SICK
Sensor Intelligence.

DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Konzeption und Implementierung einer modularen Testsuite zur Automatisierung von Softwaretests

für die Prüfung zum

Bachelor of Science

des Studienganges Informatik / Informationstechnik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Marvin Böck

Abgabedatum 30.August.2021

Bearbeitungszeitraum	6. Praxisphase
Matrikelnummer	1892597
Kurs	Tinf18B3
Ausbildungsfirma	SICK STEGMANN GmbH Donaueschingen
Betreuer der Ausbildungsfirma	Timo Bayer, M.Sc.
Gutachter der Studienakademie	Prof. Dr. Hans-Jörg Haubner

Erklärung an Eidesstatt

Hiermit erkläre ich, dass ich die vorliegende Abschlussarbeit mit dem Titel: "*Konzeption und Implementierung einer modularen Testsuite zur Automatisierung von Softwaretests*"
eigenständig und ohne fremde Hilfe angefertigt habe. Textpassagen, die wörtlich oder dem Sinn nach auf Publikationen oder Vorträgen anderer Autoren beruhen, sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.
Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

.....	.....
Ort, Datum	Unterschrift

Sperrvermerk

Der Inhalt der Arbeit darf weder als Ganzes noch in Auszügen Personen außerhalb des Prüfungsprozesses und des Evaluationsverfahrens zugänglich gemacht werden, sofern keine anderslautende Genehmigung der SICK STEGMANN GmbH vorliegt

Zusammenfassung

Die vorliegende Arbeit befasst sich mit der Kozeptionierung sowie Proof of Concept Implementierung einer modularen Testsuite für Softwaretests. Grund für die Entwicklung der Suite ist die Portierung eines Bestandsprodukts auf eine neue Mikrocontroller Generation und der damit verbundener erhöhter Testaufwand. Bei der Firma SICK STEGAMNN GmbH ist seit kurzem ein Framework zum automatischen Testen von Software im Einsatz welches die Möglichkeit bietet, entsprechende Softwaretests komfortabel abzudecken. Um die Generierung der Testfälle für das Portierungsprojekt zu erleichtern, soll im Rahmen dieser Arbeit eine Testsuite konzeptioniert, sowie prototypisch implementiert werden. Ziel hierbei ist es, den späteren Prozess der Testentwicklung zu beschleunigen, sowie die Testfiles übersichtlicher und robuster zu gestalten. Zu Beginn der Arbeit wird sich ein Überblick über den Bereich Automated Testing @ GBC07 sowie Software Tests im Embedded Bereich geschaffen. Im Anschluss daran, werden die Testfälle eines vergleichbaren Produkts analysiert und geeignete Testfälle identifiziert. Danach wird eine Softwarearchitektur für die Suite konzeptioniert und diese prototypisch implementiert. Zum Schluss werden die Ergebnisse der Arbeit evaluiert.

Abstract

This thesis deals with the conceptual design and proof of concept implementation of a modular test suite for software testing. The reason for the development of the suite is the porting of an existing product to a new microcontroller generation and the associated increased test effort. The company SICK STEGAMNN GmbH has recently implemented a framework for automatic software testing which offers the possibility to comfortably cover the corresponding software tests. In order to facilitate the generation of test cases for the porting project, a test suite is to be conceptualized and prototypically implemented within the scope of this work. The goal is to accelerate the later process of test development and to make the test files clearer and more robust. At the beginning of the work, an overview of the area of Automated Testing @ GBC07 and software tests in the embedded area is created. Subsequently, the test cases of a comparable product are analyzed and suitable test cases are identified. Afterwards a software architecture for the suite is conceptualized and prototypically implemented. Finally, the results of the work are evaluated.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

MFB Motor-Feedback-System

SRS Software Requirements Specification

CCE Continuous and Customised Engineering

LOC Lines of Code

1 Einleitung

In diesem Kapitel wird neben der Motivation, Zielsetzung und Methodik dieser Arbeit auch auf das betriebliche Umfeld eingegangen. Dabei wird die Unternehmensstruktur erläutert, um den Lesenden einen besseren Überblick über die Aufgabenbereiche sowie Organisation des Projektumfeld zu ermöglichen.

1.1 SICK

Die SICK AG mit Sitz in Waldkirch wurde 1946 durch Dr. E.h. Erwin Sick gegründet. Durch seine innovativ ausgeprägte Denkweise gelang ihm sechs Jahre später mit der Vorstellung seines Unfallschutz-Lichtvorhangs, auf der internationalen Werkzeugmaschinenmesse in Hannover, ein wirtschaftlicher Durchbruch [1]. Heute ist SICK einer der weltweit führenden Hersteller von Sensoren und Sensorlösungen in den Bereichen Logistik-, Fabrik-, und Prozessautomation. Der jährliche Umsatz des Unternehmens beläuft sich auf rund 1,7 Mrd. Euro. Aktuell werden weltweit rund 10.000 Mitarbeiter in fast 50 Tochtergesellschaften beschäftigt.[2] Um als Globales Unternehmen bestehen zu können, ist die SICK AG organisatorisch in SSU´s, GIC´s, GBC´s, Corporate Departments und Corporate Units gegliedert. Jedem Global Business Center (GBC) ist eine spezifische Produktsparte zugeordnet, welche durch die GBC betreut wird. Hierbei liegt der Fokus auf der Entwicklung und Produktion der in der Sparte angesiedelten Produkte. Die Sales and Service Units (SSU) betreuen den Vertrieb der in den GBC´s gefertigten Produkte. Ein Global Industry Center (GIC) bietet branchenspezifische Gesamtlösungen. Coporate Departments und Corporate Units werden auch als Zentralbereiche bezeichnet. Sie sind GBC übergreifend für alles zuständig was nicht direkt mit Entwicklung, Produktion und Vertrieb in Verbindung steht. Die Organisationsstruktur ist in der Abbildung "Unternehmensstruktur" dargestellt.

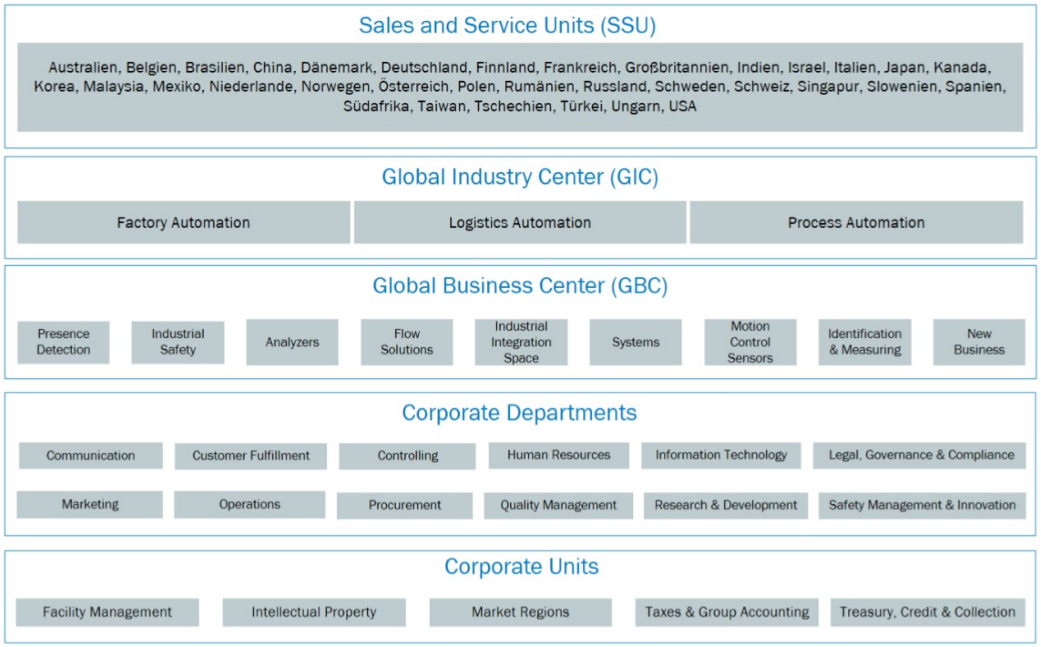

Abbildung 1: Unternehmensstruktur [2]

1.2 SICK STEGMANN GmbH

Die Firma STEGMANN GmbH mit Standort in Donaueschingen wurde 1956 von Max Stegmann gegründet. Ihr Produktportfolio bestand zu Beginn hauptsächlich aus Kleinmotoren, Getrieben, Programmsteuerungen und feinmechanischen Baugruppen. 1960 wurde der erste inkrementelle Encoder entwickelt. 2002 wurde Stegmann durch die SICK AG aufgekauft und als Tochtergesellschaft in das Unternehmen eingegliedert. Die SICK STEGMANN GmbH (GBC 07) gliedert sich in drei Business Center. Die BU71 ist zuständig für Motor Feedback Systeme, BU72 für Encoder und Neigungssensoren und die BU73 für Linear Encoder.[2]

1.3 Hardeware / CCE Team

Das Team Hardeware und Continuous and Customised Engineering (CCE) ist Bestandteil der BU71. Das Hardeware Team beschäftigt sich vorwiegend mit der Entwicklung der für Motor-Feedback-System (MFB) benötigten Hardware Komponenten. CCE steht für Continuous and Customised Engineering.

Das Team befasst sich mit der Anpassung von Bestandsprodukten nach kundenspezifischen Anforderungen sowie der Anpassung von Bestandsprodukten, zum Beispiel bei der Änderung eines Bauteils oder kleineren Anpassungen der Firmware.

1.4 Problemstellung

Der SKS bzw. SKM ist ein Motorfeedbacksystem, welches bereits seit mehreren Jahren durch die SICK STEGMANN GmbH vertrieben wird. Trotz der langen Lebenszeit des Produktes ist es immer noch eines der Absatz stärksten MFB im Portfolio von SICK. Innerhalb des Gerätes dient ein ST7 Mikrocontroller als Recheneinheit. Dieser stellt u. a. die Kommunikation via Hiperface bereit. Der ST7 ist ein 8bit Mikrocontroller der Firma STMicroelektronics, welcher auf einer Von-Neumann-Architektur basiert[3]. ST hat bekannt gegeben, dass der ST7 in naher Zukunft das End of life erreicht und somit keine neuen ST7 mehr verfügbar sein werden. Aus diesem Grund ist SICK gezwungen eine Umstellung von ST7 auf den Nachfolger ST8 vorzunehmen. Im Rahmen der Umstellung muss eine Portierung der MFB Software stattfinden. Im Zuge der Softwareportierung wird es zu einem erhöhten Testaufwand für das Produkt kommen. Da neben dem SKS bzw. SKM in Zukunft auch noch weitere MFB portiert werden müssen, ergibt sich die Notwendigkeit einer universell einsetzbaren Testsuite, welche die Kernfunktionalitäten der Software im Rahmen des Automated Testing @ GBC07 Umfeldes abprüfen kann.

1.5 Zielsetzung

Das Ziel der Arbeit ist die Entwickelung, sowie prototypische Implementierung einer universellen Testsuite. Die Testsuite soll so gestaltet werden, dass sie die Kernfunktionalitäten des Controllers, wie zum Beispiel die Kommunikation über das Hiperface Protokoll abdeckt. Diese Kernfunktionalitäten werden auch von anderen Produkten genutzt. Ziel ist es bei der späteren Portierung des Controllers und den damit verbundenen Tests, auf die Testsuite zurückgreifen zu können und mithilfe der darin vorhandenen Testfälle

den Validierungsprozess zu beschleunigen. Weiterhin soll die Testsuite so gestaltet werden, dass sie auch für später folgende weitere Portierungen von ähnlichen Drehgebern verwendet werden kann.

1.6 Vorgehensweise

Zu Beginn des Projektes wird ein Überblick über die verschiedenen Themenbereiche (Hiperface Schnittstelle, MFB, Automated-Testing etc.) geschaffen, sowie eine Projektplanung vorgenommen. Die Projektplanung umfasst hierbei u.a. die Zeitplanung. Aufgrund der Kürze des Projektes wurde hierbei auf eine Wochenplanung gesetzt. Die Planung sieht fünf Projektphasen vor (Einarbeitung, Analyse und Konzeption, Implementierung und Verifikation). Besonderes Augenmerk wird auf die Konzeptionsphase gelegt. Im Rahmen der Einarbeitungsphase werden die bereits existierenden Projekte betrachtet und eine Literaturrecherche durchgeführt. In der Analysephase werden die Testfälle anderer Produkte analysiert. Weiterhin wird geprüft, welche Testfälle ggf. übernommen werden können. In der Konzeptionsphase wird der Aufbau der Suite sowie die konkreten Testfälle definiert. Hierbei spielt die Planung der Softwarearchitektur eine große Rolle, da diese als Grundlage für den Erfolg des Projekt dient. Die Architektur soll hierbei auch später noch offen für Erweiterungen sein. In diesem Projektschritt kann teilweise auf vorhandenen Quellcode zurückgegriffen und dieser ggf. wiederverwendet werden. Die Implementierung erfolgt als proof of concept, da zum voraussichtlichen Ende des Projektes die Portierung noch nicht abgeschlossen sein wird. Um das Ergebnis des Projekts zu Verifizieren findet am Ende ein Abgleich mit den definierten Anforderungen statt.

2 Related Work

In diesem Kapitel werden verschiedene Literaturquellen diskutiert, welche im Zusammenhang mit dem Thema der Arbeit stehen. Hierbei wird der Inhalt kurz zusammengefasst und erläutert, in welchem Zusammenhang die Arbeiten stehen.

Optimized test suites for automated testing using different optimization techniques

In diesem Artikel, welcher von Manju Khari, Prabhat Kumar, Daniel Burgos und Rubén González Crespo geschrieben wurde, befassen sich die Autoren mit der Optimierung von automatisierten Testsuites unter Betrachtung verschiedener Optimierungstechniken. Hierbei werden Tools zur automatischen Generierung von Testsuites verglichen. Die Ergebnisse werden dann im Kontext des Automated Testing betrachtet. Die Arbeit zeigt auf, welche Möglichkeiten bei der Optimierung und automatischen Generierung von Softwaretests bestehen. Im Zuge des konkreten Projekts wird ebenfalls eine Testsuite erstellt. Die Betrachtung von eventuellen Optimierungsmöglichkeiten ist für den späteren Verlauf des Projekts durchaus von Interesse.[4]

Eine Technologe für das durchgängige und automatisierte testen eingebetteter Software

Die Arbeit, welche durch Dipl.-Inform. Till Fischer im Rahmen seiner Dissertation an der Fakultät für Elektrotechnik und Informationstechnik des Karlsruher Instituts für Technologie (KIT)durchgeführt wurde, werden neben den Grundlagen des Testens von eingebetteten Systemen auch die verschiedenen Testebenen diskutiert. Ziel der Arbeit ist die Verbesserung der Durchgängigkeit des Testprozesses für eingebettete Systeme. Als Lösungsstrategie wird durch Fischer eine Testlösung versiert, welche den Quelltext der ausgeführten Software, die Netzwerkkommunikation, sowie das physikalische Verhalten an elektrischen Schnittstellen und der simulierten Umgebung abdeckt. Hier ist

der Bezug zum in dieser Arbeit behandelten Automated Testing @ GBC07 Projekt zu erkennen. [5]

3 Grundlagen Softwareentwicklung und Verifikation

Dieses Kapitel beschreibt die Grundlagen aus softwaretechnischer Sicht. Da sich die Arbeit mit dem Thema Softwaretests, sowie Testautomatisierung beschäftigt, stellt dieser Themenbereich einen großen Teil der Grundlagen dar. Zu Beginn wird auf Grundlagen zur Entwicklung bzw. Architektur von Software eingegangen. Es werden die verschiedenen Teststufen nach dem V-Modell beschrieben und erläutert. Im Anschluss werden die Grundlagen des automatisierten Testens, sowie das SICK eigene Automated Testing @ GBC07 Projekt näher erläutert.

