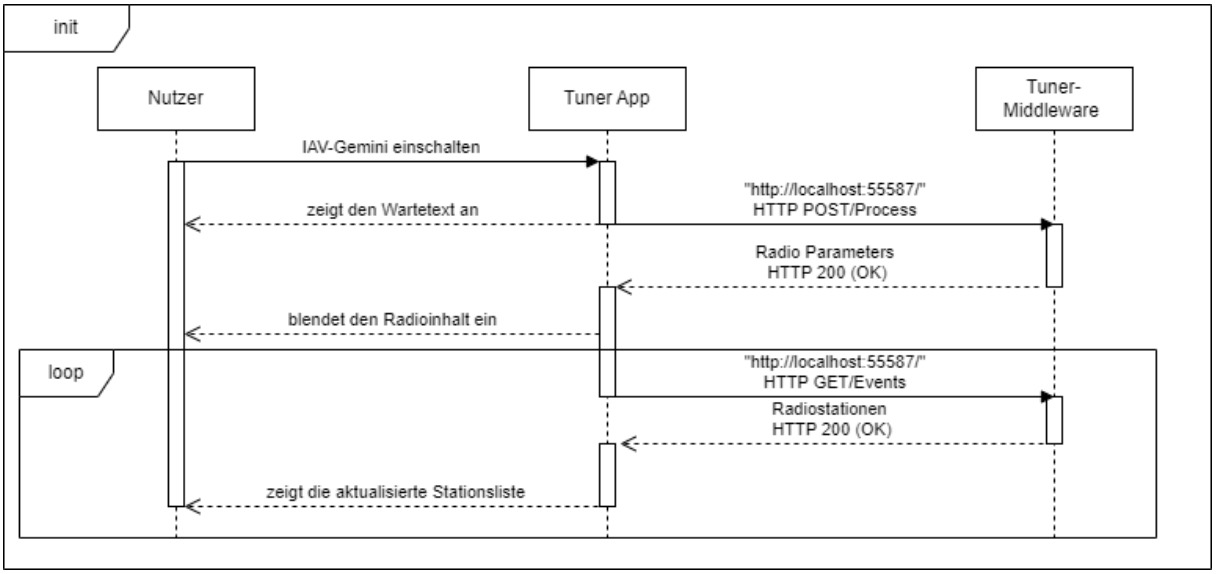

Abbildung 25: UML-Sequenzdiagramm zur Applikations-Initialisierung

Die ersten HTTP-Requests werden so lange wiederholend geschickt, bis eine Antwort empfangen ist. Diese Requests dienen sowohl als Abfrage über die Hardware-Parameter als auch das Zeichen über den Server-Zustand. Solange noch keine Antwort empfangen ist, wird der Radio-Inhalt ausgeblendet und der Wartetext angezeigt. Die späteren HTTP-GET Requests werden im Sekundentakt wiederholend abgeschickt, um aktuelle Radiostationen zu empfangen.

4.2.3.2 Wählen einer Radiostation

Der Vorgang zum Tunen auf eine bestimmte Radiostation lässt sich durch folgende Abbildung visuell darstellen:

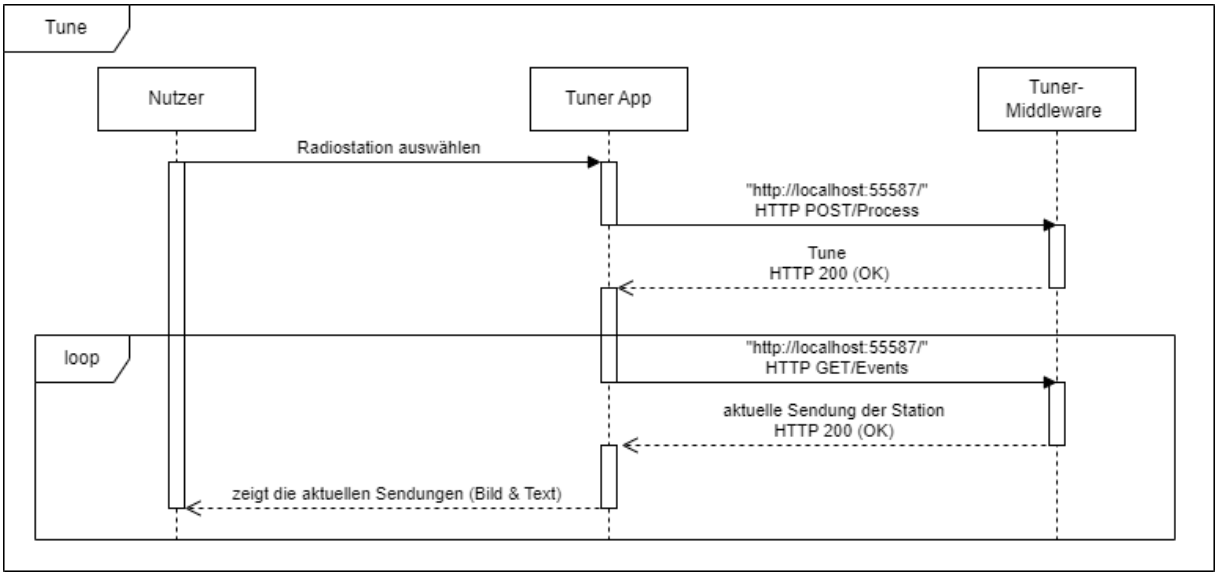

Abbildung 26: UML-Sequenzdiagramm zum Wählen einer Radiostation

Wenn der Benutzer eine Radiostation aus der Liste wählt, wird ein HTTP-Request an die Tuner-Middleware geschickt. Dieser Request beinhaltet IDs, die eine bestimmte Radiostation kennzeichnen. Im folgenden Verlauf werden GET-Requests wiederholend im Sekundentakt geschickt, diese enthalten Abfragen über den aktuellen Inhalt der Radiosendung (Radiotext und Radiobild).

