Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Fakultät Elektro- und Informationstechnik
Studiengang Elektrotechnik – Elektro- und Informationstechnik

Masterthesis

von

David Erb

Referent:	Prof. Dr. Marianne Katz
Korreferent:	Prof. Dr. rer. nat. Klaus Wolfrum
Arbeitsplatz:	
Betreuer am Arbeitsplatz:	
Zeitraum:	18.04.2017 – 18.10.2017

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Masterthesis ohne unzulässige fremde Hilfe selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Stuttgart, den 29. September 2017

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Seit dem Aufkommen des CAN-Busses in den 1980er-Jahren nahm die Anzahl der verbauten Steuergeräte im PKW stetig zu. Mehr und mehr Sensoren und Aktoren werden eingesetzt und Infotainmentsysteme gehören inzwischen zur Standardaustattung eines Autos. Aus diesen Gründen steigen die zu verarbeitenden Datenmengen. Es werden leistungsfähige Steuergeräte benötigt, die durch Bussysteme miteinander vernetzt sind und mit hohen Datenübertragungsraten kommunizieren. Im Automotive-Bereich haben sich die Bussysteme CAN, LIN, MOST und FlexRay etabliert.

Zukünftig wird sich dieser Trend fortsetzen und das Datenaufkommen innerhalb von Fahrzeugen wird weiter steigen. Als Grund ist hier beispielsweise die Zunahme von Fahrassistenzsystemen zu nennen, die den Fahrer im Verkehr unterstützen. Eine Antwort auf diese steigenden Anforderungen sind Busprotokolle wie das im Jahre 2012 veröffentlichte CAN FD-Protokoll, das eine bis zu 8-fach höhere Datenübertragungsrate ermöglicht.

Obwohl die Fahrzeugssysteme komplexer werden, müssen die Hersteller aufgrund des Konkurrenzdrucks die Entwicklungszyklen für neue Fahrzeugmodelle verkürzen und die Entwicklungs- und Herstellungskosten möglichst gering halten. Auch bei der Entwicklung von Steuergeräten muss auf diese Herausforderungen reagiert werden. Es werden Möglichkeiten benötigt, schnell und kostengünstig neue prototypische Funktionen zu entwickeln und diese innerhalb einer realen oder simulierten Fahrzeugumgebung zu testen.

Die Firma GIGATRONIK bietet mit ihren GIGABOX-Produkten dafür eine Lösung. Die Universalsteuergeräte sind ausgestattet mit Kommunikations-

schnittstellen, die im Automotive-Bereich gängig sind wie z. B. CAN und LIN. Zusätzlich besitzen sie digitale und analoge Ein- und Ausgänge zur Signalverarbeitung bzw. Steuerung und Regelung von Systemen. Für die Entwicklung von GIGABOX-Applikationen wird die integrierte Entwicklungsumgebung (IDE) *configurAIDER* eingesetzt. Dort können Steuergerätefunktionen auf einfache Weise programmiert werden unter Verwendung der Skriptsprache *Pawn*.

Allerdings wird die aktuelle Produktpalette der GIGABOX nicht mehr den neuesten technischen Anforderungen gerecht, da diese beispielsweise kein CAN FD unterstützt. Im Moment wird deshalb die GIGABOX FD entwickelt, welche die bisherige GIGABOX-Familie ablösen soll. Im Vergleich zu den Vorgängermodellen besitzt diese eine CAN FD-Schnittstelle. Ebenfalls bietet sie die Möglichkeit, zusätzliche Kommunikationstechnologien über Erweiterungsplatinen anzubinden.

Eine moderne Desktop-Anwendungssoftware sollte dem Anwender eine attraktive und gut benutzbare Benutzeroberfläche bieten. Dadurch ergeben sich für den Nutzer Vorteile wie eine kürzere Einarbeitungszeit in die Software, produktiveres Arbeiten sowie weniger Fehler bei der Bedienung. Ein gutes Beispiel für eine IDE mit einer gelungenen Benutzeroberfläche ist *Visual Studio* von Microsoft. Für die neue GIGABOX FD soll deshalb eine moderne IDE zur Verfügung stehen mit einer qualitativ hochwertigen Benutzeroberfläche.

1.2 Zielsetzung

Die Ziele der vorliegenden Arbeit sind:

- Erarbeitung einer Anforderungsanalyse für eine IDE, die zur Erstellung von GIGABOX FD-Applikationen verwendet werden kann
- Umsetzung der Anforderungen auf Basis von standardisierten Softwareentwicklungsprozessen
- Erstellung einer Softwarearchitektur mithilfe von UML-Diagrammen
- Implementierung unter dem .NET Framework mit C#. Zur Erstellung der grafischen Benutzeroberfläche bietet sich die Verwendung von WPF an.

Kapitel 2

Grundlagen

2.1 GIGABOX-Steuergeräte

2.1.1 Überblick

GIGABOX ist eine Produktfamilie von Steuergeräten der Firma GIGATRONIK. GIGABOX-Steuergeräte werden in der Fahrzeugentwicklung eingesetzt, um prototypische Kommunikationslösungen im Fahrzeug schnell und kostengünstig realisieren zu können. Beispielanwendungen sind der Einsatz als Gateway und die Ansteuerung von Treiber-ICs im Automobil.

GIGABOX-Steuergeräte werden in unterschiedlicher Hardwareausstattung und Funktionalität angeboten. Typischerweise verfügen sie über diverse Kommunikationsschnittstellen, die im Automotive-Bereich eingesetzt werden wie CAN und LIN. Außerdem stehen digitale und analoge Eingänge sowie digitale Ausgänge zur Verfügung.

Das neueste Modell der GIGABOX, die GIGABOX FD, befindet sich aktuell noch in der Entwicklung. Im Vergleich zu älteren Modellen ermöglicht sie neue Funktionen wie Kommunikation über CAN FD oder Bluetooth.

2.1.2 Einsatzszenarien

Das Use-Case-Diagramm in Abbildung 2.1 zeigt drei typische Einsatzszenarien einer GIGABOX.

1. **Einsatz als Gateway zur Modifikation von Busbotschaften**

Beispiel:

Ein Fahrzeughersteller entwickelt das Nachfolgemodell einer PKW-Modellreihe. Für das neue Fahrzeugmodell wurde eine neue Klimaanlage entwickelt, deren Funktionsfähigkeit im realen neuen Modell überprüft werden soll. Einige Busbotschaften, die die Steuergeräte untereinander austauschen, wurden für das neue Modell verändert oder wurden neu hinzugefügt. Der Großteil der Busbotschaften sind allerdings identisch wie beim alten Modell.
Das neue Modell ist allerdings noch nicht so weit entwickelt, dass es eine sinnvolle Testumgebung darstellen kann. Die neue Klimaanlage wird deshalb im alten Fahrzeugmodell getestet. Die für das Klimasteuergerät relevanten Busbotschaften können dann mithilfe einer GIGABOX so manipuliert werden, dass sie den Botschaften entsprechen, die im neuen Fahrzeugmodell am Klimasteuergerät ankommen würden.

2. **Funktionalität des Fahrzeugs erweitern**

   - Rapid Prototyping: Eine neue Funktionalität soll möglichst schnell und unkompliziert gestestet werden

     Beispiel:

     Ein Fahrzeughersteller möchte eine neue Idee ausprobieren. Die LEDs des Fahrzeugblinkers sollen dabei nicht mehr gleichzeitig aufleuchten, sondern nacheinander von innen nach außen mit kurzer zeitlicher Verzögerung. Zur Umsetzung dieser Idee kann eine GIGABOX verwendet werden. Registriert die GIGABOX eine Botschaft auf dem CAN-Bus, die eine Anweisung zum Blinken enthält, können die einzelnen LEDs des Blinkers zeitlich verzögert in der gewünschten Reihenfolge über die digitalen Ausgänge aktiviert werden.

   - Serieneinsatz: Eine neue zusätzliche Funktion soll realisiert werden bei Fahrzeugumbau

     Beispiel:

     Ein PKW soll zu einem Leichenwagen umgebaut werden, der elektrisch verstellbare Fensterjalousien besitzen soll. Über einen Schalter soll die Höhe der Jalousien eingestellt werden können. Dazu wird ein Steuergerät benötigt, das die Stellung der Schalter einliest und die zur Höhenverstellung zuständigen Motoren ansteuert. Eine GIGABOX kann diese Aufgaben übernehmen, da sie über die benötigten digitalen Ein- und Ausgänge verfügt.

3. **Restbussimulation**

Eine GIGABOX kann Steuergeräte innerhalb eines Bussystems simulieren. Steuergeräte können damit getestet werden, ohne das komplette Bussystem aufbauen zu müssen.

Restbussimulationen können auch mit PC-Tools wie zum Beispiel „CANoe“ von Vector Informatik GmbH und passendem Bus-Interface durchgeführt werden. Die Kosten für Softwarelizenzen und benötigter Hardware übersteigen allerdings die Kosten einer konfigurierten GIGABOX und bedingen einen komplizierteren Systemaufbau.

Quelle: http://www.samtec.de/hauptmenu/loesungen-fuer/restbussimulation.html

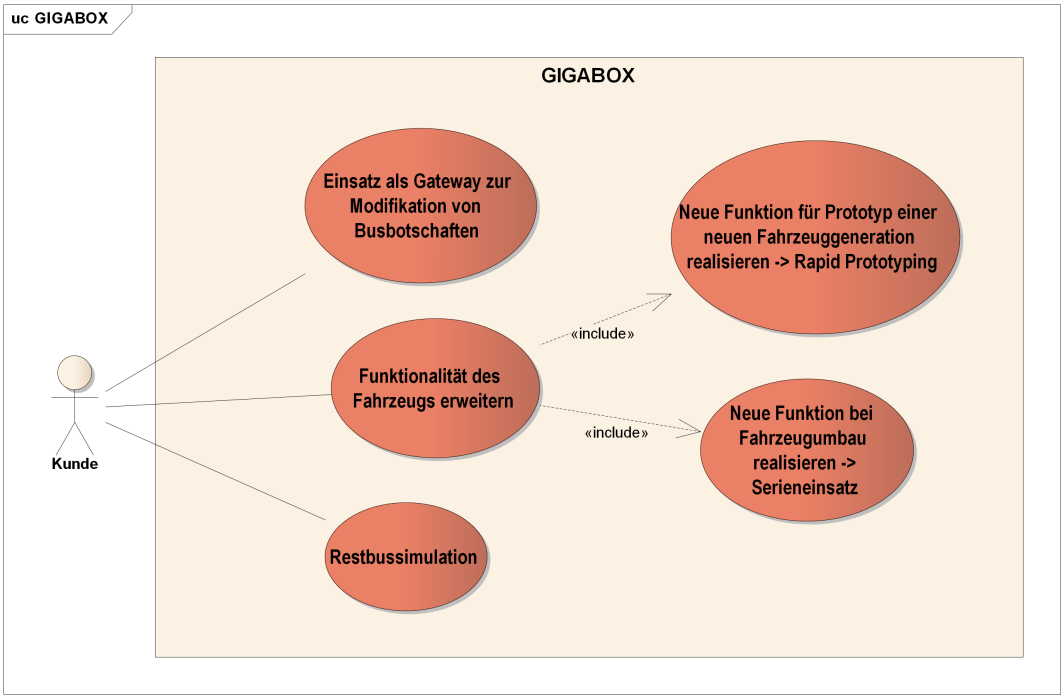

Abbildung 2.1 – Use-Case-Diagramm GIGABOX

2.1.3 Hardware der GIGABOX FD

Die GIGABOX FD ist modular aufgebaut bestehend aus einer Basisplatine und diversen Applikationsplatinen, mit denen zusätzliche Funktionalitäten zur Verfügung gestellt werden können. Auf der Basisplatine befindet sich ein Spannungsregler, ein 32-bit Mikrocontroller ARM Cortex M4 STM32F427IIT6, Kommunikationsschnittstellen sowie analoge und digitale Ein- und Ausgänge. Der Spannungsregler wird eingesetzt, um die Batteriespannung des Fahrzeugs auf die Versorgungsspannung des Mikrocontrollers herunterzuregeln.

Der Mikrocontroller wird mit einer Taktfrequenz von 168MHz betrieben und stellt 2MB Flash Speicher sowie 192 kB SRAM + 64 kB core coupled memory (CCM) zur Verfügung.

Auf der GIGABOX FD sind als Kommunikationsschnittstellen 2 CAN-, 2 CAN FD- und 2 LIN-Kanäle implementiert. Zusätzlich wird eine USB-Schnittstelle bereitgestellt, die mit einem UART-to-USB-Converter realisiert wird. Es sind 4 analoge Eingänge vorhanden um Sensorwerte einzulesen wie z. B. von einem Temperatursensor. Mit den 8 digitalen Eingängen können z. B. Schalterzustände eingelesen werden. Als Ausgänge stehen 8 Halbbrückenausgänge und 2 Highsideausgänge zur Verfügung (Wann werden Halbbrücken- wann Highside benutzt?).

Einen Überblick über die eingesetzte Hardware bietet Abbildung 2.2 (Von Shruti, Abb neu erstellen wegen schlechter Bildqualität).

Der Einsatz von einer Applikationsplatine ermöglicht die Funktionserweiterung der GIGABOX FD um beispielsweise zusätzliche Kommunikationsschnittstellen wie Bluetooth bereitzustellen.

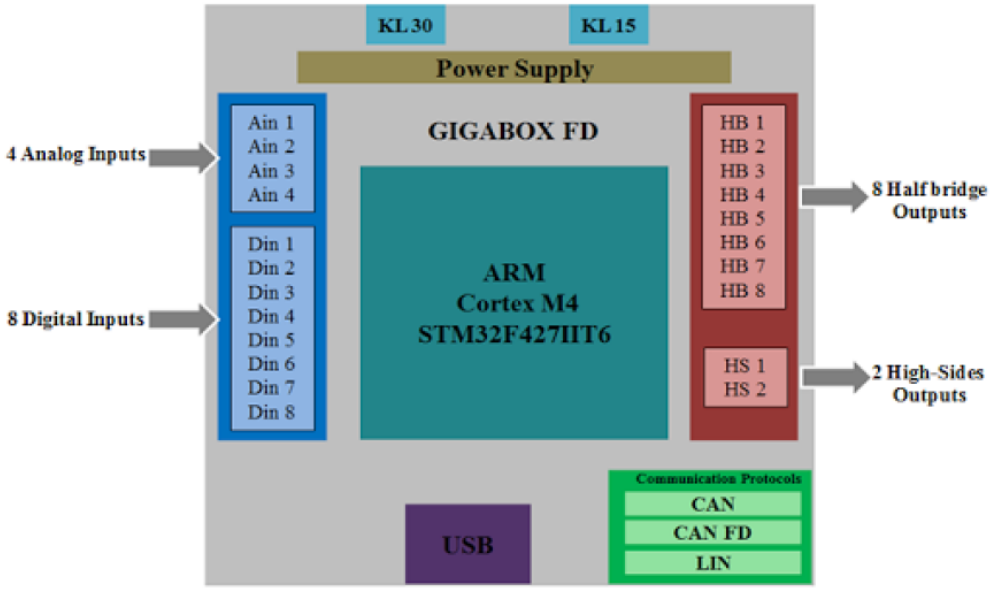

Abbildung 2.2 – Hardwareüberblick GIGABOX FD

2.1.4 Softwarearchitektur der GIGABOX FD

Abbildung 2.3 gibt einen Überblick über die Softwarearchitektur der GIGABOX FD.

Der Hardware Abstraction Layer (HAL) ist eine Softwareschicht, die übergeordnete Schichten von der Hardware abstrahiert. Übergeordnete Schichten sind damit unabhängig von der eingesetzten Hardware. Dies bietet den Vorteil, dass bei Hardwareänderungen nur der HAL angepasst werden muss, nicht aber die übergeordneten Schichten. Auf dem HAL sind die Treiber des Mikrocontrollers und der zugehörigen Peripherie implementiert. Die Treiber stellen ein Application Programming Interface (API) bereit, die Zugriff auf den Speicher oder die Peripherie des Mikrocontrollers wie z.B. Analog-Digital-Converter (ADC) ermöglicht.

Vom Middleware Layer aus kann über die API des Hardware Abstraction Layers auf die Hardware zugegriffen werden. Auf dieser Schicht ist ein USB-HID Treiber implementiert, der benötigt wird, damit die GIGABOX FD bei Verbindung mit einem Computer über USB als Human Interface Device (HID) erkannt wird. Außerdem befindet sich dort das Transportprotokoll ISO-TP für CAN-Bus. Das Protokoll wird benötigt, um Botschaften zu verschicken, die die maximale Nutzdatengröße von 8 Byte eines CAN-Frames überschreiten. Um die Pawn virtuelle Maschine (VM) an den HAL anzubinden, befindet sich auf dem Middleware Layer zusätzlich die PAWN Driver Abstraction.

Auf dem Application Layer ist die Pawn VM und der Bootloader implementiert. Eine Abstraktionsebene über der Pawn VM liegt die Skript Applikation, die vom Endanwender auf den Flash-Speicher der GIGABOX FD aufgespielt werden kann. Das Skript ermöglicht die Ansteuerung bzw. Abfrage der Kommunikationsschnittstellen sowie das Steuern bzw. Beobachten der digitalen/analogen Ein-/Ausgänge der GIGABOX FD. Auf dieser Ebene ist außerdem die Konsole angesiedelt, an die der Anwender Befehle senden kann über USB.

2.2 Pawn

Pawn ist eine Skriptsprache mit einer C ähnlichen Syntax sowie eine zugehörige virtuelle Maschine, auf der kompilierter Pawn-Quellcode ausgeführt werden kann. Das Pawn-Paket, das Kompiler und virtuelle Maschine beinhaltet, ist kostenlos verfügbar und wurde unter der Apache Lizenz 2.0 veröffentlicht. Die Sprache wurde abgeleitet von der Sprache „Small-C“, die wiederum eine Teilmenge von C darstellt und für Mikrocontroller entwickelt wurde. Pawn wurde erstmal 1998 öffentlich zugänglich gemacht, damals noch unter dem Namen „SMALL“ und wird seitdem regelmäßig weiterentwickelt.

Pawn-Quellcode kann mithilfe des Pawn-Kompilers in Bytecode kompiliert werden. Der Bytecode wird anschließend auf der zugehörigen virtuellen Maschi-

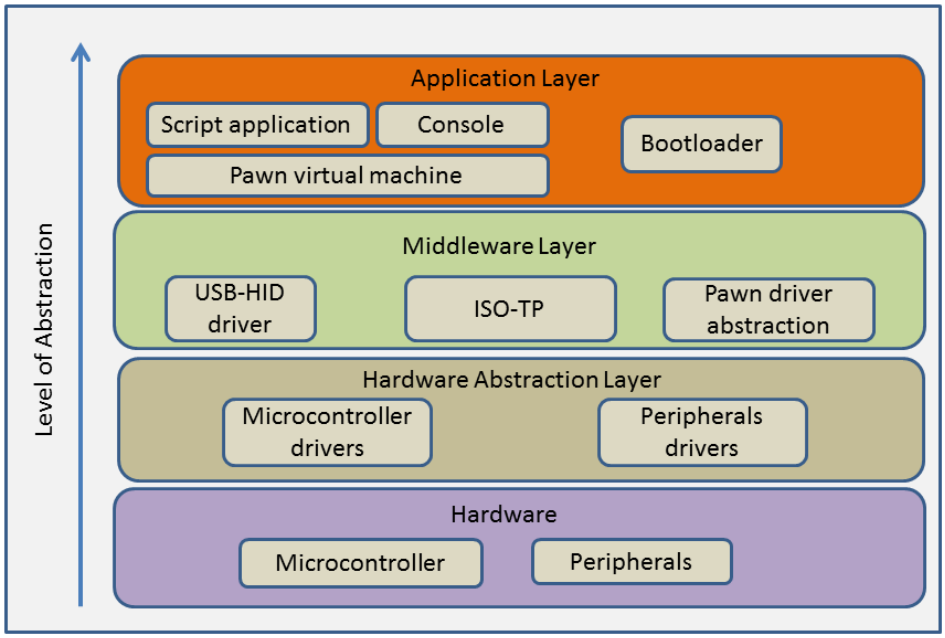

Abbildung 2.3 – Softwarearchitektur der GIGABOX FD

ne interpretiert. Dies bedeutet, dass für jede Bytecode-Anweisung definierte Funktionen aus einer C-Klassenbibliothek aufgerufen werden. Die Ausführung von Bytecode auf einer virtuellen Maschine bietet im Vergleich zur Ausführung von nativem Code auf der Zielhardware den Vorteil, dass der Code unabhängig von der eingesetzten Hardware lauffähig ist. Der Code kann folglich plattformunabhängig eingesetzt werden. Die virtuelle Maschine muss dagegen an die eingesetzte Plattform angepasst werden. Dieser Vorteil wird allerdings mit Geschwindigkeitseinbußen bei der Programmausführung bezahlt. Bei der Entwicklung von Pawn wurde besonders auf eine einfache Syntax, Schnelligkeit in der Codeausführung und Robustheit wert gelegt. Die virtuelle Maschine eignet sich aus diesem Grund auch zur Einbindung in Mikrocontroller mit geringer Rechenleistung und Speicherkapazität.

In Bezug auf GIGABOX-Steuergeräte wird Pawn benutzt, um GIGABOX-Applikationen zu erstellen. Mithilfe von GIGABOX-Applikationen kann der Anwender Funktionen für GIGABOX-Steuergeräte entwickeln und die Applikation anschließend auf GIGABOX-Steuergeräte flashen, auf denen eine Pawn-VM implementiert ist. GIGABOX-Applikationen werden mithilfe eventgetriebener Programmierung entwickelt. Sobald ein definiertes Event wie z. B. der Empfang einer CAN-Botschaft eintritt, wird die dem Event zugeordnete Funktion aufgerufen.

[1], [2]

Siehe Pawn_Language_Guide.pdf, Foreword
Pawn_Implementation_Guide.pdf, The Compiler, The abstract machine

2.3 Softwareentwicklungsprozess

2.3.1 V-Modell 97

Das V-Modell 97 ist ein Vorgehensmodell für IT-Entwicklungsprojekte, das 1997 von der Bundesrepublik Deutschland veröffentlicht wurde. Es hat sich als Entwicklungsstandard in der Softwareentwicklung etabliert.

Der Entwicklungsprozess nach dem V-Modell durchläuft verschiedene Phasen. Es beginnt mit einer Anforderungsanalyse, bei der die Anforderungen an ein System ermittelt und dokumentiert werden.