3.1 Software Entwicklung

Der Softwareentwicklungsprozess ist ein komplexer Vorgang welcher bei jedem größeren Softwareprojekt durchlaufen wird. Er dient dazu die Entwicklung einer Software klar zu strukturieren.[6] Heute gibt es eine Vielzahl verschiedener Vorgehensmodelle zur Softwareentwicklung, zum Beispiel Kanban, Wasserfallmodell, RUP etc., welche alle verschiedene Ansätze verfolgen. Eines der ältesten, jedoch noch gebräuchlichen Modelle, ist das V-Modell. Das V-Modell beschreibt die fünf Phasen des Entwicklungsprozesses in absteigender Reihenfolge:[7]

- Anforderungsanalyse
- Systementwurf
- Architekturentwurf
- Modulentwurf
- Codierung

3.1.1 Anforderungsanalyse

Im Rahmen der Anforderungsanalyse werden die Leistungsanforderungen an das Softwareprojekt ermittelt. Hierzu ist ein enger Kontakt zum Auftraggeber

notwendig. Man unterscheidet zwei Kategorien von Anforderungen:

- Funktionale
- Nicht Funktionale

Funktionale Anforderungen beschreiben Funktionen, welche das System konkret anbieten muss, zum Beispiel addieren und subtrahieren bei der Entwicklung eines Taschenrechners. Nicht funktionale Anforderungen sind zum Beispiel das Erfüllen einer Coding-Guideline, oder die universelle Verwendbarkeit einer Software. Die nicht funktionalen Anforderungen lassen sich weiterhin in Prozess und Qualitätsanforderungen unterteilen.[8]. Das Ergebnis der Anforderungsanalyse ist ein Pflichtenheft (Software Requirements Specification (SRS), welches im Anschluss als Grundlage für die Entwicklung der Software und weiterhin auch bei der Endabnahme durch Kunden dient.

3.1.2 Systementwurf

Der Systementwurf im V-Modell ist die Beschreibung des prinzipiellen Aufbaus und der querschnittlichen Eigenschaften des zu entwickelnden Systems, ohne hierbei auf die Funktionsweise einzelner Systemelemente im Detail einzugehen[9]. Hierbei wird die grobe Architektur des Systems entwickelt und zum Beispiel auch die Verteilung von Servern und Clients berücksichtigt.

3.1.3 Architekturentwurf

Die Architektur der Software ist die Grundlage jedes Programms. Durch sie werden die einzelnen Komponenten, sowie deren Schnittstellen zueinander beschrieben.

> "Die grundlegende Organisation eines Systems, dargestellt durch dessen Komponenten, deren Beziehungen zueinander und zur Umgebung sowie den Prinzipien, die den Entwurf und die Evolution des Systems bestimmen." [10]

Im Unterschied zum Systementwurf liegt hierbei der Fokus rein auf den verschiedene Softwarekomponenten. Bei der Entwicklung einer Architektur sollten die Prinzipien der Softwareentwicklung wie zum Beispiel KISS, Kopplung und Kohäsion, Seperation of Concerns etc. beachtet und angewendet werden.[8]. Die Architektur hat einen großen Bestandteil daran, die Software skalierbar, persistent und effizient zu gestalten. Für die Entwicklung einer konkreten Architektur stehen verschiedene Architekturmuster zur Verfügung. Schichtenarchitekturen, Pipe-Filter Modelle, das MVC Design und das Plugin-Architekturmuster sind nur einige Beispiele.

Schichtenarchitekturen

Eine der am häufigsten verwendeten Architekturen ist das Schichtenmodell.[11] Hierbei wird das System in verschiedene Schichten eingeteilt. Die einzelnen Schichten bauen aufeinander auf, sodass eine höhere Schicht auf die niedrigeren zugreifen kann, jedoch nicht auf die Schichten über ihr. Bei einer strikten Schichtenarchitektur darf darüber hinaus nur auf die direkt darunter liegende Schicht zugegriffen werden. Man unterscheidet bei dieser Art der Architektur verschiedene Modelle, so kommen zum Beispiel zwei-Schicht, drei-Schicht usw. Modelle zum Einsatz. Bei einer zwei-Schicht Architektur wird das Projekt im Allgemeinen in eine Anwendungs- und eine Datenerhaltungsschicht gegliedert. Bei einer drei-Schicht wird die Anwendungsschicht in eine Dialog- (Benutzeroberfläche) und eine Fachkonzeptschicht aufgeteilt. Je mehr Schichten zur Architektur hinzugefügt werden, desto feiner kann die Software gegliedert werden.[11] Besonders bekannt sind Schichtenarchitekturen aus der Netzwerktechnik. Das ISO/OSI Modell ist eine sieben-Schichten Architektur.

3.1.4 Modulentwurf

Nachdem die konkrete Softwarearchitektur entwickelt wurde, liegt der Fokus in diesem Schritt darauf die Software in einzelne Teile zu zerlegen und diese der Architektur entsprechend, aufzubauen. Einzelne Softwarebestandteile, welche sich eigenständig Kapseln lassen, werden als Module bezeichnet. Die verschiedenen Module haben Schnittstellen durch welche auf ihre Funktiona-

lität zugegriffen werden kann. Die umsichtige Klassifikation in Komponenten ist von großer Bedeutung, da diese gerade in Bezug auf die Wiederverwendbarkeit der Komponenten für andere Softwareprojekte eine große Rolle spielt. Eine Möglichkeit der Darstellung verschiedener Module ist zum Beispiel die Form des Klassendiagramms.

3.2 Warum Software Tests?

Bei der professionellen Softwareentwicklung sind Softwaretests heute unabdingbar. Fehler in Software haben besonders in den letzten Jahren vermehrt dramatische Folgen gehabt. Der Absturz mehrerer Boeing 787 Max, für welchen wohl Softwareprobleme die Ursache waren [12] ist nur ein Beispiel aus vielen. Aber auch wenn Softwarefehler nicht solch katastrophale Folgen haben können sie hohe Kosten verursachen. Werden Fehler frühzeitig entdeckt und behoben, lassen sich diese Probleme abmildern. Die Kosten eines Fehlers steigen mit der Zeit seiner Existenz.[13] Um die Folgen fehlerhafter Software abzumildern spielt das Testen von Software in heutigen Entwicklungsprozessen eine zentrale Rolle. Als Leitlinie für das Durchführen und Planen von Softwaretests stehen verschiedenen Normen zur Verfügung. Als wegweisende Norm für Software-Systemtests wird der IEEE Standard 829 (Test Documentation) gesehen. Dieser wurde bereits 1983 eingeführt und in der zwischen Zeit mehrfach überarbeitet. Der Standard beschreibt den Systemtestprozess und die Testdokumentation. [14, S. 21] Weiterhin werden Softwaretests im ISO-Standard 9126 sowie IEC-29119 behandelt. Der Testprozess wird nach IEEE 829 in sieben Schritte eingeteilt:

- Testplanung
- Testfallspezifikation
- Testentwurf
- Testdarstellung (Ist und Soll)
- Testausführung

- Testauswertung
- Testabnahme

Der ISO-Standard 9126 hingegen befasst sich eher mit Softwarequalität im Allgemeinen. In der Norm ist definiert was für die Messung der Softwarequalität erforderlich ist. Im ISO Standard IEC-29119 hingegen ist der Testbegriff, Dokumentation, Testprozess, sowie die einzelnen Testtechniken definiert. Hier ist zum Beispiel der Begriff sowie der Aufbau von Unit Tests definiert.[14] Aufgrund der zentralen Rolle von Softwaretests verfolgen einige Entwickler das Konzept des Testdriven Development, welches vorsieht den produktiv Code auf Grundlage der zuvor erstellten Tests zu entwickeln.

3.3 Arten von Softwaretests

In der modernen Softwareentwicklung wird zwischen verschiedenen Arten von Software Tests unterschieden. Die Einteilung gestaltet sich schwierig, da eine große Vielfalt an verschiedenen Begriffen bzw. Bezeichnungen besteht[13]:

- Abnahmetests
- Systemtests
- Integrationstests
- Modultests (Unit Tests)

Das "V-Model" zeigt die Entwicklungsphasen und die entsprechenden Testphasen in hierarchischer Ordnung auf.

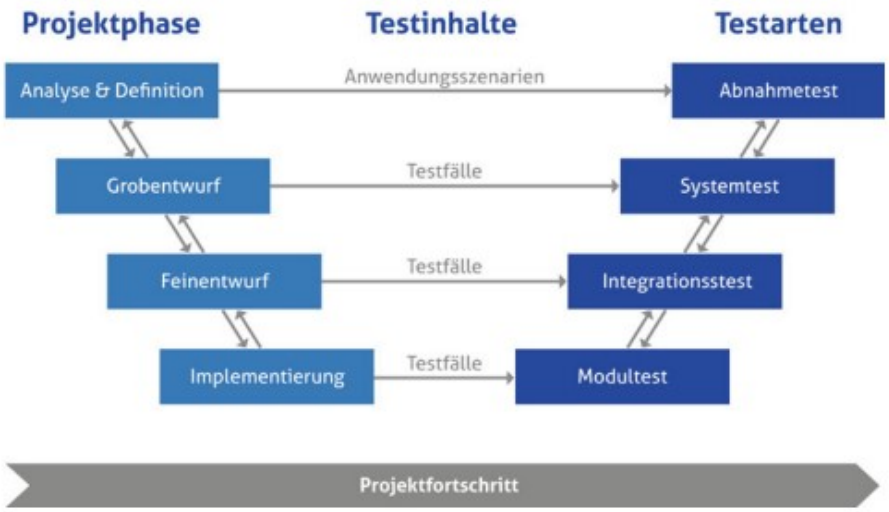

Abbildung 2: V-Model [14]

3.3.1 Modultests

Als Modultests oder auch Unit Tests werden Tests beschrieben, welche das Design einer Softwarekomponente testen. Der Test läuft also gegen die Funktionalität der Komponente selbst. Man versucht dabei, die kleinste Sinnvolle Einheit zu überprüfen [15]. Unit Tests sind ein dynamisches Testverfahren. Meist werden sie als erste Tests nach dem Schreiben des Codes angewendet. Das Prinzip des Unit-Testing beruht darauf, dass Software aus einzelnen Modulen besteht, welche zu einem gesamten Programm zusammengesetzt werden. Durch Unit Tests werden diese einzelnen Module (Komponenten) isoliert getestet und geprüft, ob jede Komponente für sich korrekt arbeitet. Ein Beispiel wäre zum Beispiel das Abprüfen eines Umrechnungstools für Einheiten in einer Kalorienzähler App. Ein beispielhafter Unit Test ist in Abbildung "Einfacher Unit Test" dargestellt.

```
[Fact]
| 0 Verweise | tobi4321, vor 11 Tagen | 1 Autor, 2 Änderungen | 0 Ausnahmen
public void ConvertLiterToMl()
{
    // Arrange
    // input variable for the parser function
    double liter = 1.6;

    // Act
    // parse the value of liter into an other unit
    double milliliter = UnitParser.LToMl(liter);

    // Assert with precision of 4 decimal places
    Assert.Equal(1600.0,milliliter,4);
}
```

Abbildung 3: Einfacher Unit Test [16]

Bei einem Unit Test sollte zuerst die Funktion der Komponente geprüft werden. Im Falle eines Umrechnungstools wären hier zum Beispiel Tests mit erwarteten Daten zielführend. Nachdem die Grundfunktionalität getestet ist, sollten die Grenzfälle betrachtet und Fehlerfälle eingebaut werden. Grenzfälle können zum Beispiel Werte am Rand des definierten Wertebereiches der Funktion sein. Fehlerfälle können zum Beispiel die Eingabe eines negativen Wertes statt eines erwarteten positiven, oder die Verwendung von Werten außerhalb des Wertebereichs sein. In der Regel wird für jede Teilkomponente ein eigener Unit Test erstellt, welcher die Funktion dieser Teilkomponente unabhängig von den anderen Komponenten betrachtet. In der Realität lassen sich jedoch die Komponenten oft nur schwer von einander Abgrenzen. Ein Beispiel hierfür ist der Aufruf einer anderen Komponente und das Verarbeiten des Rückgabewertes. Bei der Implementierung muss der Aufruf der Unterfunktion bzw. deren Rückgabewert simuliert werden, um daraus resultierende Fehlerfälle auszuschließen bzw. mit Absicht herbeizuführen. Die Simulation einer Unterkomponente wird auch als Stub bzw. Mock bezeichnet.[15] Für die Entwicklung und Durchführung von Unit Tests wird meist ein unterstützendes Testframeworke verwendet. Beispiele für Testframeworkes sind u.a. JUnit, TestNG oder Spock.

Bei der Erstellung kann die AAA-Normalform herangezogen werden, diese definiert die drei Hauotkomponenten jedes Unit Tests[13]:

- Arrange -> Initialisieren der Test-"Welt"
- Act -> Ausführen der zu testenden Aktion
- Assert -> Prüfen der Test-Zusicherung

Die ATRIP Regeln gelten als Grundlage für die Erstellung qualitativ guter Unit Tests[13]:

- Automatic
- Thorough
- Repeatable
- Independent
- Professional

Unit Tests sollten automatisch ablaufen (Automatic), sie sollten gründlich sowie wiederholbar sein (Thorough & Repeatable). Weiterhin sollten keine Abhängigkeiten zwischen den einzelnen Tests bestehen (Independent) und die einzelnen Tests sollten mit Sorgfalt erstellt worden sein (Professional). Die Testabdeckung kann nach verschiedenen Methoden ermittelt werden. Zu den gängigen Methoden zählen unter anderem die Line Coverage und die Branch Coverage. Bei der Line Covergage wird lediglich gemessen, wie viele Codezeilen mittels Tests durchlaufen werden. Bei der Branch Coverage wird geprüft, ob der Test alle Verzweigungen zum Beispiel bei einem if Statement mindestens einmal durchlaufen hat.[15] Je nach Art und Umfang des Programmes kann es sehr aufwendig sein jede Komponente einer Software mit Unit Tests abzudecken. Dennoch ist eine Testabeckung von über 80% anzustreben. Bei der Erstellung von Tests sollt jedoch stets das Optimum zwischen Aufwand und Kosten gefunden werden.[13]

3.3.2 Integrationstests

Bei einem Integrationstest wird im Gegensatz zu Unit Tests nicht eine einzige Komponente getestet, sondern das Zusammenspiel mehrerer Komponenten. Dies bedeutet, dass in der Regel bereits verschiedenen Unit Tests von den Komponenten durchlaufen wurden. Somit kann davon ausgegangen werden, dass die Komponenten für sich isoliert fehlerfrei funktionieren. Ziel der Integrationstests ist es Fehler aufzudecken, welche das Zusammenspiel verschiedener Komponenten betreffen. Mithilfe eines Integrationstests lassen sich zum Beispiel inkompatible Schnittstellenformate oder Timing Probleme bei der Übergabe von Daten aufdecken [14]. Es wird zwischen verschiedenen Integrationsteststrategien unterschieden. Bei der Bottom-Up Methode ("Ablauf Bottom-Up Test") erfolgen die Tests in umgekehrter Richtung zur Benutzer Beziehung, also von innen nach außen. Durch dieses Vorgehen spart man sich das Erstellen der Testrümpfe, da nur die Testtreiber programmiert werden müssen. Ein Nachteil hierbei ist, dass Fehler in der obersten Schicht

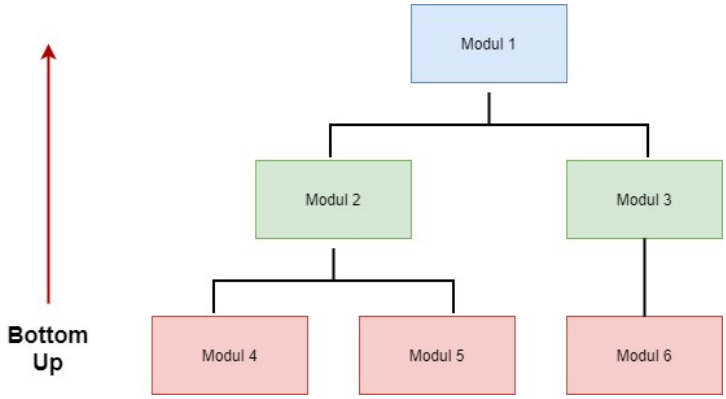

Abbildung 4: Ablauf Bottom-Up Test

erst sehr spät erkannt werden. Im Gegensatz dazu wird bei der Top-Down Methode in Richtung der Benutzer Beziehung getestet. Für diese Strategie werden zwar Testrümpfe benötigt, jedoch werden Fehler in der obersten Schicht früher erkannt. Eine weitere Möglichkeit ist der Urknalltest, hierbei werden alle Komponenten getrennt entwickelt und in einem Schritt integriert. Dieses Vorgehen erschwert allerdings die Fehlersuche. Neben diesen drei Methoden kommt auch eine Hybride Mischung aus Bottom-Up bzw. Top-Down

zum Einsatz.