4.2.4 Modellierung einer Radiostation

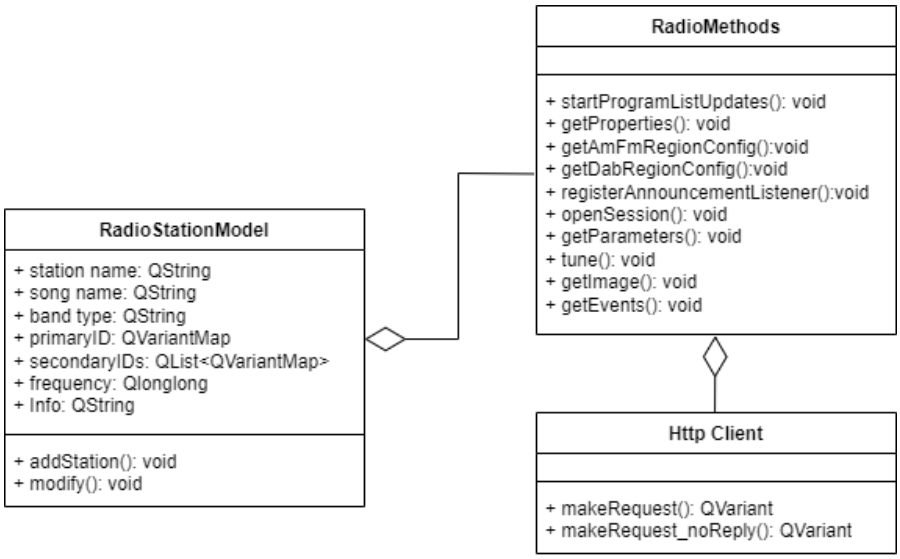

Abbildung 27: UML-Klassendiagramm zur Radiostation-Struktur

Abbildung 28 zeigt die Struktur der Radiostationen. Diese Radiostationen werden als JSON-Arrays von der Tuner-Middleware (HTTP-Server) empfangen. Die empfangenen JSON-Arrays enthalten umfangreiche Informationen zu den jeweiligen Radiostationen, die werden von den Radio Methods-Klasse gefiltert und sortiert.

Die Radio Methods-Klasse verarbeitet nicht nur die empfangenen JSON-Arrays, sondern speichert die auch dementsprechend auf der Abstraktionsschicht über die `addStation` Methode, damit man eine bestimmte Radiostation wählen kann. Diese Klasse bereitet zusätzlich JSON-Nachrichten zur Steuerung der Tuner-Middleware vor, diese ist z.B. die `Tune` Methode.

Zusammenfassend wird bei jeder Radiostation folgende Informationen extrahiert und gespeichert:

Information	Beschreibung
Station Name	Name der Radiostation
Song Name	Aktueller Text der Radiosendung
Band Type	Übertragungstechnologie (FM/DAB)
Frequency	Übertragungsfrequenz
Info	Information über die aktuelle Sendung z.B. Verkehrsfunk
Primary ID	Primary ID zum Tunen
Secondary IDs	Secondary IDs zum Tunen

Tabelle 4: extrahierte Informationen über eine Radiostation

Die gespeicherten Informationen wie Name, Text und Übertragungstechnologie jeder Radiostation werden für das Anzeigen auf der Benutzeroberfläche benötigt. Die Primary und Secondary IDs werden hingegen für das Tunen auf eine bestimmte Radiostation verwendet.

Jede Radiostation wird für 5 Sekunden zwischen gespeichert, falls diese keine weitere Aktualisierung bekommt, wird diese aus der Liste gelöscht. Die empfangenen HTTP-Nachricht enthält zusätzlich eine ID für die Aktuellen Bilder. Diese Bilder werden abgerufen und in binärer Form übertragen. Diese Bilder werden mithilfe der Stationsnamen im lokalen Filesystem gespeichert und mit der passenden Radiostation zugeordnet. Sobald es Aktualisierungen für den Radiotext bzw. Information werden diese über die `modify` Methode an die Radiostationen übertragen. Somit kann die Änderung auf der Benutzeroberfläche erkannt werden.

Die Attribute in `Radio Station Model` und `Radio Methods` sind voneinander getrennt, um Methoden für die Benutzeroberfläche und die Tuner-Middleware zu separieren. `Radio Methods` soll ausschließlich Kommunikations-Nachrichten in Form von JSON über HTTP schicken und empfangen. `Radio Station Model` soll hingegen empfangene Radiostationen sowie deren Aktualisierung für die Benutzeroberfläche bereitstellen.