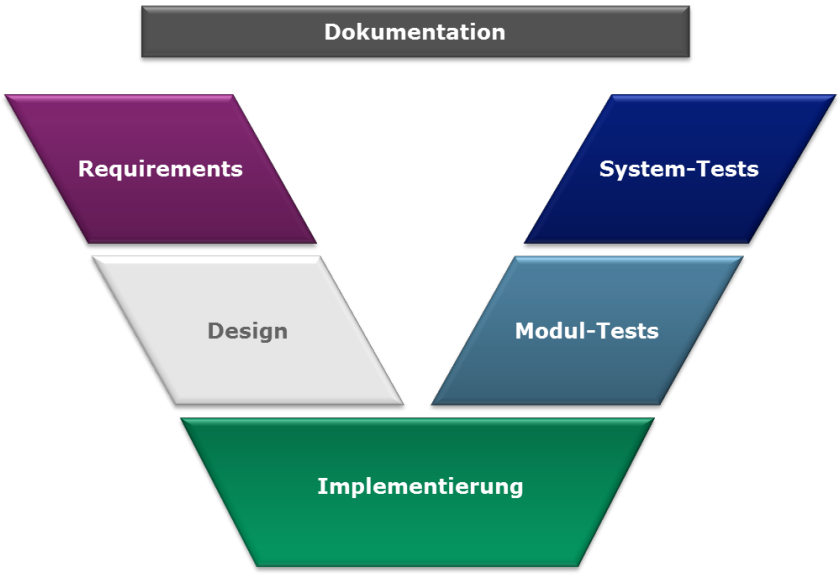

Abbildung 2.4 – Vorgehensschritt im V-Modell

2.3.2 Qualitätskriterien an Software

Um Softwarequalität definierbar zu machen, wurden diverse Software-Qualitätsmodelle entwickelt. Die ISO/IEC 25010 definiert unter anderem das Product Quality Model. Dieses Modell definiert acht Qualitätmerkmale für Software, die im folgenden erläutert werden:

- *Functional Suitability:*

  „Das Qualitätsmerkmal der funktionallen Eignung fordert, dass ein Software-System die von ihm erwartete Funktionalität in angemesserner und konkreter Art und Weise und unter Berücksichtigung der festgelegten Randbedingungen und Eigenschaften umsetzt. Dabei geht es um Fragestellungen nach der Angemessenheit der Funktionalitäten für den vorgegebenen Einsatzbereich des Systems.“ [3]

- *Performance Efficiency*

  „Das Merkmal der Performanz und Effizienz betrachtet das Leistungsniveau einer Anwendung in Abhängigkeit der eingesetzten Betriebsmittel und der geforderten Bedingungen. Dabei geht es insbesondere um das Zeitverhalten der Anwendung (Anwendungszeit, Durchsatz) sowie das Verbrauchsverhalten in Bezug auf Speicher, Prozessorzeit und andere Ressourcen.“ [3]

- *Compatibility*

  „Das Merkmal der Kompatibilität betrachtet einerseits die explizite Zusammenarbeit des Systems mit anderen Systemen und andererseits die impliziten Auswirkungen des Systems auf andere Systeme, die auf der gleichen Hardware laufen.“ [3]

- *Usability*

  „Das Qualitätsmerkmal der Benutzerfreundlichkeit beschäftigt sich damit, wie gut ein System seine Nutzer dabei unterstützt ihre Ziele effizient und effektiv zu erreichen. Zu den zentralen Teilmerkmalen zählt hier beispielsweise der Aufwand für die Erlernung des Systems sowie für die Bedienung. Ein weiteres Teilmerkmal ist die Attraktivität des Systems für die Nutzer.“ [3]

- *Reliability*

  „Das Qualitätsmerkmal der Zuverlässigkeit beschreibt die Fähigkeit des Systems sein Leistungsniveau unter festgelegten Bedingungen über einen festgelegten Zeitraum zu bewahren. Zu den Teilmerkmalen zählen hier beispielsweise die Robustheit gegenüber fehlerhaften Eingaben, die Stabilität beim Umgang mit fehlerhaften Systemzuständen und die Wiederherstellbarkeit bei Systemausfällen.“ [3]

- *Security*

  „Das Merkmal umfasst alle in Bezug auf Datensicherheit relevanten Teilmerkmale wie die Datenintegrität, Authentizität und Vertraulichkeit.“ [3]

- *Maintainability*

  „Das Merkmal der Wartbarkeit, teilweise auch als Änderbarkeit bezeichnet, beschäftigt sich vorrangig mit der internen Qualität eines Systems. Zentrale Teilmerkmale hier sind der Aufwand zur Analyse eines Fehlers oder Fehlverhaltens, der Aufwand zur Durchführung von Modifikationen mit entsprechender Testbarkeit sowie die Stabilität des Systems gegenüber unerwarteten Seiteneffekten.“ [3]

- *Portability*

  „Die Portabilität als weiteres Qualitätsmerkmal beschäftigt sich mit dem Aufwand, um ein System strukturell zu verändern, auf eine andere Umgebung zu verlagern oder einzelne Systemkomponenten auszutauschen. Als Teilmerkmale nennt der Standard die Anpassbarkeit eines Systems an unterschiedliche Umgebungen, den Aufwand zur Installation und die Austauschbarkeit von Komponenten.“ [3]

Abbildung 2.5 zeigt eine Übersicht über die acht Qualitätsmerkmale und löst jedes Merkmal feingranular weiter auf.

Um das Qualitätsmerkmal „Usability“ zu verfeinern, kann die Norm EN ISO 9241-110 (Grundsätze der Dialoggestaltung) herangezogen werden. Diese beschreibt, welche Eigenschaften Dialoge erfüllen sollen, um eine gute Usability zu erreichen. Die sieben Grundsätze der Dialoggestaltung werden im Folgenden nach Quelle [4], Kapitel 8.3 zitiert.

- Aufgabenangemessenheit: „Ein interaktives System ist *aufgabenangemessen*, wenn es den Benutzer unterstützt, seine Arbeitsaufgabe zu erledigen, d.h. wenn Funktionalität und Dialog auf den charakteristischen Eigenschaften der Arbeitsaufgabe basieren, anstatt auf der zur Aufgabenerledigung eingesetzten Technologie.“

- Selbstbeschreibungsfähigkeit: „Ein Dialog ist in dem Maße *selbstbeschreibungsfähig*, in dem für den Benutzer zu jeder Zeit offensichtlich ist, in welchem Dialog, an welcher Stelle er sich befindet, welche Handlungen unternommen werden können und wie diese ausgeführt werden können.“

- Lernförderlichkeit: „Ein Dialog ist *lernförderlich*, wenn er den Benutzer beim Erlernen der Nutzung des interaktiven Systems unterstützt und anleitet.“

- Steuerbarkeit: „Ein Dialog ist *steuerbar*, wenn der Benutzer in der Lage ist, den Dialogablauf zu starten sowie seine Richtung und Geschwindigkeit zu beeinflussen, bis das Ziel erreicht ist.“

- Erwartungskonformität: „Ein Dialog ist *erwartungskonform*, wenn er den aus dem Nutzungskontext heraus vorhersehbaren Benutzerbelangen sowie allgemein anerkannten Konventionen entspricht.“

- Individualisierbarkeit: „Ein Dialog ist *individualisierbar*, ween Benutzer die Mensch-System-Interaktion und die Darstellung von Informationen ändern können, um diese an ihre individuellen Fähigkeiten und Bedürfnisse anzupassen.

- Fehlertolerant: „Ein Dialog ist *fehlertolerant*, wenn das beabsichtigte Arbeitsergebnis trotz erkennbar fehlerhafter Eingaben entweder mit keinem oder mit minimalem Korrekturaufwand durch den Benutzer erreicht werden kann.“

  Quelle: Wikipedia -> überprüfen ob andere Quelle verfügbar ist, am Besten die Norm direkt Als Quelle kann Paper von PH Weingarten „Usability

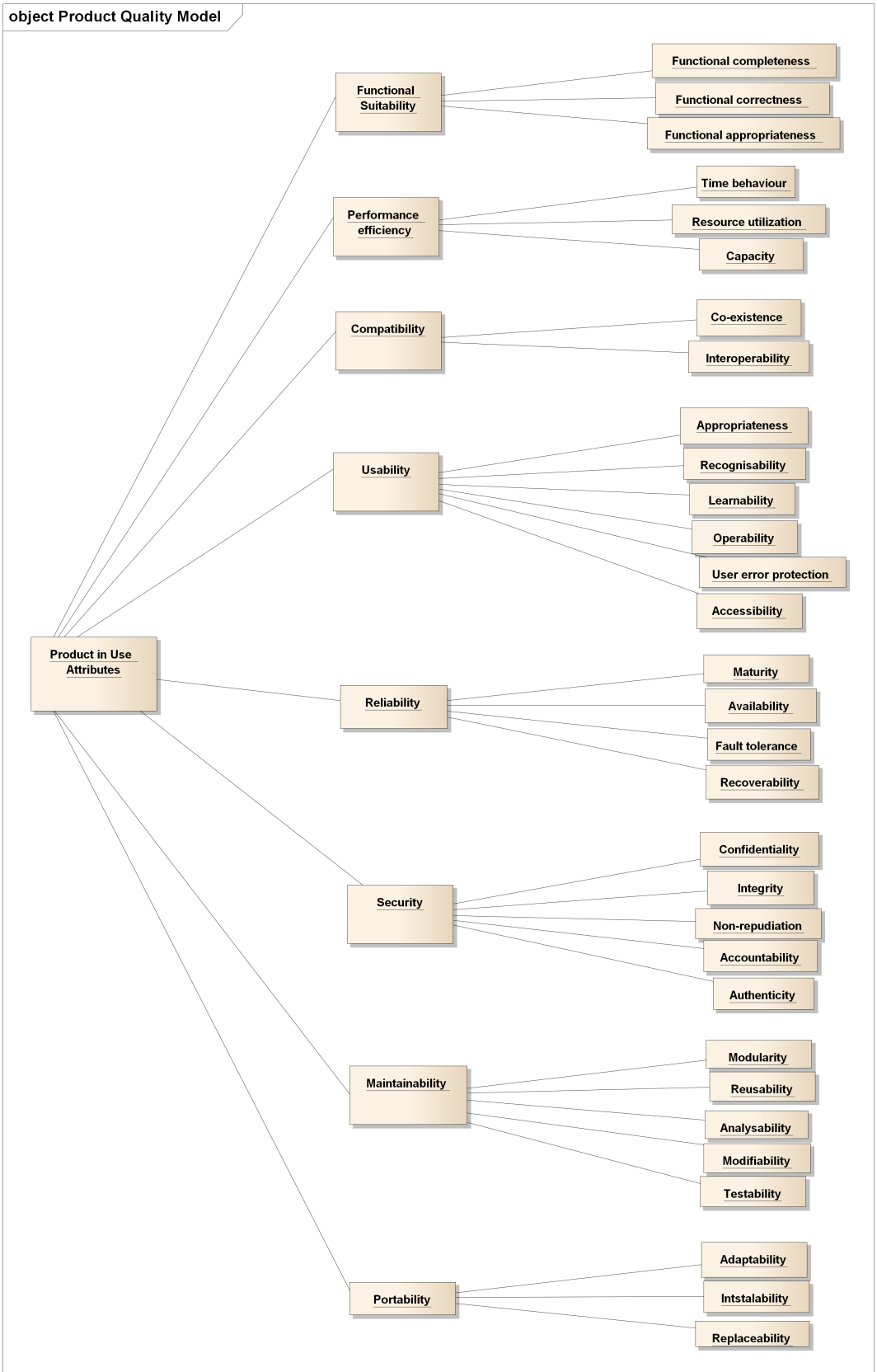

Abbildung 2.5 – Qualitätsmerkmale des Product Quality Model nach ISO/IEC 25010

und Usability Engineering genommen werden“. Hieraus die Stichpunkte noch verfeinern

2.4 Entwicklungsframework

2.4.1 .NET Framework

Es wurden im folgenden Abschnitt die Quellen [5],[6] verwendet.

Das .NET Framework ist eine von Microsoft entwickelte Entwicklungsplattform für Anwendersoftware, die erstmals im Januar 2002 in der Version 1.0 erhältlich war. Es stellt die Umsetzung des Common Language Infrastructure (CLI) Standards dar, der sprach- und plattformunabhängige Anwendungsentwicklung spezifiziert. Wesentliche Bestandteile des .NET Frameworks sind die Laufzeitumgebung Common Language Runtime (CLR) und die Framework Class Library (FCL). Die CLR verwaltet Speicher, Thread- und Codeausführung, Überprüfung der Codesicherheit und Kompilierung. Von der CLR verwalteter Code wird dabei Managed Code genannt, außerhalb der CLR laufender Code Unmanaged Code. Es wird eine Vielzahl von Programmiersprachen unterstützt, unter anderem C#, C++ und Visual Basic .NET. Beim Kompilieren der verschiedenen Hochsprachen wird der Code zunächst in eine Zwischensprache übersetzt, der Common Intermediate Language (CIL). Aus dem daraus entstehenden Intermediate Language Code (IL-Code) wird dann von einem Just-In-Time-Compiler (JIT-Compiler) zur Laufzeit Maschinencode erzeugt (Abbildung 2.6).

Die FCL ist eine objektorientierte Klassenbibliothek mit einer Auflistung wiederverwendbarer Typen, von denen eigener Code abgeleitet werden kann.

2.4.2 Windows Presentation Framework (WPF)

Mit Einführung des .NET Frameworks 3.0 im Jahr 2006 wurde Windows Presentation Foundation (WPF) veröffentlicht, ein Framework zur Erstellung von grafischen Benutzeroberflächen. WPF stellt den Nachfolger für das seit Version 1.0 im .NET Framework enthaltene Windows Forms. Es können Desktop- sowie Webanwendungen erstellt werden.

WPF nutzt die Leistungsressourcen von Grafikkarten mit 3D-Beschleunigern besser aus als Windows Forms, da zum Rendern der GUI-Inhalte DirectX benutzt wird anstatt Graphics Device Interface+ (GDI+). Mithilfe von DirectX kann WPF grafische Elemente selbst zeichnen anstatt dass sie durch das

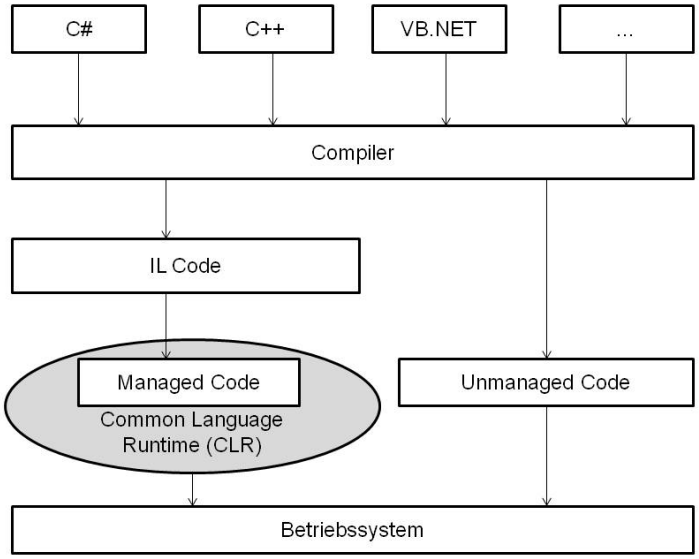

Abbildung 2.6 – Managed Code und Unmanaged Code

Betriebssystem gezeichnet werden. Ein weiterer Vorteil von WPF ist das vektorbasierte Zeichnen der Anwendungsinhalte. Dies ermöglicht eine beliebige Skalierung der Inhalte ohne Verpixelung.

Eines der zentralen Konzepte von WPF ist die Verwendung der XML-basierten Beschreibungssprache Extensible Application Markup Language (XAML) zur Beschreibung der Benutzeroberfläche. XAML ermöglicht eine übersichtlichere und kompaktere Beschreibung der Benutzeroberfläche als die Programmierung der Oberfläche in C#.

Die Beschreibung einer Benutzeroberfläche mit XAML wird in einer XAML-Datei vorgenommen. Jede XAML-Datei ist untrennbar mit einer Codebehind-Datei gekoppelt, in der die Logik der Benutzeroberfläche in C# programmiert werden kann. Die Logik kann allerdings auch in einer eigenen C#-Quellcodedatei programmiert werden und über sogenannte DataBindings und Commands an die XAML-Datei angebunden werden. Darstellung und Logik können auf diese Weisen getrennt werden, was unter anderem eine effektivere Zusammenarbeit zwischen Designern der Benutzeroberfläche und Entwicklern der Anwendungslogik ermöglicht.

Ein Designer kann mithilfe eines Design-Tools wie z. B. Microsoft Expression Blend die Benutzeroberfläche entwerfen und daraus eine XAML-Datei generieren. Der Entwickler der Anwendungslogik kann anschließend auf Grundlage

der XAML-Datei des Designers die gewünschte Logik zu der GUI entwickeln.

Kapitel 3

Anforderungsanalyse

3.1 Überblick

In diesem Kapitel soll eine Anforderungsanalyse für eine Entwicklungsumgebung für GIGABOX-Steuergeräte durchgeführt werden. In Abbildung 2.1 wurde gezeigt, für welche Anwendungen GIGABOXEN eingesetzt werden. Die Anforderungen an die Entwicklungsumgebung wurden so gestellt, dass sie Softwareentwicklern für GIGABOX-Funktionen die Umsetzung dieser Anwendungen ermöglicht und möglichst große Unterstützung bei der Codeentwicklung bietet.

Zuerst wurden Gespräche mit Entwicklern geführt, vorhandener Programmcode aus realisierten Projekten analysiert und selbstständig Codeentwicklung mit dem bestehenden configurAIDER betrieben um herauszufinden, welche Use Cases eine IDE zur Entwicklung von Funktionen für GIGABOX-Steuergeräte abdecken soll. Aus dieser Analyse entstanden die in Abbildung 3.1 veranschaulichten Use-Cases.

Von den ermittelten Use Cases wurden direkt die funktionalen Anforderungen abgeleitet, die im folgenden Abschnitt näher beleuchtet werden.

Auf Basis der Qualitätsmerkmale für die Anforderungsspezifikation im Standard IEEE 830-1998 werden die ermittelten funktionalen Anforderungen eingeteilt in drei Prioritärsklassen:

- Essential: Das Software-System kann nicht akzeptiert werden, wenn diese Anforderung nicht in der vereinbarten Form geliefert wird.
- Conditional: Diese Anforderungen erweitern das Software-System in geeigneter Form. Wenn sie fehlen, würden sie den Einsatz des Systems jedoch nicht gefährden.

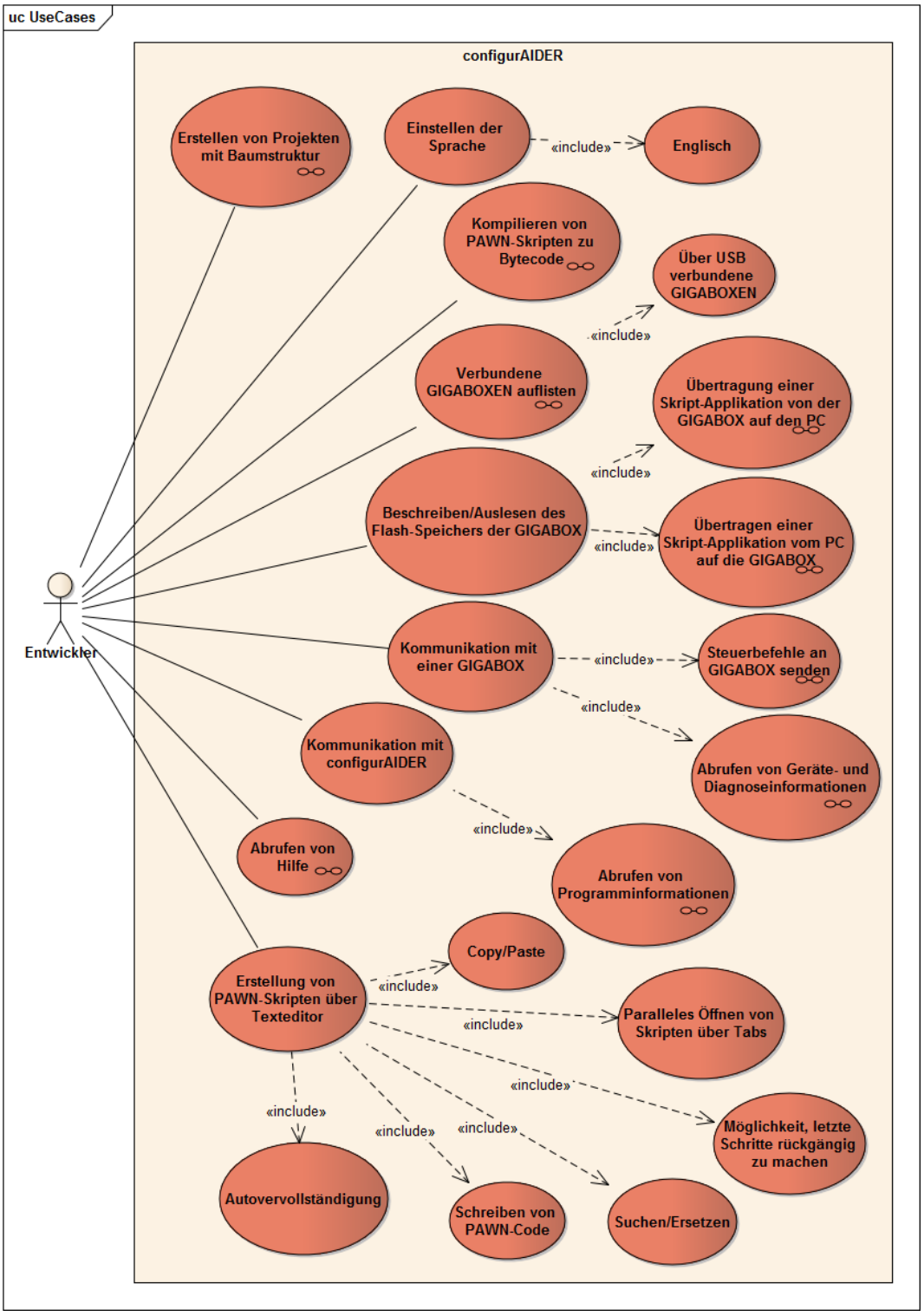

Abbildung 3.1 – Use-Case-Diagramm funktionale Anforderungen

- Optional: Diese Anforderungen sind nicht unbedingt notwendig, für die Anwender jedoch eine angenehme Erweiterung (nice to have).

Zur Ermittlung der nichtfunktionalen Anforderungen wurden bekannte Vorgehensmodelle und Normen der Softwareentwicklung herangezogen. Die hergeleiteten nichtfunktionalen Anforderungen sollen bei der Entwicklung beachtet werden, um eine hohe Qualität der zu entwickelnden Software sicherzustellen. Auf eine Bewertung wird verzichtet.

Quelle: [3]

3.2 Ermittlung und Bewertung von funktionalen Anforderungen

3.2.1 Erstellung von Projekten mit Baumstruktur

Beschreibung:

Innerhalb eines Projektes können verschiedene GIGABOX-Modelle angelegt werden. Für jedes angelegte GIGABOX-Modell können PAWN-Skripte erstellt werden mit der zum Modell passenden Dateiendung .p. Für jedes angelegte GIGABOX-Modell können Include-Files hinzugefügt werden. Der Benutzer erhält einen strukturierten Überblick über alle seine GIGABOX-Projekte und alle für ein Projekt entwickelten Skripte. Das Öffnen eines Skriptes erfolgt komfortabel über Doppelklick auf ein Skript in der Projektstruktur. Lästiges navigieren durch die Dateistruktur des PCs beim Öffnen eines Skriptes entfällt + der Benutzer muss den Ablageort im Dateisystem nicht wissen.

Bewertung:

Erhöht den Benutzerkomfort signikfikant, deshalb Einstufung in Prioritätsklasse „Conditional“.

3.2.2 Einstellung der Sprache

Beschreibung:

Die Sprache der Benutzeroberfläche kann angepasst werden. Es soll Deutsch und Englisch zur Verfügung stehen.