3.3.3 Systemtests

Systemtests sind Tests welche das gesamte System gegen dessen Anforderungen testen. Üblicherweise wird als Testumgebung ein Abbild der späteren Produktivumgebung gewählt und der Test mit entsprechenden Testdaten (zum Beispiel Datenbankeinträgen, etc.) durchgeführt. Wichtig hierbei ist, dass ein Abbild der Produktivumgebung geschaffen wird und diese nicht selbst zu verwenden. Unter Umständen kann es sonst zu Schäden bzw. Ausfällen an der Produktivumgebung kommen. Weiterhin sind die Test dadurch aber auch schwer reproduzierbar. Bei einem Systemtest werden sowohl funktionale als auch nicht funktionale Qualitätsmerkmale der Software getestet[14]. Hierzu wird zwischen funktionalen und nicht funktionalen Tests unterschieden. Funktionale Tests prüfen, ob die Software, die in der Spezifikation geforderten Funktionen bereitstellt. Nicht funktionale Tests prüfen Zuverlässigkeit, Benutzbarkeit, Effizienz, Veränderbarkeit und Übertragbarkeit. In Abbildung "Systemtest" wird ein Überblick über mögliche Systemtests dargestellt.

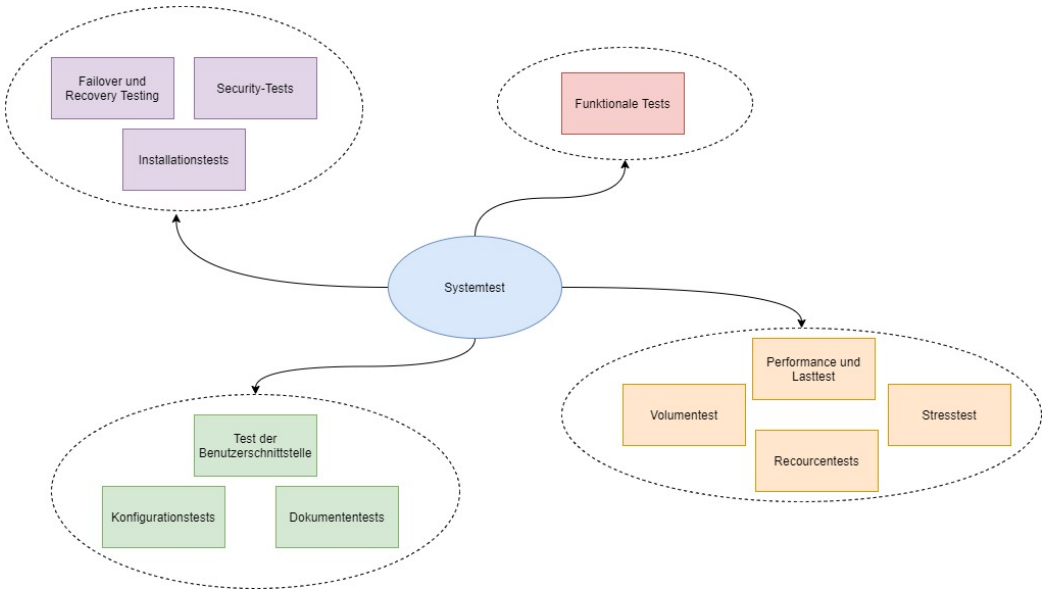

Abbildung 5: Systemtest

Systemtests gelten als der wichtigste Testschritt:

> "Systemtests sind die wichtigste Teststufe. In Systemtests werden Designfehler,Inkompatibilitäten zur Hardware, zeitabhängige Fehler und zum Teil auch Lücken in der Spezifikation gefunden." [15, S. 195]

Systemtests können aufgrund der breiten Testabdeckung sehr umfangreich und aufwendig sein. Eine Automatisierung der Tests macht jedoch nur Sinn, wenn die Tests mehrfach wiederholt werden sollen, da die Implementierung automatischer Tests einen etwa zehnfach höheren Aufwand bedeutet[17]. Die Messung der Testabdeckung lässt sich bei Systemtests ebenfalls nur schwer realisieren. [15]

3.3.4 Abnahmetests

Im Gegensatz zu den bisher behandelten Teststufen wird der Abnahmetest meist nicht vom Hersteller, sondern direkt vom Kunden durchgeführt. Der Test hat nicht das Ziel eventuelle Fehler aufzudecken, sondern das Vertrauen des Kunden in das Produkt bestärken. Der Abnahmetest ist hierbei fester Bestandteil des Vertrags zwischen Kunde und Lieferant, sowie ein Meilenstein in der Projektplanung.
Durch einen erfolgreichen Abnahmetest werden verschiedene Ereignisse, wie zum Beispiel Zahlungsfristen oder Garantievereinbarungen getriggert. Um Probleme während des Abnahmetests zu verhindern sollten die Rahmenbedingungen (Test Umgebung und Anforderungen an den Test) im Vorhinein Vertraglich geregelt sein. Um repräsentative Ergebnisse zu erhalten kann es auch angebracht sein, den Abnahme- bzw. Akzeptanztest mit verschiedenen Benutzergruppen durchzuführen.[14]

3.4 Testsuite

Als Testsuite wird eine Zusammenstellung mehrerer Testfälle für den Test einer Komponente oder eines System bezeichnet, bei der die Nachbedingungen des einen Tests als Vorbedingungen des folgenden Tests genutzt werden sollen. [18] Durch die Verwendung einer Testsuite lassen sich Software Tests einfacher automatisieren und wiederverwenden. Durch den Einsatz von Testsuites lassen sich die Testfälle aus dem Testplan besser clustern. Ein möglicher Aufbau eines Testkonzepts ist in Abbildung "Testsuite" schematisch dargestellt.

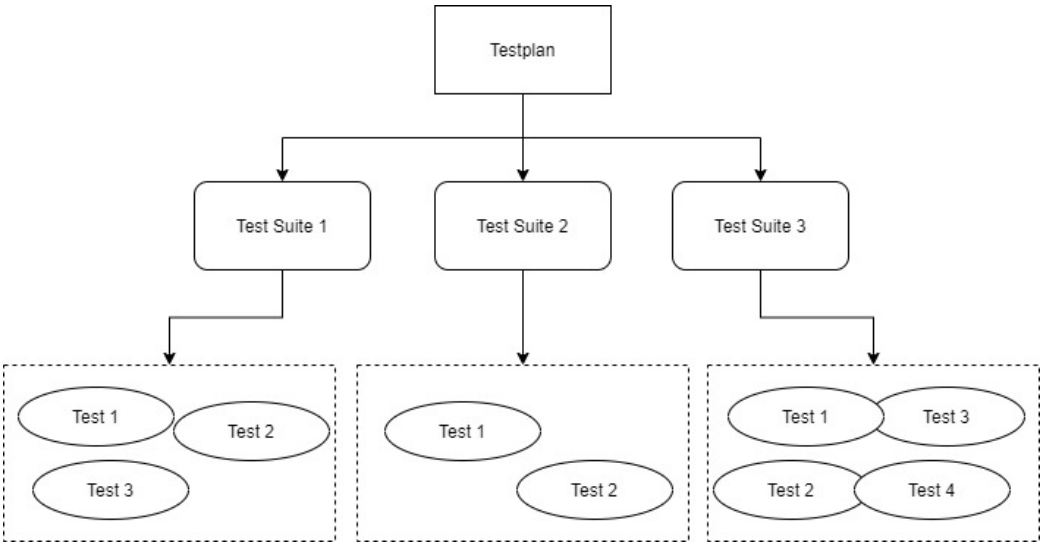

Abbildung 6: Testsuite

3.5 Testautomation

Im Rahmen der Softwareentwicklung sollten Softwartests schon frühzeitig in den Entwicklungsprozess eingebunden werden. Das Konzept einer CI Pipeline, wie es heute in vielen Entwicklerteams zum Einsatz kommt, sieht Softwaretests als klaren Bestandteil des Entwicklungsprozess an. Eine CI Pipeline ist in Abbildung "CI/CD Prozessablauf" zu sehen.

Abbildung 7: CI/CD Prozessablauf [19]

Bei einer strikten Umsetzung des CI Prozesses befindet sich im Repository nur getesteter Code[19]. Zur Vereinfachung dieses Prozesses werden die Testvorgänge automatisiert. Durch die Automatisierung von Tests wird sowohl die Reproduzierbarkeit gesteigert, als auch die Testabdeckung erhöht. Weiterhin werden durch automatisierte Tests Kosten und Zeit gespart[14]. Jedoch ergeben sich durch automatisiertes Testen auch Nachteile. So können Testergebnisse nicht schon während der Testdurchführung interpretiert werden. Außerdem ist der Aufwand für die Implementierung höher als bei manuellen Test[14]. Bei der Automatisierung von Tests geht es vorwiegend um die automatische Ausführung und weniger um die maschinelle Erstellung von Tests. Hierfür stehen eine Vielzahl von verschiedenen Tools zur Verfügung. Im Kapitel "Projekt Automated Testing @ GBC07" wird das bei SICK STEGMANN verwendete Tool näher erklärt.

3.6 Projekt Automated Testing @ GBC07

Das Projekt Automated Testing @ GBC07 ist ein Projekt, welches bereits seit mehren Jahren durch die Abteilung Research and Development der SICK STEGMANN GmbH vorangetrieben wird. Ziel des Projektes ist es, die Testprozesse im Unternehmen auch über verschiedene Abteilungen hinweg zu vereinfachen und gemeinsame Standards zu definieren. Das Projekt umfasst die gesamte Testumgebung sowohl im Software als auch Hardware Bereich. Der grundsätzliche Aufbau des Projekts ist in der Abbildung "Aufbau Automated Testing @ GBC07" dargestellt. Zu erkennen ist, dass der komplette Testablauf automatisiert wurde. Durch den Aufbau des Projekts lassen sich die verschiedene Testarten vom Unit Test bis hin zum Systemtest durchführen. Für den Standardentwickler ist der Testaufwand so deutlich geringer. Er hat die Möglichkeit die entsprechenden Tests in einem Einheitlichen Frameworke zu Entwicklern und auch Tests von anderen Projekten bzw. Entwicklern wiederzuverwenden. Die Tests werden in der SICK eigenen Programmiersprache ITE oder in Python geschrieben. ITE ist an C++ angelehnt, wurde jedoch an einigen Stellen speziell auf die Bedürfnisse des SICK Konzerns angepasst. Herzstück des Testsystems ist der Test Automation Slave. Dieser enthält alle Komponenten, welche zur Konfiguration und Ausführung der Tests benötigt werden. Das Modul Access and Test Creation dient zur Konfiguration des Testablaufs. Die Persistenzschicht dient zur Ablage der erstellten Testpläne. Der Test Controller dient im Anschluss zur Ausführung der Testpläne. Das Modul Test Cases enthält die konkreten Testfälle auf welche der Test Controller je nach Anforderung des Testplans zurück greift.[20]

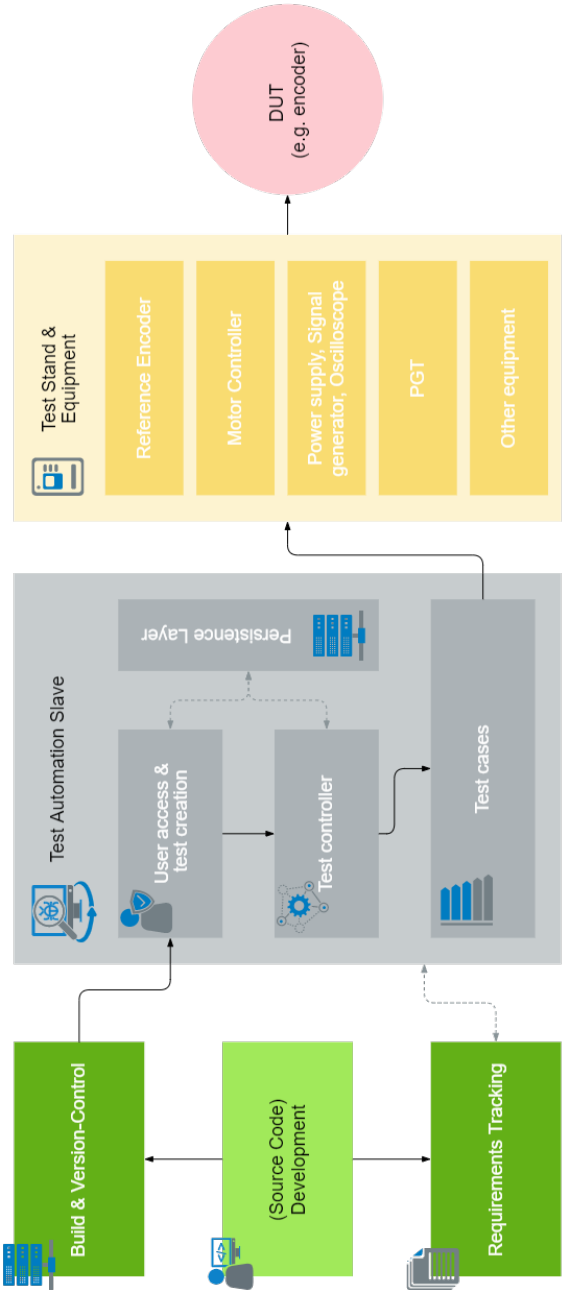

Abbildung 8: Aufbau Automated Testing @ GBC07 [20]

4 Technische Grundlagen

Im Kapitel Technische Grundlagen werden die Hardware Grundlagen wie zum Beispiel Aufbau und Funktionsweise eines Drehgebers oder die Funktionsweise des Hiperface Protokolls beschrieben. Durch dieses Kapitel soll ein Grundwissen über die verwendete Hardware geschaffen werden, welches für das Verständnis der folgenden Konzeptionierung, sowie verschiedener Designentscheidungen notwendig ist.