4.2.5 Modellierung der Abstraktionsschicht

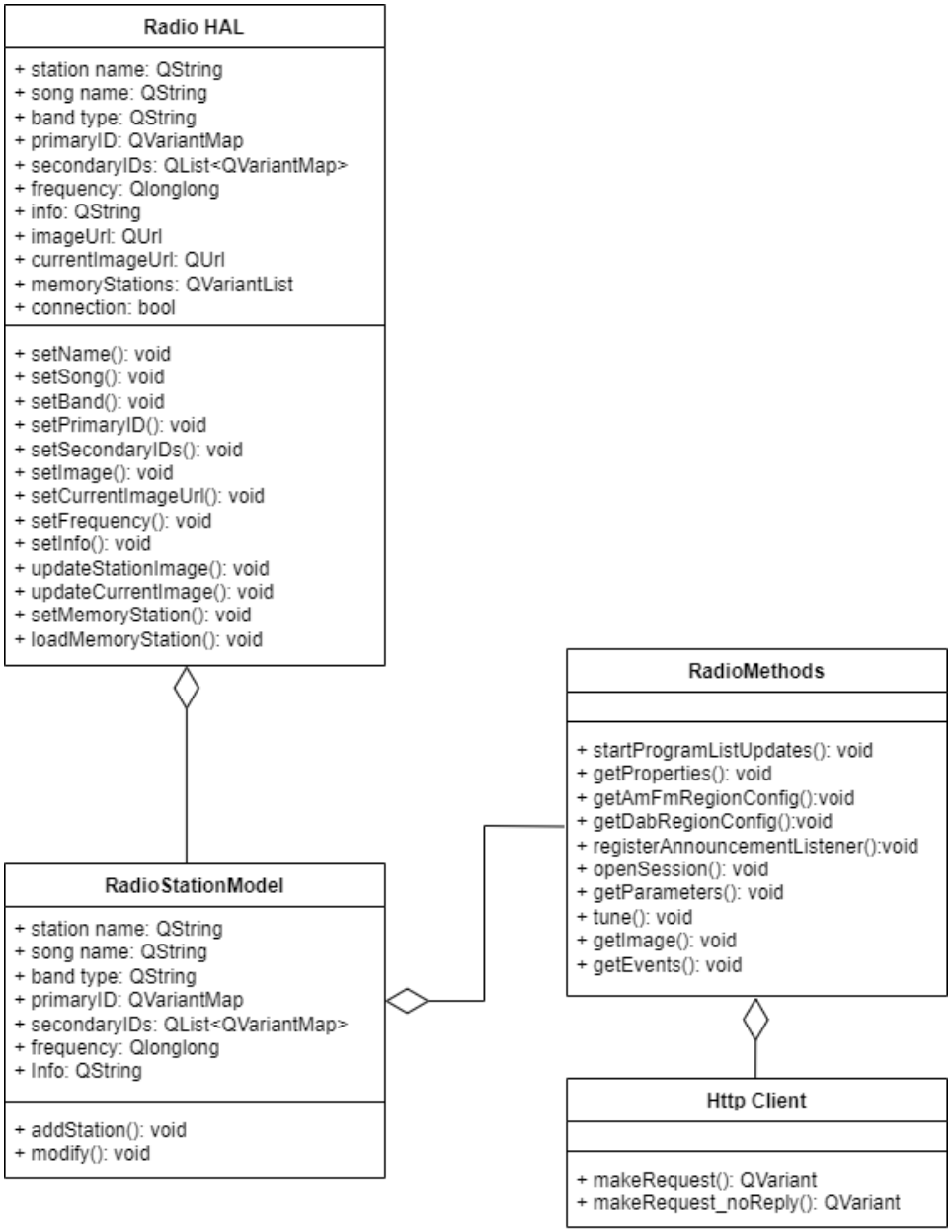

Abbildung 28: UML-Klassendiagramm zur gesamten Abstraktionsschicht

Die Radio Hardware Abstraction Layer (HAL) soll zusammen mit den anderen Klassen eine Abstraktionsschicht zwischen dem Backend und dem UI dienen. Diese Abstraktionsschicht speichert die letzten empfangenen Informationen von dem HTTP-Backend und zeigt sie auf dem UI. Radio HAL besitzt im Vergleich zum Stationsmodel zusätzliche Elemente wie Memory und Radiobilder. Radiobilder werden in binärer Form von der Tuner-Middleware übertragen. Die empfangenen Bilder werden von „Tuner App“ lokal im Filesystem gespeichert. Die Bilder werden mit den Stationsnamen zugeordnet. Es existiert im HAL zusätzlich das Objekt `Connection`, diese enthält die Information über die HTTP-Verbindung des HTTP-Backend. Je nachdem wie die Verbindung ist, wird die HAL entscheiden, ob der Radioinhalt ein- bzw. ausgeblendet werden soll.

5 Entwicklung

In diesem Kapitel wird die Realisierung des in Kapitel 4 besprochenen Entwurfs erläutert. Es wird zunächst der Projektaufbau erklärt, anschließend wird die Realisierung der Software in Code erklärt.

5.1 Projektaufbau

Die gesamte Applikationssoftware für das IAV-Gemini, namens „Demo App Mise“ wurde als ein Repository mit Sub-Modulen realisiert. „Tuner App“ ist ein eigenes Repository und gehört zu den Sub-Modulen von „Demo App Mise“. „Demo App Mise“ ist eine Sammlung von Demonstrationsapplikationen, welche von der Micro-Service Architektur verwaltet werden. Aufgrund von Datenschutz Gründen darf in dieser Arbeit nur „Tuner App“ veröffentlicht werden. Der Aufbau von „Tuner App“ lässt sich in folgende drei Haupt-Verzeichnisse unterteilen:

1. **Bilder**

   Unter diesem Verzeichnis liegen Standardbilder von „Demo App Mise“ wie Logos und Icons, diese sollen für die Realisierung des einheitlichen Designs verwendet werden.

2. **QML**

   Unter diesem Verzeichnis sind Codes in QML-Sprache für die Definition der grafischen Benutzeroberfläche.

3. **Sources**

   Sources enthält sowohl den Quellcode als auch die Header-Dateien für das Backend dieser Radioapplikation.

Außerhalb dieser Haupt-Verzeichnisse befinden sich Konfigurationsdateien für den Build-Prozess. Es sind zum einen die `.qrc` Dateien, diese bettet Ressourcen wie Bilder und QML in das kompilierte Programm ein. Zum anderen befinden sich Build-Scripts wie die `CMakeLists.txt` und `RadioUI.pro` Dateien in dieser Ebene. Diese Build-Scripts definieren welche Dateien gebaut werden müssen, und wie die miteinander gelinkt werden. Während die `CMakeLists.txt` Datei für das *CMake*-Buildsystem spezifisch ist, ist `RadioUI.pro` *QMake* spezifisch.

5.2 Implementierung des UIs

In diesem Abschnitt wird die Implementierung des im Abschnitt 4.1 erläuterten Entwurfs detaillierter erklärt.

5.2.1 Main-View

Die Benutzeroberfläche des Radios wurde mithilfe der Modellierungssprache QML realisiert. Die UI besteht grundsätzlich aus zwei verschiedenen Teilen, dem Main und dem Radio. Der Main ist das Grundfenster dieser Applikation, in diesem Fenster befindet sich eine weitere Swipe View, diese enthält den ganzen Radioinhalt. Durch diese Swipe View könnten zukünftig

weitere Media-Applikationen hinzugefügt werden, zwischen denen durch „Swipen“ gewechselt werden kann. Diese Mediaapplikationen könnten z.B. Spotify oder andere Musik-Player sein. Ein weiterer Vorteil dieser Swipe-View ist die Möglichkeit, den gesamten Radioinhalt während des Wartevorgangs auszublenden.

```
Rectangle {
    width: 1920
    height: 1080
    visible: true
    color: "black"

    Image {
        //Hintergrundbild
        id: backgroundImage
        source: "qrc:/bilder/common/background.png"
        anchors.fill: parent
        fillMode: Image.PreserveAspectCrop
    }
    Image {
        //IAV-Logo
        id: iavLogo
        source: "qrc:/bilder/common/iav-logo.svg"
        scale: 0.2
        anchors.centerIn: parent
        anchors.verticalCenterOffset: -500
        fillMode: Image.PreserveAspectCrop
    }
    Text {
        //Wartetext während des Wartevorgangs
        id: connectionText
        anchors.centerIn: parent
        text: "Verbindung zum Tuner wird aufgebaut..."
        font.pixelSize: 40
        color: "white"
        visible: !radio_HAL.connection //Sichtbar falls keine Verbindung ist
    }
    SwipeView {
        //Radioinhalt
        id: swipeView
        width: parent.width
        height: parent.height
        anchors.centerIn: parent
        visible: radio_HAL.connection //Sichtbar falls Verbindung da ist
        Loader {
            source: "qrc:/qml/radio.qml"
        }
    }
```

Code Beispiel 3: Realisierung der Main-View in QML

Code Beispiel 3 stellt die Definition von der Main-View in QML-Sprache dar. Das Grundfenster besitzt eine Auflösung von 1920x1080 Pixeln (Voll-HD). Mittig im Hauptfenster befindet sich eine Swipe View, diese kann durch verschiedene Inhalte gefüllt werden. In diesem Fall beinhaltet die Swipe-View nur das Radio.