Bewertung:

Die Darstellung der Benutzeroberfläche in der Muttersprache des Benutzers erhöht die Usability, da es die Selbstbeschreibungsfähigkeit und Individualisierbarkeit fördert. Englisch muss auf jeden Fall zur Verfügung stehen, da es die international am weitesten verbreitete Sprache ist und auch in Deutschland von einem Großteil der Menschen verstanden wird. Die Darstellung der Benutzeroberfläche in Englisch wird deshalb in Prioritätsklasse „Essential“ eingeteilt. Die Darstellung in Deutsch wird in Klasse „Optional“ eingeteilt, da die Usability dadurch nur mäßig erhöht wird. Das liegt daran, dass der potenzielle Benutzerkreis des Tools Entwickler sind, denen größtenteils der Umgang mit englischen GUIs vertraut ist.

3.2.3 Texteditor zur Erstellung von PAWN-Skripten

- Schreiben von PAWN-Code

  Beschreibung:

  Es steht ein Textfeld zur Verfügung, in dem PAWN-Code editiert werden kann.

  Bewertung:

  Klasse „Essential“, da als Grundfunktionalität einer IDE einzustufen.

- Copy/Paste

  Beschreibung:

  Codefragmente können kopiert und an anderer Stelle eingefügt werden.

  Bewertung:

  Macht die Codeentwicklung bedeutend effektiver, da dem Benutzer Tipparbeit erspart bleibt. Weiterer Vorteil ist die Fehlervermeidung durch Tippfehler. Deshalb Einstufung in Klasse „Conditional“.

- Suchen/Ersetzen

  Beschreibung:

  Ein Skript kann nach Zeichenketten durchsucht werden. Die gefundenen Ergebnisse werden farblich hervorgehoben und es kann zu den Ergebnissen gesprungen werden. Alle gefundenen Ergebnisse können ersetzt werden durch eine neue Zeichenkette.

  Bewertung:

  Das Auffinden von Variablen und Methoden im Skript wird erheblich erleichtert. Bei Umbenennung von Variablen oder Methoden wird viel

Zeit eingespart und es werden Tippfehler vermieden. Einteilung in Klasse „Conditional“.

- Letzte Schritte rückgängig machen

  Beschreibung:

  Die letzten vom Benutzer vorgenommenen Anweisungen können rückgängig gemacht werden.

  Bewertung:

  Fördert die Fehlertoleranz. Einteilung in Klasse „Conditional“

- Paralleles Öffnen von Skripten

  Beschreibung:

  Mehrere Skripte lassen sich parallel über Tabs öffnen.

  Bewertung:

  Erhöht die Usability (effektives arbeiten). Es kann einfacher Code zwischen verschiedenen Skripten kopiert werden. Klasse „Conditional“

- Anzeige aller instanziierten Variablen im Skript

  Beschreibung:

  Alle instanziierten Variablen werden aufgelistet. Doppelklick auf einen Variablennamen in der Liste markiert im Skript alle Verwendungen der Variable. Per Drag&Drop kann ein Variablenname aus der Liste in das Skript eingefügt werden.

  Bewertung:

  Benutzer hat alle instanziierten Variablen im Blick und kann leicht zu Verwendungsstellen im Skript navigieren. Das Einfügen per Drag&Drop bietet eine komfortable Möglichkeit der Variablenverwendung und verhindert Fehleingaben durch Tippfehler. Einteilung in Klasse „Optional“.

- Anzeige aller implementierten Funktionen zur Navigation im Skript

  Beschreibung:

  Alle implementierten Funktionen werden gelistet. Doppelklick auf eine Funktion führt zu einem Sprung zur Funktionsimplementierung. Per Drag&Drop kann ein Funktionsname in das Skript gezogen werden, ein Aufruf der Funktion wird in das Skript eingefügt. Es können alle Stellen im Skript markiert werden, an denen die gewählte Funktion aufgerufen wird.

  Bewertung:

Benutzer hat alle implementierten Funktionen im Blick und kann leicht zu Verwendungsstellen im Skript navigieren. Komfortable Möglichkeit der Funktionsverwendung, Verhindert Fehleingaben durch Tippfehler. Einteilung in Klasse „Optional“.

- Bei der Codeentwicklung unterstützende Funktionen

  Beschreibung:

  Es sollen Funktionen integriert werden, die schnelleres und effektiveres Schreiben von Code ermöglichen. Bsp: Autovervollständigung von Variablen, Funktionen und Anweisungen bei der Tastatureingabe

  Bewertung:

  Einteilung in Klasse „Conditional“.

- Routingeditor

  Beschreibung:

  Beschreibung für CAN hier. Auch für LIN gibt es Dokumente, die Botschaften definieren. Die Dokumente könnten geparst werden und darauf ein PAWN-Skript generiert werden, das die Botschaften als Variablen enthält.

  Aus einer Datenbankdatei (.dbc/.arxml) können dort definierte Bussignale geladen werden. Die Signale können in verschiedenen CAN-Botschaften enthalten sein. Aus den Bussignalen kann im Routingeditor eine eigene CAN-Busbotschaft konfiguriert werden mit einer vom Benutzer definierten ID. Dazu können die Signale innerhalb der Nutzdatenfeldes einer PDU frei angeordnet werden.

  Aus der im Routing-Editor konfigurierten Botschaft kann PAWN-Code generiert werden und in die Funktion OnCanRxEvent() eines geöffneten Skriptes im Texteditor eingefügt werden.

  Bewertung:

  Beim Programmieren der GIGABOX tritt des öfteren der Use-Case auf, dass aus Signalen, die in verschiedenen auf dem Bus ankommenden Botschaften enthalten sind, eine neue Botschaft erstellt und verschickt werden soll.

  Dabei werden oft Bitverschiebungen benötigt, deren Programmierung viel Zeit- und Denkaufwand bedeuten. Der Routingeditor ermöglicht die grafische Konfiguration von Busbotschaften und erleichtern die Umsetzung dieser Routingaufgaben. Einteilung in Klasse „Optional“.

3.2.4 Konfigurieren einer GIGABOX ohne PAWN-Kenntnisse

Beschreibung:

Grundlegende Funktionen einer GIGABOX sollen ohne Kenntnisse von PAWN konfiguriert werden können. Dazu bietet sich die Funktionsbausteinsprache an.

Bewertung:

Erleichtert es Kunden ohne Programmierkenntnisse, selbständig GIGABOX-Funktionalitäten zu konfigurieren. Erweitert möglicherweise den Kreis an potentiellen Käufern der GIGABOX. Im Vergleich zur direkten Codeentwicklung können allerdings nur eingeschränkte Funktionalitäten realisiert werden. Erfahrene Softwareentwickler werden aufgrund dieser Tatsache PAWN-Code direkt entwickeln. Einteilung in Klasse „Optional“.

3.2.5 Verbundene GIGABOXEN auflisten

Beschreibung:

Alle GIGABOXEN, die mit dem PC per USB oder Bluetooth verbunden sind, werden aufgelistet. Es werden Geräteinformationen wie z.B. Modellname, Seriennummer bereitgestellt. Der Benutzer kann aus der Liste eine GIGABOX anwählen, mit der kommuniziert werden soll.

Bewertung:

Verbindung zu GIGABOX per USB herstellen wird als Grundfunktionalität eingestuft, Klasse „Essential“. Verbindung per Bluetooth wird in Klasse „Optional“ eingestuft.

3.2.6 Kompilieren von Pawn-Quellcodedateien

Beschreibung:

PAWN-Skripte können in Bytecode übersetzt werden. Wenn Include-Files verwendet werden, soll ein Linker den Bytecode des Skriptes und des Include-Files nach dem Übersetzen zusammenfügen.

Bewertung:

Das Kompilieren ist eine Grundfunktionalität und wird deshalb in die Klasse „Essential“ eingestuft.

3.2.7 Beschreiben/Auslesen des Flash-Speichers der GIGABOX

Beschreibung:

Kompilierte Skript-Applikationen können vom PC auf den Flash-Speicher einer GIGABOX übertragen werden. Sich bereits auf dem Flash-Speicher befindliche Skripte können auf den PC geladen und angezeigt werden. Die Übertragung soll jeweils per Kabel über die USB-Schnittstelle und per Bluetooth möglich sein.

Bewertung:

Die Übertragung von Skripten vom PC auf eine GIGABOX über die USB-Schnittstelle wird als Grundfunktionalität bewertet und in Klasse „Essential“ eingeteilt. Die Übertragung von der GIGABOX auf den PC über die USB-Schnittstelle ist für den Entwickler sehr nützlich, aber nicht unbedingt notwendig und wird deshalb als Klasse „Conditional“ eingestuft. Bluetooth-Übertragung wird in Klasse „Optional“ eingestuft, da es wenige Use-Cases gibt, bei denen eine Funkübertragung Vorteile gegenüber der Kabelübertragung bietet. Ein möglicher Use-Case wäre eine schwer zugängliche GIGABOX im Fahrzeug, auf die ein neues Skript aufgespielt werden soll. Eine Übertragung per Bluetooth würde einen aufwändigen Ausbau der GIGABOX aus dem Fahrzeug ersparen.

3.2.8 Kommunikation mit einer GIGABOX

Beschreibung:

Benutzer soll Befehle an eine GIGABOX senden können, z. B. einen Reset der GIGABOX veranlassen, den Bootloader aufrufen. Außerdem sollen Diagnose- und Geräteinformationen abgerufen werden können. Die Datenübertragung soll per Kabel über die USB-Schnittstelle und per Bluetooth möglich sein.

Bewertung:

Das Senden von Befehlen und Abrufen von Diagnose- und Geräteinformationen per Kabel über die USB-Schnittstelle wird in Klasse „Essential“ eingestuft. Die Datenübertragung per Bluetooth wird in Klasse „Optional“ eingestuft, da es wenige Use-Cases gibt, bei denen eine Funkübertragung Vorteile gegenüber der Kabelübertragung bietet. Ein möglicher Use-Case wäre eine schwer zugängliche GIGABOX im Fahrzeug, auf die ein neues Skript aufgespielt werden soll. Eine Übertragung per Bluetooth würde einen aufwändigen Ausbau der GIGABOX aus dem Fahrzeug ersparen.

3.2.9 Kommunikation mit dem Benutzer

Beschreibung:

Benutzer soll Befehle über die Konsole an die IDE senden können, z. B. das Kompilieren eines Skriptes veranlassen. Außerdem sollen Diagnose- und Programminformationen der IDE abgerufen werden können.

Bewertung:

Einstufung in Klasse „Essential“.

3.2.10 Steuern und Beobachten von Ein-/Ausgängen und Timern

Beschreibung:

Die Zustände der digitalen und analogen Ausgänge sowie der internen Timer sollen über die Oberfläche der Entwicklungsumgebung gesteuert und beobachtet werden können. Die Zustände der digitalen und analogen Eingänge sollen über die Oberläche der Entwicklungsumgebung beobachtet werden können. Außerdem sollen vom Benutzer konfigurierte CAN- und LIN-Botschaften über die IDE gesendet werden können. Dabei soll einmaliges und zyklisches Senden von Botschaften möglich sein. Auf dem Bus liegende Botschaften einer definierten ID (CAN) bzw. für eine definierte Adresse (LIN) sollen beobachtet werden können.

Zusätzlich soll die Möglichkeit bestehen, Buskommunikation innerhalb eines definierten Zeitfensters mitzuloggen. Alle Busbotschaften, die innerhalb dieses Zeitfensters auf dem Bus liegen, sollen dabei mit Zeitstempel gelistet werden.

Bewertung:

Die beschriebenen Funktionen sind für den Nutzer hilfreich bei der Codeentwicklung, aber nicht unbedingt notwendig. Damit können die Funktionen in die Prioritätsklasse „Optional“ eingeteilt werden.

3.2.11 Debugging

Beschreibung:

Es sollen Breakpoints gesetzt werden können. Wenn ein Breakpoint erreicht wird, wird die Codeausführung auf der virtuellen Maschine der GIGABOX angehalten. Da der Codeausführung eventbasiert ist, werden die gesetzten Break-

points nur erreicht, wenn das Event ausgelöst wird, in dem sich der Breakpoint befindet. Events, die auslösen während die Codeausführung angehalten ist, müssen ignoriert werden. -> führt zu Problemen, da Timer ablaufen aber nicht neu aufgezogen werden -> Timerevents müssten trotzdem im Hintergrund auslösen und neu aufgezogen werden. Aber Anweisungen in den Event außer TimerSet() werden ignoriert.

CAN und LIN-Events werden ignoriert.

Der Benutzer kann zeilenweise zu den folgenden Anweisungen springen.

Bewertung:

Die beschriebenen Funktionen sind für den Nutzer hilfreich bei der Codeentwicklung, aber nicht unbedingt notwendig. Damit können die Funktionen in die Prioritätsklasse „Optional“ eingeteilt werden.

3.2.12 Bereitstellung von Dokumentationen

Beschreibung:

Die IDE soll Hilfedokumentation zu den GIGABOX-Modellen, zum configurAIDER und zu PAWN bereitstellen.

Bewertung:

Ein direkter Zugriff von der IDE aus auf die Hilfedokumentation erhöht die Usability. Dies bietet dem Benutzer den Vorteil, dass er die Oberfläche der IDE nicht verlassen muss um Dokumentationen aufzurufen. Einteilung in Klasse „Conditional“.

3.2.13 Individualisierbare Oberfläche

Beschreibung:

Die Fenster innerhalb der IDE sind frei andockbar. Somit kann der Benutzer die Oberfläche individuell nach seinen Wünschen aufbauen.

Bewertung:

Steigert die Usability (Individualisierbarkeit), ist aber keine unbedingt notwendige Funktion und wird deshalb als „Optional“ eingestuft.

3.3 Ermittlung von nichtfunktionalen Anforderungen

- Entwicklung des Tools mit WPF unter Verwendung MVVM-Pattern
- Entwicklung für Windows 32bit + 64bit als Zielbetriebssystem
- Entwicklung nach V-Modell
- Qualitätskriterien an Software nach ISO 9126 (aus Wiki)/ siehe auch Kapitel 6 Moderne Softwarearchitektur
  - Wartbarkeit
    - Analysierbarkeit:
      Coding-Richtlinien sollen eingehalten werden: Verständliche Kommentare, einheitliche Namensgebung etc
    - Modifizierbarkeit:
      SOLID-Prinzipien einhalten, um möglichst wenig Abhängigkeiten unter den Klassen zu erreichen. Dadurch treten bei Modifikationen innerhalb einer Klasse weniger schwer vorhersehbare Fehler auf
    - Testbarkeit:
      Keine Ahnung auf was bei der Entwicklung geachtet werden soll um gute Testbarkeit zu erreichen
  - Benutzbarkeit (genauer definiert in Norm EN ISO 9241-110)
  - Effizienz
    - Akzeptable Zeit bis Tool gestartet ist und verwendbar ist
    - Unmittelbare und flüssige Reaktion auf Benutzereingaben
  - Übertragbarkeit
    - Anpassbarkeit: Möglichkeit, Software an verschiedene Umgebungen anzupassen
      Verschiedene Betriebssysteme? 32/64Bit?
  - Zuverlässigkeit
    - Reife: Geringe Versagenshäufigkeit durch Fehlerzustände: Ausreichendes Testing
    - Wiederherstellbarkeit: Fähigkeit, bei einem Versagen das Leistungsniveau wiederherzustellen und die direkt betroffenen Daten wiederzugewinnen. Zu berücksichtigen sind die dafür benötigte Zeit und der benötigte Aufwand.

Kapitel 4

Ist-Zustand

4.1 Überblick

In diesem Kapitel wird der Ist-Zustand des configurAIDERs beschrieben. Es wird die Benutzeroberfläche vorgestellt und die Funktionalität beschrieben. Im Anschluss wird die Funktionalität und die Usability bewertet. Abbildung 4.1 veranschaulicht die Module, die im configureAIDER enthalten sind.

4.2 Elemente des configurAIDER

4.2.1 Rahmenfenster

Der configurAIDER besteht aus einem Rahmenfenster, in das weitere Fenster eingebettet werden können. Das Rahmenfenster besitzt im oberen Fensterbereich eine Menüleiste in horizontaler Ausrichtung und ist in einem Hintergrund gehalten, der Firmenlogo und -farbe zeigt. Die Menüeinträge ändern sind dynamisch in Abhängigkeit der vom Benutzer geöffneten Fenster. Beim Start des confiurAIDERs ist das Fenster „Verfügbare Geräte“ standardmäßig in das Rahmenfenster eingebettet. Eingebettete Fenster können nicht über die Grenzen des Rahmenfensters hinausgezogen werden.

4.2.2 Fenster „Verfügbare Geräte“

Im Fenster „Verfügbare Geräte“ werden alle GIGABOXen angezeigt, die an der USB-Schnittstelle des PC angeschlossen sind. Es wird der Modellname, die

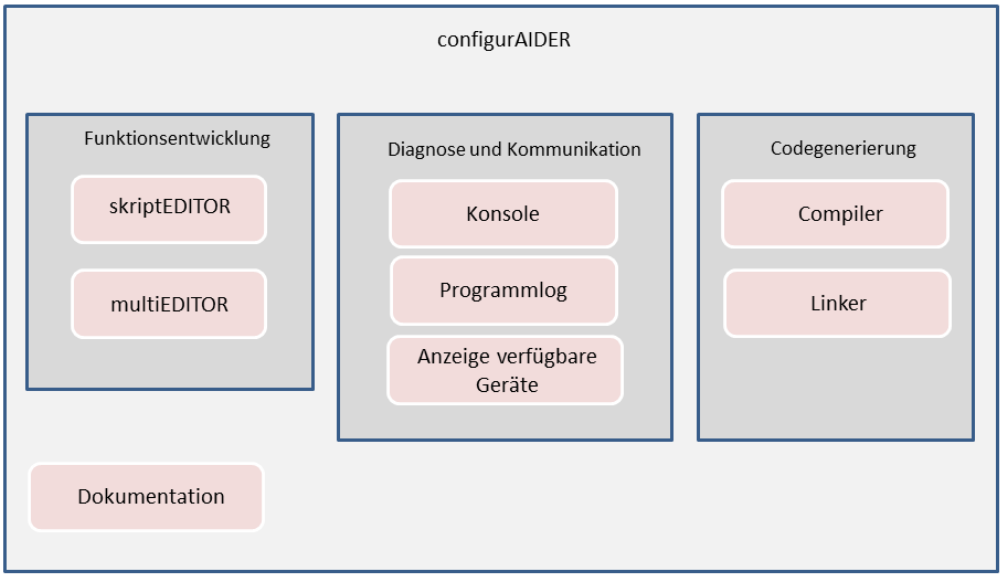

Abbildung 4.1 – Übersicht configurAIDER

Seriennummer und der Verbindungsstatus von GIGABOXen ausgegeben, die am PC erkannt wurden. Über dieses Fenster kann eine GIGABOX angewählt werden, mit der eine Kommunikation gestartet werden soll. Wenn keine reale GIGABOX zur Verfügung steht, kann eine GIGABOX simuliert werden. Die Auswahl eines zu simulierenden Gerätes erfolgt über ein eigenes Dialogfenster „Simuliertes Gerät“ (siehe Abbildung 4.2).

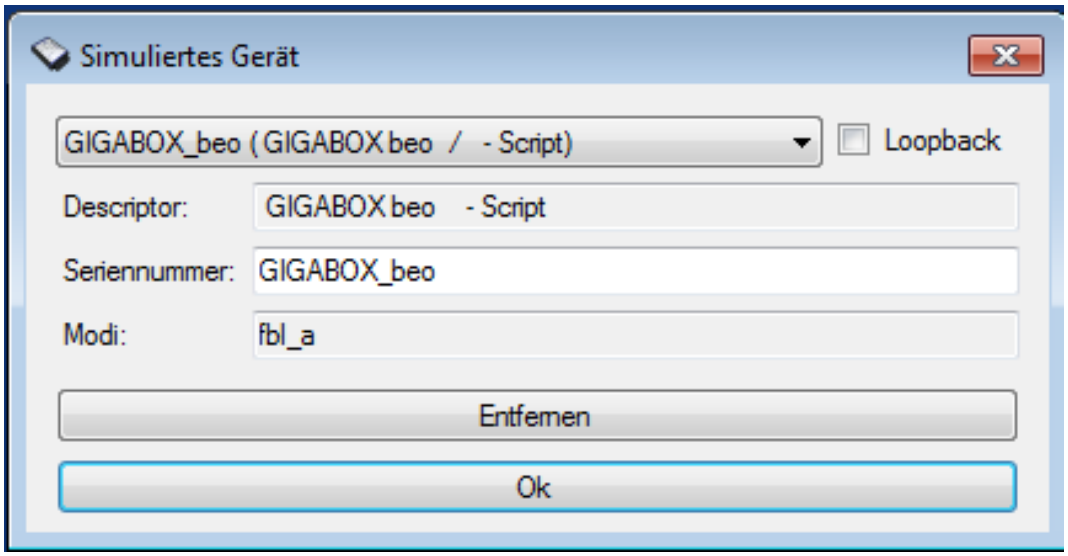

Abbildung 4.2 – Fenster „Simuliertes Gerät“

Die simulierte GIGABOX wird wie eine reale GIGABOX gelistet.

Nach der Anwahl eines GIGABOX-Modelles wird versucht, eine Verbindung zu dem Gerät herzustellen (wird das tatsächlich hardwaremäßig versucht oder ist die Anzeige nur dazu da, dem Benutzer zu signalisieren, welches Gerät er konfiguriert). Bei erfolgreicher Verbindungsherstellung wird das Gerät grün hinterlegt und es öffnet sich ein Fenster zur Editorauswahl. Dort kann ausgewählt werden, ob die Konsole, der scriptEDITOR oder der multiEDITOR geöffnet werden soll. Bei einigen GIGABOX-Modellen steht kein multiEDITOR zur Verfügung.

4.2.3 Fenster „Konsole“

Die Konsole besteht aus einer Textbox zur Eingabe von Befehlen und einer Textbox zur Ausgabe von Informationen. Sie ermöglicht Kommunikation mit einer GIGABOX. Die Konsole kann z.B eingesetzt werden zur Abfrage von Diagnoseinformationen und um einen Reset zu veranlassen.

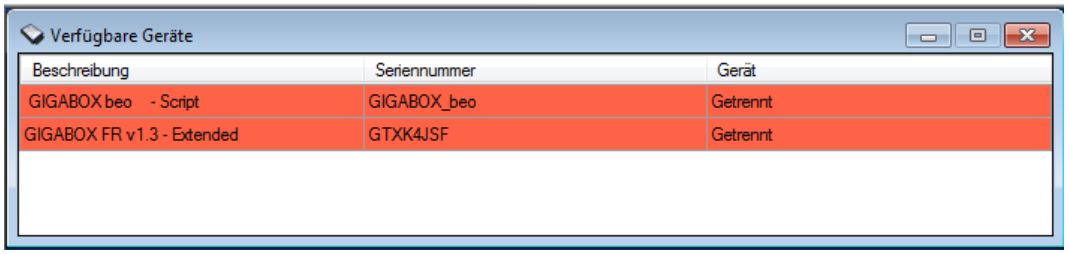

Abbildung 4.3 – Fenster „Verfügbare Geräte“

Über die Schaltflächen „Folgen“ und „Terminal“ kann gewählt werden, wie die Textein- und ausgabe erfolgen soll. Wenn „Folgen“ ausgewählt wird, erfolgt die Texteingabe in einem von der Textausgabe getrennten Fenster. Wenn „Terminal“ ausgewählt wird, erfolgt die Textein- und ausgabe in einem gemeinsamen Fenster.