4.1 Encoder/MFB

Encoder und Motor-Feedback-Systeme dienen klassischerweise zur Messung rotatorischer Bewegungen. Sie wandeln hierbei die Winkel zweier sich relativ zueinander drehender Objekte in ein elektrisches Signal um. MFB und Encoder unterscheiden sich hierbei vorwiegend in ihren Einsatzgebieten. MFB werden üblicherweise in Elektromotoren eingesetzt, wohingegen das Einsatzgebiet von Encodern breiter gefasst ist. Ein Encoder wir hierbei als Lastgeber (misst an der Lastachse), ein MFB als Motorgeber bezeichnet.[17, S.1] Der technische Aufbau der Systeme ist im Grundsatz gleich, er gliedert sich in folgende drei wesentliche Bestandteile:

- Sender
- Modulator
- Empfänger

Drehgeber messen die Drehbewegung einer rotativen Achse in Bezug zu einem Referenzpunkt, d.h. den Winkel zwischen der Aktuellen Position und dem Referenzpunkt. Die Abbildung "Winkel bei rotatorischer Bewegung" stellt einen Einheitskreis mit einem Radius von eins dar, in welchem sich ein Zeiger entsprechend der Rotation des Motors bewegt. Die x und y-Komponente des Zeigers können wiederum als Sinus- bzw. Cosinus-Werte dargestellt werden. Dies ist in Abbildung "Sinus- und Cosinus Darstellung beim Einheitskreis" abgebildet. Durch diese Betrachtung lässt sich der Winkel zwischen

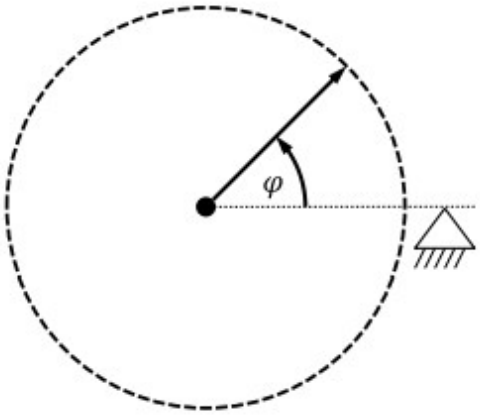

Abbildung 9: Winkel bei rotatorischer Bewegung[17]

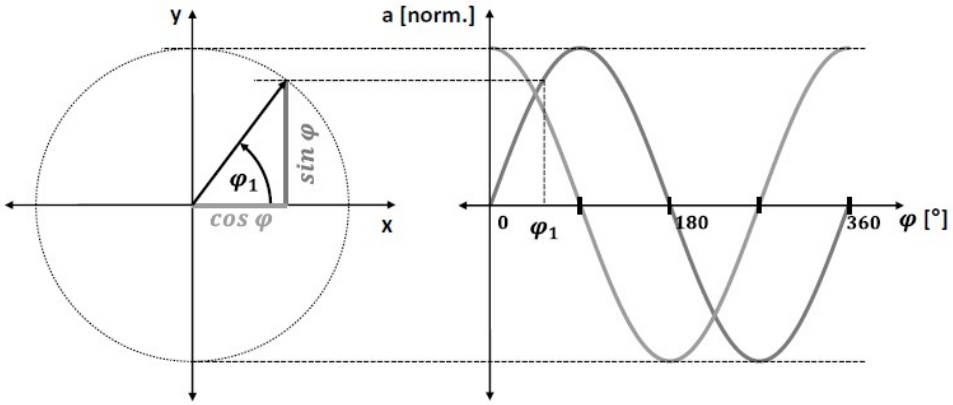

Abbildung 10: Sinus- und Cosinus Darstellung beim Einheitskreis [17]

Ausgangsposition und aktueller Position mittels Trigonometrischer Funktionen berechnen (Goniometrie). Die Formel lautet dementsprechend:[17]

$$\varphi = arctan\left(\frac{a_{sin}}{a_{cos}}\right)$$

Um eine höhere Auflösung zu erreichen, wird in der Praxis eine Umdrehung nicht als eine Sinus bzw. Cosinus dargestellt, sondern der Vollwinkel in mehrere Teilwinkel unterteilt. Diese werden dann wiederum durch eine Sinus- bzw. Cosinus-Periode repräsentiert. Ein Drehgeber liefert demnach als Aus-

gangssignal Sinus bzw. Cosinus Werte, aus welchen sich die aktuelle Lage der Welle berechnen lässt. Im Bereich der Drehgeber gibt es eine Vielzahl verschiedener Ansätze zur Bestimmung dieses Winkels, bzw. der aktuellen Position des Zeigers. Eine Klassifizierung ist zum Beispiel anhand des sensorischen Messprinzips möglich. Hierbei wird zwischen optisch, magnetisch, induktiv, kapazitiv und resistiv potentiometrisch unterschieden. Da sich diese Arbeit konkret mit dem SKS bzw. SKM beschäftigt, welcher als optischer Drehgeber realisiert ist, wird im Verlauf lediglich auf optische Drehgeber eingegangen.

Der grundsätzliche Aufbau optischer Drehgeber setzt sich aus einer Codescheibe (Modulator), einer Sende- Empfangseinheit sowie einer Leiterplatte zusammen. Die Leiterplatte nimmt die Signale der Sende- Empfangseinheit auf und verarbeitet diese, darüber hinaus stellt sie eine Kommunikationsschnittstelle zur Verfügung. Die einzelnen Komponenten sowie ihr Zusammenspiel sind in Abbildung "Komponenten eines MFB" dargestellt.

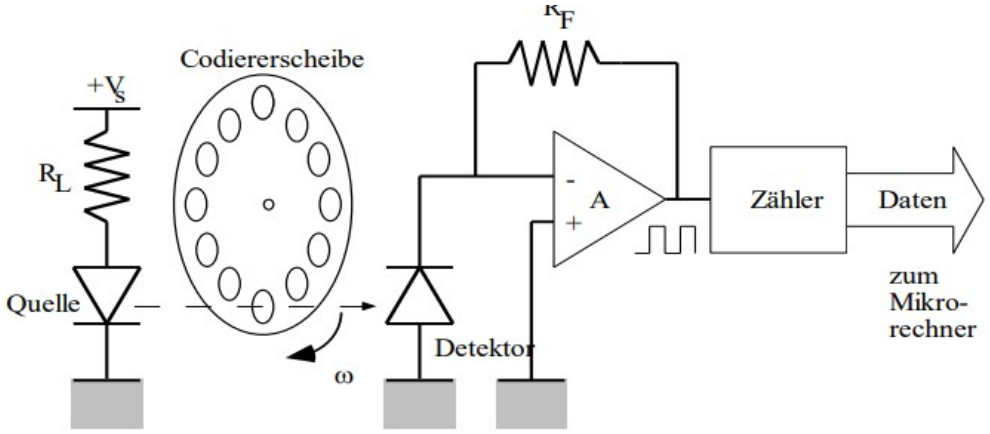

Abbildung 11: Komponenten eines MFB [21]

Der Sender bzw. Empfänger sind fest montiert, die Codescheibe ist beweglich. Die Codescheibe ist mit der Welle verbunden und dreht sich somit äquivalent zu dieser. Durch die Codescheibe wird der vom Sender ausgesendete Lichtstrom moduliert. Die Modulation erfolgt durch den Aufbau der Code-

scheibe mittels lichtdurchlässiger und undurchlässiger Felder. Der modulierte Lichtstrom (Hell-Dunkel-Muster) fällt auf den Empfänger. Durch diesen wird die optische Energie in elektrische gewandelt. Das hier erläuterte Verfahren wird als Schattenbildverfahren bezeichnet. Daneben kommen auch andere Verfahren, wie das Basis diffraktive (basierend auf optischer Beugung), oder Verfahren basierend auf dem Morie-Effekt zum Einsatz. Bei der Modulation des Lichtstrom ist die Bauart der Codescheibe bzw. deren Codierung maßgeblich. Man unterscheidet hierbei zwischen inkrementellen und absoluten Codierungen. Bei der inkrementellen Codierung wird nicht die absolute Winkelinformation, sondern lediglich eine Winkeländerung ausgegeben. Eine inkrementelle Codescheibe ist meist durch regelmäßige, trapezförmige Bereiche, welche abwechselnd lichtdurchlässig bzw. -undurchlässig sind realisiert. So lässt sich ein Sinus- bzw. Rechtecksignal erzeugen. Bei inkrementellen Drehgebern ist eine Referenzfahrt im Startvorgang des Drehgebers notwendig. Bei absoluten Drehgebern kann jederzeit die absolute Winkelposition ermittelt werden. Der Aufbau der Codescheibe gliedert sich hier in zwei Bestandteile, eine inkrementelle Codespur, sowie weitere Signalspuren, welche gemeinsam einen Code (zum Beispiel binär-, Gray-, oder Pseudo-Random-Code)ergeben. Durch diesen Aufbau entfällt die Referenzfahrt. Ein Beispiel für unterschiedliche Codespuren ist Abbildung "Codescheiben, links Graycode, rechts Binär".

Neben der Unterscheidung zwischen inkrementell und absolut können Drehgeber auch in Single- und Multiturn Drehgeber unterteilt werden. Bei einem Singelturn Geber wird immer nur eine Umdrehung mittels der Codescheibe aufgelöst. Bei Multiturn Gebern wird die Codescheibe um ein untersetztes Getriebe ergänzt. Das Getriebe wird wie die Codescheibe durch die Welle angetrieben. Die aktuelle Position des Getriebes wird über Hall-Sensoren ermittelt. Durch diesen Aufbau wird ein Messbereich von bis zu 4096 Umdrehungen erreicht. [17] In dieser Arbeit wird der SKx behandelt. Der SKS bzw. SKM ist ein durch die Firma SICK STEGAMNN entwickeltes MFB. Der SKx ist ein absolut arbeitender Drehgeber, und bereits seit mehreren Jahren eines der meistverkauften Produkte der SICK STEGMANN GmbH. Die Bezeichnung SKS steht für die Ausführung als Singleturn Geber,

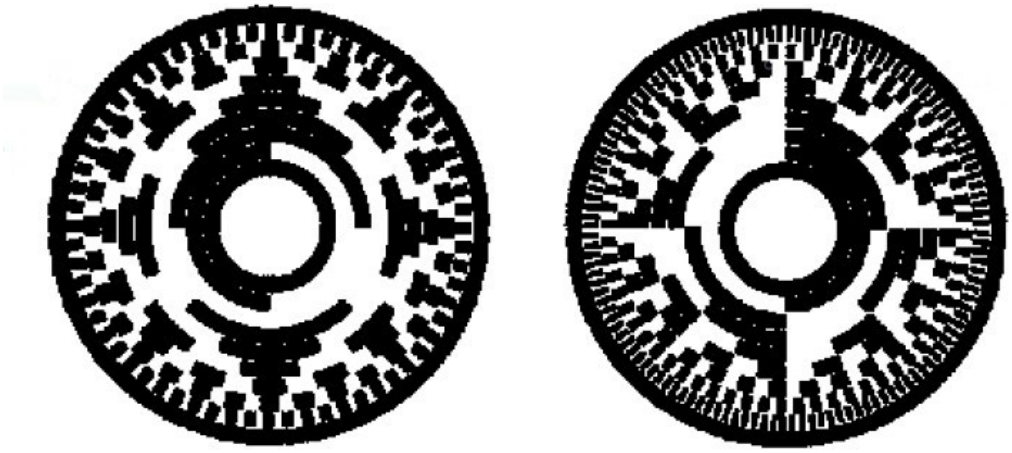

Abbildung 12: Codescheiben, links Graycode, rechts Binär [21]

SKM für Multiturn. Als Besonderheit des Gebers ist die Hiperface Schnittstelle zu erwähnen.

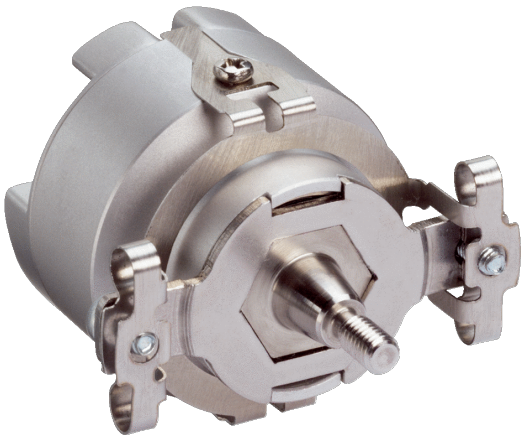

Abbildung 13: SKM36 [22]

4.2 Hiperface Protokoll

Das Hiperface Protokoll ist eine durch die Firma SICK entwickelte, offene Schnittstelle. Die Hiperface Schnittstelle gilt als Standard Schnittstelle im Bereich der Motorfeedbacksysteme Der Name Hiperface steht für High Performance Interface. Die Hiperface Schnittstelle setzt sich aus einer busfähigen RS485 Schnittstelle (Digital) Sowie einer analogen Schnittstelle zur Übertragung der Sinus und Cosinus Signale zusammen. In Abbildung "Aufbau Hiperface MFB" ist der Aufbau eines Hiperface Gerätes schematisch dargestellt. Die Codescheibe wird bei diesem Aufbau neben der absoluten Codespur um eine Inkrementelle ergänzt. Zur Realisierung der Schnittstelle sind lediglich acht Leitungen zwischen MFB und verarbeitendem System notwendig. Zwei Leitungen dienen zur Übertragung der Versorgungsspannung (7-12V), vier Leitungen werden zur Übertragung der Inkrementellen Sinus und Cosinus benötigt und die verbleibenden zwei Leitungen dienen zur Kommunikation mittels RS485 Schnittstelle. Die Kommunikation mittels der bidirektionalen RS485-Schnittstelle beginnt jeweils mit der Angabe der Zieladresse. Darauf folgt der entsprechende Befehl. Die Nachricht endet mit einer Prüfsumme. Jede Nachricht beginnt mit einem Startbit und endet

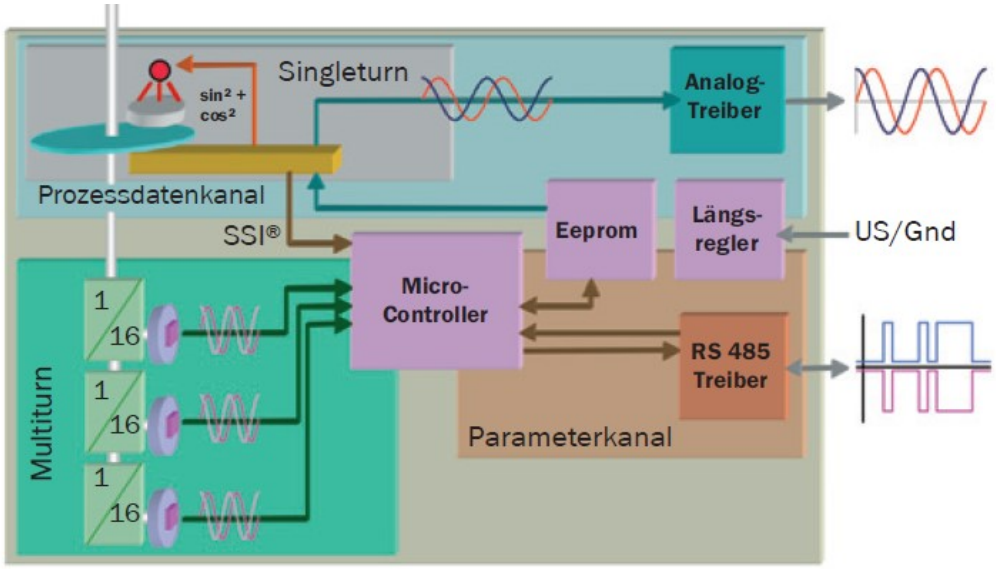

Abbildung 14: Aufbau Hiperface MFB [22]

mit einem Stopbit, dazwischen befinden sich acht Datenbit. Es wird zwischen zwei Arten von Nachrichtenformaten unterschieden (Adress und Command). Bei Adress wird lediglich die Teilnehmernummer übertragen, hierfür werden fünf Bit verwendet. Die default Teilnehmernummer lautet 40h. Alle weiteren Teilnehmernummern beginnen ab 40h, was dazu führt, dass lediglich fünf Bit Variabel gestaltet sind. Wie in Abbildung "Aufbau Adresse Hiperface" zu sehen bleibt durch die Beschneidung des Adressbereichs das Bit sieben immer eins und das Bit sechs immer null. Lediglich die Bits 1-5 sind variabel. Bei

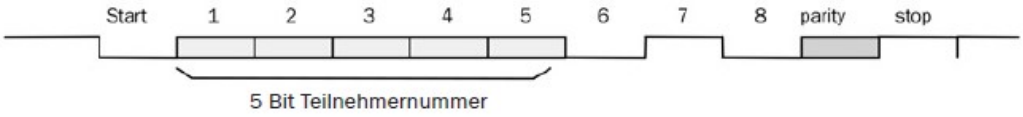

Abbildung 15: Aufbau Adresse Hiperface [23]

Command Befehlen ist das Vorgehen sehr ähnlich, hier sind jedoch sechs Bit variabel und das siebte Bit immer eins. Die einzelnen Befehle sind von 42h bis 67h codiert. Ein einfacher Befehl ist in "Lesen der aktuellen Position über RS485" zu sehen.