5.2.2 Radio-View

Die Radio-View definiert das im Abschnitt 4.1 erläuterte Design in QML-Sprache. Die Radio-View ist in drei verschiedene Bereiche geteilt. Diese Bereiche sind die dynamische Stationsliste, aktuelle Radioinformation und die Shortcuts.

5.2.2.1 Radio Stationsliste

Die dynamische Radio Stationsliste soll die aktuell empfangenen Radiostationen anzeigen. Diese Radiostationen werden aus dem Radiostationsmodell entnommen. Die Elemente dieser Liste werden bei QML als `Delegate` genannt, die werden als Knöpfe realisiert.

```
ListView {
            id: listView
            width: parent.width / 2.3
            height: parent.height - 100
            anchors.centerIn: parent
            anchors.horizontalCenterOffset: 450
            anchors.verticalCenterOffset: 100
            spacing: 5

            model: radioStationModel //Model für die Radiostation
            delegate: Button {       //Jede Radiostation wird als ein Knopf dargestellt
                id: stationMemoryButton
                width: ListView.view.width
                height: 130

                background: Rectangle {
                    id: buttonBackground
                    width: parent.width
                    height: parent.height
                    color: "transparent"
                    border.color: "lightblue"
                    radius: 20
                }
```

Code Beispiel 4: Dynamische Stationsliste

Durch das Radiostationsmodell können die Radiostationen mit ihren Namen, Übertragungstechnologie und Bild in der Liste angezeigt werden. Beim Klicken auf eine Radiostation wird die gewählte Station in die Abstraktionsschicht geladen und gespeichert.

```
        onClicked: {
                    //Informationen über die aktuelle Radiostation wird
                    radio_HAL.song = model.song
                    radio_HAL.band = model.band
                    radio_HAL.frequency = model.frequency
                    radio_HAL.primaryId = model.primaryId
                    radio_HAL.secondaryIds = model.secondaryIds
                    radio_HAL.info = model.info
                    radio_HAL.name = model.name
                    listView.currentIndex = index //Der Index für die Step-Knöpfe wird aktuallisiert
                }
```

Code Beispiel 5: Wählen einer Radiostation

Informationen wie das Stationslogo wird mithilfe der Stationsname geladen. Falls das Bild von einer Radiostation erfolgreich abgerufen ist, wird das Bild im lokalen Filesystem mit dem Namen von der Radiostation gespeichert. Diese Bilder werden später von der QML-Datei auf das UI geladen.

```
                Text {
                    text: model.band
                    color: "white"
                    font.pixelSize: 40
                    anchors.centerIn: parent

                    anchors.horizontalCenterOffset: 180
                }

                Image {
                    width: 100
                    height: 100
                    source: "file:" + model.name + ".png"
                    anchors.centerIn: parent
                    anchors.horizontalCenterOffset: 330
                }
```

Code Beispiel 6: Stationslogo auf der Senderliste

5.2.2.2 Aktuelle Radioinformation

Das Feld für die aktuellen Radioinformation enthält dynamische Texte und Bilder. Sowohl der Text als auch das Bild wird ständig aktualisiert. Damit das Bild nicht aus dem Cache geladen wird, wurde das QML Property *cache* auf `False` gesetzt.

```
        Image {
            id: currentSongImage
            anchors.centerIn: parent
            anchors.horizontalCenterOffset: -720
            anchors.verticalCenterOffset: 0
            width: 200
            height: 200
            fillMode: Image.PreserveAspectFit
            cache: false
            source: radio_HAL.currentImageUrl.toString()
        }
```

Code Beispiel 7: aktuelle Sendungsbild

Der aktuelle Radiotext wird mithilfe einer Scroll-View realisiert, somit kann eine seitliche Verschiebung des Textes ermöglicht werden. In dieser Scroll-View befindet sich ein *Flickable* element, diese ermöglicht das Scrollen des Inhalts. Jede 50 Millisekunden wird der Text-Inhalt um 1 Pixel nach links verschoben. Sobald das Ende erreicht ist, wird der Text wieder zur Anfangsposition mit einem Delay von 1 Sekunde geschoben.

```
            ScrollView {
                id: view
                width: parent.width
                anchors.fill: parent
                clip: true

                Flickable {
                    id: flickable
                    width: parent.width
                    height: parent.height
                    contentWidth: label.width
                    contentHeight: parent.height
                    interactive: false

                    Label {
                        id: label
                        text: radio_HAL.song
                        font.pixelSize: 30
                        color: "white"
                        anchors.verticalCenter: parent.verticalCenter

                        wrapMode: Text.NoWrap
                    }
                }
            }
```

Code Beispiel 8: scroll Text

5.2.2.3 Shortcuts

Im Feld für die Shortcuts befinden sich fünf Memory Knöpfe, diese werden mithilfe eines *Repeaters* realisiert. Die enthaltene Information über die gespeicherte Station wird durch das Stationslogo gekennzeichnet. Falls die Radiostation kein Logo besitzt, wird dieses durch ein Standard Logo ersetzt.

```
Repeater {
        model: 5
        delegate: Button {
            id: memoryButton
            width: 130
            height: 130
        Image {
                id: memoryButtonImage
                width: 100
                height: 100
                source: "file:" + radio_HAL.memoryStations[index].name + ".png"
                anchors.centerIn: parent

                onStatusChanged: {
                    console.log("Loading image: " + source)
                    if (status == Image.Error) {
                        source = "qrc:/bilder/icons/radio.PNG"
                    }
                }
            }
```

Code Beispiel 9: Memory-Knöpfe

Das Setzen und Laden des Memorys werden mithilfe von zwei verschiedenen Click-Funktionen. Beim gedruckt Halten des Knopfes wird die aktuell gewählte Radiostation gespeichert. Beim einfachen Drücken des Knopfes wird aus dem Memory die Radiostation gewählt.

```
        onClicked: {
                    radio_HAL.loadMemoryStation(index)
                    console.log("Memory " + (index + 1) + " clicked")
                }
                onPressAndHold: {
                    radio_HAL.setMemoryStation(index)
                    console.log("Memory " + (index + 1) + " set")
                    memoryButtonImage.source = "file:"
                            + radio_HAL.memoryStations[index].name + ".png"
                    memoryButtonBackground.border.color = "blue"
                }
```

Code Beispiel 10: Setzen und Laden des Memorys

5.3 Implementierung des Backends

In diesem Abschnitt wird die Realisierung des HTTP-Backends und der Abstraktionsschicht erklärt.