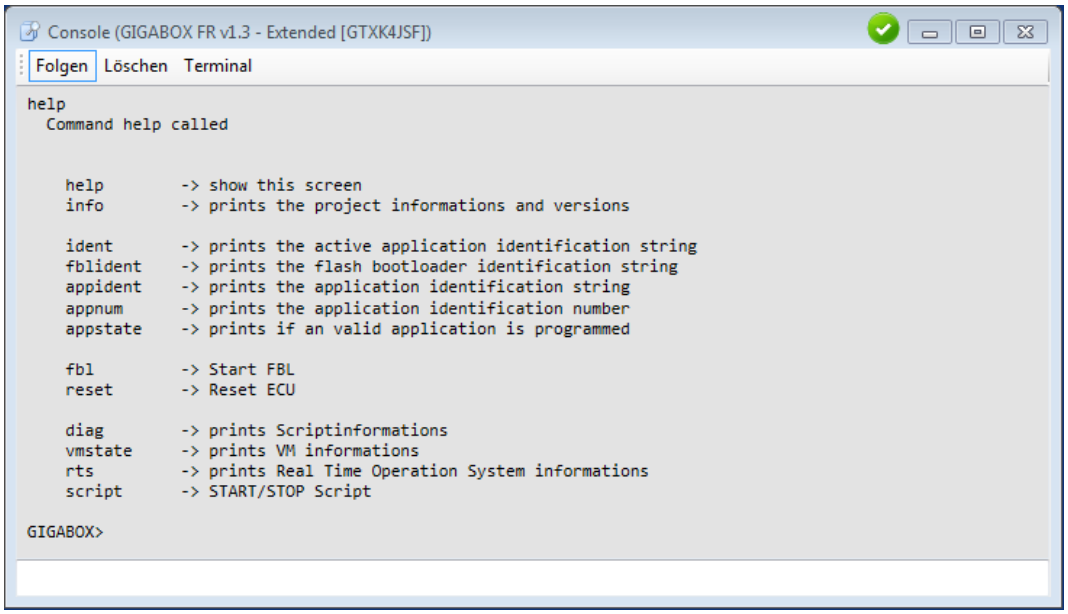

Abbildung 4.4 – Fenster „Konsole“

4.2.4 Fenster „Programmlog“

Das „Programmlog“-Fenster besteht aus einer Textbox, über die tabellarisch Erfolgs- und Fehlermeldungen über die ausgeführten Aktionen des configurAIDERs ausgegeben werden.

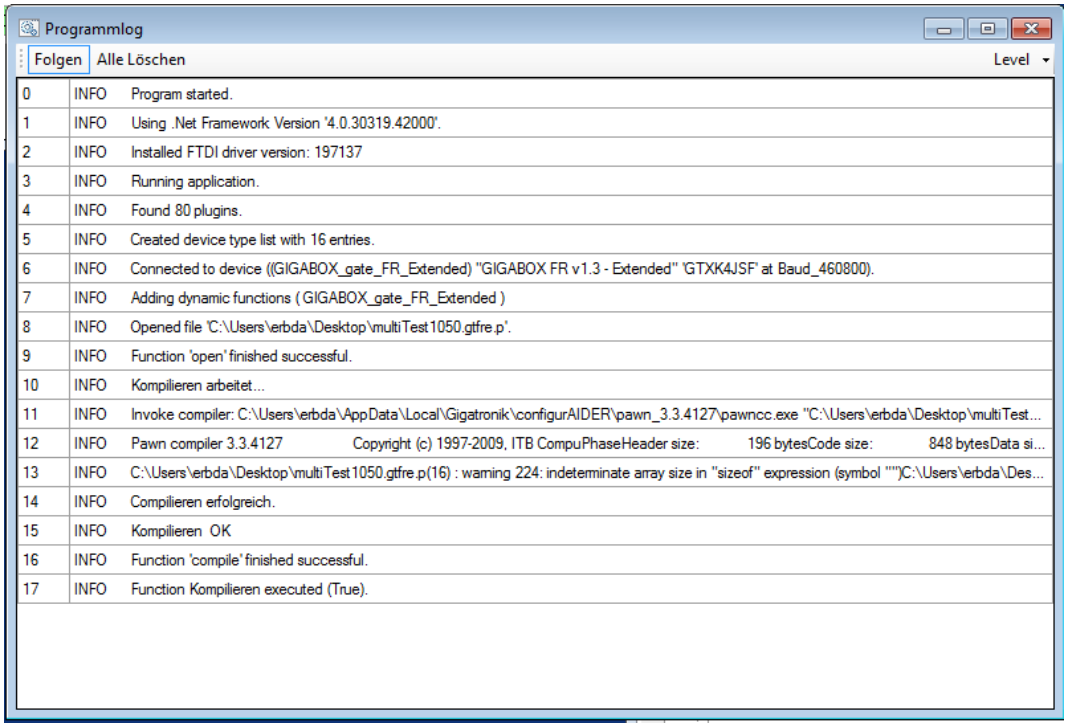

Abbildung 4.5 – Fenster „Programmlog“

4.2.5 Fenster „scriptEDITOR“

Der scriptEDITOR ist ein Texteditor zum Erstellen und Editieren von Funktionen in PAWN. Um den scriptEDITOR aufzurufen, muss zuerst eine Verbindung zu einer realen oder simulierten GIGABOX hergestellt werden über das Fenster „Verfügbare Geräte“. Anschließend öffnet sich automatisch ein Fenster zur Editorauswahl. Dort kann der scriptEDITOR angewählt werden. Mit dem scriptEDITOR erstellte Skriptdateien haben die Dateiendung .gt**.p. Die Sterne sind zu ersetzen mit einem Kürzel, das abhängig ist vom verbundenen GIGABOX-Modell. Für die unterschiedlichen GIGABOX-Modelle werden also Skripte mit verschiedenen Dateiendungen erstellt, da die Modelle unterschiedliche Funktionen implementieren. Bsp: GIGABOX gate FR extended: .gtfre.p GIGABOX gate XL: .gtxl.p GIGABOX gate gateway: .gtfrg.p

In der obersten Fensterzeile ist eine Buttonleiste angeordnet mit den folgenden Buttons:

- Neu: Öffnet ein neues Skript
- Öffnen: Öffnet ein vorhandenes Skript

- Speichern: Speichert das geöffnete Skript unter einem bereits festgelegt Dateinamen und Pfad
- Speichern unter: Speichert das geöffnete Skript unter einem anzugebenden Dateinamen und Pfad
- Kompilieren: Interpreter erzeugt Bytecode aus dem PAWN-Quellcode
- Herunterladen: Interpreter erzeugt Bytecode aus dem PAWN-Quellcode. Die PAWN-Skriptdatei wird als ZIP-Datei komprimiert. Anschließend wird der Bytecode und das gezippte Skript auf die virtuelle Maschine der GIGABOX übertragen.
- Heraufladen: Die als ZIP-Datei komprimierte Skriptdatei auf dem Steuergerät wird auf den PC übertragen, entzippt und im Texteditor geöffnet.
- Reset: Führt einen Reset der GIGABOX durch
- Log: Öffnet Konsole

Aufzählung der Buttons zu detailreich Mittig ist der Texteditor angeordnet. Hier kann ein Skript angezeigt und bearbeitet werden. Es stehen bei der Entwicklung typische Unterstützungswerkzeuge wie Codefolding zur Verfügung. (Aufzählung der unterstützenden Werkzeuge wie Autovervollständigung etc.)

Unten befindet sich ein Fenster zur Ausgabe von Fehlern, Warnungen und Erfolgsmeldungen.

4.2.6 Fenster „multiEDITOR“

Der „multiEDITOR“ ist ein Editor zur Funktionskonfiguration mit WENN-DANN-Anweisungen in Tabellen. Er soll es Benutzern mit geringer Programmiererfahrung ermöglichen, einfache Funktionen für die GIGABOX zu entwickeln. Aus den konfigurierten Funktionstabellen kann anschließend automatisch ein PAWN-Skript generiert werden. Das erstellte PAWN-Skript kann dann kompiliert und der Bytecode auf den Flash-Speicher der GIGABOX übertragen werden.

In der obersten Fensterzeile ist eine Buttonleiste angeordnet mit den folgenden Buttons:

- Neu: Öffnet ein neues Skript
- Öffnen: Öffnet ein vorhandenes Skript

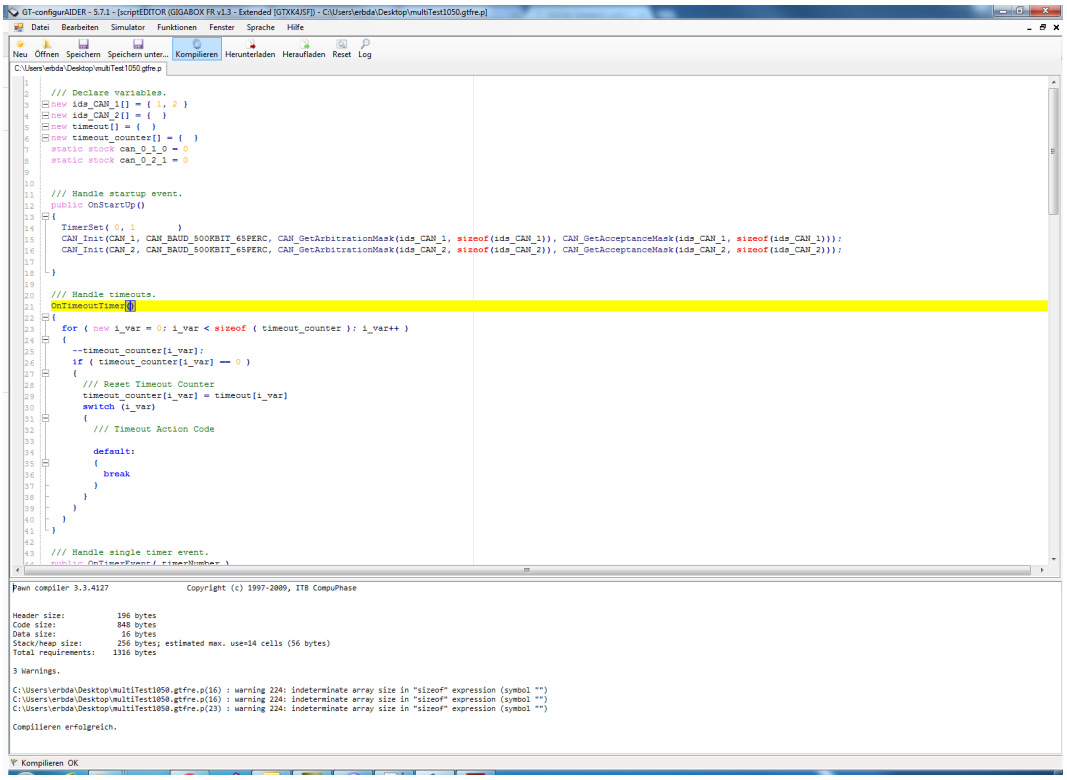

Abbildung 4.6 – Fenster „scriptEDITOR“

- Speichern: Speichert das geöffnete Skript unter einem bereits festgelegt Dateinamen und Pfad
- Speichern unter: Speichert das geöffnete Skript unter einem anzugebenden Dateinamen und Pfad
- Generieren: Aus den mittels Tabelle konfigurierten Funktionen wird ein PAWN-Skript generiert
- Kompilieren: Interpreter erzeugt Bytecode aus dem PAWN-Quellcode
- Herunterladen: Aus den mittels Tabelle konfigurierten Funktionen wird ein PAWN-Skript generiert. Interpreter erzeugt Bytecode aus dem PAWN-Quellcode. Die PAWN-Skriptdatei wird als ZIP-Datei komprimiert. Anschließend wird der Bytecode und das gezippte Skript auf die virtuelle Maschine der GIGABOX übertragen.
- Heraufladen: Die als ZIP-Datei komprimierte Skriptdatei auf dem Steuergerät wird auf den PC übertragen und entzippt. PAWN-Code wird rückgewandelt in Tabelle mit WENN-DANN-SONST-Anweisungen. Die funktioniert nur, wenn der von der GIGABOX hochgeladenen Code ursprünglich mithilfe des multiEDITORs erzeugt wurde.
- DBC: Es können Vector-DBC Dateien oder AUTOSAR .arxml Dateien erstellt oder geöffnet werden. Dabei handelt es sich um Datenbanken, in denen CAN-Botschaften und Signale definiert sind
- Reset: Führt einen Reset der GIGABOX durch
- Log: Öffnet Konsole

Aufzählung der Buttons zu detailreich

Unter der Buttonleiste befindet sich eine Tabelle, in der CAN-Botschaften und Signale erstellt und bearbeitet werden können. Botschaften und Signale, die aus Datenbank-Dateien (.dbc oder .arxml) geladen wurden, werden hier angezeigt.

Darunter befindet sich eine Tabelle zur Konfiguration von Funktionen mithilfe von WENN-DANN-SONST-Anweisungen.

4.3 Funktionalität

Abbildung 4.8 gibt einen Überblick über die verfügbaren Funktionen des configurAIDERs.

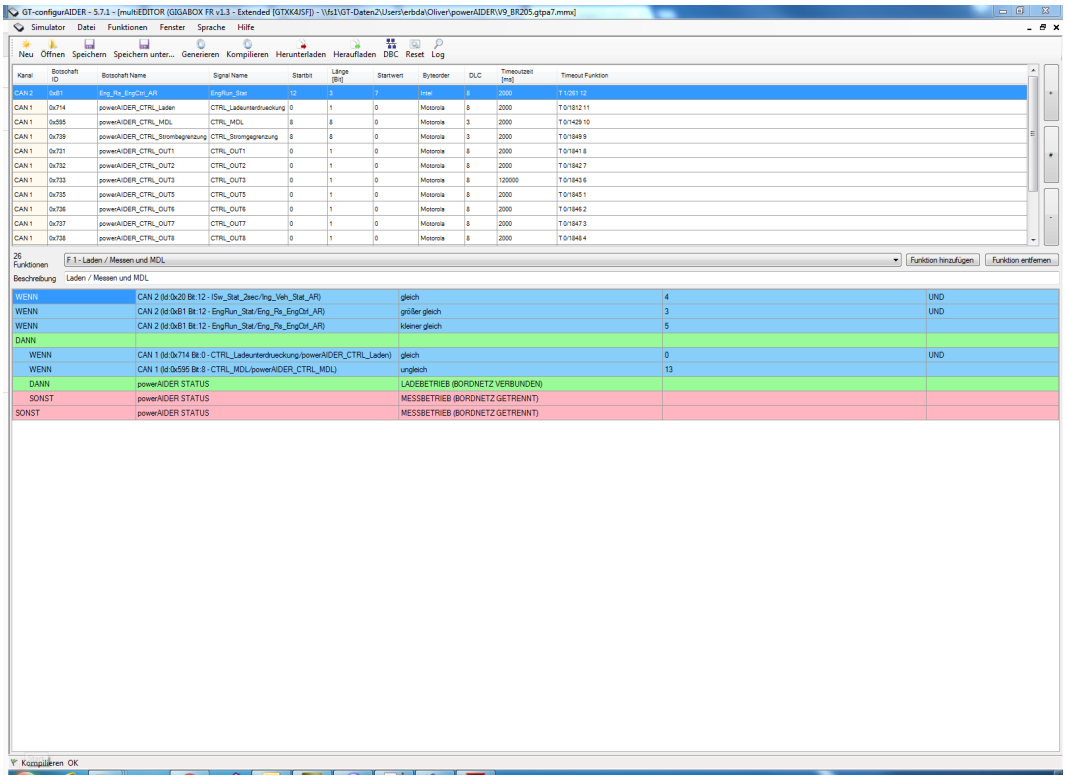

Abbildung 4.7 – Fenster „multiEDITOR“

Abbildung 4.8 – Überblick Funktionen des configurAIDERs

4.3.1 Einstellung der Sprache

Die Sprache der Fensterdialoge des configurAIDERs kann in der Menüleiste des Rahmenfenster unter dem Eintrag „Sprache“ eingestellt werden. Es steht Deutsch und Englisch als Sprache zur Verfügung. Nach der Sprache von Deutsch auf Englisch wird ein Dialog eingeblendet, dass die Sprachänderung erst nach einem Neustart übernommen wird. Nach ausgeführtem Neustart werden nicht alle Menüeinträge auf Englisch dargestellt. So werden beispielsweise im scriptEDITOR die Menüeinträge unter „Edit“ weiterhin komplett in Deutsch dargestellt. Die Funktionstabellen im multiEDITOR werden auch weiterhin in Deutsch dargestellt.

4.3.2 Bereitstellung von Dokumentation zur GIGABOX, configurAIDER und PAWN

Über den Menüeintrag „Hilfe“ können PDF-Dokumente zu den verschiedenen GIGABOX-Modellen, configurAIDER und zur Skriptsprache PAWN abgerufen werden. Für den configurAIDER steht ein allgemeines Handbuch, Hinweise zur Installation und jeweils für scriptEDITOR und multiEDITOR ein eigenes Handbuch zur Verfügung. Der Menüeintrag „linEDITOR“ enthält nur einen ausgegrauten Eintrag. Zu den GIGABOX-Modellen stehen Handbücher, Flyer und Informationen zu Firmwareupdates zur Verfügung. Die Dokumentationen werden nicht durchgängig für alle GIGABOX-Modelle zur Verfügung gestellt. So wird für die GIGABOX gate xl kein Handbuch angeboten, für die GIGABOX gate powerAIDER kein Flyer.

4.3.3 Verbindung zu einer GIGABOX herstellen

GIGABOXEN, die an der USB-Schnittstelle des PCs erkannt werden sowie simulierte GIGABOXEN werden im Fenster „Verfügbare Geräte“ (Abbildung 4.3) gelistet. Hier kann die Verbindung zu einem Gerät hergestellt werden. Falls keine reale GIGABOX zur Verfügung steht, kann eine GIGABOX simuliert werden.

Die verschiedenen Modelle besitzen unterschiedliche Hardwarefunktionen (unterschiedliche Anzahl von Ein-/Ausgängen, Bussysteme), die zu entwickelnden Funktionen müssen daran angepasst werden. Dem configurAIDER muss deshalb vor der Funktionsentwicklung bekannt sein, für welches Modell Code entwickelt werden soll.

Es wurden verschiedene Fehler entdeckt, die im Folgenden aufgelistet werden.

1. Das Fenster „Verfügbare Geräte“ wird vom Benutzer geschlossen. Bei einem Wiederaufruf des Fensters kann kein neues Gerät mehr simuliert werden. Lösung: configurAIDER neu starten

2. Im Fenster „Verfügbare Geräte“ sind mehrere Geräte gelistet. Bei dem Versuch, die Verbindung zu einem Gerät herzustellen über Rechtsklick auf das Gerät und „Verbinden“, wird keine Verbindung zum angewählten Gerät hergestellt sondern zum ersten Gerät in der Liste. Lösung: Verbindung herstellen über Doppelklick auf Gerät

3. Die Simulation von Geräten schlägt gelegentlich fehl, es wird über das Programmlog-Fenster ein Fehler ausgegeben (ERROR: Internal device table obscurred!) Lösung: Neuer Versuch der Herstellung einer Verbindung

4.3.4 PAWN-Code kompilieren

Aus scriptEDITOR und multiEDITOR heraus kann mithilfe des Compilers PAWN-Code in Bytecode übersetzt werden. Der erzeugte Bytecode kann auf der virtuellen Maschine der GIGABOX ausgeführt werden.

4.3.5 PAWN-Code entwickeln

PAWN-Skripte schreiben

PAWN-Skripte können im scriptEDITOR geschrieben werden. Erstellte Skriptdateien besitzen die Dateiendung .gt**.p. Die Sterne sind zu ersetzen mit einem Kürzel, das abhängig ist vom verbundenen GIGABOX-Modell. Für die unterschiedlichen GIGABOX-Modelle werden also Skripte mit verschiedenen Dateiendungen erstellt, da die Modelle unterschiedliche Funktionen implementieren. Bsp: GIGABOX gate FR extended: .gtfre.p GIGABOX gate XL: .gtxl.p GIGABOX gate gateway: .gtfrg.p

Um den Benutzer bei der Codeentwicklung zu unterstützen, sind aus anderen IDEs bekannte Funktionen integriert. Einige ausgewählte Funktionen sind hier beispielhaft angeführt:

- Codeabschnitte sind auf- und zuklappbar
- Unterschiedliche Farbgebung (Anweisungen in blau, Kommentare in grün etc)

- Eventfunktionen sind als Rumpf vorimplementiert
- Markierung der Zeile, in der sich der Cursor aktuell befindet

Im Vergleich zu weit verbreiteten IDEs wie Visual Studio ist der Umfang an unterstützenden Funktionen aber klein gehalten.

PAWN-Skripte aus Funktionstabellen generieren

Im multiEDITOR können tabellarisch Funktionen konfiguriert werden mit WENN-DANN-SONST-Anweisungen. Aus den konfigurierten Funktionstabellen kann anschließend automatisch ein PAWN-Skript generiert werden. Das erstellte PAWN-Skript kann dann kompiliert und der Bytecode auf den Flash-Speicher der GIGABOX übertragen werden.

Für jede Funktion wird eine neue Tabelle erstellt. Jede Tabelle besteht mindestens aus 3 Zeilen mit den Anweisungen WENN, DANN und SONST. Es können mehrere WENN-Anweisungszeilen erstellt werden, die mit dem Bedingungsoperator UND oder ODER verknüpft werden müssen. Außerdem können mehrere DANN und SONST-Anweisungen erstellt werden, die jeweils miteinander über den UND-Operator verknüpft werden. Unter jeweils jede DANN und SONST-Anweisung kann eine neue, untergeordnete WENN-DANN-SONST-Anweisung eingefügt werden. Damit sind verschachtelte Anweisungen möglich.

In den Spalten der WENN-Anweisungszeile können digitale und analoge Eingänge, digitale Ausgänge sowie eingehende Busbotschaften auf definierte Werte geprüft werden. Wenn die dort definierten Werte angenommen werden, wird die DANN-Anweisungenszeile ausgeführt. Hier können digitale Ausgänge durchgeschaltet oder ausgeschaltet werden. Das Absetzen von Busbotschaften ist nicht möglich.

Wenn die in der WENN-Anweisungszeile definierten Werte nicht angenommen werden, wird die SONST-Anweisungszeile ausgeführt. Auch hier können digitale Ausgänge durchgeschaltet oder ausgeschaltet werden.

Bewertung

4.3.6 Übertragen von Skript-Applikationen

Es kann eine Skript-Applikation auf die GIGABOX übertragen werden. Die Skriptdatei wird als ZIP-Datei komprimiert und zusammen mit dem Bytecode auf den Flash-Speicher geschrieben. Wenn ein Skript mithilfe des multiEDITORs generiert wurde, wird zusätzlich zu der Skriptdatei (.gt**.p) auch

die multiEDITOR-Konfigurationsdatei (.gt**.mmx) (+.tmp -> was steht da drin?) mit in die ZIP-Datei gepackt.

Ein auf der GIGABOX ausgeführtes Skript kann auch von der GIGABOX auf den PC geladen werden. Dabei wird die auf der GIGABOX gespeicherte ZIP-Datei auf den PC geladen, entpackt und im scriptEDITOR angezeigt. Wenn das Skript mit dem multiEDITOR erzeugt wurde, liegt der ZIP-Datei zusätzlich die multiEDITOR-Konfigurationsdatei (.gt**.mmx) bei und kann folglich auch im multiEDITOR geöffnet werden.