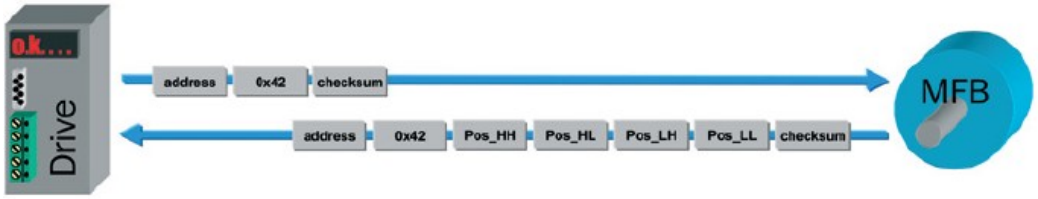

Abbildung 16: Lesen der aktuellen Position über RS485 [23]

Neben der aktuellen Position des Gebers lassen sich über die Hiperface Schnittstelle noch weitere Informationen wie zum Beispiel der Status des Gerätes, oder das elektronische Typenschild auslesen. Weiterhin ist es ebenfalls möglich verschiedene Register im MFB über die Schnittstelle zu beschreiben, oder einen Reset des Gerätes durchzuführen.

4.3 ST7/ STM8

Der ST7 ist ein durch die Firma STMicroelektronics vertriebener Mikrocontroller. Zur ST7 Familie gehören verschiedene Derivate, so ist zum Beispiel eine lite- oder eine automotiv Variante verfügbar. Alle Modelle der ST7 Familie gehören zu den 8 Bit Mikrocontrollern. Für den SKS bzw. SKM wird konkret der ST7234BK6T3 verwendet. Der Controller verfügt über einen integrierten A/D Wandler, Timer und eine SPI sowie SCI Schnittstelle. Er basiert auf einer Von-Neumann-Architektur.[24]. Der ST7 stellt das Herzstück des SKS bzw. SKM dar. Er dient hier zur Kommunikation via Hiperface und übernimmt die Berechnung der absoluten Position. Der STM8 ist die neueste acht Bit Controllerlinie von STMicroelektronics. Die STM8 Familie basiert auf auf einer modifizierten Harvard-Architektur[25]. Neben mehr Speicherplatz verfügt der Controller auch über deutlich mehr peripherie Möglichkeiten im Vergleich zu seinem Vorgänger.

5 Konzept

Zu Beginn der Konzeptionsphase werden die Anforderungen an das Projekt definiert. Hierzu werden Absprachen mit dem Auftraggeber getroffen, sowie die bereits vorhandenen Testfälle anderer Projekte analysiert. Aus den daraus erhaltenen Informationen wird im Anschluss eine Anforderungsdefinition erstellt. Nachdem die Anforderungen an das Projekt definiert sind, folgt die Entwicklung einer geeigneten Softwarearchitektur.

5.1 Analyse bisheriger Testfälle

Zur Ermittlung der Testfälle, welche durch die Suite abgedeckt werden sollen, werden die bisherigen Testpläne analysiert. Schwierigkeit hierbei ist, dass der Geber vor ca. 20 Jahren entwickelt wurde. Zum Entwicklungszeitpunkt fand keine, den heutigen Standards entsprechende, Dokumentation der Testfälle (Testplan) statt. Weiterhin ist zum Projektstart noch kein Testplan für die Portierung des Controllers vorhanden. Aus diesem Grund wird für die Analyse der Testfälle auf den Testplan eines ähnlichen Produktes (SEY) zurückgegriffen. Der SEY eignet sich hier, da er ebenfalls über eine Hiperface Schnittstelle verfügt.
Die im Testplan festgelegten Tests werden in eine Excel Tabelle übertragen. Die Testfälle sind im Testplan in verschiedene Kategorien unterteilt (zum Beispiel General Requirements, HW Architecture, Functional Requirements). Im Unterschied zum SK verfügt der SEY über zwei Mikrocontroller (Master und Slave, dies führt dazu das einige Testfälle im Testplan doppelt vorhanden sind. Da diese Dopplung für den SKx nicht notwendig ist werden zu Beginn der Analyse die entsprechenden Dopplungen entfernt. In einem zweiten Schritt wird die Beschreibung jedes Testfalls analysiert. Die Beschreibung der Testfälle erfolgt in Textform. Jeder Testfall ist durch eine Testfallnummer (zum Beispiel SW00077) eindeutig identifizierbar. Bei der Analyse werden die Kriterien aus "Kriterien für Testauswahl" angewendet. Hierdurch wird geprüft ob ein Testfall unter den derzeit gegebenen Rahmenbedingungen des Projekts Automated Testing @ GBC07 realisierbar ist. Ein Beispiel für einen nicht automatisierbaren Testfall ist zum Beispiel beim Eingriff mittels De-

bugger während des Tests gegeben. Als Ergebnis der Bewertung stehen drei Auswahlmöglichkeiten zur Verfügung (gegeben, nicht gegeben, mit Änderungen). Das Ergebnis "mit Änderungen" dient dazu, Testfälle zu identifizieren welche im Moment zum Beispiel aufgrund fehlender Hardware am Teststand nicht realisierbar sind, zu identifizieren. Bei diesen Testfällen wäre eine spätere Realisierung in einer weiteren Ausbaustufe des Teststandes möglich.

	Nummer des Tests aus Testplan:		SW000016
Anford. Nr.	Anford. Bezeichnung	Beschreibung	Bewertung
01	Externe Software	Externe Software wie zum Beispiel der Einsatz eines Debuggers darf nicht gefordert werden	gegeben
02	Timing Test	Zeitmessungen sind aufgrund der nicht Echtzeitfähigen BS schwer zu realisieren	gegeben
03	Manueller Eingriff	es darf kein manueller Eingriff in den Test (zum Beispiel umstecken, auslesen eines Registers, Prüfungen direkt im Quellcode) notwendig sein	nicht gegeben
04	Externe Hardware	Für den Test benötigte Hardware muss am Teststand vorhanden sein	mit Änderungen
05	Produkt spezifisch	Test ist auf spezielle Produkt Eigenschaft ausgelegt	gegeben
Ergebnis			nicht bestanden

Tabelle 1: Kriterien für Testauswahl

Aus der Analyse der Testfälle mittels "Kriterien für Testauswahl" ergibt sich eine Gruppe von Testfällen, welche sich grundsätzlich zur Automatisierung unter den derzeitigen Gegebenheiten eignet. Die Gruppe beinhaltet hierbei eine Vielzahl verschiedener Tests (Hiperface Schnittstelle, Positionsbestimung, Power Supply Tests). Um eine bessere Übersicht zu erhalten, werden die Testfälle in Cluster gegliedert. Die Einteilung erfolgt in die sechs Kategorien Error Codes, Hiperface Commands, Position and external Sensors, Temperatur and Power Supply, Settings und Other. Um den weiteren Verlauf der Analyse- und Konzeptionsphase zu vereinfachen, werden im Anschluss die benötigten Hard- und Softwareschnittstellen für die verschiedenen Cluster identifiziert. Die Tabelle "Testcluster" zeigt beispielhaft die Cluster der Tests, verschiedene Cluster sind hierbei farblich abgegrenzt. Die vollständige Tabelle befindet sich im Anhang.

Test	Kategorie	Inhalt	Bemerkung
SW00001	General Requirements	Firmware version format	
SW00030	HW Architecture	Master IcDSL IcDSL resolution is set by EEPORIM parameter Part 1	
SW00031	HW Architecture	Master IcDSL IcDSL resolution is set by EEPORIM parameter Part 2	
SW00058	Functional Requirements	Boot-loader	
SW00132	Functional Requirements	General user cmds WEnterBootloader (70h) NO support of safety variants	Nur für Safety Variante
SW00017	HW Architecture	Master mic. Supply Voltage Monitoring Part 1 (Accuracy)	
SW00018	HW Architecture	Master mic. Supply Voltage Monitoring Part 2 (Lower limit)	
SW00021	HW Architecture	Master mic. Internal Voltage Rail 1.8V	
SW00078	Functional Requirements	Error handlig referred to Hiperface specification	

Tabelle 2: Testcluster

5.2 Anforderungsdefinition

Eine klare Definition der Anforderungen an das Softwareprojekt dient im Verlauf dazu. die Endabnahme des Projekts zu erleichtern. Weiterhin werden durch die Anforderungen SMARTe [26] Ziele definiert, welche die weitere Projektstruktur positiv beeinflussen soll. Die während der Analyse erhaltenen Ergebnisse dienen hierbei als Grundlage für die Anforderungsdefinition. Weitere Anforderungen werden darüber hinaus durch den Auftraggeber (Betreuer der Ausbildungsfirma) vorgegeben. Auf die Erstellung einer vollständigen SRS wird im Rahmen dieses Projekt aus Zeitgründen bewusst verzichtet. Bei den Anforderungen wird zwischen funktionalen und nicht funktionalen unterschieden. Funktionale Anforderungen sind all jene, welche eine konkrete Funktion der Software definieren. Nicht funktionale Anforderungen sind Anforderungen wie zum Beispiel Zuverlässigkeit und Zeitverhalten. Die Anforderungen sind in den Tabellen "Anforderungen funktional" und "Anforderungen nicht-funktional" dargestellt. Die Reihenfolge und Nummerierung der Anforderungen spiegeln gleichzeitig ihre Relevanz für das Projekt wieder.

Nr.	Bezeichnung	Beschreibung	Zuordnung
F01	Integration in Automated Testing @ GBC07	Die Suite muss in das Projekt Automated Testing @ GBC07 Integrierbar sein (Implementierung in ITE, Aufbau nach Vorgaben des Projekts)	Funktional
F02	Schnittstellen Config	Auslesen und prüfen der Schnittstellen Parametrisierung und Funktionalität	Funktional
F03	Diagnose Funktionalität	Überprüfung der vorhandenen Diagnose Funktionalität (zum Beispiel Spannungs Monitoring)	Funktional
F04	Core Funktionalität	Prüfung der externen Temperatur Schnittstelle, sowie der single- und multiturn Positionsbestimmung	Funktional
F05	Identifikation des DUT	Überprüfung der Firmware Version, des elektronischen Typenschilds auf Korrektheit	Funktional
F06	Hiperface Befehle	Überprüfung der Funktion der verschiedenen Hiperface Befehle sowie der entsprechenden Antwort	Funktional
F07	Error Codes	Eignung zur Stimulation der verschiedenen Fehlerfälle und dadurch Produktion der entsprechenden Fehlermeldungen (01h-23h)	Funktional

Tabelle 3: Anforderungen funktional

Nr.	Bezeichnung	Beschreibung	Zuordnung
NF01	Externes Equipment	Möglichkeit zur Integration des am Teststand vorhandenen externe Equipment	Nicht Funktional
NF02	Universell einsetzbar	Der Aufbau der Suite soll so gestaltet sein, dass diese auch zukünftig für die Portierung anderer (ähnlicher) Produkte genutzt werden kann	Nicht Funktional
NF03	Coding Guidelines	Der Aufbau der Suite soll so gestaltet sein, dass diese auch zukünftig für die Portierung anderer (ähnlicher) Produkte genutzt werden kann	Nicht Funktional

Tabelle 4: Anforderungen nicht-funktional

5.3 Softwarearchitektur

Um die Anforderungen aus der vorherigen Anforderungsanalyse erfüllen zu können, bedarf es der Planung einer entsprechenden Softwarearchitektur. Hierzu werden die Ergebnisse aus der Anforderungsanalyse und der Analyse der Testfälle zusammengetragen. Das Resultat ist eine Beschreibung der benötigten Hard- und Softwareschnittstellen welche die Suite abdecken muss. Diese gliedern sich in:

- ICP Con (RS232 auf RS485 Converter)
- Motor
- Schaltnetzteil
- Logging Funktion

Nachdem ein Überblick über die entsprechenden Softwarekomponenten geschaffen ist, werden die bisher verwendeten Skripte analysiert. Dies soll dazu dienen Softwarekomponenten zu identifizieren die für das aktuelle Projekt wiederverwendet werden können, sowie sich einen Überblick über die bisherige Softwarearchitektur zu erarbeiten. Ein Beispiel der alten Architektur ist in Abbildung "SW-Architektur HW Komponenten Alt" dargestellt.