5.3.1 Implementierung des HTTP-Backends

Die Kommunikation zwischen „Tuner App“ und der Tuner-Middleware werden durch synchrone HTTP-Anfragen realisiert. Die synchrone Kommunikation funktioniert auf eine ähnliche Art und Weise wie die von „Tuner Test App“. Durch die synchronen Anfragen ist es möglich, sofortige Antworten vom Server zu empfangen. Die Anfragen werden wiederholend im Sekundentakt geschickt, solange der Server nicht antwortet.

```
QVariant HttpClient::makeRequest(const QJsonObject &json, const QString &endPoint) {

    QVariant responseData;
    if (m_serverStrUri.isEmpty()) {
        return responseData;
    }

    QUrl url(m_serverStrUri + endPoint);
    QNetworkRequest request(url);
    request.setHeader(QNetworkRequest::ContentTypeHeader, "application/json");

    const int retryDelayMs = 1000;
    bool success = false;
    QString lastHttpError="";

    while (!success) {
        QNetworkReply *reply = nullptr;
        QEventLoop eventLoop;

        if (endPoint == "process") {
            QJsonDocument doc(json);
            QByteArray data = doc.toJson();
            reply = m_manager.post(request, data);
        } else if (endPoint == "events") {
            reply = m_manager.get(request);
        } else {
            return responseData;
        }
```

Code Beispiel 11: HTTP-Request

Jede Request wird mithilfe des Endpoints klassifiziert, es gibt Requests ohne und mit zusätzlichen Nachrichten. Das Endpoint bildet zusammen mit der URI den URL. Die URI lautet `http://localhost:55587/`.

```
        connect(reply, &QNetworkReply::finished, &eventLoop, &QEventLoop::quit);
        eventLoop.exec();

if (reply->error() == QNetworkReply::NoError) {
            QByteArray response = reply->readAll();
            if (!response.isEmpty()) {
                QJsonDocument responseDoc = QJsonDocument::fromJson(response);
                responseData = responseDoc.toVariant();
                success = true;
                m_connection = true;
                lastHttpError="";
                emit connectionChanged(m_connection);
                if (m_Callback)
                {
                    m_Callback(m_connection);
                }
            }
        } else {
            QThread::msleep(100);
            if (lastHttpError != reply->errorString()){
                //do not spam: only trace when errorString has been changed
                lastHttpError = reply->errorString();
                qDebug() << "HTTP Error:" << reply->errorString();
            }
            success = false;
            m_connection = false;
```

Code Beispiel 12: HTTP-Error Behandlung

Bei einem Fehler wird die Variable `m_connection` auf `False` gesetzt, dadurch wird der Inhalt des Radios ausgeblendet und der Warte-Text eingeblendet. Der Log Text soll nicht ständig ausgedrückt werden, wenn sich der Status nicht ändert. Durch solche kleinen Details ist eine Einsparung an Rechenleistung möglich.

5.3.2 Radio Methods

Die Radio Methods Klasse soll bestimmte JSON-Nachrichten über das HTTP-Backend für spezifische Methoden durchführen. Diese soll zusätzlich auch empfangene Nachrichten filtern und Sortieren, somit ist das Anzeigen von aktuellen Radioinformationen möglich. Die genaue Nachrichtenstruktur darf allerdings aus Datenschutz Gründen nicht veröffentlicht werden. Grob beschrieben gibt es Methoden zum Abfragen der Hardware-Fähigkeiten und Methoden, die als Kommando für die Hardware dienen. Einige dieser Methoden werden nur einmal beim Starten der Applikation ausgeführt, und manche werden wiederholend geschickt.

```
class RadioMethods : public QObject {
    Q_OBJECT

public:
    explicit RadioMethods(RadioStationModel *model, HttpClient *client, QObject *parent = nullptr);
    void startProgramListUpdates();
    void getProperties();
    void getAmFmRegionConfig(bool full);
    void getDabRegionConfig();
    void registerAnnouncementListener();
    void openSession();
    void getParameters();
    Q_INVOKABLE void tune(const QVariantMap &primaryId, const QList<QVariantMap> &secondaryIds);
    void getImage(int id, const QString name, bool stationImage);
    void getEvents();
```

Code Beispiel 13: Radio Methods

5.3.3 Radiostation Model

Das Radiostationsmodel soll die empfangenen Radiostationen in eine Klasse speichern. Die empfangenen Informationen über die Radiostationen werden von Radio Methods gefiltert und mit den Methoden in Radiostation Model gespeichert.

```
class RadioStationModel : public QAbstractListModel {
    Q_OBJECT
    Q_PROPERTY(int count READ rowCount NOTIFY countChanged)

public:
    static RadioStationModel* instance();
    enum RadioStationRoles {
        NameRole = Qt::UserRole + 1,
        SongRole,
        BandRole,
        PrimaryIdRole,
        SecondaryIdsRole,
        FrequencyRole,
        InfoRole
    };
        explicit RadioStationModel(QObject *parent = nullptr);
        int rowCount(const QModelIndex &parent = QModelIndex()) const override;
        QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const override;
        QHash<int, QByteArray> roleNames() const override;
        void addStation(const RadioStation &station);
        void modify(const QString &name, const QString &newSong, const QString &newBand, const
        qlonglong &newFreq, const QString &newInfo);
```

Code Beispiel 14: Radiostation Model

```
class RadioStation {
public:
    RadioStation(const QString &name, const QString &song, const QString &band, const QVariantMap
&primaryId, const QList<QVariantMap> &secondaryIds, qlonglong frequency, const QString &info)
        : m_name(name), m_song(song), m_band(band), m_primaryId(primaryId),
m_secondaryIds(secondaryIds), m_freq(frequency), m_info(info) {}

    QString name() const { return m_name; }
    QString song() const { return m_song; }
    QString band() const { return m_band; }

    QVariantMap primaryId() const { return m_primaryId;}
    QList<QVariantMap> secondaryIds() const { return m_secondaryIds;}

    qlonglong frequency() const {return m_freq;}
    QString info() const {return m_info; }
```