4.3.7 Kommunikaton mit verbundener GIGABOX

Über die Konsole können Befehle an eine reale GIGABOX gesendet werden, z.B. Befehl zum Reset, Aufrufen des Bootloaders. Dort können auch Informationen der GIGABOX abgerufen und ausgeben werden, z.B. Diagnoseinformationen über Bussysteme oder Timer.

Wenn die Schaltfläche „Terminal“ aktiviert ist, kann eine erfolgte Texteingabe nicht mehr korrigiert werden. Bei deaktivierter Schaltfläche ist die Korrektur möglich.

4.3.8 Ausgabe von Ereignismeldungen

Es kann über das Programmlog-Fenster nachverfolgt werden, welche Aktionen vom configurAIDER durchgeführt wurden. Wichtige Erfolgs- und Fehlermeldungen werden dort ausgegeben. Bsp: Erfolgreich kompiliert.

4.3.9 Ergebnis

Kapitel 5

Evaluation

5.1 Überblick

5.2 Funktionalität und Usability des Ist-Zustands

Es sollen die Funktionen des configurAIDERs bewertet werden hinsichtlich Umfang, Umsetzung und Fehlerzuständen. Zusätzlich soll die Usability nach EN ISO 9241-110 (siehe Abschnitt 2.3.2) bewertet werden.

5.2.1 Einstellung der Sprache

Es kann Deutsch und Englisch als Sprache eingestellt werden. Hier wäre zu überlegen, ob noch weitere Spracheinstellungen zur Verfügung gestellt werden sollten, falls es eine größere Anzahl an Benutzern aus nicht deutsch- oder englischsprachigen Ländern gibt. Die englische Spracheinstellung ist nicht konsistent umgesetzt, hier sollte darauf geachtet werden, dass durchgängig alle Bedienelemente englisch beschriftet werden.

5.2.2 Bereitstellung von Dokumentation zur GIGABOX, configurAIDER und PAWN

Über den Menüeintrag „Hilfe“ können PDF-Dokumente zu den verschiedenen GIGABOX-Modellen, configurAIDER und zur Skriptsprache PAWN abgerufen werden. Für den configurAIDER steht ein allgemeines Handbuch, Hinweise zur Installation und jeweils für scriptEDITOR und multiEDITOR ein

eigenes Handbuch zur Verfügung. Hier ist zu kritisieren, dass der Menüeintrag „linEDITOR“ nur einen ausgegrauten Eintrag enthält und somit unnötig ist. Zu den GIGABOX-Modellen stehen Handbücher, Flyer und Informationen zu Firmwareupdates zur Verfügung. Leider werden die Dokumentationen nicht durchgängig für alle GIGABOX-Modelle zur Verfügung gestellt. So wird für die GIGABOX gate xl kein Handbuch angeboten, für die GIGABOX gate powerAIDER kein Flyer.

5.2.3 Verbindung zu einer GIGABOX herstellen

Für den Benutzer ist nicht zu unterscheiden, ob ein Gerät in der Liste real oder simuliert ist. Simulierte Geräte werden in der Liste nicht hervorgehoben oder markiert. Die gewünschte Aufgabenangemessenheit ist für dieses Fenster damit nicht gegeben, da der Benutzer die Auswahl einer GIGABOX nicht effizient erledigen kann.

Die Auswahl eines zu simulierenden Gerätes erfolgt über ein eigenes Dialogfenster „Simuliertes Gerät“ (siehe Abbildung 4.2). Für den Benutzer ist in diesem Dialog nicht ersichtlich, was die Checkbox „Loopback“ bewirkt. Die Selbsterklärungsfähigkeit des Dialogs ist nicht ausreichend.

Bei der Auswahl eines zu simulierenden Gerätes im Fenster „Simuliertes Gerät“ wird das angewählte Gerät der Liste im Fenster „Verfügbare Geräte“ hinzugefügt, bevor die Auswahl mit „Ok“ bestätigt wird. Diese Reaktion entspricht nicht den üblichen, dem Benutzer bekannten Konventionen von Software und ist damit nicht erwartungskonform.

Aufgrund von diverse Fehlerzuständen, die beobachtet wurden, kann die Verbindung zu einem Gerät nicht zuverlässig und fehlerfrei hergestellt werden. Das Tool produziert Ergebnisse, die vom Benutzer so nicht erwartet werden und erfordert lästiges Neustarten des Programmes.

5.2.4 PAWN-Code kompilieren

Die Kompilierung funktioniert ohne Probleme.

5.2.5 PAWN-Code entwickeln

PAWN-Skripte schreiben

Keine Möglichkeit, letzte Aktion rückgängig zu machen -> schlechte Steuerbarkeit

Keine Erkennung von Syntaxfehlern -> schlechte Selbstbeschreibungsfähigkeit

Buttons Herunterladen/Heraufladen sind missverständlich -> schlechte Selbstbeschreibungsf ähigkeit

Nur ein Skript kann geöffnet werden, nicht mehrere gleichzeitig -> Aufgabenangemessenheit

Autovervollständigung nur teilweise umgesetzt -> Aufgabenangemessenheit

Codeblöcke können nicht auskommentiert werden über Buttons/Tastenkombination -> Aufgabenangemessenheit

Die Buttons „Herunterladen“ und „Heraufladen“ in der oberen Buttonleiste sind nicht klar verständlich benannt. Dem Benutzer wird nicht klar, in welche Richtung der Datenaustausch erfolgt. Der Button „Log“ öffnet die Konsole, die Erwartung wäre, dass sich das Programmlog-Fenster öffnet. Die Selbstbeschreibungsfähigkeit und Erwartungskonformität der Buttonleiste ist negativ zu bewerten.

PAWN-Skripte aus Funktionstabellen generieren

Im Signaleditor ist Bit/CAN-Bit Unterschied nicht klar ersichtlich -> Selbstbeschreibungsfähigkeit

Es können keine CAN-Nachrichten gesendet werden -> eingeschränkte Funktionalität

Keine Reaktion auf LIN-Nachrichten möglich/ kein Senden von LIN-Nachrichten möglich

Keine Möglichkeit, SWITCH CASE-Anweisung einzufügen. Deshalb müssen verschachtelte WENN DANN-Anweisungen erstellt werden, die schnell unübersichtlich werden

Es ist nicht ersichtlich, dass über den DBC-Button auf .arxml-Dateien geladen werden können

Bei Klick auf DBC-Button öffnet sich Fenster zum Laden von .dbc und .armxl-Dateien. In diesem Fenster befindet sich eine Tabelle zur Anzeige der geladenen

Signale. Bei Anwahl einer .dbc-Datei bleibt die Tabelle allerdings leer. Die im ausgewählten DBC definierten Signale werden dort nicht angezeigt

5.2.6 Übertragen von Skript-Applikationen

Das Übertragen von Skript-Applikationen funktioniert ohne Probleme.

5.2.7 Kommunikaton mit verbundener GIGABOX

Wenn die Schaltfläche „Terminal“ aktiviert ist, kann eine erfolgte Texteingabe nicht mehr mit der Rücktaste korrigiert werden. Bei deaktivierter Schaltfläche ist die Korrektur möglich. Die Konsole mangelt es folglich an Fehlertoleranz, was einer schlechten Usability entspricht.

5.2.8 Ausgabe von Ereignismeldungen des configurAIDERs

Die Ausgabe von Ereignismeldungen über das Programmlog-Fenster ist zufriedenstellend.

5.3 Differenz zwischen Anforderungen und Ist-Zustand

	Anforderungen	Priorität	Ist-Zustand
Erstellung von Projekten mit Baumstruktur	Innerhalb eines Projektes können verschiedene GIGABOX-Modelle angelegt werden . Für jedes angelegte GIGABOX-Modell können PAWN-Skripte erstellt werden mit der zum Modell passenden Dateiendung. Für jedes angelegte GIGABOX-Modell können Include-Files hinzugefügt werden.	Conditional	Nicht realisiert
Einstellung der Sprache	Deutsch	Optional	Realisiert
	Englisch	Essential	Realisiert, aber nicht konsistent umgesetzt
Oberfläche individualisierbar	Frei andockbare Fenster	Optional	Fenster sind frei verschiebbar, aber nicht andockbar
Verbindung zu einer GIGABOX herstellen	Alle GIGABOXEN, die mit dem PC per USB verbunden sind, werden aufgelistet. Es werden Geräteinformationen wie z.B. Modellname, Seriennummer bereitgestellt. Der Benutzer kann aus der Liste eine GIGABOX anwählen, mit der kommuniziert werden soll.	Essential	Realisiert im Fenster „Verfügbare Geräte“. Es treten Fehler auf.
	Alle GIGABOXEN, die mit dem PC per Bluetooth verbunden sind, werden aufgelistet. Es werden Geräteinformationen wie z.B. Modellname, Seriennummer bereitgestellt. Der Benutzer kann aus der Liste eine GIGABOX anwählen, mit der kommuniziert werden soll.	Optional	Nicht realisiert

PAWN-Compiler	Compiler zum Übersetzen von PAWN-Code in Bytecode. Linker um Bytecode zusammenzufügen.	Essential	Realisiert
Beschreiben/Auslesen des Flash-Speichers der GIGABOX	Übertragen der Skript-Applikation vom PC auf den Flash-Speicher der GIGABOX per USB	Essential	Realisiert
	Übertragen der Skript-Applikation vom PC auf den Flash-Speicher der GIGABOX per Bluetooth	Optional	Nicht realisiert
	Übertragen der Skript-Applikation vom Flash-Speicher der GIGABOX auf den PC per USB	Conditional	Realisiert
	Übertragen der Skript-Applikation vom Flash-Speicher der GIGABOX auf den PC per Bluetooth	Optional	Nicht realisiert

Kommunikation mit einer GIGABOX	Benutzer kann Geräte- und Diagnoseinformationen der GIGABOX abrufen per USB	Essential	Realisiert im Fenster "Konsole"
	Benutzer kann Geräte- und Diagnoseinformationen der GIGABOX abrufen per Bluetooth	Optional	Nicht realisiert
	Benutzer kann Befehle an eine GIGABOX senden per USB	Essential	Realisiert im Fenster "Konsole"
	Benutzer kann Befehle an eine GIGABOX senden per Bluetooth	Optional	Nicht realisiert
Kommunikation mit dem Benutzer	Benutzer kann Diagnose- und Programminformationen der IDE abrufen	Essential	Realisiert im Fenster "Programmlog" sowie im Ausgabefenster des scriptEDITOR und multiEDITOR
	Benutzer kann über Konsole Befehle an IDE senden	Essential	Nicht realisiert
Steuern- und Beobachten von Ein- und Ausgängen und Timern	Steuern und Beobachten der Zustände der digitalen und analogen Ausgänge über GUI der IDE. Beobachten der Zustände der digitalen und analogen Eingänge über GUI der IDE.	Optional	Nicht realisiert
	Steuern und Beobachten der internen Timer über GUI der DIE.	Optional	Nicht realisiert
	CAN-/LIN-Botschaften senden (einmalig und zyklisch) über GUI der DIE.	Optional	Nicht realisiert
	Auf dem Bus liegende Botschaften einer definierten ID (CAN) bzw. für eine definierte Adresse (LIN) sollen beobachtet werden können.	Optional	Nicht realisiert
	Buskommunikation kann innerhalb eines definierten Zeitfensters mitgeloggt werden.	Optional	Nicht realisiert
Debugging	Setzen von Breakpoints im Code	Optional	Nicht realisiert
	Zeilenweise Steuerung der Anweisungsausführung	Optional	Nicht realisiert
Bereitstellung von Dokumentationen	Dokumentation zur IDE, GIGABOX-Modellen und PAWN	Conditional	Realisiert

Texteditor zur Erstellung von PAWN-Skripten	Schreiben von PAWN-Code	Essential	Realisiert
	Paralleles Öffnen von Skripten über Tabs	Conditional	Nicht realisiert
	Copy/Paste	Conditional	Realisiert
	Suchen/Ersetzen	Conditional	Realisiert
	Möglichkeit, letzte Schritte rückgängig zu machen	Conditional	Nicht realisiert
	Anzeige aller implementierten Funktionen zur Navigation im Skript	Optional	Nicht realisiert
	Anzeige aller instanziierten Variablen. Einfügen des Variablennamens per Drag&Drop	Optional	Nicht realisiert
	Routing-Editor	Optional	Nicht realisiert
	Funktionen, die bei der Codeentwicklung unterstützen, z.B. Autovollständigung	Conditional	Realisiert, aber Funktionsumfang ist mäßig
Konfigurieren einer GIGABOX ohne PAWN-Kenntnisse	Realisierung von Funktionen mit Funktionsbausteinsprache	Optional	Nicht mit Funktionsbausteinsprache realisiert sondern mit Anweisungstabellen

5.4 Quellcode des configurAIDERs

Was wurde mit WinForms gemacht, was mit WPF?

Entstand aus mehreren studentischen Arbeiten

Diagramm mit bestehender Architektur erstellen

5.5 Ergebnis

Kapitel 6

Konzeption

6.1 Überblick

Ziel des Kapitels ist es, Konzepte aufzuzeigen, wie die in der Anforderungsanalyse erarbeiteten Anforderungen umgesetzt werden können.

6.2 Erweiterung des bestehenden configurAIDERs um neue Features

Der bestehende Code des configurAIDERs wird erweitert um neue Features, die bei der Bewertung der Anforderungen in Abschnitt **??** mit (+) oder (++) bewertet wurden. Die an bereits bestehenden Features in Abschnitt 4.3 kritisierten Punkte sollen verbessert werden.

Neue Funktionen:

- Projektverwaltung: Es können Projekte angelegt werden, in denen unterschiedliche Modelle der GIGABOX projektiert sind und die zugehörigen Skripte+Include-Files
- Texteditor wird erweitert um:
  1. Möglichkeit, letzte Schritte rückgängig zu machen
  2. Anzeige aller implementierten Funktionen zur Navigation im Skript
  3. Bei der Codeentwicklung unterstützende Funktionen

4. Routing-Editor

- Frei andockbare Fenster
- Steuern und Beobachten von DIN, AIN, DOUT, SWITCH

Vorteil:

- Das Tool erhält damit einen erhöhten Funktionsumfang, als wichtig bewertete funktionale Anforderungen können umgesetzt werden.

Nachteil:

- Nichtfunktionale Anforderungen wie Qualitätskriterien an Software nach ISO9126 (Wartbarkeit, Usability, Effizienz, Übertragbarkeit, Zuverlässigkeit) können nicht erfüllt werden, sondern werden sich verschlechtern durch eine weitere Funktionserweiterung des Tools
- configurAIDER bleibt schwer testbar, da Sollverhalten des Codes nicht immer bekannt ist
- Viel Einarbeitung in bestehenden Code nötig

6.3 Neuentwicklung des configurAIDERs

Das Tool wird von Grund auf neu entwickelt. Es sollen die funktionalen und nichtfunktionalen Anforderungen aus Kapitel 3 umgesetzt werden. Im Rahmen dieser Arbeit sollen vorerst nur funktionalen Anforderungen umgesetzt werden, die in die Klasse „Essential“ oder „Conditional“ eingeordnet wurden. Allerdings sollen alle nichtfunktionalen Anforderungen erfüllt werden.

Vorteil:

- Tool wird einfacher erweiterbar als bisheriger Stand. Dies ermöglicht es, zukünftig einfacher neue Features zu integrieren.
- Besser wartbar: Bugs können in Zukunft besser behoben werden, da Verhalten des neuen Codes besser bekannt ist
- Usability kann verbessert werden
- Tool konsistent mit WPF realisiert unter aktueller .NET Version 4.6

- Keine lange Einarbeitung in alten Code nötig

Nachteil:

- Softwarearchitektur muss neu entworfen werden
- Tool bietet keinen direkten Mehrwert in Form von höherem Funktionsumfang
- Hoher Implementierungsaufwand um die gestellten funktionalen Anforderungen zu erreichen

6.4 Ergebnis

Es wird entschieden, das Konzept der Neuentwicklung des configurAIDERs umzusetzen.

Kapitel 7

Design

7.1 Überblick

In Kapitel 5 wurde entschieden, eine Neuentwicklung des configurAIDERs weiter zu verfolgen. Hierzu muss die Benutzeroberfläche neu entworfen werden, eine neue Software-Architektur entwickelt werden und anschließend eine Umsetzung in Programmcode erfolgen. Dieses Kapitel beschäftigt sich mit dem Entwurf einer neuen Benutzeroberfläche und dem Herleiten einer neuen Softwarearchitektur.

7.2 Entwurf der Benutzeroberfläche

Die Benutzeroberfläche soll so entworfen werden, dass die in der Anforderungsanalyse erarbeiteten Use Cases erfüllt werden können. Zusätzlich sollen die in Abschnitt 2.3.2 definierten Grundsätze der Dialoggestaltung aus ISO ISO/IEC 25010 beachtet werden.

Abbildung 7.1 zeigt ein Mockup der Benutzeroberfläche, das diese Kriterien erfüllt.

Die Hauptelemente der Benutzeroberfläche sind ein Texteditor, ein Projektexplorer, eine Geräteliste sowie eine Konsole. Zur Bedienung dieser Hauptelemente verfügt die GUI über eine Menüleiste, eine Toolbar mit Buttons sowie über Kontextmenüs. Die Menüleiste stellt Funktionen bereit, die im jeweiligen Bedienungskontext zur Verfügung stehen. Den Zugriff auf häufig benötigte Funktionen soll eine Toolbar ermöglichen.

Die Toolbar enthält folgende Buttons:

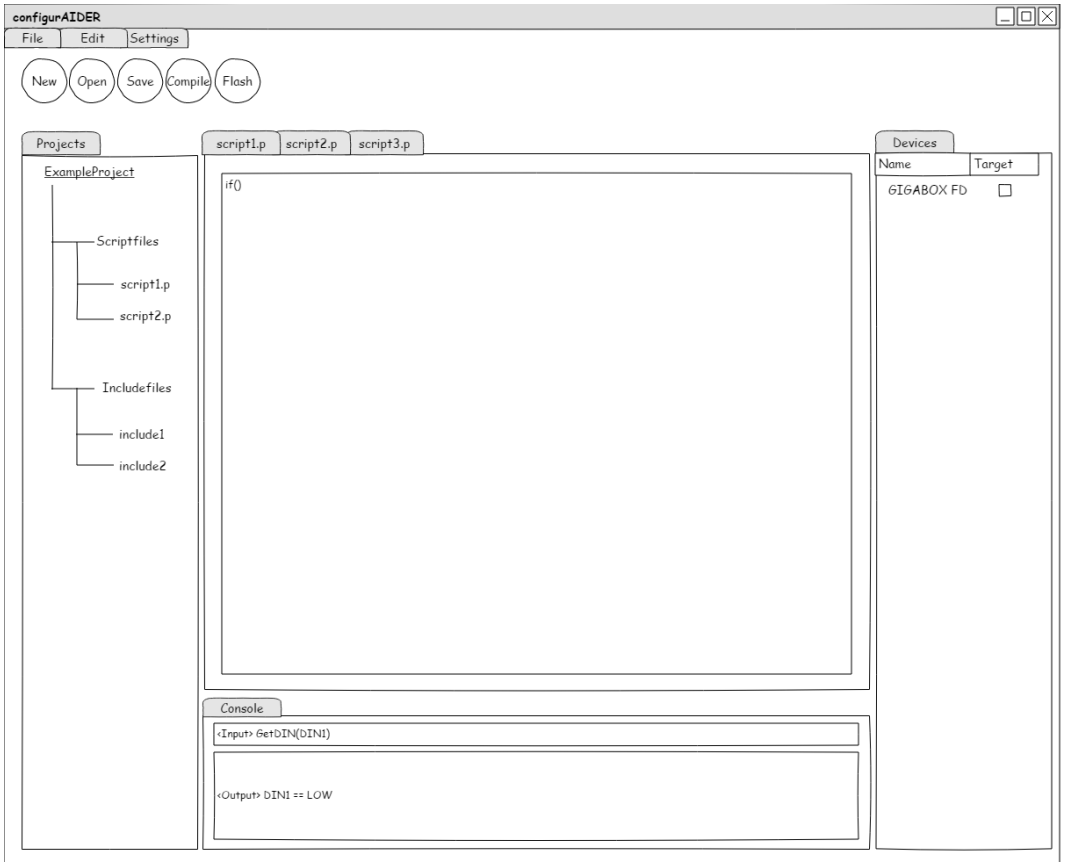

Abbildung 7.1 – Mockup der Benutzeroberfläche des configurAIDERs

- Neues Projekt erstellen
- Projekt öffnen
- Datei öffnen
- Datei speichern
- Datei speichern unter
- Datei kompilieren
- Datei flashen

Die vier GUI-Hauptelemente werden innerhalb eines einzelnen Programmfensters dargestellt, um dem Benutzer einen zentralen Zugriff auf alle zur Verfügung stehenden Funktionen zu ermöglichen.

Als zentrales Element ist in der Mitte des Fensters der Texteditor angeordnet, da für die Anzeige von Quellcode viel Platz beansprucht wird. Um innerhalb von größeren Programmskripten zu navigieren, muss der Texteditor scrollbar sein.

Die Konsole ist unterhalb des Texteditors ausgerichtet, da ein breites Eingabefeld für Befehle benötigt wird. Sie besteht aus einer Eingabezeile, die als oberstes Steuerelement angeordnet ist sowie aus einem darunterliegenden Ausgabefeld. Da in vertikale Richtung wenig Platz zur Verfügung steht, muss die Textausgabe der Konsole scrollbar sein.

Der Projektexplorer kann ein Projekt darstellen, das eine Baumstruktur aus Ordnern und Pawn-Dateien beinhaltet. Das Projekt repräsentiert das Wurzelelement des Baumes und stellt Ebene 1 der Struktur dar. Die Kindelemente des Projektes können Ordner sowie Pawn-Dateien sein, sie repräsentieren Ebene 2 der Struktur. Ordner wiederum können nur Pawn-Dateien als Kindelemente enthalten, somit ist die Baumstruktur auf drei Ebenen beschränkt. Das Hinzufügen von Kindelementen soll über ein Kontextmenü des Elternelementes möglich sein.

In der Geräteliste werden über USB verbundene GIGABOX FD-Geräte untereinander aufgelistet. Zu jedem Gerät wird eine Checkbox dargestellt, mit der eine GIGABOX als Zielgerät ausgewählt werden kann. Wurde ein Zielgerät festgelegt, steht im configurAIDER die Flash-Funktion zur Verfügung und in die Konsole eingegebene Befehl werden an das Zielgerät gesendet.

(Vereinzelt Designbegründung nach ISO einfügen)

Erläuterung des Entwurfes mit Bezugnahme auf ISO, die gute Usability definiert

7.3 Entwurf der Softwarearchitektur

„The software architecture of a program or computing system is the structure or structures of the system, which comprises software components, the externally visible properties of those components and the relationship among them.“ [7]

Nach dieser in der Literatur oft zitierten Definition legt die Softwarearchitektur die Komponenten eines Systems fest, beschreibt deren wesentliche, von außen sichtbare Merkmale und charakterisiert die Beziehungen dieser Komponenten. Sie beschreibt den statischen Aufbau einer Software im Sinne eines Bauplans und den dynamischen Ablauf einer Software im Sinne eines Ablaufplans.