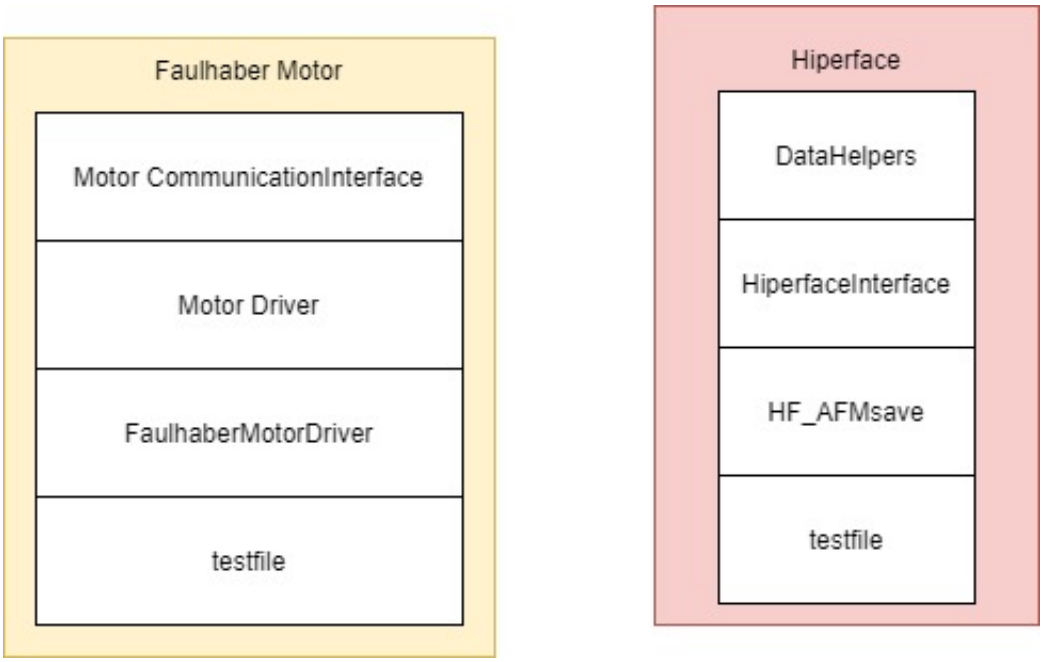

Abbildung 17: SW-Architektur HW Komponenten Alt

Zu erkennen ist hierbei der schichtweise Aufbau der entsprechenden Module. Jedoch ergibt sich keine klare Softwarearchitektur, welche einheitlich auf die verschiedenen Komponenten angewendet werden kann. Die oberste Ebene jedes Testfalls wird durch das Testfile, welches die konkrete Implementierung des Testfalls enthält, gebildet. In den darunterliegenden Ebenen wird jeweils die Kommunikation mit den entsprechenden Hardwarekomponenten realisiert. Hierbei unterscheidet sich der architektonische Aufbau für die einzelnen Komponenten stark. Dies führt dazu, dass es für den Entwickler eines Tests sehr aufwendig ist zu prüfen welche konkreten Files bzw. Funktionen er für sein Testequipment nutzen muss. Weiterhin führt dieser Umstand dazu, dass die Testfallimplementierungen nur schwer für andere Tests mit anderen Hardware Anforderungen verwendet werden können.
Die Architektur der Suite soll es dem Benutzer ermöglichen die benötigten Hard- bzw. Softwarekomponenten einfach einzubinden. Weiterhin soll die Suite die Wiederverwendung der Testfallimplentierungen fördern. Konkret ergeben sich also die folgenden Anforderungen an die Softwarearchitektur:

- Erweiterbarkeit
- Einfache Wartbarkeit
- Benutzerfreundlichkeit
- Wiederverwendbarkeit

Um ein geeignetes Architekturmodell zu identifizieren werden drei Modelle verglichen. Als Modelle werden das MVC Pattern, das Pipe-Filter Modell und die Schichtenarchitektur gewählt. In Tabelle "Architekturmodelle im Vergleich" werden die Vor- und Nachteile der einzelnen Modelle bewertet, hierbei werden die Architekturen nach den Kriterien des aktuellen Projekt bewertet. Die Vor- und Nachteile können für andere Projekte variieren.

Kriterium	Pipe Filter	MVC	Schichten
Erweiterbarkeit	++	+	+++
Wartung	+++	++	+++
Benutzerfreundlichkeit	+	+++	+++
Wiederverwendbarkeit	+++	++	+++

Tabelle 5: Architekturmodelle im Vergleich

Durch den Vergleich ergibt sich eine Schichtenarchitektur als geeignetes Modell. Die Schichtenarchitektur lässt in diesem Fall eine gute Abstraktion für den Benutzer zu, ohne das System zu komplex für eine Erweiterung bzw. Wartung zu machen. Hierdurch lässt sich auch das Prinzip von Kopplung und Kohäsion besser umsetzen. Als Herausforderung bei der Schichtenarchitektur zeigt sich die Auswahl der konkreten Anzahl Schichten. Hier wird eine drei-Schicht mit einer vier-Schicht Architektur verglichen. Es zeigt sich, dass

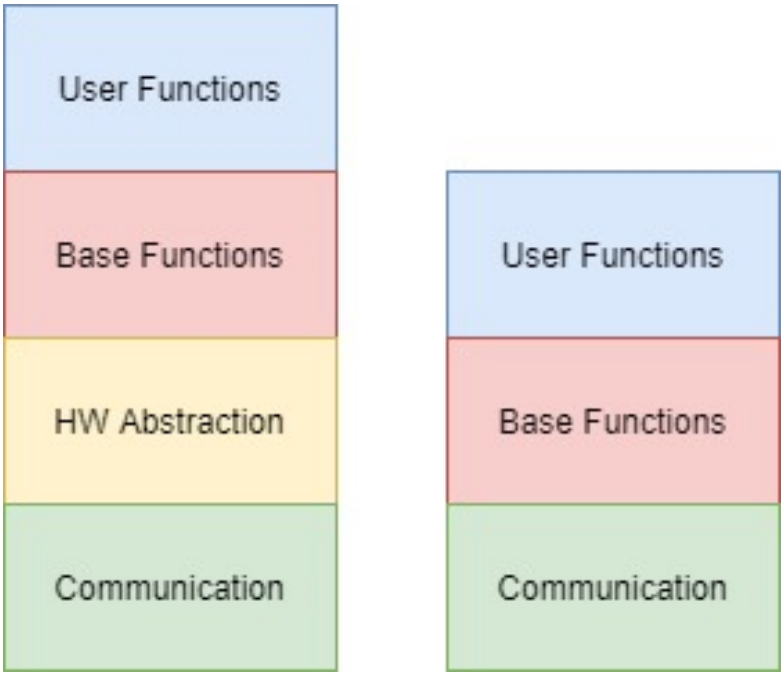

Abbildung 18: Vergleich drei und vier Schichten

eine drei-Schicht Architektur prinzipiell möglich ist, jedoch gerade bei den Motor Modulen eine vier-Schicht Architektur für eine bessere Strukturierung

und Trennung sorgt. Grund hierfür ist die Möglichkeit verschiedene Motoren mit jeweils unterschiedlichen Zusatzfunktionen nutzen zu können. Durch den zusätzlichen Hardware Abstraction Layer bei der vier Schicht Architektur lassen sich die unterschiedlichen Motoren besser abbilden. Als unterste Schicht der Architektur wird die Communication Schicht gewählt. Diese übernimmt die Bereitstellung der entsprechenden Communication Interfaces. Als zweite Schicht wird eine Hardware Abstraktionsschicht eingeführt. Welche u.a die Treiber für die Hiperface Schnittstelle, sowie für die verschiedenen Motoren beinhaltet. Die dritte Schicht (BaseFunctions) enthält die grundlegenden Befehle, welche zum Beispiel zum Starten eines Motors notwendig sind. Weiterhin enthält diese Schicht die Implementierung der grundlegenden Hiperfacebefehle. Die Userfunctions bilden die oberste Schicht der Architektur. Diese Schicht beinhaltet die Funktionalitäten, welche der Testentwickler für die konkrete Testfallimplementierung später benötigt. Beispiele hierfür sind z.B das auslesen des Typenschildes via Hiperface oder das Rotieren des Motors mit einen definierten Geschwindigkeit für eine gewisse Zeit. Der Nutzer der Suite kann also in Zukunft im Testverlauf einen Motor starten indem er die entsprechende Funktion der Suite aufruft, ohne dass er genaue Kenntnis über die konkrete Implementierung der Motortreiber oder geeigneter Motorfunktionen haben muss. Neben den vier Schichten werden horizontal über alle vier Schichten hinweg zwei weitere Komponenten vorgesehen. Zum Einen ist hier eine Klasse DataHelers welche, grundlegende Umrechnungsfunktionen sowie bestimmte globale Datenstrukturen enthält, vorgesehen. Zum anderen eine Klasse, welche die allgemeinen Testdaten speichert. Durch die horizontale Anordnung über alle Schichten hinweg ist ein Zugriff aus allen Schichten und Modulen auf diese Hilfsklassen möglich. Als Besonderheit beim Architekturentwurf wird die Logging Funktion gesehen. Dies ist eine reine Softwarekomponenten und benötigt aus diesem Grund weder eine Communication noch eine Hardwareschicht. Durch die gewählte Schichtenarchitektur lässt sich aber auch dieser Fall gut realisieren. Das Ergebnis dieses Grobentwurfs ist in Abbildung "Softwarearchitektur Neu" dargestellt:

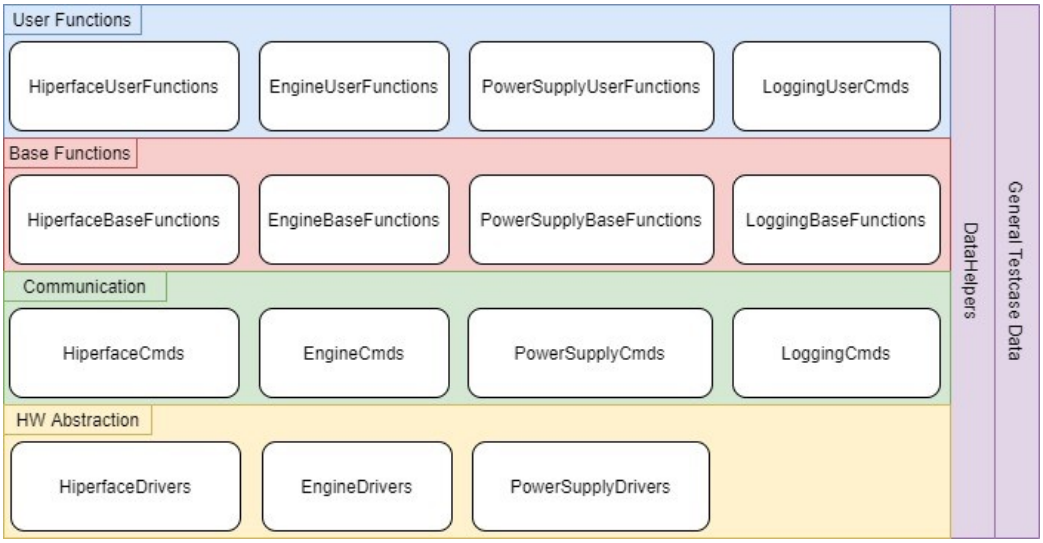

Abbildung 19: Softwarearchitektur Neu

Die einzelnen Module der Software werden in den weißen Kästen dargestellt. Durch die hier entwickelte Architektur lassen sich die Module beliebig austauschen. Weiterhin ist die Architektur gut auf eventuelle spätere Erweiterungen übertragbar. Nach der Fertigstellung des Grobentwurfs erfolgt der Feinentwurf der Architektur. Hierzu werden für die einzelnen Funktionalitäten jeweils Klassendiagramme angefertigt. Durch die angewendete Schichtenarchitektur lassen sich die Klassendiagramme einfach erstellen. Die Anzahl und Reihenfolge der Abhängigkeiten ist beschränkt. In Abbildung "Klassendiagramm Hiperface Neu" ist das Klassendiagramm der Hiperface Funktionalität vereinfacht dargestellt. Auf die Abbildung der Klassenvariablen wurde aus Übersichtsgründen hierbei verzichtet. Die Aufteilung der Klassen entspricht den Modulen der Softwarearchitektur im Schichtenmodell. Die oberste Ebene "HiperfaceUserFunctions" stellt die Benutzerebene dar. In dieser werden nur Funktionalitäten implementiert, welche für den Benutzer direkt relevant sind. In der darunterliegenden "HiperfaceBaseFunctions" Klasse befinden sich die verschiedenen Kommandos, welche durch die Hiperface Schnittstelle spezifiziert sind. Die "HiperfaceDrivers" Klasse enthält alle Funktionen, welche zum senden und empfangen der verschiedenen Kommandos benötigt werden. Dies ist zum Beispiel die Berechnung der Checksumme. Als unterste

Ebene kommt das "HiperfaceComIterface" zum Tragen. Dies stellt die Funktionalitäten für das initialisieren, öffnen und schließen der Schnittstelle zur Verfügung.

Als Grundlage für die Konzeption der neuen Architektur dient eine bisherige Implementierung der Hiperface Schnittstelle. Diese ist in Abbildung "Klassendiagramm Hiperface Alt" zu sehen. Beim Vergleich der beiden Architekturen fällt auf, dass durch die neue Architektur eine deutlich bessere Abstraktion für den Benutzer erzielt wird. Weiterhin ist es mithilfe der neuen Architektur deutlich einfacher das Programm um weitere Module zu ergänzen, da einheitliche Schnittstellen geschaffen werden.