Code Beispiel 15: Radiostation Klasse

Die Methode zum Hinzufügen einer Radiostation namens `addStation` nimmt ein Parameter von Typ `RadioStation`, diese ist wiederum eine Klasse.

```
void RadioStationModel::addStation(const RadioStation &station) {
    if (stationExists(station)) {
        resetStationTimeout(station.name());
        return;
    }
    RadioStation newStation = station;
    beginInsertRows(QModelIndex(), rowCount(), rowCount());
    m_stations << newStation;
    endInsertRows();
    emit countChanged();
    startStationTimeout(station.name());
}
```

Code Beispiel 16: `addStation` Methode

Beim Hinzufügen einer Radiostation durch die Methode `addStation` wird es zunächst geprüft, ob eine Station mit dem gleichen Namen bereits vorhanden ist. Danach wird die Station hinzugefügt und durch das Emittieren des Signals `countChanged` wird die Liste auf dem UI aktualisiert. Zusätzlich wird ein Timer gestartet, dieser soll die Radiostation nach einer Zeit von 5 Sekunden aus der Liste wieder löschen. Falls die Station bereits existiert, gilt dieser Aufruf als Aktualisierung und startet den Timer wieder neu mit einer Dauer von 5 Sekunden.

```
void RadioStationModel::startStationTimeout(const QString &stationName) {
    removeStationAfterTimeout(stationName, 5000);
}

void RadioStationModel::removeStationAfterTimeout(const QString &stationName, int timeout) {
    QTimer *timer = new QTimer(this);
    timer->setSingleShot(true);
    connect(timer, &QTimer::timeout, this, [this, stationName, timer]() {
        for (int i = 0; i < m_stations.size(); ++i) {
            if (m_stations[i].name() == stationName) {
                beginRemoveRows(QModelIndex(), i, i);
                m_stations.removeAt(i);
                endRemoveRows();
                emit parameterChanged(stationName, ""," ", 0, "MUTED");
                emit countChanged();
                break;
            }
        }
        timer->deleteLater();
        m_stationTimers.remove(stationName);
    });
    timer->start(timeout);
    m_stationTimers[stationName] = timer;
}
```

Code Beispiel 18: Löschen einer Radiostation

```
void RadioStationModel::modify(const QString &name,const QString &newSong,const QString &newBand,
                               const qlonglong &newFreq,const QString &newInfo) {
    for (int i = 0; i < m_stations.size(); ++i) {
        if (m_stations[i].name() == name) {
            m_stations[i].setSong(newSong);
            m_stations[i].setBand(newBand);
            m_stations[i].setFreq(newFreq);
            m_stations[i].setInfo(newInfo);
            QModelIndex topLeft = index(i);
            QModelIndex bottomRight = index(i);
            emit dataChanged(topLeft, bottomRight, {SongRole});
            emit parameterChanged(name, newSong, newBand, newFreq, newInfo);
            break;
        }
    }
}
```

Code Beispiel 17: Modifizieren einer Radiostation

Sobald die aktuell gewählte Radiostation aus der Liste genommen ist, wird der Status „MUTED“ auf dem UI angezeigt. Wenn die Station bereits existiert und aktualisiert wird, wird

der Timer zurückgesetzt, somit wird die Station nicht aus der Liste genommen. Mithilfe der `modify` Methode lässt sich aktuelle Information von der Radiosendung wie Text ändern. Nach dieser Änderung wird das Signal `parameterChanged` emittiert, dieses aktualisiert die Benutzeroberfläche.

5.3.4 Radio HAL

Radio HAL gilt als die oberste Schicht der gesamten Abstraktionsschicht. Radio HAL enthält den aktuellen Inhalt der Radiosendungen. Radio HAL wurde als eine Klasse mit *Q_Properties* definiert, diese sind von Qt definierte Properties mit vordefinierten Signalen und Slots. Durch diese Properties können Backend und Benutzeroberfläche zusammenarbeiten.

```
class Radio_HAL : public QObject{
    Q_OBJECT
    Q_PROPERTY(QString song READ song WRITE setSong NOTIFY songChanged)
    Q_PROPERTY(QString band READ band WRITE setBand NOTIFY bandChanged)
    Q_PROPERTY(QString name READ name WRITE setName NOTIFY nameChanged)
    Q_PROPERTY(QVariantMap primaryId READ primaryId WRITE setPrimaryId NOTIFY primaryIdChanged)
    Q_PROPERTY(QList<QVariantMap> secondaryIds READ secondaryIds WRITE setSecondaryIds NOTIFY
            secondaryIdsChanged)
    Q_PROPERTY(QUrl imageUrl READ imageUrl NOTIFY imageChanged)
    Q_PROPERTY(QUrl currentImageUrl READ currentImageUrl NOTIFY currentImageUrlChanged)
    Q_PROPERTY(qlonglong frequency READ frequency WRITE setFrequency NOTIFY frequencyChanged)
    Q_PROPERTY(QString info READ info WRITE setInfo NOTIFY infoChanged)
    Q_PROPERTY(QVariantList memoryStations READ memoryStations NOTIFY memoryStationsChanged)
    Q_PROPERTY(bool connection READ connection NOTIFY connectionChanged)
```

Code Beispiel 19: Informationen über die aktuelle Radiosendung

Code Beispiel 19 zeigt die *Q_Properties* für die aktuell gespielte Radiosendung. Zu jedem Property gehören Funktionen zum Setzen und Lesen dieses Properties. Es gibt zusätzlich Signale, die zum Aktualisieren der UI emittiert werden.

```
signals:
    void songChanged();
    void bandChanged();
    void nameChanged();
    void primaryIdChanged();
    void secondaryIdsChanged();
    void imageChanged();
    void currentImageUrlChanged();
    void frequencyChanged();
    void infoChanged();
    void memoryStationsChanged();
    void connectionChanged();
```