Die in diesem Kapitel vorgestellte Software-Architektur des configurAIDERs wurde entworfen auf Basis der funktionalen und nichtfunktionalen Anforderungen an Software in Kapitel 3. Wie im vorigen Kapitel „Konzeption“ erläutert, werden nur Funktionen umgesetzt, die als „Essential“ oder „Conditional“ priorisiert wurden.

Die Softwarearchitektur wird aus verschiedenen Sichten dargestellt, da eine einzelne Darstellung die Komplexität eines Systems nicht wiedergeben kann. Jede Sicht zeigt das System detailliert aus einer bestimmten Perspektive. Details, die für eine Sicht nicht von Bedeutung sind, werden vernachlässigt. Es werden drei Arten von Sichten verwendet: die Kontextabgrenzung, die Bausteinsicht und die Laufzeitsicht. Die verschiedenen Sichten werden in UML-Diagrammen dargestellt. Für die Darstellung der Kontextabgrenzung wird ein Deployment Diagramm verwendet, für die Bausteinsicht Klassendiagramme und für die Laufzeitsicht Sequenzdiagramme. Die Einteilung in diese Sichtarten wird in Buch [8] empfohlen.

Zur Verdeutlichung kann man sich die Erstellung einer Gebäudearchitektur anschauen, bei der ähnlich vorgegangen wird. Das Gebäude wird aus verschiedenen Sichten dargestellt, z. B. Grundriss, Gebäudeplan und Elektroplan. Jede Sicht konzentriert sich auf die Darstellung von Details, die für diese Sicht wichtig sind und vernachlässigt dafür anderes, um die Darstellung nicht zu überladen.

Quelle: [8]

7.3.1 Model-View-ViewModel-Pattern (MVVM-Pattern)

Das Model-View-ViewModel-Pattern (MVVM) ist ein Software-Architekturmuster. Das Muster basiert auf der Idee des Model-View-Controller-Patterns (MVC), die Verantwortlichkeiten für Darstellung und Layout der Benutzeroberfläche

von der Logik der Benutzeroberfläche zu trennen. Das MVVM-Pattern verwendet drei Komponenten, die jeweils unterschiedliche, voneinander getrennte Verantwortlichkeiten besitzen. Diese drei Komponenten sind die View, das Model und das ViewModel.

Die View ist zuständig für Aussehen und Struktur der Elemente der Benutzeroberfläche. In WPF wird die View in einer XAML-Datei beschrieben. Jede XAML-Datei ist untrennbar mit einer Codebehind-Datei gekoppelt, in der UI-Logik implementiert werden kann. Nach dem MVVM-Pattern soll im Codebehind keine Logik implementiert werden, die außerhalb der Verantwortlichkeiten der View liegt.

Das Model enthält die Daten, die dem Benutzer über die View angezeigt werden und von diesem bearbeitet werden können. Hier wird auch eine Validierung der eingegebenen Daten vorgenommen.

Das ViewModel stellt das Bindeglied zwischen View und Model dar. Hier wird die UI-Logik implementiert. Die Daten des Models werden im ViewModel manipuliert, um sie in geeigneter Weise über die View darstellen zu können. View und ViewModel werden nach dem MVVM-Pattern ausschließlich mithilfe von Data Bindings und Commands gekoppelt. Data Binding ermöglich der View, sich an Properties des ViewModels zu binden und damit auf benötigte Daten zugreifen zu können. Beispielsweise kann ein Texteingabefeld der View über Data Binding an eine `string`-Property im ViewModel gebunden werden. Im ViewModel steht dann der Inhalt der Textbox als `string` zur Verfügung, kann dort weiterverarbeitet und anschließend im Model gespeichert werden. Commands werden eingesetzt, um auf Benutzeraktionen wie einen Klick auf einen Button reagieren zu können. Jedes in der View definierte UI-Element, das das Interface `ICommandSource` implementiert, löst bei einer definierten Aktion ein Command aus. Über Data Binding kann auf ein Property vom Typ `ICommand` im ViewModel gebunden werden. Dort wird die *Execute*-Methode, die im `ICommand`-Interface enthalten ist, aufgerufen. In der *Execute*-Methode wird schließlich die Logik definiert, die die Benutzeraktion auslösen soll.

Ändern sich im Model oder im ViewModel Daten, die über Data Binding an die View gebunden sind, muss die View über Änderungen informiert werden. Sie kann dann von dem Property, das die geänderten Daten bereitstellt, die Daten neu abfragen und die aktualisierten Daten auf der Benutzeroberfläche ausgeben. Ein solcher Benachrichtigungsmechanismus kann realisiert werden, indem alle Model- und ViewModel-Klassen das Interface `INotifyPropertyChanged` implementieren. Jede dieser Klassen muss dann ein Event deklarieren, das ein Delegat vom Typ `PropertyChangedEventHandler` verwendet. Wird ein Property neu gesetzt, wird das Event ausgelöst und als Folge davon die View darüber benachrichtigt, in welchem Property Änderungen vorgenommen wur-

den.

Abbildung 7.2 zeigt beschriebene Prinzip aus Data Binding, Commands und Benachrichtigungen.

Quelle: [9], [10]

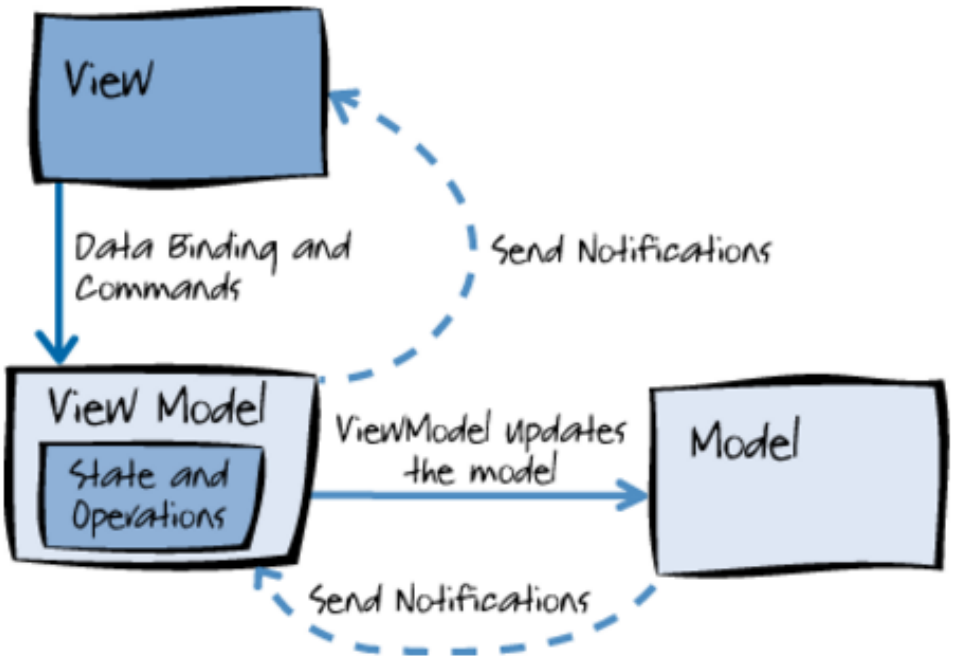

Abbildung 7.2 – Prinzip des MVVM-Patter. Quelle: [9]

7.3.2 Entwurf der Kontextabgrenzung

Die Kontextabgrenzung zeigt das System als Blackbox und stellt das System in Kontext zu seiner Umgebung dar. Abbildung 7.3 zeigt die Kontextabgrenzung des configurAIDERs als UML Deployment Diagramm. Es werden alle den configurAIDER umgebenden Systeme dargestellt, die zur Erfüllung der in der Anforderungsanlayse hergeleiteten Use-Cases (nur die als „Essential“ oder „Conditional“ klassifizierten) benötigt werden.

GIGABOX Steuergeräte stehen in einer bidirektionalen Kommunikationsbeziehung mit dem configurAIDER über USB. Wenn eine GIGABOX an die USB-Schnittstelle des Computers angeschlossen wird, soll sie Geräteinformationen an den configurAIDER senden. Alle verbundenen GIGABOXEN können dann im configurAIDER mit den zugehörigen Geräteinformationen auf-

gelistet werden (Use Case „Verbundene GIGABOXEN auflisten“ 3.2.5). Die USB-Verbindung wird auch genutzt, um Befehle von der Konsole des configurAIDERs an eine GIGABOX zu senden und die Antwort von der GIGABOX zu empfangen (Use Case „Kommunikation mit einer GIGABOX“ 3.2.8). Zusätzlich wird die Kommunikationsverbindung für den Flashvorgang benötigt (Use Case „Beschreiben/Auslesen des Flash-Speichers der GIGABOX“ 3.2.7).

Der Benutzer steht ebenfalls in einer bidirektionalen Kommunikationsbeziehung mit dem configurAIDER. Er tätigt Eingaben über Eingabegeräte wie Maus und Tastatur und erhält Rückmeldung über Ausgabegeräte wie einen Bildschirm.

Zur Erfüllung des Use Cases „Erstellung von Projekten mit Baumstruktur“ 3.2.1 wird eine Projektdatei benötigt, in der die Projektstruktur und Informationen über die im Projekt enthaltenen Dateien wie Name und Pfad abgebildet sind. Das XML-Format ist hervorragend geeignet um Informationen und hierarchische Strukturen maschinenlesbar abzubilden und wird deshalb für die Projektdatei verwendet.

GIGABOX-Applikationen liegen als Pawn-Quellcodedatei (*.p) vor. Bei der Kompilierung wird eine Pawn-Quellcodedatei verwendet und in Bytecode kompiliert, der in eine Binärdatei (*.amx) geschrieben wird (Use Case „Kompilieren von Pawn-Quellcodedateien“ 3.2.6).

Zum Flashen einer GIGABOX-Applikation auf die GIGABOX muss der Bytecode der vom Compiler erzeugten Binärdatei (*.amx) im Intel Hex-Format vorliegen anstatt im Binärformat. Eine Umwandlung ist vonnöten. Die Umwandlung des Bytecodes in das Intel Hex-Format erzeugt eine Hex-Datei (*.hex). Diese Hex-Datei kann nun auf den Flash-Speicher der GIGABOX geschrieben werden.

Der Benutzer soll aus dem configurAIDER heraus Hilfedokumentationen aufrufen können (Use Case Bereitstellung von Dokumentationen 3.2.12). Die Dokumente liegen im PDF-Format vor und werden bei diesem Use Case aus dem configurAIDER aufgerufen und in dem auf dem Betriebssystem installierten PDF-Viewer angezeigt.

7.3.3 Entwurf der Bausteinsicht

Die Bausteinsicht stellt den Quellcode eines Softwaresystems in unterschiedlichen Abstraktionsebenen dar. Sie veranschaulicht die Struktur und die Zusammenhänge der unterschiedlichen Bausteine eines Systems. Bausteine werden hier innerhalb von UML-Diagrammen als Pakete, Komponenten und Klassen

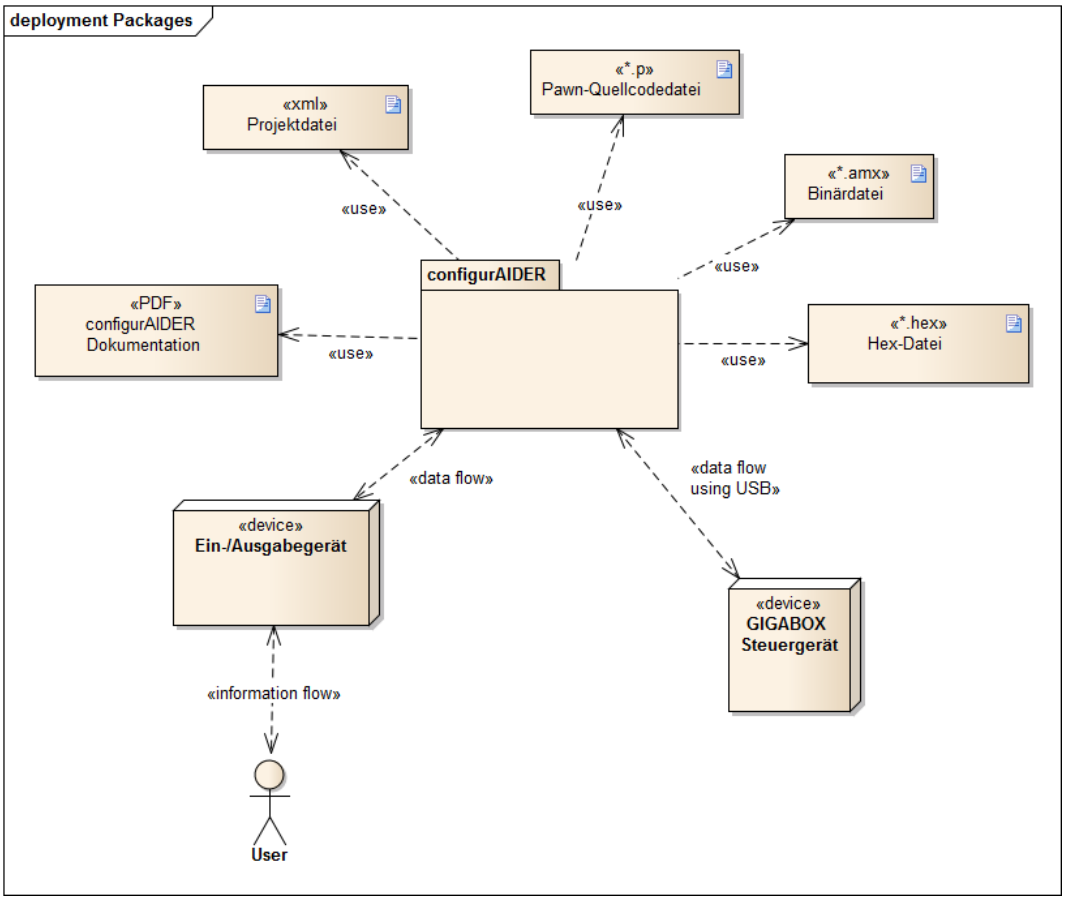

Abbildung 7.3 – Darstellung der Kontextabgrenzung des Systems im UML-Deployment Diagramm

dargestellt.

Abstraktionsebene 1

Abbildung 7.4 zeigt die oberste Abstraktionsebene der Bausteinsicht des configurAIDERs. Das `Core`-Paket repräsentiert die ausführbare Datei (.exe) der Software. Hier ist sind die verschiedenen Views und die zugehörigen ViewModels enthalten. Die ViewModels übernehmen die Verantwortlichkeiten der Models mit. Es wurde auf Models verzichtet, da die Software nicht auf große Datenmengen zugreifen muss. Die Komplexität wird dadurch reduziert.

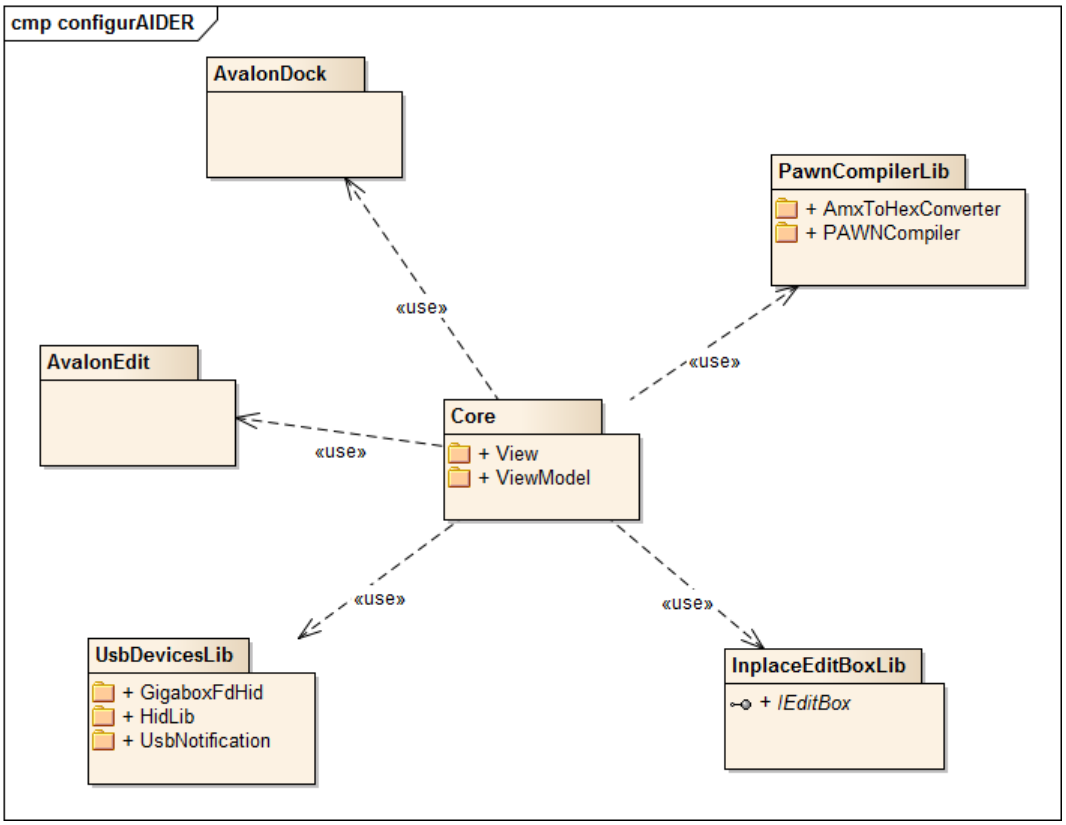

Abbildung 7.4 – Darstellung der ersten Ebene der Bausteinsicht

Das `Core`-Paket benutzt die Pakete `AvalonEdit`, `AvalonDock`, `PawnCompilerLib`, `InplaceEditBoxLib` und `UsbDevicesLib`.

Bei `AvalonEdit` handelt es sich um einen WPF basierten Texteditor, dessen Quellcode frei einsehbar ist (Open Source) und für die verwendete Version 4.3.0 unter der GNU Lesser General Public License veröffentlicht wird. Ent-

wickelt wurde `AvalonEdit` für die Open Source IDE „SharpDevelop“, die eine kostenlose Alternative zu Microsofts IDE „Visual Studio“ darstellt. [11]

`AvalonDock` ist ein Paket, das die Darstellung von Inhalten in frei andockbaren Fenstern ermöglicht. Seit der Veröffentlichung in der Version 2.0 kann `AvalonDock` auch unter Einhaltung des MVVM-Patterns verwendet werden. Es wird veröffentlicht unter der BSD-Lizenz und ist Open Source. [12]

`InplaceEditBoxLib` ist ein Paket, das editierbare Textboxen zur Verfügung stellt. Die Textbox kann Text ausgeben und bei Bedarf ein Eingabefeld einblenden, über das der Benutzer den angezeigten Text ändern kann. Die Textboxen aus diesem Paket werden innerhalb des Projektexplorers des configurAIDERs benutzt, um dem Benutzer eine Neubenamung der Objekte zu ermöglichen. Der Quellcode ist frei zugänglich von Quelle [13] zu beziehen und steht unter der „Code Project Open License (CPOL)“.

`PawnCompilerLib` ist zuständig für die Kompilierung von Pawn-Quelldateien sowie für die Konvertierung der vom Compiler erzeugten Binärdateien in Hex-Dateien. Der Pawn-Compiler „pawncc.exe“ wird als Konsolenanwendung verwendet, frei beziehbar von Quelle [14]. Zur Umwandlung der Binärdateien in flashbare Hex-Dateien wird die Konsolenanwendung „srec_cat.exe“ verwendet, das im Paket „SRecord 1.64“ von Quelle [15] enthalten ist.

`UsbDevicesLib` ist verantwortlich für die USB-Kommunikation des configurAIDERs mit GIGABOX-Steuergeräten.
Eine Abstraktionsebene unter `UsbNotification` liegen die Pakete `UsbDevicesLib`, `HidLib` und `GigaboxFdLib`. `UsbDevicesLib` ermöglicht es, über die Windows API eine Benachrichtigung zu bekommen, sobald ein USB-Gerät an der USB-Schnittstelle des PCs angeschlossen oder entfernt wird. Die Geräteliste des configurAIDERs kann somit stets aktualisiert werden, sobald eine solche Benachrichtigung registriert wird.
`HidLib` ermöglicht es, eine Verbindung zu einem USB-HID (Human Interface Device) Gerät herzustellen und mit diesem Daten auszutauschen. Das Paket enthält Wrapper-Klassen, die die zur Kommunikation benötigten nativen APIs kapseln. Die Klassen stellen ihrer Umgebung Programmfunktionen bereit, mithilfe derer Datenpakete an ein verbundenes USB-HID Gerät gesendet und empfangen werden können. `HidLib` basiert auf Quellcode des Projektes „MightyHID“ verwendet, das von Quelle [16] stammt.
Mithilfe des Paketes `GigaboxFdHid` wird die Kommunikation des configurAIDERS mit einer GIGABOX realisiert. Der Benutzer soll einer GIGABOX Befehle über die Konsole des configurAIDER senden können sowie eine Hex-Datei auf den Flash-Speicher des Gerätes schreiben zu können. GIGABOXEN werden von einem PC als USB-HID Gerät erkannt, deshalb kann zur Datenübertragung das `HidLib`-Paket verwendet werden.

Abstraktionsebene 2

Aufgrund der Vielzahl an Paketen wird die Verfeinerung der Abstraktion auf die Pakete `View` und `ViewModel` beschränkt. Der Code dieser Pakete wurde vollständig selbst entwickelt und basiert nicht auf Quellcode von Dritten.

Grundsätzlich besteht der configurAIDER aus vier Elementen. Einer Menüleiste und Buttons in der Kopfzeile, einem Texteditor, einem Projektexplorer und einer Liste zur Anzeige aller angeschlossener GIGABOX-Geräte. Jedes der Elemente besitzt eine eigene View, die mithilfe einer XAML-Datei beschrieben wird. Daten und Logik zu jeder View werden durch verschiedene ViewModels bereitgestellt, die über Data Binding und Commands lose miteinander gekoppelt sind.

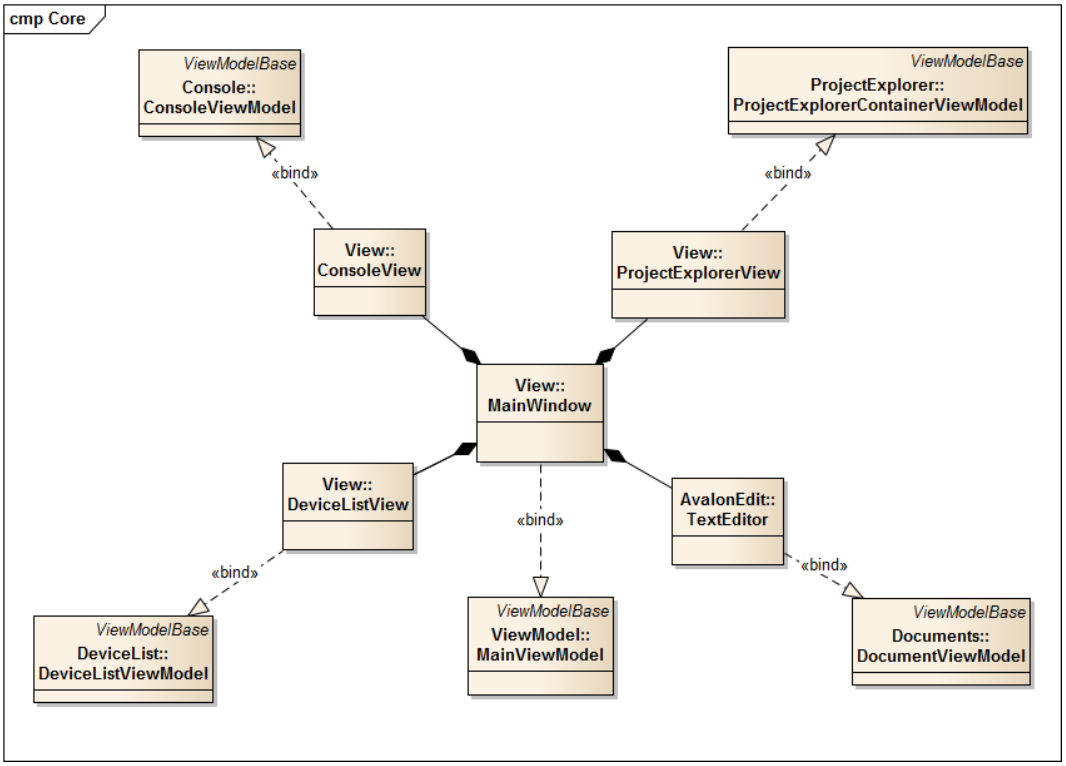

Abbildung 7.5 – Kopplung zwischen Views und ViewModels

Das UML Klassendiagramm 7.6 zeigt die Struktur der ViewModel-Klassen im UML Klassendiagramm.