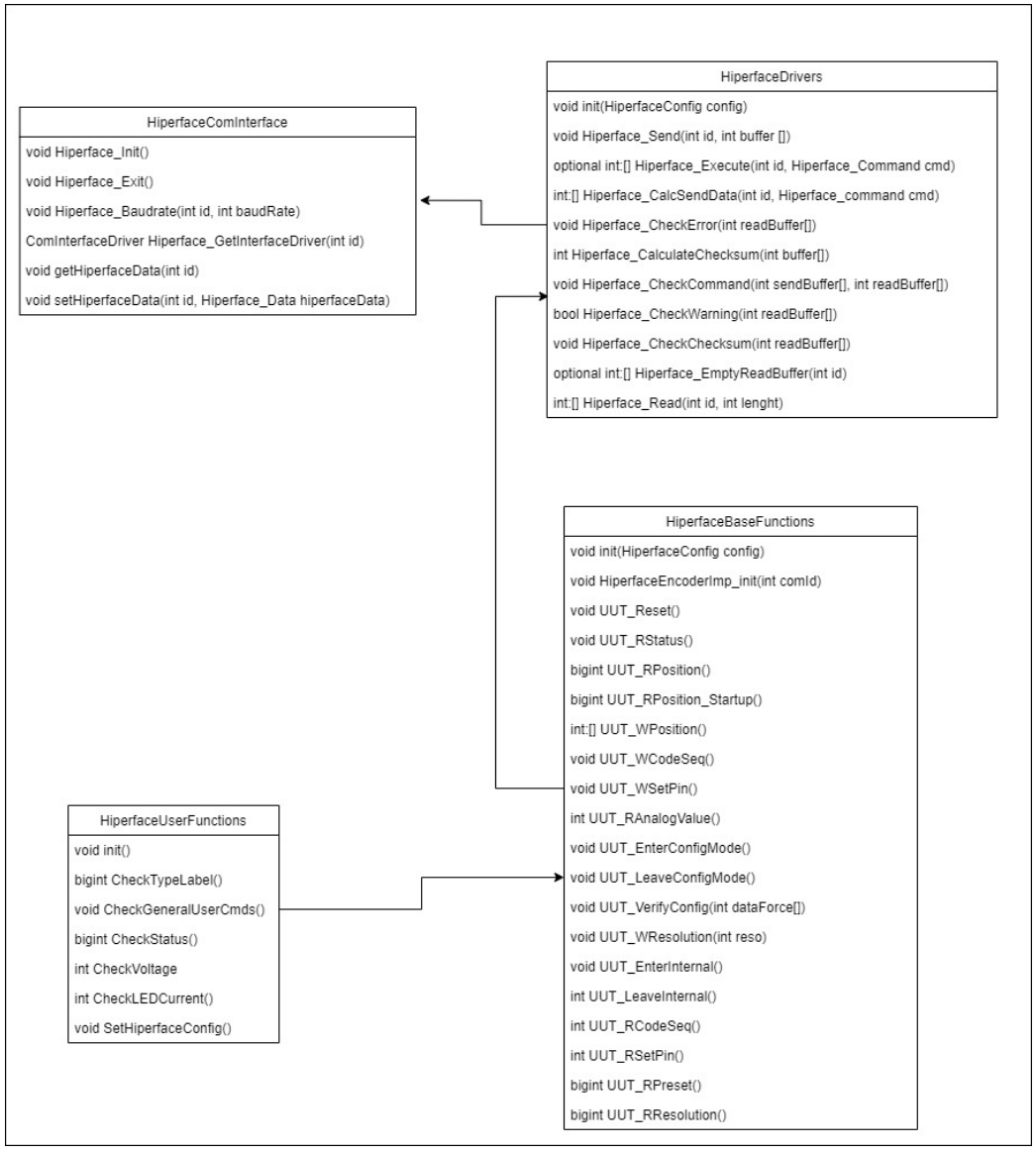

Abbildung 20: Klassendiagramm Hiperface Neu

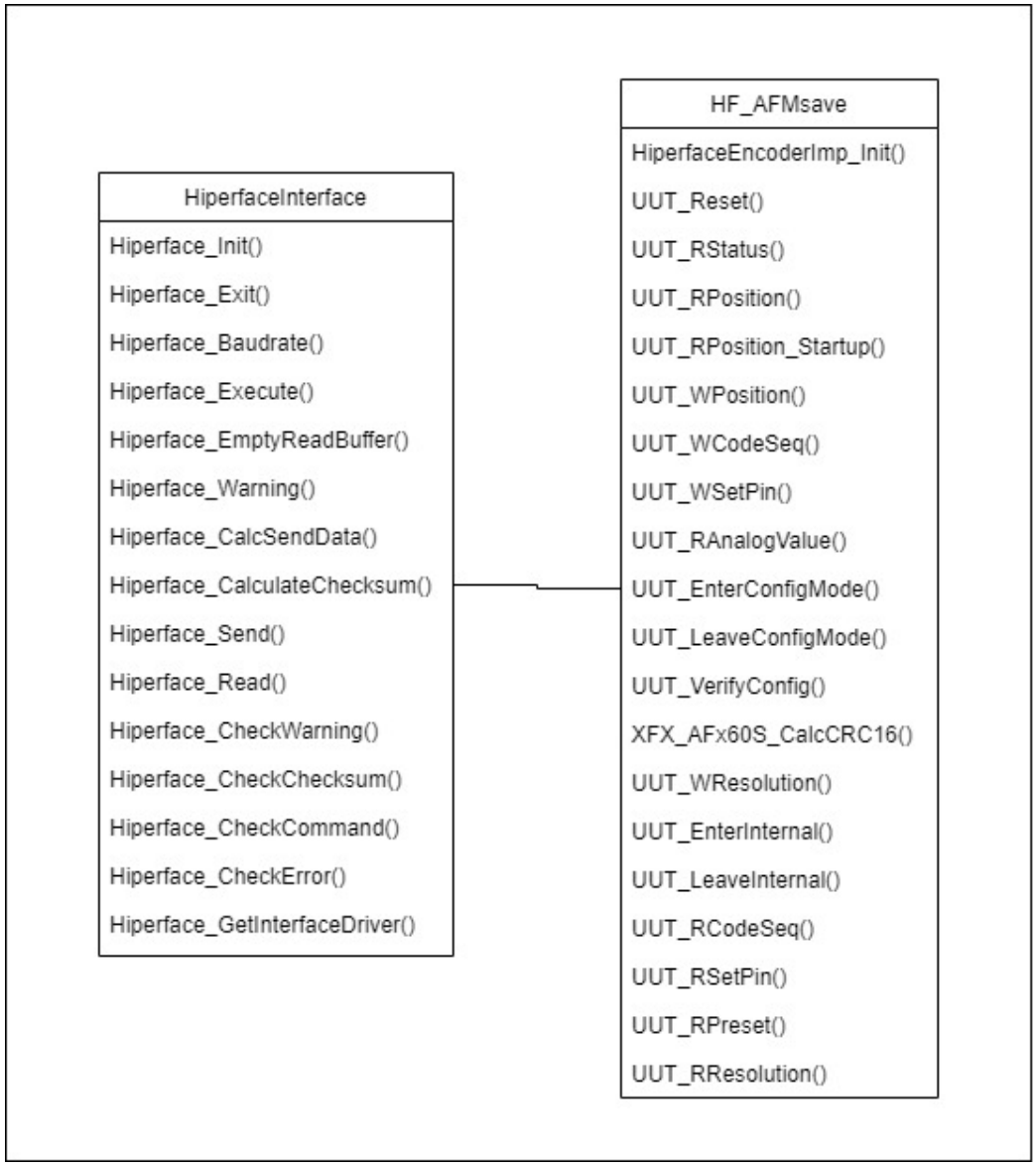

Abbildung 21: Klassendiagramm Hiperface Alt

Zum Zeitpunkt dieser Arbeit sind am Teststand zwei verschiedene Arten von Motoren vorhanden. Hierbei handelt es sich um einen Motor der Marke Faulhaber, sowie einen der Marke Metronix. Zwar sind die Basisbefehle dieser Motoren identisch, jedoch verfügen beide über dasselbe Kommunikationsinterface und dieselben Grundfunktionalitäten. Jedoch besitzen sie verschiedene, spezielle Befehle wie etwa das Setzen eines Operationsmode oder das Spezifizieren der Beschleunigung. Aufgrund dieser Unterschiede gestaltet sich die Erstellung der Driverschicht hier aufwendiger. Das "EngineDriver" Modul gliedert sich in die drei Bestandteile "EngineDriver", "MetronixEngineDriver" und "FaullhaberEngineDriver". Die Klasse "EngineDriver" stellt die Grundfunktionalitäten bereit, von welcher die beiden spezifischen Motoren dann abgeleitet werden.

5.4 Wirtschaftlichkeit

Im Rahmen des Projektmanagements wird weiterhin eine wirtschaftliche Betrachtung des Projekts versiert. Diese gestaltet sich jedoch aufgrund der mangelnden Datenlage schwierig. Die Kosten des Projektes werden durch die benötigte Arbeitszeit definiert. Weitere Kosten zum Beispiel für Hardwarekomponenten sind vernachlässigbar, da es sich bei der Testsuite um eine reine Softwarelösung handelt. Zwar wird das Projekt im Rahmen einer Bachelorarbeit bearbeitet, jedoch sind keine Angaben zu den Kosten eines Bacheloranten pro Stunde vorhanden. Aus diesem Grund wird für die Berechnung auf den in der Softwareentwicklung für solche Projekte vorgegeben Stundensatz von 90€ angenommen und halbiert. Durch die Vorgabe der DHWB, welche einen Stundenaufwand von mindestens 360 Stunden für eine Bachelorarbeit vorsieht, ergeben sich daraus Kosten in Höhe von:

$$45€ * 360h = 16200€$$

Um beurteilen zu können ob das Projekt wirtschaftlich effizient ist, müssen diese Kosten nun mit den durch die Testsuite eingesparten Kosten zum Beispiel durch beschleunigte Testentwicklung etc. verglichen werden. Eine konkrete Abschätzung des durch die Testsuite reduzierten Aufwand, ist jedoch

zu diesem Zeitpunkt nicht möglich. Es wird erwartet, dass die Zeitersparnis gerade in Bezug auf die in Zukunft kommenden Portierungen, das Projekt als wirtschaftlich darstellen wird.

6 Implementierung

In diesem Kapitel werden die verschiedenen Implementierungsschritte erläutert. Beispielhaft wird auf einzelne Schlüsselstellen eingegangen und die entsprechenden Lösungsstrategien hierfür beschrieben. Weiterhin wird kurz erläutert welche Programmierumgebung bzw. Sprache verwendet wurde. Da es sich bei der Implementierung um ein Proof of Concept handelt, werden nicht alle Softwarekomponenten aus dem erstellten Konzept implementiert. Bei der Implementierung dienen die Regeln des Clean Code Development als Grundlage [27]. Auch wird bei der Implementierung auf die Einhaltung der Programming Principles wie DRY, KISS, SOLID etc. geachtet. Als Code Management System wird ein GitLab Repository verwendet. Dies bietet Schutz gegen Dateiverlust und ermöglicht eine Versionierung der Software. Für das verfassen von Kommentaren werden die Doxygen Richtlinien angewendet. Dies ermöglicht später die automatische Generierung einer entsprechenden Code Dokumentation.

6.1 Entwicklungsumgebung/Programmiersprache

Die Auswahl der Programmiersprache gestaltete sich aufgrund der Anforderung zur Integrationsfähigkeit in das Automated Testing @ GBC07 Projekt relativ einfach. Durch das Projekt Automated Testing @ GBC07 wird als Programmiersprache ITE vorgegeben. Da ITE eine objektorientierte Sprache ist, ähneln die meisten Sprachkonstrukte denen von C++. Jedoch sind bestimmte Funktionalitäten wie zum Beispiel Interfaces in ITE noch nicht implementiert. Als Entwicklungsumgebung kommt Visual Studio Code zum Einsatz. Hier ist die Integration des ITE Compilers sehr einfach. Weiterhin ist Visual Studio Code einfach zu bedienen und kostenlos verfügbar.

6.2 Hiperface Implementierung

Die Hiperface Komponente ist im späteren Einsatz für die Kommunikation mit dem MFB zuständig. Durch diese Aufgabe stellt sie die Kernkomponente der Testsuite dar. Da außerhalb der Hiperface Schnittstelle keine Möglichkeit zur Kommunikation, oder Datenaustausch mit dem zu testenden MFB besteht. Durch die Kozeptionierung ist eine entsprechende Klassenstruktur für die Implementierung vorgegeben (siehe "Klassendiagramm Hiperface Neu"). Bei der Umsetzung der konkreten Funktionen kann teilweise auf bestehenden Code zurückgegriffen werden. Dies bietet den Vorteil, dass zum Beispiel die einzelnen Hiperface Befehle nicht von Grund auf neu implementiert, sondern lediglich angepasst und in die neue Softwarearchitektur integriert werden müssen. Diese Befehle werden in der "HiperfaceBaseFunctions" Klasse zusammengefasst. Die "HiperfaceUserFunctions" Klasse enthält zusammengesetzte Funktionen, welche auf die Funktionen der "HiperfaceBasFunctions" Klasse zugreifen. Dies geschieht mittels einer klassischen objektorientierten Programmierung und der Erzeugung eines Objekts vom Typ "HiperfaceBaseFunctions". Ein Beispiel für dieses Vorgehen ist die Abfrage des LED Current. In Listing "Auslesen eines Analogwertes" ist zu erkennen, wie der Aufruf der Funktion "RAnalogValue" mit dem entsprechenden Übergabeparameter stattfindet. Auch ist hier zu erkennen, dass bei jedem Befehl geprüft wird, ob bereits eine Initialisierung der Schnittstelle stattgefunden hat.

```
1    public void checkLedCurrent(){
2        if(this.initDone == false){
3             this.init();
4             int ledCurrent = this.hiperfaceBaseFunctions.
                  UUT_RAnalogValue(0x50);
5         }
6         else{
7             int ledCurrent = this.hiperfaceBaseFunctions.
                  UUT_RAnalogValue(0x50);
8         }
9
10   }
```

Listing 1: Auslesen eines Analogwertes

Die Implementierung der entsprechenden Hiperface Befehle erfolgt in der "HiperfaceBaseFunctions" Klasse. Es wird zwischen lesenden und schreibenden Befehlen unterschieden. Der Aufbau der verschiedenen Befehle ist entsprechend der jeweiligen Kategorie sehr ähnlich. Ein Beispiel für einen lesenden Befehl ist die Abfrage des MFB Status.

```
1  public int UUT_RStatus()
2    {
3        int receive[];
4        Hiperface_Command cmd = { 0x50 => command,
5                                  true => responseRequired
                                      };
6        receive =  this.hiperfaceDriver.Hiperface_Execute(
             this.hiperfaceId, cmd);
7
8        return receive[0];
9
10   }
```

Listing 2: Abfragedes aktuellen Geräte Status

Das Listing "Abfragedes aktuellen Geräte Status" zeigt den entsprechenden Quellcode. Das Hiperface Kommando setzt sich aus dem Befehl (Angabe als Hex-Code) sowie der Angabe das eine Antwort erwartet wird (bool) zusammen. Im Anschluss wird die Funktion "Hiperface_Execute" aufgerufen und ihr das Kommando als Parameter (struct) übergeben. Die "Hiperface_Execute" Funktion übernimmt dann das Berechnen der Checksumme, sowie das Senden des Befehls an das MFB. Hierbei wird das von der Klasse "HiperfaceComInterface" zur Verfügung gestelltes Interface genutzt.

Durch die objektorientierte Programmierung besteht die Möglichkeit mehrere Instanzen der Hiperface Schnittstelle zu erzeugen. Da dies für den späteren Gebrauch jedoch nicht zielführend ist und es hierdurch zu Schwierigkeiten bei der Verwendung der COM Ports kommen kann, wird das Erzeugen mehrerer Instanzen der Klasse "HiperfaceComInterface" mittels eines Singelton Entwurfsmusters verhindert. Die Implementierung ist in Listing "Beispiel Singelton Entwurfsmuster" zu sehen.

```
1  private New(){
2
3      }
4
5      static HiperfaceComInterface hiperfaceComInterfaceObject
           ;
6
7      public static getInstance(){
8          if(hiperfaceComInterfaceObject == null){
9              hiperfaceComInterfaceObject =
                   HiperfaceComInterface.New();
10         }
11         return hiperfaceComInterfaceObject;
12     }
```

Listing 3: Beispiel Singelton Entwurfsmuster

Durch das Singelton Pattern wird sichergestellt, dass eine Klasse nur genau ein Exemplar besitzt. Darüber hinaus stellt es einen globalen Zugriffspunkt auf dieses dar.[28] Die Implementierung des Singelton erfolgt hier durch das deklarieren des Konstruktors als "Privat". Somit ist kein Zugriff von außerhalb der Klasse auf ihn möglich. Der Zugriff auf den Konstruktor erfolgt im Anschluss über die Funktion "getInstance". In dieser Funktion wird geprüft, ob es bereits ein Objekt der Klasse gibt. Falls nicht, wird ein neues erstellt und dieses zurückgegeben. Ein Zugriff auf die Klasseninstanz ist in Listing "Beispiel Zugriff Singelton " zu sehen.

```
1  HiperfaceComInterface hiperfaceCominterface =
       HiperfaceComInterface.getInstance();
```

Listing 4: Beispiel Zugriff Singelton

Die Verwendung des Singelton Entwurfsmusters ist teilweise umstritten, da es zum Beispiel die Anwendung von Unit Tests erschwert, jedoch ist es komfortabel und unkompliziert zu implementieren. Weiterhin bietet es die Möglichkeit

sehr gut zu Steuern und zu Kontrollieren wann auf die Instanz zugegriffen wird.