Code Beispiel 20: Signale für die Aktualisierung der UI

In dieser Ebene gibt es zusätzlich mithilfe von *Q_Settings* die Möglichkeit, den letzten Stand des Programms zu speichern. Somit kann z.B. die zuletzt gewählte Radiostation bei einem Neustart wieder gespielt werden. Außerdem werden die fünf gespeicherten Radiostationen bei einem Reboot auch nicht verschwinden.

```
void Radio_HAL::saveSettings() {
    m_settings.setValue("song", m_currentStationSong);
    m_settings.setValue("band", m_currentStationBand);
    m_settings.setValue("name", m_currentStationName);
    m_settings.setValue("primaryId", m_primaryId);
    QVariantList secondaryIdsList;
        for (const QVariantMap &map : m_secondaryIds) {
            secondaryIdsList.append(map);
        }
    m_settings.setValue("secondaryIds", secondaryIdsList);
    m_settings.setValue("image", m_currentStationImageUrl);
    m_settings.setValue("frequency", m_freq);
    m_settings.setValue("info", m_info);


    for (int i = 0; i < m_memoryStations.size(); ++i) {
        m_settings.setValue(QString("memoryStation%1/name").arg(i), m_memoryStations[i].name);
        m_settings.setValue(QString("memoryStation%1/frequency").arg(i),
m_memoryStations[i].frequency);
        m_settings.setValue(QString("memoryStation%1/song").arg(i), m_memoryStations[i].song);
        m_settings.setValue(QString("memoryStation%1/band").arg(i), m_memoryStations[i].band);
        m_settings.setValue(QString("memoryStation%1/primaryId").arg(i),
m_memoryStations[i].primaryId);


        QVariantList variantList;
        for (const QVariantMap &map : m_memoryStations[i].secondaryIds) {
            variantList.append(map);
        }
    m_settings.setValue(QString("memoryStation%1/secondaryIds").arg(i), variantList);
    }
}
```

Code Beispiel 21: Speichern des letzten Radiozustands

5.4 Bereitstellung

Um die „Tuner App“ und die Tuner-Middleware auf der Hardware laufenzulassen, muss der Quellcode von beiden Programmen mit dem entsprechenden SDK kompiliert werden. „Tuner App“ wird als Service in Abhängigkeit von der Tuner-Middleware beim Bootvorgang des Betriebssystems gestartet. Die Tuner-Hardware wird direkt über USB an das IAV-Gemini angeschlossen, davor muss allerdings der passende FTDI-Treiber auf dem Target installiert sein. Dieser Treiber ermöglicht die Kommunikation über SPI und I2C über die USB-Schnittstelle. Dieser Abschnitt darf allerdings aus Datenschutzgründen nur so abstrakt beschrieben werden.

6 Validierung

Dieses Kapitel stellt das Ergebnis dieser Arbeit vor. Anschließend werden die in Kapitel 3.5 und 3.6 definierten Anforderungen verifiziert.

6.1 Ergebnis

Die Applikation „Tuner App“ konnte erfolgreich umgesetzt und die im Abschnitt 1.1 definierten Probleme somit gelöst werden. Die Tuner-Middleware konnte zusätzlich für das Target portiert werden. Zur Demonstration der realisierten Funktionen wurden Videos im Anhang aufgenommen.

Abbildung 29: Das finale Design. eigene Darstellung

Abbildung 30: Implementierung auf dem Target

6.2 Verifizierung der Anforderungen

Die Erfüllung der Anforderungen wird anhand der in dieser Arbeit angehängten Videos validiert. In der Spalte „Zeitstempel“ der folgenden Tabellen lassen sich die Zeitabschnitte des Videos wiederfinden, in welchen die Erfüllung einer spezifischen Anforderung gezeigt wird.

6.2.1 Verifizierung der funktionalen Anforderungen.

In Tabelle 5 werden die Muss-Anforderungen aufgelistet und anhand der Videos validiert.

Alle Muss-Anforderungen wurden erfüllt.

Anforderung	Validierung	Zeitstempel
FA#1 HTTP-Verbindung	Erfüllt	Software.mkv 00:10-00:20
FA#2 Verbindungswiederherstellung	Erfüllt	Software.mkv 00:10-00:20
FA#3 HTTP-Anfragen	Erfüllt z.B. bei dem Tune Vorgang	Software.mkv 00:42-00:45
FA#4 Empfangen von Nachrichten	Erfüllt z.B. beim Empfangen von Radiostationen	Software.mkv 00:18-00:25
FA#5 Parameter abfragen	Erfüllt	Im Hintergrund
FA#6 Parameter setzen	Erfüllt	Im Hintergrund
FA#7 Radioinformation empfangen	Erfüllt z.B. Name, Frequenz, Text	Software.mkv 00:20-00:25
FA#8 Senderliste anzeigen	Erfüllt	Software.mkv Ab 00:18
FA#9 Senderliste aktualisieren	Erfüllt	Software.mkv 00:18-00:22
FA#10 Radiosender entfernen	Erfüllt Nach Abziehen der Antenne	Software.mkv 01:50-02:00
FA#11 Radiofrequenz anzeigen	Erfüllt Wird bei jeder Radiostation gesehen	Software.mkv

FA#12 Übertragungstechnologie anzeigen	Erfüllt Wird bei jeder Radiostation gesehen	Software.mkv
FA#13 Radiotext anzeigen	Erfüllt z.B. bei Oldie Antenne	Software.mkv 01:03-01:07
FA#14 Stationslogo anzeigen	Erfüllt Wird bei DAB-Stationen gesehen	Software.mkv
FA#15 Stationslogo aktualisieren	Erfüllt Bei jedem Neustart des Systems	Im Hintergrund
FA#16 Aktuelle Sendungsbild anzeigen	Erfüllt z.B. bei Oldie Antenne	Software.mkv 00:53-00:55
FA#17 Sendungsbild abrufen	Erfüllt	Im Hintergrund
FA#18 Radiostation wählen	Erfüllt z.B. bei Dlf	Software.mkv 00:40-00:45
FA#19 Auf die nächste bzw. vorherige Radiostation wechseln	Erfüllt	Software.mkv 02:30-02:40
FA#20 Radiostationen speichern	Erfüllt Bis zu 5 Stationen	Software.mkv 01:08-01:12
FA#21 Radiostation aus dem Memory laden	Erfüllt	Software.mkv 01:15-01:17
FA#22 Tuner App muss auf dem Gemini laufen	Erfüllt	Vorstellung.MOV

Tabelle 5: Validierung der funktionalen Muss-Anforderungen

Die funktionalen Soll-Anforderungen werden in Tabelle 6 verifiziert. Alle funktionalen Soll-Anforderungen wurden erfüllt.

Anforderung	Validierung	Zeitstempel
FA#23 Integration in die Micro-Service Architektur	Erfüllt Integration von „Tuner App“ mit dem Master App	Aufbau.MOV
FA#24 Portierung der Tuner-Middleware auf das Gemini	Erfüllt	Aufbau.MOV
FA#25 Starten des Radios während dem Bootvorgang	Erfüllt	Aufbau.MOV

Tabelle 6: Validierung der funktionalen Soll-Anforderungen

6.3 Verifizierung der nicht-funktionalen Anforderungen

In Tabelle 7 werden alle nicht-funktionalen Anforderungen validiert. Alle nicht-funktionalen Anforderungen wurden erfüllt, allerdings konnten manche davon aufgrund Ihrer Form nicht auf dem Video dargestellt werden. Es wurden sowohl die Windows als auch Linux Version der Entwicklungsumgebung erstellt, allerdings bleibt diese firmenintern.