`ViewModelBase` ist die abstrakte Basisklasse aller ViewModels. Sie implementiert das Interface `INotifyPropertyChanged` und stellt Funktionen bereit, mithilfe derer die View über Datenänderungen innerhalb eines ViewModels infor-

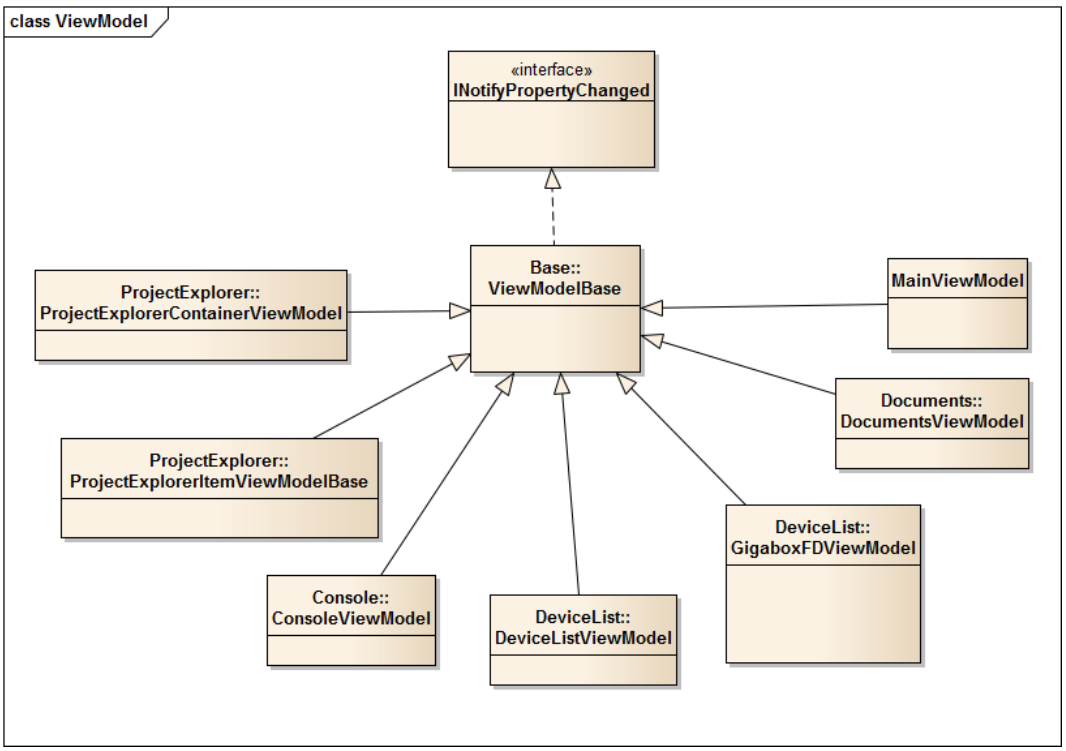

Abbildung 7.6 – Struktur der ViewModel-Klassen im UML Klassendiagramm

miert werden kann. Dieser Benachrichtigungsmechanismus, der im Rahmen des MVVM-Patterns verwendet wird, wurde in Abschnitt 7.3.1 näher beschrieben.

Abstraktionsebene 3

Auf Abstraktionsebene 3 wird ausschließlich der Projektexplorer beschrieben, dieser besitzt trotz niedriger Abstraktion noch eine komplexe Struktur besitzt. Im Projektexplorer kann der Benutzer eine hierarchische Dateistruktur aus Elementen anlegen, wie sie beispielsweise vom Windows Explorer bekannt ist. Diese Elemente können ein Projekt, Ordner, Pawn-Skriptdateien und Pawn-Includedateien sein.

Das Projektexplorerfenster wird durch die Klasse `ProjectExplorerContainerViewModel` repräsentiert, die auch die Kopplung mit `ProjectExplorerView` realisiert (siehe Abbildung 7.5). Innerhalb des Projektexplorers kann ein Projekt angelegt werden, das durch die Klasse `ProjectViewModel` repräsentiert wird. Zu jedem Projekt können entweder Ordner (repräsentiert durch `FolderViewModel`), Pawn-Skriptdateien (repräsentiert durch `ScriptFileViewModel`) und Pawn-Includedateien (repräsentiert durch `IncludeFileViewModel`) hinzugefügt werden. Innerhalb eines Ordners können Pawn-Skriptdateien und Pawn-Includedateien angelegt werden.
Abbildung 7.7 zeigt diese Zusammenhänge der Klasseninstanzen in einem UML-Klassendiagramm mithilfe von Kompositionen und den zugehörigen Multiplizitäten.

Alle beschriebenen Klassen, die Elemente im Projektexplorer repräsentieren, leiten von `ProjectExplorerItemViewModelBase` ab. `ProjectExplorerItemViewModelBase` ist als abstrakte Klasse definiert und implementiert das Interface `IProjectExplorerItem`. Das Interface legt fest, welche Properties und Methoden ein Element des Projektexplorers enthalten muss.
Abbildung 7.8 zeigt die Struktur der Klassen, die für den Projektexplorer von Bedeutung sind.

7.3.4 Entwurf der Laufzeitsicht

Die Laufzeitsicht veranschaulicht, wie die in der Bausteinsicht gezeigten Klassen zur Laufzeit interagieren. Für die wichtigsten Use Cases wurde jeweils eine Laufzeitsicht erstellt mithilfe von UML Sequenzdiagrammen.

Beispielhaft wird hier die Laufzeitsicht des Use Cases „Steuerbefehle an GIGABOX senden“ angeführt. Abbildung 7.9 zeigt das zugehörige Sequenzdiagramm. In der Darstellung des Diagrammes wird die Annahme getroffen, dass

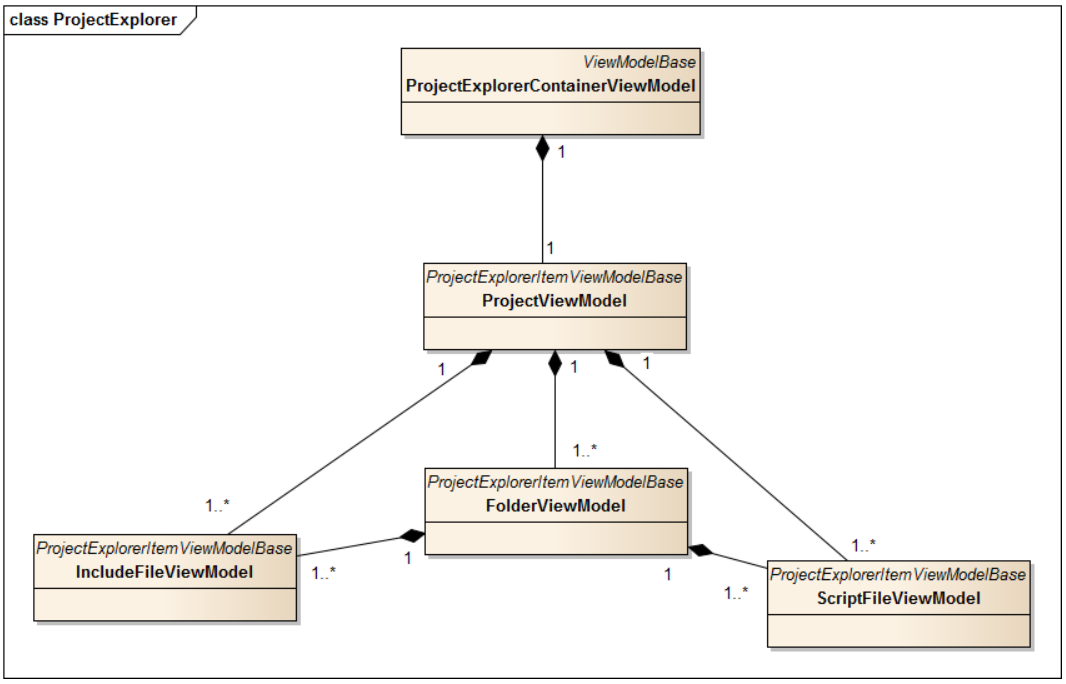

Abbildung 7.7 – Kompositionsbeziehungen zwischen den ViewModels des Projektexplorers in einem UML-Klassendiagramm

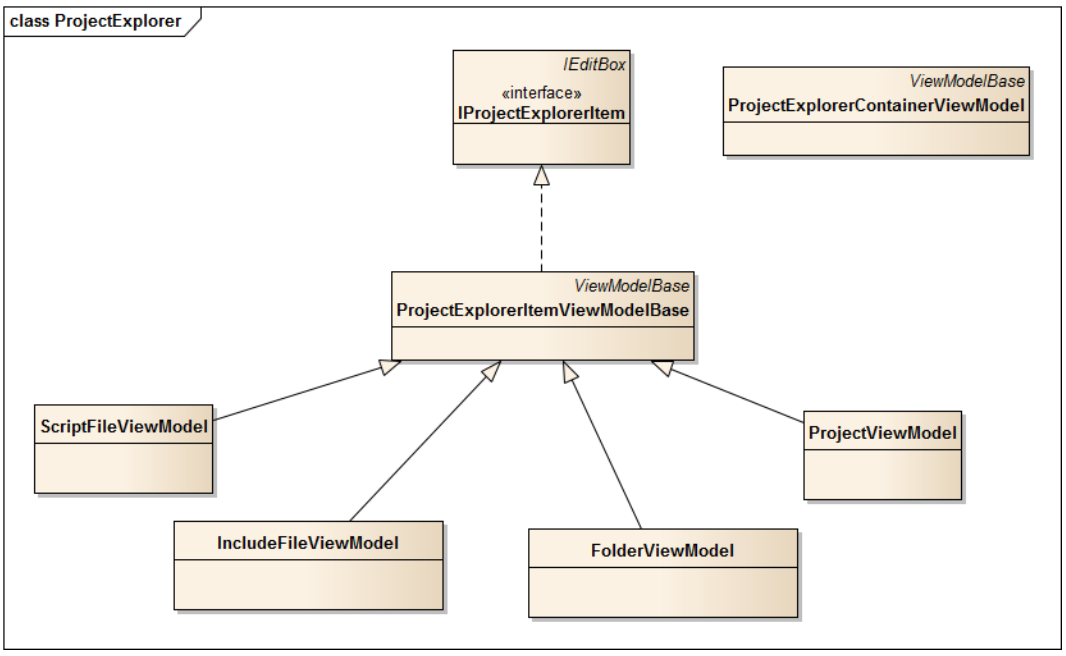

Abbildung 7.8 – Struktur der Klassen des Projektexplorers in einem UML-Klassendiagramm

der Benutzer bereits eine GIGABOX angewählt hat, mit der kommuniziert werden soll.

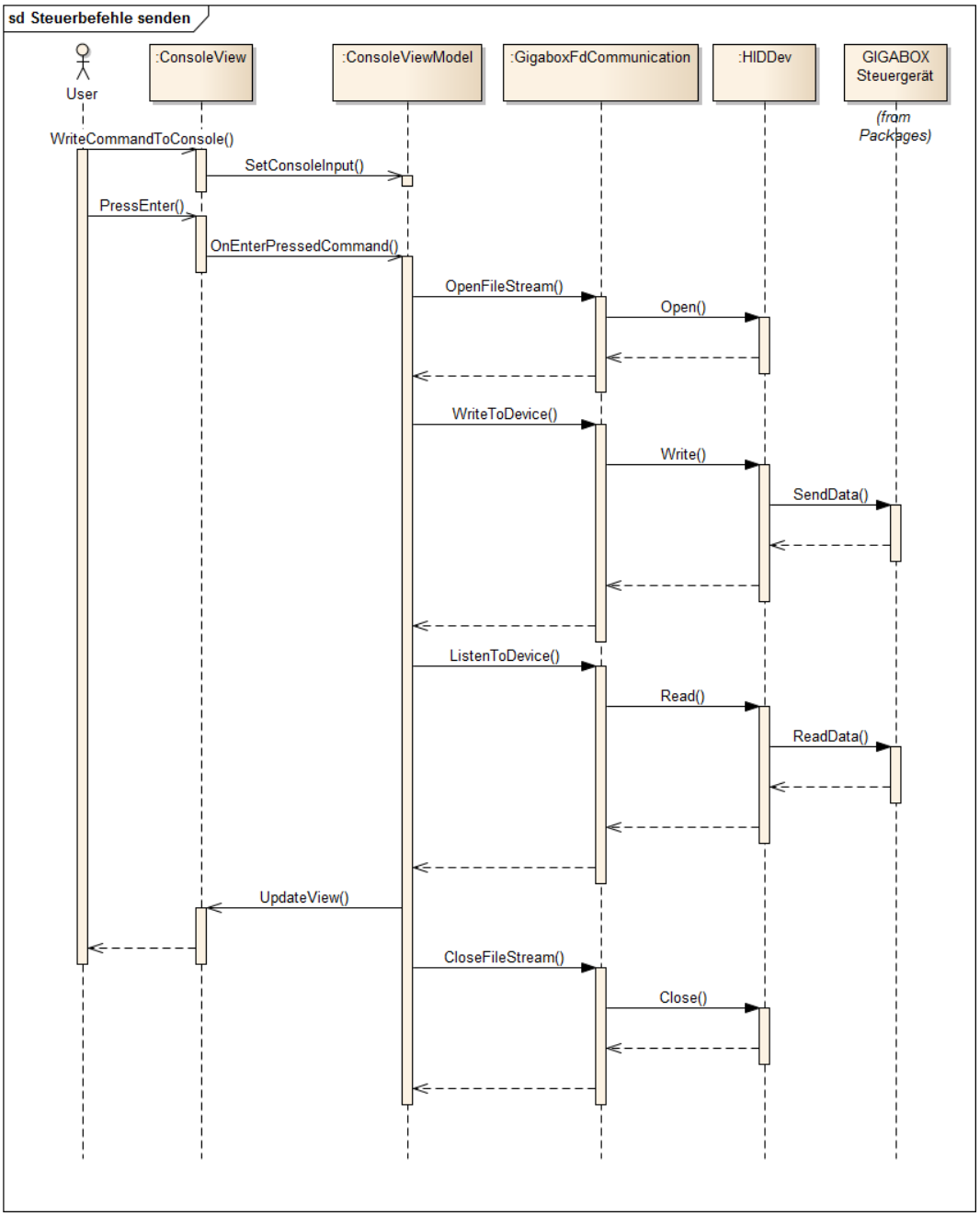

Abbildung 7.9 – Darstellung der Laufzeitsicht des Systems im UML Sequenzdiagramm

Als initiale Handlung schreibt der Benutzer einen Befehl in die Konsole des

configurAIDERs. Im Diagramm wird dies dargestellt durch die Nachricht *WriteCommandToConsole()* an eine Instanz vom Typ `ConsoleView`, die die Benutzeroberfläche repräsentiert. Der eingegebene Befehl wird in einer Instanz von `ConsoleViewModel` gespeichert. Sobald der Benutzer die Eingabe mit Enter bestätigt, wird über einen Command die Instanz von `ConsoleViewModel` benachrichtigt, dass die Benutzereingabe ausgeführt werden soll. Die Funktion *Open()* wird aufgerufen, die die Klasse `HIDDev` implementiert. Diese Funktion ermöglicht einen Lese- und Schreibzugriff auf das angewählte Gerät. Der vom Benutzer eingegebene Befehl wird nun an die GIGABOX gesendet. Für ein definiertes Zeitfenster wird anschließend auf eine Antwort gewartet. Wird eine Antwort empfangen, wird diese in der Instanz von `ConsoleViewModel` gespeichert und auf die Konsole geschrieben. Abschließend wird der Lese- und Schreibzugriff zu dem Gerät wieder beendet.

Kapitel 8

Realisierung

8.1 Überblick

In diesem Kapitel wird vorgestellt, wie das in Abschnitt 6.3 vorgestellte Konzept, den configurAIDER vollständig neu zu entwickeln, umgesetzt wurde. Innerhalb des V-Modells beschreibt dieses Kapitel die Implementierungsphase. In dieser Phase werden die in der Anforderungsanalyse festgelegten Anforderungen in Code umgesetzt unter Beachtung der in der vorherigen Phase entworfenen Softwarearchitektur.

In Abschnitt 7.3.3 „Abstraktionsebene 1“ wurde ein Überblick über die für die Applikation verwendeten Pakete gegeben. Dieses Kapitel beschreibt die Pakete `Core`, `PawnCompilerLib` und `UsbDevicesLib`. Zusätzlich wird vorgestellt, wie die Pakete `AvalonDock`, `AvalonEdit` und `InplaceEditBoxLib` in die Applikation eingebunden wurden. Diese basieren auf frei zugänglichem Quellcode von Dritten.

In Abschnitt 7.2 wurde ein Mockup der Benutzeroberfläche entworfen, an dem sich sich die tatsächlich realisierte GUI orientiert. Sie besteht aus einem Hauptfenster, in das ein Texteditor, eine Geräteliste, eine Konsole und ein Projektexplorer eingebettet werden können. In der Beschreibung des `Core`-Paketes wird deshalb für jedes dieser Fensterelemente erklärt, wie sie Benutzeroberfläche („View“) und die zugehörige Programmlogik („ViewModel“) erstellt wurden.

8.2 Core-Paket

8.2.1 Erstellung des Hauptfensters

View

Die View des Hauptfensters besteht aus einer horizontalen Menüleiste als oberstes Element, einer Symbolleiste sowie freier Fläche, in die andere Fensterelemente eingebettet werden können. Das Hauptfenster ist in einer XAML-Datei mit dem Namen „MainWindow“ definiert.

Zur Erstellung der Menüleiste wird eine Instanz der Klasse `Menu` erzeugt, die sich im Namespace `Systems.Windows.Controls` befindet. Menüeinträge können innerhalb eines `Menu`-Objektes mit Objekten der Klasse `MenuItem` erstellt werden. Durch Schachtelung von `MenuItem`-Objekten ineinander kann sehr einfach ein Menü mit ineinander geschachtelten Inhalten erstellt werden. Sobald ein Mausklick auf das Element registriert wird, kann `MenuItem` einen Command auslösen. Die Klasse stellt dazu die Property `Command` bereit, die über Data Binding an eine Property vom Typ `ICommand` eines ViewModels gebunden wird. Das Prinzip von Commands und Data Binding in WPF-Applikationen wurde in Abschnitt 7.3.1 „MVVM-Pattern“ näher erläutert. Das folgende Codebeispiel in XAML erstellt eine Menüleiste mit geschachtelten Menüeinträgen. (Evtl. Bild des erzeugten Menüs einfügen)

```
1 <Menu IsMainMenu="True" >
2         <MenuItem Header="_Code" >
3                 <MenuItem Header="Compile" Command="{Binding
                  ↪  OnCompileActiveDocumentCommand}"/>
4                 <MenuItem Header="Flash" Command="{Binding
                  ↪  OnFlashActiveDocumentCommand}"/>
5         </MenuItem>
6 </Menu>
```

Um Symbolleisten zu erstellen, stellt das .NET-Framework die Klasse `Toolbar` bereit. Innerhalb der Symbolleiste wird die Klasse `Button` benutzt, um Schaltflächen zu kreieren. Objekte vom Typ `Button` können Commands auslösen, sobald ein Mausklick auf das Objekt registriert wird. Innerhalb der `Button`-Objekte werden Objekte der Klasse `Image` erstellt, über deren Property `Source` der Pfad zu einer Bilddatei angegeben wird. Die verwendeten Symbole stammen von der „Visual Studio Image Library“, die von Microsoft frei zur Verfügung gestellt wird [17].

Innerhalb der verbleibenden leeren Fläche sollen andere Elemente eingebettet werden können. Hierzu wird *AvalonDock* verwendet. Das Paket ermöglicht es, Fensterelemente an den Rändern eines Hauptfensters anzudocken, frei herauszulösen oder auszublenden. Mithilfe der `DockingManager`-Klasse können die Fensterelemente in das Hauptfenster eingebettet werden. In *AvalonDock* gibt es zwei verschiedene Typen von Rahmenfenstern, in die selbst jeweils mehrere Fenster eingebettet können.
Der erste Typ von Rahmenfenster ist `LayoutDocumentPane` und kann Fenster vom Typ `LayoutDocument` aufnehmen. Es kann nicht an Ränder angedockt werden. `LayoutDocument`-Elemente werden im Rahmenfenster als Tab dargestellt und sind prädestiniert dafür, Textdokumente anzuzeigen.
Der zweite Typ von Rahmenfenster ist `LayoutAnchorablePane` und kann Fenster vom Typ `LayoutAnchorable` aufnehmen. Dieser Typ kann an Ränder angedockt oder am Rand versteckt werden.
Beide Fenstertypen können aus ihrem Rahmenfenster herausgelöst und als frei verschiebbares `LayoutFloatingWindow` dargestellt werden.

Abbildung 8.1 zeigt die Benutzeroberfläche des configurAIDERs. Die Rahmenfenster vom Typ `LayoutAnchorablePane` sind rot markiert. Das am linken Fensterrand angedockte `LayoutAnchorablePane` enthält zwei `LayoutAnchorable`-Fenster, den Projektexplorer und die Geräteliste. Über die schwarz markierten Tabs kann angewählt werden, welches `LayoutAnchorable` angezeigt werden soll. Das am unteren Fensterrand angedockte `LayoutAnchorablePane` enthält ein `LayoutAnchorable`, die Konsole.
Das grün markierte Rahmenfenster ist vom Typ `LayoutDocumentPane`. Es enthält zwei `LayoutDocument`-Fenster, in denen ein im Texteditor geöffnetes Dokument angezeigt werden kann. Über die hellblau markierten Tabs kann angewählt werden, welches der beiden Dokumente angezeigt wird.

Die Klasse `DockingManager` besitzt die Properties `DocumentsSource` und `AnchorablesSource`, an die jeweils eine Property vom Typ `IEnumerable` gebunden werden kann. In `IEnumerable` enthaltene Objekte, die an `DocumentsSource` gebunden sind, werden als Fenster vom Typ `LayoutDocument` dargestellt. Analog dazu werden in `IEnumerable` enthaltene Objekte, die an `AnchorablesSource` gebunden sind, als Fenster vom Typ `LayoutAnchorable` dargestellt.
An `DocumentsSource` wird die Property `Documents` von `MainViewModel` gebunden, die vom Typ `ObservableCollection<DocumentViewModel>` ist.
An `AnchorablesSource` wird die Property `Anchorables` von `MainViewModel` gebunden, die vom Typ `IEnumerable<IAnchorable>` ist.

Über die Properties `DocumentsSource` und `AnchorablesSource` wird allerdings nur festgelegt, welche Daten dargestellt werden sollen, nicht aber wie diese Daten visualisiert werden sollen. Für die darzustellenden Elemente Tex-

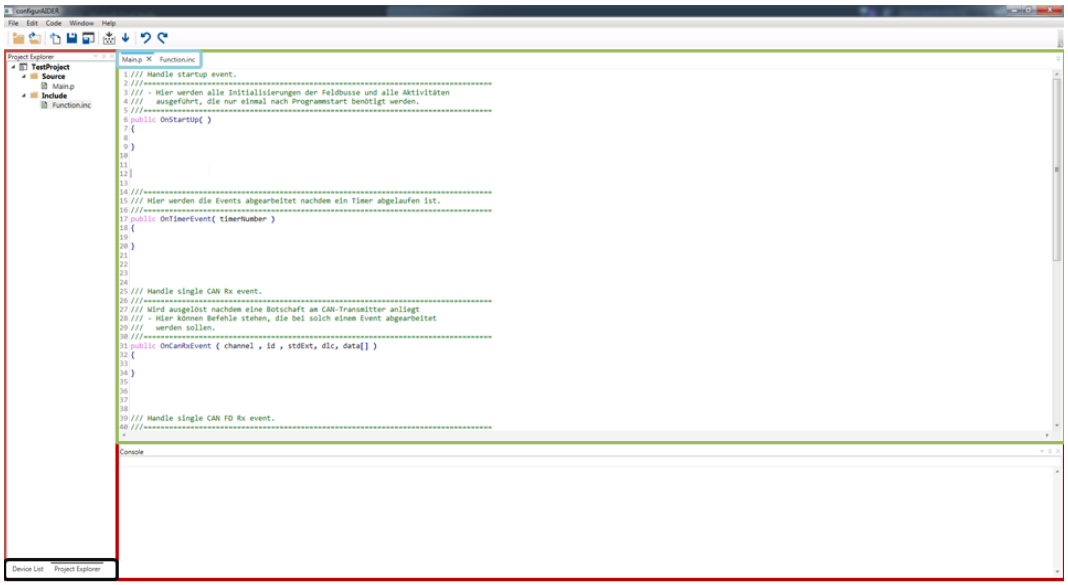

Abbildung 8.1 – Mockup der Benutzeroberfläche des configurAIDERs

teditor, Projektexplorer, Konsole und Geräteliste muss deshalb Darstellung und Aussehen der zugehörigen Daten definiert werden. Für diese Elemente wurde deshalb jeweils eine eigene View erstellt, die im späteren Verlauf der Arbeit noch beschrieben werden. Die Views wurden als `DataTemplate` erstellt. Mit `DataTemplates` lassen sich Vorlagen definieren, die die visuelle Präsentation von Daten beschreiben.
Die Klasse `PanesTemplateSelector` stellt die `DataTemplate`-Objekte zur Verfügung. Um *AvalonDock* das Aussehen der darzustellenden Daten mitzuteilen, muss eine Instanz von `PanesTemplateSelector` der `LayoutItemTemplateSelector`-Property von `DockingManager` zugewiesen werden.

LayoutItemContainerStyle von DockMAnager beschrieben

PanesTemplateSelector / LayoutItemTemplateSelector setzt den DataContext der Views auf das zugehörige ViewModel

LayoutPanel beschreiben, Layout beim Starten wird aus XML-Datei serialisiert, deserialisierung bei Schließen des Programms.

ViewModel

`MainViewModel` ist das ViewModel, mit der die View des Hauptfensters `MainWindow` über Data Binding und Commands gekoppelt ist. Es stellt Logik und Daten für `MainWindow` bereit, wie z.B. Objekte von ViewModel-Klassen, die die Inhalte

der *AvalonDock*-Elemente definieren.

Die Property `Documents` stellt eine `ObservableCollection` bereit, in der sich Objekte vom Typ `DocumentViewModel` befinden. Diese Objekte repräsentieren Textdokumente, die in einem *AvalonDock*-Fenster vom Typ `DocumentLayout` dargestellt werden. `DocumentViewModel` wird im Abschnitt 8.2.2 Texteditor näher beschrieben. Die Klasse `ObservableCollection` ist eine generische Collection, die die Interfaces `INotifyCollectionChanged` und `INotifyPropertyChanged` implementiert, die für den Benachrichtigungsmechanismus von ViewModel zu View benötigt werden (siehe Abschnitt 7.3.1, „MVVM-Pattern“).

Die Property `Anchorables` stellt eine generische Auflistung vom Typ `IEnumberable` zur Verfügung, in der sich Objekte vom Typ `IAnchorable` befinden. Im Gegensatz zu `Documents` wird hier keine `ObservableCollection` benötigt, da die Anzahl der `LayoutAnchorable`-Fenster während der Laufzeit der Applikation stets gleich ist und somit kein Benachrichtigungsmechanismus über hinzugefügte oder gelöschte Objekte benötigt wird. Das Interface `IAnchorable` wird von allen ViewModels implementiert, deren zugehörige View in `LayoutAnchorable`-Elementen dargestellt werden. Dies sind die ViewModels `ProjectExplorerContainerViewMode` `ConsoleViewModel` und `DeviceListViewModel`. Sie repräsentieren in dieser Reihenfolge den Projektexplorer, die Konsole sowie die Geräteliste.

Außerdem wird in `MainViewModel` die Logik zu folgenden Steuerelementen des Menüs und der Toolbar festgelegt:

- SAVE: Speichert die im Texteditor angezeigte Pawn-Datei auf dem Dateipfad, auf dem das Dokument aktuell gespeichert ist.
- SAVE AS: Speichert die im Texteditor angezeigte Pawn-Datei auf einem Dateipfad, den der Benutzer in einem sich öffnenden Dialogfeldes angeben kann.
- OPEN: Öffnet eine Pawn-Datei von einem Dateipfad, den der Benutzer in einem sich öffnenden Dialogfeldes angeben kann.
- COMPILE: Kompiliert die im Texteditor angezeigte Pawn-Datei mit dem Pawn-Comiler
- FLASH: Kompiliert die im Texteditor angezeigte Pawn-Datei und überträgt die erzeugte Hex-Datei auf den Flash-Speicher einer über USB mit dem PC verbundenen GIGABOX FD

Die Kopplung zwischen den Steuerelementen der View und der zugehörigen Logik im ViewModel geschieht über Commands. Die `Command`-Property eines

Steuerelementes der View wird dabei an eine Property vom Typ `ICommand` des ViewModels gebunden. Ein Klick auf eines dieser Steuerelemente ruft die darauf gebundene `ICommand`-Property des ViewModels auf, worauf die dort implementierte Logik ausgeführt wird. Klickt der Benutzer beispielsweise auf den Button SAVE der Toolbar, wird die Property `OnSaveFileCommand` aufgerufen. Die dort implementierte Logik speichert das im Texteditor angezeigte Dokument auf dem aktuellen Dateipfad des Dokuments.

8.2.2 Erstellung des Texteditors

Es wurde der in Abschnitt 7.3.3 erwähnte Texteditor *AvalonEdit* in der Version 4.3.0 verwendet. An dieser Stelle wird nicht näher auf den Quellcode von *AvalonEdit* eingegangen, sondern beschrieben, wie der Editor in die Applikation integriert wurde.

AvalonEdit stellt als Schnittstelle nach außen die Klasse `TextEditor` bereit, mit deren Hilfe der Texteditor in eine Applikation eingebettet werden kann. Die Einbindung von `TextEditor` erfolgt in der View `MainWindow` über die Property `LayoutItemTemplateSelector` der `DockingManager`-Klasse von *AvalonDock*. Der von *AvalonEdit* erzeugte Inhalt wird in einem *AvalonDock*-Fenster vom Typ `LayoutDocument` dargestellt.

Über Properties von `TextEditor` kann der angezeigte Text festgelegt sowie Einstellungen zu Aussehen und Verhalten des Texteditors gemacht werden. Die benötigten Properties wurden in `MainWindow` mithilfe von Data Binding an Properties der Klasse `DocumentViewModel` gebunden. In `DocumentViewModel` können somit Einstellungen von *AvalonEdit* gesetzt werden. Dabei repräsentiert jedes Objekt von `DocumentViewModel` ein *AvalonDock*-Element vom Typ `LayoutDocument`.

Die folgende Aufzählung zeigt, welche Eigenschaften des Texteditors in `DocumentViewModel` vorgenommen werden können:

- Festlegung des angezeigten Textes.
- Festlegung der Schriftgröße.
- Festlegung der Schriftart.
- Festlegung von Syntax-Highlighting. Für gängige Programmiersprachen bestehen vordefinierte Regeln des Syntax-Highlightings oder es können eigene Regeln in einer XML-Datei definiert werden.

- Anzeige von Zeilennummern ein- oder ausschalten.
- Abfrage, ob angezeigtes Dokument modifiziert wurde und sich von der ursprünglich geöffneten Version unterscheidet.

`DocumentViewModel` enthält außerdem Funktionen, die das Öffnen von Dokumenten im Texteditor ermöglichen. Mit *OpenDocumentFromPath()* kann ein Dokument von einem beliebigen Dateipfad geöffnet werden, die Funktion erwartet den Pfad als Argument. Mit *OpenDocumentFromProjectExplorer()* kann ein Textdokument vom Projektexplorer aus geöffnet werden, indem ein Objekt vom Typ `IProjectExplorerItem` als Argument übergeben wird.

Die Funktion *Update()* dient dem Zweck, Daten im Texteditor zu aktualisieren, die im Projektexplorer geändert wurden. Als Beispiel sei hier ein Projekt angeführt, in dem ein Textdokument angelegt wurde. Wird nun der Name des Projektes geändert, ändert sich gleichzeitig der Dateipfad des Textdokumentes, da dieses sich in einem Unterverzeichnis des Projektverzeichnisses befindet. Folglich muss das Objekt von `DocumentViewModel`, das das Textdokument repräsentiert, über die Änderung des Projektnamens informiert werden und der dort hinterlegte Dateipfad des Textdokumentes aktualisiert werden. Würde der Pfad nicht aktualisiert werden, würde ein Speichern des Dokumentes fehlschlagen, da der in `DocumentViewModel` hinterlegte Dateipfad nicht mehr existiert. Dieser Aktualisierungsmechanismus wurde nach dem Prinzip des Observer-Patterns implementiert.

8.2.3 Erstellung des Projektexplorers

View

Die View des Projektexplorers ist in der XAML-Datei `ProjectExplorerView` definiert. `ProjectExplorerView` wird in einem AvalonDock-Fenster vom Typ `LayoutAnchorable` dargestellt, der Aufruf erfolgt in der View des Hauptfensters `MainWindow`.

Im Projektexplorer kann ein Projekt erstellt und geöffnet werden, das als Baumstruktur dargestellt wird und aus Ordnern, Pawn-Skriptdateien sowie Pawn-Includedateien besteht. Das Projekt repräsentiert das Wurzelelement des Baumes und stellt Ebene 1 der Struktur dar. Die Kindelemente des Projektes können Ordner sowie Pawn-Dateien sein, sie repräsentieren Ebene 2 der Struktur. Ordner wiederum können nur Pawn-Dateien als Kindelemente enthalten, somit ist die Baumstruktur auf drei Ebenen beschränkt.

WPF stellt die Klasse `TreeView` zur Verfügung, um Objekte in einer Baumstruktur anzuordnen. Objekte in einer `TreeView` können ein- und aufgeklappt werden, um Kindelemente ein- oder auszublenden.
An die Property `ItemsSource` von `TreeView` kann ein Objekt vom Typ `IEnumerable` gebunden werden, die alle Objekte enthalten muss, die in der Baumstruktur dargestellt werden sollen. `ItemsSource` wird deshalb an die Property `Project` gebunden, die in `ProjectExplorerContainerViewModel` enthalten ist. Diese stellt eine `ObservableCollection` zur Verfügung, die Objekte vom Typ `IProjectExplorerItem` enthalten muss.
Die Struktur der Klassen, mit denen der Projektexplorer realisiert wurde, wurde in Abschnitt 7.3.3 „Abstraktionsebene 3“ vorgestellt. Wie in dem Klassendiagramm in Abbildung 7.8 zu erkennen ist, wird `IProjectExplorerItem` von der abstrakten Basisklasse `ProjectExplorerViewModelBase` implementiert. Die Klassen `ProjectViewModel`, `FolderViewModel`, `ScriptFileViewModel` sowie `IncludeFileViewModel` leiten von dieser Basisklasse ab und repräsentieren jeweils ein Baumelement des Projektexplorers. Die `ObservableCollection`, die `Project` zur Verfügung stellt, kann somit ausschließlich Instanzen dieser Klassen enthalten.

Mithilfe der Klasse `HierarchicalDataTemplate` können Vorlagen erstellt werden, in denen definiert wird, wie die Objekte von `Projects` visuell präsentiert werden. Für jede Klasse, die `IProjectExplorerItem` implementiert, wurde deshalb eine solche Vorlage definiert.

Die visuelle Präsentation jedes Baumelementes erfolgt durch ein Icon und eine Textbox.
Die Icons wurden mithilfe der `Image`-Klasse eingebunden. Sie stammen von der „Visual Studio Image Library“, die von Microsoft frei zur Verfügung gestellt wird [17].

Die Textbox soll über zwei Modi verfügen, „Editierbar“ und „Nichteditierbar“. Im Modus „Nichteditierbar“ soll der Name des Baumelementes ausgegeben werden, eine Editierung des Namens soll nicht möglich sein. In Modus „Editierbar“ soll der Elementname umbenannt werden können, wobei ein Eingabefeld eingeblendet werden soll. Der bisherige Elementname soll im Eingabefeld ausgewählt und markiert sein. Der Wechsel von Modus 1 in Modus 2 soll entweder über einen zweimaligen Linksklick auf die Textbox oder über das Kontextmenü des Elementes möglich sein. WPF enthält keine Textbox, die die beschriebenen Anforderungen erfüllt. Es wurde deshalb das Paket `InplaceEditBoxLib` verwendet, das in Abschnitt `??` „Abstraktionsebene 1“ beschrieben wurde. Um die Textbox innerhalb von `ProjectExplorerView` einzubinden, wurde die Klasse `EditBox` aus `InplaceEditBoxLib` verwendet. Die Textbox ermöglicht es,

ViewModel

Realisierung mit TreeView

Beschreibung der Funktionen

8.2.4 Erstellung der Konsole

View

Die View der Konsole ist in der XAML-Datei `ConsoleView` definiert. Die von `ConsoleView` erzeugten grafischen Elemente werden in einem *AvalonDock*-Fenster vom Typ `LayoutAnchorable` dargestellt.

Die Konsole besteht aus einer Eingabezeile, die als oberstes Fensterelement angeordnet ist, sowie aus einem scrollbaren Ausgabefeld. Die Ausgabemeldungen sollen untereinander angeordnet werden, wobei die neueste Meldung das unterste Textelement im Ausgabefeld sein soll.

Das Eingabefeld wird mithilfe der Klasse `TextBox` erstellt. Der eingegebene Text wird die `Text`-Property von `TextBox` mit Data Binding an die Property `ConsoleInput` gebunden, die in `ConsoleViewModel` implementiert ist. Der im View eingegebene Text ist damit auch im ViewModel verfügbar.
Eine Texteingabe kann mit der Enter-Taste bestätigt werden, so wie es Anwender von Konsolenanwendungen gewöhnt sind. Um dieses Verhalten zu erreichen, wird in der Property `InputBindings` von `TextBox` ein `KeyBinding` definiert. Dort wird ein Command definiert, der auf die Property `OnEnterPressedCommand` von `ConsoleViewModel` gebunden ist und bei Drücken der Enter-Taste ausgelöst wird.

Das Ausgabefeld könnte auf zwei verschiedene Arten realisiert werden.
Erstens könnte es als großes Textfeld realisiert werden, in dem jede neue Textausgabe in einer neuen Zeile unter der letzten Ausgabe hinzugefügt wird.
Zweitens könnte für jede Textausgabe ein eigenes Textfeld erstellt werden, die einzelnen Textfelder werden dann übereinander gestapelt dargestellt.

Die Implementierung der View wäre für die erste Lösung sehr einfach. Es muss ein Objekt der Klasse `TextBox` definiert werden und an eine `string`-Property im ViewModel gebunden werden. Allerdings muss im ViewModel ein Algorithmus implementiert werden, der dafür sorgt, dass neue Textausgaben stets in einer neuen Zeile dargestellt werden.
Es wurde die zweite Lösung umgesetzt, da dabei im ViewModel auf diesen

Algorithmus verzichtet werden kann. Die Implementierung der View ist bei dieser Lösung dagegen komplexer und wird im Folgenden beschrieben.

Das Ausgabefeld wird bei dieser Lösung mithilfe der Klasse `ItemsControl` erstellt, die über die `ItemsSource`-Property an die Property `ConsoleOutput` vom Typ `ObservableCollection<string>` gebunden wird. `ItemsControl` wird innerhalb eines `ScrollViewer`-Objektes erstellt, dadurch werden die in `ConsoleOutput` enthaltenen Objekte auf einer scrollbaren Fläche dargestellt. `ItemsControl` muss allerdings wissen, wie die Objekte aus `ConsoleOutput` visuell dargestellt werden sollen. Dazu muss ein `DataTemplate` definiert werden. Ein `DataTemplate` ist eine Vorlage, die festlegt, wie ein Objekt präsentiert werden soll.

Innerhalb des `DataTemplate` wird ein `TextBox`-Objekt erstellt, deren Property `Text` auf die `string`-Elemente in `ConsoleOutput` binden. Jeder `string` in `ConsoleOutput` wird somit von einem eigenen `TextBox`-Objekt als Text ausgegeben. Standardmäßig besitzt jede `TextBox` eine schwarze Umrahmung. Indem die Property `BorderThickness` von `TextBox` auf „0“ gesetzt wird, wird dieser Rahmen ausgeblendet. Die Textausgaben erscheinen somit auf einer gemeinsamen Ausgabefläche und sind nicht voneinander abgetrennt.

ViewModel

Die View `ConsoleView` der Konsole ist mit dem ViewModel `ConsoleViewModel` nach dem MVVM-Pattern gekoppelt. `ConsoleViewModel` leitet von `ViewModelBase` ab (siehe Klassendiagramm in Abbildung 7.6) und implementiert das Interface `IAnchorable`, da `ConsoleView` in einem `LayoutAnchorable`-Fenster dargestellt wird. `ConsoleViewModel` stellt verschiedene Properties bereit, an die sich die View bindet. Die wichtigsten Properties sind hier `ConsoleOutput` vom Typ `ObservableCollection` und `ConsoleInput` vom Typ `string`. `ConsoleOutput` stellt der View die `string`-Elemente zur Verfügung, die im Ausgabefeld der Konsole angezeigt werden. `ConsoleInput` enthält alle Zeichen, die aktuell im Eingabefeld der Konsole angezeigt werden.

Die Funktion *WriteOutput()* schreibt einen als Argument übergebenen `string` in das Ausgabefeld der Konsole. Die Funktion ist mit dem Zugriffsmodifizierer PUBLIC deklariert und ist damit die einzige Funktion, auf die von außerhalb der Klasse zugegriffen werden kann. Sie kann deshalb dazu benutzt werden, von anderen Klassen aus Text auf der Konsole auszugeben. *WriteOutput()* wird beispielsweise benutzt, um das vom Pawn-Compiler erzeugten Protokoll (Log) auf der Konsole auszugeben.

Sobald der Benutzer eine Konsoleneingabe mit Enter bestätigt, wird `OnEnterPressedCommand` vom Typ `ICommand` aufgerufen. Dort wird die Funktion *WriteOutput()* auf-

gerufen, die den vom Benutzer eingegebenen Textbefehl in das Ausgabefeld schreibt. Anschließend wird die Funktion *ExecuteInput()* aufgerufen, die den Befehl an eine per USB verbundene und in der Geräteliste als Zielgerät markierte GIGABOX FD sendet. Dabei wird die Klasse `GigaboxFDCommunication` verwendet, die in Abschnitt 8.3 „Kommunikation mit GIGABOX“ näher beschrieben wird.

8.2.5 Erstellung der Geräteliste

View

ViewModel

8.3 `UsbDevicesLib`-Paket

Mit dieser Klasse wird UseCase „Kommunikation mit einer GIGABOX“ realisiert.

Beschreibung der API zu GIGABOX

8.4 `PawnCompilerLib`-Paket

Kapitel 9

Fazit und Ausblick

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] ITB CompuPhase. Pawn Implementer´s Guide , 09.2016. Erhältlich unter `https://www.compuphase.com/pawn/pawn.htm`, abgerufen am 06.09.2017.

[2] ITB CompuPhase. Pawn Language Guide , 01.2016. Erhältlich unter `https://www.compuphase.com/pawn/pawn.htm`, abgerufen am 06.09.2017.

[3] Ulrike Hammerschall and Gerd Beneken. *Software Requirements*. Pearson Deutschland GmbH, 2013.

[4] Andreas M. Heinecke. *Mensch-Computer-Interaktion - Basiswissen für Entwickler und Gestalter* . Springer-Verlag , 2012. 2. Auflage.

[5] D. Louis, S. Strasser, and K. Löffelmann. *Microsoft Visual C# 2005 - Das Entwicklerbuch*. Hrsg.: Microsoft Press Deutschland, 2006.

[6] Microsoft. Übersicht über .NET Framework. `https://msdn.microsoft.com/de-de/library/vstudio/zw4w595w(v=vs.11)`, Stand: 13.10.2015.

[7] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice* . Addison-Wesley, 1998.

[8] Gernot Starke. *Effektive Software Architekturen*. Hanser Verlag, 2009.

[9] Microsoft. The MVVM Pattern. `https://msdn.microsoft.com/en-us/library/hh848246.aspx`, Stand: 04.09.2017.

[10] Thomas Claudius Huber. *Windows Presentation Foundation - Das umfassende Handbuch*. Rheinwerk Verlag, 2016.

[11] icsharpcode. AvalonEdit. Erhältlich unter `http://avalonedit.net`, abgerufen am 06.09.2017.

[12] Xceed. AvalonDock. Erhältlich unter `https://avalondock.codeplex.com`, abgerufen am 06.09.2017.

[13] Dirk Bahle. A WPF MVVM In-Place-Edit TextBox Control. `https://www.codeproject.com/Articles/802385/A-WPF-MVVM-In-Place-Edit-TextBox-Control`, Stand: 06.09.2017.

[14] ITB CompuPhase. Pawn . Erhältlich unter `https://www.compuphase.com/pawn/pawn.htm`, abgerufen am 06.09.2017.

[15] Peter Miller and Scott Finneran. SRecord 1.64 . Erhältlich unter `http://srecord.sourceforge.net/download.html`, abgerufen am 06.09.2017.

[16] Tomasz Watorowski. MightyHID . Erhältlich unter `https://github.com/MightyDevices/MightyHID`, abgerufen am 06.09.2017.

[17] Microsoft. Visual Studio Image Library . Erhältlich unter `https://www.microsoft.com/en-us/download/details.aspx?id=35825`, abgerufen am 12.09.2017.