Anforderung	Validierung	Zeitstempel
NFA#1 Einheitliches Design	Erfüllt	Anschicht1.jpeg
NFA#2 Zurverfügungstellung	Erfüllt Für Messen und Akquisen	-
NFA#3 Entwicklungsumgebung	Erfüllt Linux VM & Windows	-
NFA#4 Dokumentation der Entwicklungsumgebung	Erfüllt	Auf der firmeninternen Cloud
NFA#5 Ressourcenverbrauch	Erfüllt	Aufbau.MOV Ab 00:50

Tabelle 7: Validierung der nicht-funktionalen Anforderungen

7 Fazit

Die im Rahmen dieser Arbeit entwickelte Applikation namens „Tuner App“ konnte erfolgreich als Demonstrator umgesetzt werden. Alle zuvor definierten Anforderungen wurden erfüllt. Somit wurden auch die zu Beginn erläuterte Probleme, wie das Aussehen der Benutzeroberfläche und das Starten der Software verbessert und gelöst. Die Applikation kann nun auf einem echten Infotainment System laufen und wird während des Bootvorgangs automatisch gestartet. Dazu wurde die Tuner-Middleware für das IAV-Gemini portiert und das ARM64 Dev Board konnte dadurch gespart werden. Diese Software-Portierung gilt auch als Forschung für die zukünftige Produktion des Geräts. Dieses System kann nun als Messe-Demonstrator für andere Abteilungen bei IAV GmbH verwendet werden, sowohl zum Demonstrieren der HMI als auch der Telematik. Die dokumentierte Entwicklungsumgebung soll zusätzlich dazu dienen, dass weitere Entwickler Anpassungen und Erweiterungen vornehmen können. Durch die Entwicklung mit Qt konnte die Applikation plattformübergreifend erstellt werden, somit ist die Implementierung von „Tuner App“ auf andere Betriebssysteme wie Android Automotive möglich.

7.1 Ausblick

Da „Tuner App“ sich um eine Prototyp Software handelt, enthält die noch viele Schwächen und benötigt Verbesserung im Kontext der Zuverlässigkeit. Die Software kann noch optimiert werden, dass die im laufenden Betrieb weniger Ressourcen verbraucht. Dies kann bei einem eingebetteten System zur Verbesserung der Performance dienen. Da „Tuner App“ hauptsächlich als Demonstrator auf Messen und Akquisen eingesetzt wird, kann die Weiterentwicklung mehr auf HMI tendieren, es könnten z.B. neue Media-Funktionen wie Spotify oder YouTube eingebunden werden. Einige für die Messe attraktiven Funktionen könnte Sprachsteuerung oder die Möglichkeit zur Kopplung eines Handys sein.

Fachliteratur

[Abb 06] Abbott, D. (2006). *Linux for embedded and real-time applications* (2. Aufl.). Newnes.

[Gou 02] Gourley, D. & Totty, B. (2002*). HTTP: The definitive guide* (1. Aufl.). O'Reilly Media.

[Her 02] Herold, H. (2002). *Das Qt-Buch: Portable GUI-Programmierung unter Linux / Unix / Windows* (2., überarb. Aufl.). SuSE Linux AG.

[Jae 20] Jäger, M. (2020). *Betriebssysteme*.

[Lau 04] Lauterbach, T. (2004). *Digitaler Hörfunk: DAB und DVB im Einsatz*. Springer.

[Ma 24] Ma, J., & Gong, Z. (2024). *Automotive Human-Machine Interaction (HMI) Evaluation Method*. Springer Nature Singapore Pte Ltd.

[Nav 08] Navet, N., & Simonot-Lion, F. (Eds.). (2008). *Automotive embedded systems handbook*. CRC Press.

[Sie 12] Siemers, C. (2012). *Handbuch Embedded Systems Engineering* (Version 0.61a). Technische Universität Clausthal, Fachhochschule Nordhausen.

[Yag 13] Yaghmour, K. (2013). *Embedded Android (2. Aufl.)*. O'Reilly Media.

Onlinequellen

[Ber 1] berliner-radiosender.de. (n.d.). DAB+ Radioübersicht - Berliner Radiosender. Abgerufen am 14.10.2024, von *https://www.berliner-radiosender.de*

[Lin 1] Linux Devices. (2008). Trolltech and Qtopia. Abgerufen am 28.10.2024, von *https://linuxdevices.org/trolltech-and-qtopia-revd-and-renamed*

[Qt1] Qt Company. (2024). Signals & Slots. In Qt Core 6.8.0 Documentation. Abgerufen am 11.20.2024, von *https://doc.qt.io/qt-6/signalsandslots.html*

[Wik 1] Wikipedia. (2023). Linux (Kernel). Abgerufen am 26.11.2024, von *https://de.wikipedia.org/wiki/Linux_(Kernel)*

[Wik 2] Wikipedia. (2024). Android Automotive. Abgerufen am 26.10.2024, von *https://de.wikipedia.org/wiki/Android_Automotive*

[Wik 3] Wikipedia. (2023). Digital Audio Broadcasting. Abgerufen am 14.10.2024, von *https://de.wikipedia.org/wiki/Digital_Audio_Broadcasting*

[Wik 4] Wikipedia. (n.d.). Framework. Abgerufen am 18.10.2024, von *https://de.wikipedia.org/wiki/Framework*

[Wik 5] Wikipedia. (2024). Qt Bibliothek. Abgerufen am 3.10.2024, von *https://de.wikipedia.org/wiki/Qt_Bibliothek*

[Wik 6] Wikipedia. (2024). QML. Abgerufen am 4.10.2024, von *https://de.wikipedia.org/wiki/QML*

Abbildungsverzeichnis

Tabellenverzeichnis

Codebeispielverzeichnis

Anhang

A. Demonstrationsvideos

Videos zur Demonstration der realisierten Funktionen befinden sich unter:
https://cloud.bht-berlin.de/index.php/s/ZZn4sFjaXDx9dmE

B. Quellcode

Der Quellcode mit abgeschnittenen Abschnitten befindet sich unter:
https://cloud.bht-berlin.de/index.php/s/rr6wp6LqjGg5BZB