6.3 Motor Implementierung

Bei vielen Testfällen wird neben den Hiperface Befehlen auch ein Motor benötigt um die Welle des MFB zu drehen und so entsprechende Messdaten zu erhalten. Die Motoren verfügen über entsprechende Schnittstellen, welche es dem Benutzer ermöglichen, diese mittels einer Seriellen COM Schnittstelle zu steuern. Bei der Implementierung der Motorfunktionalität wird besonderen Wert auf die Erweiterbarkeit gelegt. Im Moment sind zwei verschiedene Motoren am Teststand vorhanden. Jedoch ist eine Erweiterung dieses Bereichs in Zukunft möglich. Dieser Umstand wurde bereits bei der Kozeptionierung eingeplant und wird auch bei der Implementierung beachtet. Um dem Benutzer einheitliche Funktionen zur Verfügung zu stellen, sodass dieser im Betrieb später nicht die Motorspezifischen Befehle verwenden muss, wird hier auf überladenen Funktionen (virtuelle Funktionen) und somit polymorphe Funktionsaufrufe gesetzt. Durch dieses Vorgehen wird das Prinzip der Protected Variations erfüllt. In der "EngineDriver" Klasse werden die virtuellen Funktionen definiert. In Listing "Virtuelle Methoden des Motors " ist ein Teil der virtuellen Funktionen, welche jeder verwendete Motoren zur Verfügung stellen muss, zu sehen.

```
1    virtual void InitImplementation();
2    virtual void CloseImplementation();
3    virtual void SendCommandImplementation(string command);
4    virtual void WriteBaudrateImplementation(int baudrate);
5    virtual void EnableDriveImplementation();
```

Listing 5: Virtuelle Methoden des Motors

In den Klassen "FaulhaberEngineDriver" und "MetronixEngineDriver" findet die konkrete Implementierung der entsprechenden Funktionen statt. Die

beiden Klassen leiten jeweils von "EngineDriver" ab. Ein Beispiel hierfür ist in Listing "Faulhaber Implementierung der virtuellen Funktionen " zu sehen.

```
1  virtual void InitImplementation() {
2      this.communicationInterface.Init();
3  }
4
5  virtual void CloseImplementation() {
6      this.communicationInterface.Disconnect();
7  }
8
9  virtual void SendCommandImplementation(string command) {
10     this.communicationInterface.Write(command);
11     Std.Print("Command response from Faulhaber: " + this.
           communicationInterface.Read() + "\n");
12 }
```

Listing 6: Faulhaber Implementierung der virtuellen Funktionen

Um die Erzeugung der unterschiedlichen Implementierungen zu realisieren, wird hier das Factory Entwurfsmuster angewendet. Das Factory Entwurfsmuster definiert eine Klassenschnittstelle mit Methoden zum Erzeugen eines Objektes. Jedoch wird durch Unterklassen entschieden, von welcher Klasse das erzeugte Objekt sein soll.[28] Die Umsetzung des Factory Entwurfsmusters erfolgt in der Funktion "setupEngine".

```
1     public EngineDriver setupEngine(string engineType)
2     {
3         this.configuration = this.loadConfig();
4
5         if(engineType == "Faulhaber")
6         {
7             return this.configureFaulhaberEngine();
8         }
9         elseif(engineType == "Metronix"){
10            return this.configureMetronixEngine();
11        }
12
13        return this.configureFaulhaberEngine();
14
15    }
```

Listing 7: Anwendung Factory Entwurfsmuster

In der Funktion "configureFaulhaberEngine" bzw."configureMetronixEngine" wird im Anschluss der konkrete Aufruf bzw. die Erzeugung des entsprechenden Objektes durchgeführt. Dies ist in Listing "Erzeugung eines Faulhaber Motor Objekts" zu sehen.

```
1  EngineDriver configureFaulhaberEngine(){
2
3          int comPort = this.configuration.Equipment.
               Faulhaber_1.Com.value.AsString().AsInt();
4          int baudRate = this.configuration.Equipment.
               Faulhaber_1.Baudrate.value.AsString().AsInt();
5          int dataBits = this.configuration.Equipment.
               Faulhaber_1.DataBits.value.AsString().AsInt();
6          string parityString = this.configuration.Equipment.
               Faulhaber_1.Parity.value.AsString();
7          int parity;
8
9          int inBuffSize = this.configuration.Equipment.
               Faulhaber_1.InputBufferSize.value.AsString().
               AsInt();
10         int outBuffSize = this.configuration.Equipment.
               Faulhaber_1.OutputBufferSize.value.AsString().
               AsInt();
11         int comConfig[] = [comPort, baudRate, dataBits,
               parity, stopbit, inBuffSize, outBuffSize];
12         string framingConfiguration[] = ["", "\x0d", "", "\
               x0d"];
13
14         EngineComInterface mci = EngineComInterface.New(
               comConfig, framingConfiguration);
15         int feedbackResolution = this.configuration.
               Equipment.Faulhaber_1.FeedbackResolution.value.
               AsString().AsInt();
16         return FaulhaberEngineDriver.New(mci,
               feedbackResolution);
17     }
```

Listing 8: Erzeugung eines Faulhaber Motor Objekts

Durch die Verwendung des Factory Entwurfsmusters ergibt sich eine Architektur wie in Abbildung "Architektur durch Factory Entwurfsmuster". Am linken Rand sind die entsprechenden Schichten der Architektur abgebildet. Die den Schichten zugeordneten Module, sowie deren Relationen sind daneben zu sehen. Aus Gründen der Lesbarkeit wird auf die Auflistung der Klassenmethoden und Variablen verzichtet.

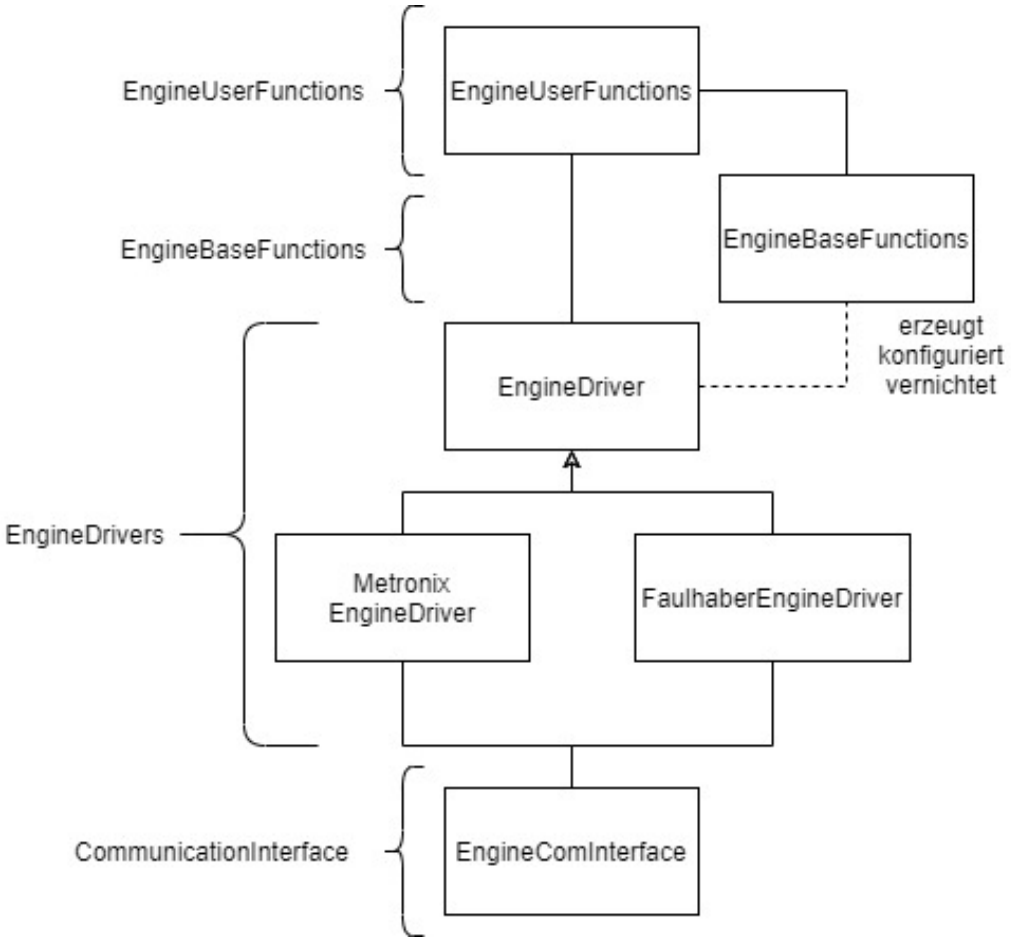

Abbildung 22: Architektur durch Factory Entwurfsmuster

7 Evaluation

In diesem Kapitel wird die Evaluation des Projekts beschrieben. Die Evaluation gliedert sich in zwei Bestandteile. Zuerst wird eine Nutzenanalyse durchgeführt. Mit dieser soll geprüft werden, welchen realen Nutzen das Projekt im späteren Einsatz liefert. Im zweiten Schritt wird das Ergebnis der Arbeit mit den erstellten Anforderungen aus Kapitel "Anforderungsdefinition" abgeglichen.

7.1 Nutzen Analyse

Um zu bestimmen, welchen Nutzen die Einführung der Suite auf die Entwicklung eines Tests hat, werden zwei Kriterien verwendet. Als Erstes wird verglichen wie viele Zeilen Code durch die Verwendung der Testsuite im eigentlichen Testfile eingespart werden können. Um zu bestimmen wie viele LOC durch den Einsatz der Suite eingespart werden können, werden verschiedene bisherige Testfälle betrachtet. Hierbei handelt es sich um die Testfälle, welche in Tabelle "Auswertung LOC pro Testfall" aufgeführt sind.

Name	Geräte	LOC
CriticalSpeed	Faulhaber Motor	659
ExtendedFuncHFSinCos	Faulhaber Motor, Hiperface	865
MultiturnAngleVerification	Faulhaber Motor	749

Tabelle 6: Auswertung LOC pro Testfall

In Tabelle "Auswertung Einsparung" sind die Testfälle, sowie deren LOC bei Verwendung der Testsuite dargestellt:

Name	LOC	Einsparung	Einsparung %
CriticalSpeed	659	ca. 50	7,59
ExtendedFuncHFSinCos	865	ca. 80	9,24
MultiturnAngleVerification	749	ca. 35	4,67

Tabelle 7: Auswertung Einsparung

Zu erkennen ist hierbei, dass es in allen betrachteten Testfällen zu einer Reduzierung der LOC kommt. Hierbei ist jedoch anzumerken, dass es sich bei den drei Testfällen nicht um Testfälle für den SKx handelt. Bei Testfällen für den SKx ist mit einer deutlich höheren Einsparung zu rechnen. Der Hauptvorteil welcher sich durch die Anwendung der Suite ergibt ist jedoch eher im Bereich des Clean Code zu sehen. Durch die Suite wird die Verwendung der verschiedenen Hardwarekomponenten standardisiert und ausgelagert, was zu einem übersichtlicheren Testfile führt. Auch ist die Komplexität für den Entwickler selbst deutlich geringer, da dieser sich nicht mit der expliziten Implementierung zum Beispiel des Motortreibers auseinandersetzen muss.

7.2 Abgleich der Anforderungen

Die Anforderungen welche in "Anforderungsdefinition" definiert sind, werden hier mit der realen Implementierung abgeglichen. In den Tabellen "Abgleich Anforderungen funktional" und "Anforderungen nicht-funktional" sind die Ergebnisse aufgeführt. Hierbei wird jeweils zwischen Kriterien "erfüllt", "teilweise erfüllt" und "nicht erfüllt" unterschieden. Alle Anforderungen werden grundlegend erfüllt. Ausnahme hierbei sind die Anforderungen "F07" und "NF03 ". Anforderung F07 ist nur teilweise erfüllt, da im Rahmen des Proof of Concept der Bereich der Spannungsversorgung zwar geplant, jedoch nicht implementiert wird. Weiterhin ist im Rahmen des Proof of Concept kein ausführliches Code Review vorgesehen, sodass Anforderung NF03 zwar umgesetzt, die Umsetzung jedoch noch nicht verifiziert ist.

Nr.	Bezeichnung	Status	Bemerkung
F01	Integration in Automated Testing @ GBC07	erfüllt	durch Verwendung von ITE als Sprache sowie entsprechende Architektur
F02	Schnittstellen Config	erfüllt	Hiperface Funktionalität vollständig implementiert
F03	Diagnose Funktionalität	erfüllt	Hiperface Funktionalität vollständig implementiert
F04	Core Funktionalität	erfüllt	Hiperface Funktionalität vollständig implementiert
F05	Identifikation des DUT	erfüllt	Hiperface Funktionalität vollständig implementiert
F06	Hiperface Befehle	erfüllt	Hiperface Funktionalität vollständig implementiert
F07	Error Codes	teilweise erfüllt	Hiperface Funktionalität vollständig implementiert, Motorfunktionalität vollständig implementiert, Spannungsversorgung geplant

Tabelle 8: Abgleich Anforderungen funktional

Nr.	Bezeichnung	Status	Bemerkung
NF01	Externes Equipment	erfüllt	Einbinden von bisher vorhandenem Equipment geplant und teilweise implementiert, durch geplante Architektur ist das Einbinden weiterer HW möglich
NF02	Universell einsetzbar	erfüllt	Durch geplante Architektur ist ein Einsatz auch in andern Projekten möglich, die einzelnen Module sind so gekapselt das sie wiederverwendet werden können.
NF03	Coding Guidelines	teilweise erfüllt	Die Coding Guidelines werden beachtet. Jedoch fand bis jetzt kein entsprechendes Refactoring des Codes statt, um dies zu verifizieren.

Tabelle 9: Anforderungen nicht-funktional

8 Fazit / Ausblick

In diesem Kapitel wird rückwirkend auf das Projekt eingegangen. Der Ablauf des Projekts wird kritisch reflektiert. Im weiteren Verlauf des Kapitels wird auf den zukünftigen Verlauf des Projekts eingegangen.

8.1 Fazit

Ziel dieser Arbeit war es eine Testsuite für die anstehende Portierung des SKx zu erstellen. Die Testsuite soll den Entwickler von Softwaretests bei seiner Aufgabe unterstützen und somit den Prozess beschleunigen. Die Testsuite sollte in das Projekt Automated Testing @ GBC07 integriert werden. Zu Beginn des Projektes wurde ein Überblick über die verschiedenen Softwaretests geschaffen. Als Herausforderung hierbei stellte sich heraus, dass es noch keine Dokumentation der SKx Softwaretests gibt. Aus diesem Grund wurde auf die Dokumentation der Softwaretests am SEY zurückgegriffen. Die verschiedenen Softwaretests wurden nach definierten Kriterien analysiert und geprüft welche von ihnen sich mit der Suite abdecken lassen. Hierbei lag das Augenmerk auf der Automatisierbarkeit der Tests. Die in diesem Arbeitsschritt identifizierten Tests wurden im Anschluss geclustert und herausgearbeitet welche Hardware Komponenten verwendet werden. Aus den erarbeiteten Informationen wurde Anforderungen an das Projekt definiert und diese im Anschluss mit den Kunden abgestimmt. Die Requirements wurden hierbei in funktional und nicht-funktionale unterteilt. Die Analyse- und Konzeptionsphase zog sich real deutlich länger, als ursprünglich geplant. Grund hierfür war die aufwendige Analyse der Testfälle. In der folgenden Konzeptionsphase wurden zuerst die bisher vorhandenen Softwarekomponenten (Motor-, Hiperfacetreiber etc.) analysiert. Hier lag das Augenmerk darauf, zu identifizieren welche Teile der bisherigen Software eventuell wiederverwendet werden können und welche Architektur bisher verwendet wurde. Im zweiten Teil der Kozeptionierung erfolgte die Erstellung einer entsprechenden neuen. Hierfür wurden verschiedene Architekturmodelle (MVC, Pipe-Filter, Schichtenarchitektur) verglichen und das am Besten geeignete Modell ausgewählt. Das Ergebnis

der Konzeption ist eine vier-Schichtarchitektur. Im Zuge der Konzeptionsphase wurde darüber hinaus eine wirtschaftliche Betrachtung des Projekts durchgeführt. Diese gestaltet sich jedoch schwierig, da sich nur schwer abschätzen lässt, welchen realen finanziellen Vorteil das Projekt in Zukunft bringen wird. In der folgenden Implementierungsphase wurde ein Teil des Konzepts im Rahmen eines Proof of Concept umgesetzt. Die Programmierung erfolgte hierbei in der durch SICK entwickelten Programmiersprache ITE. Durch die Verwendung von ITE kam es bereits während der Konzeption zu verschiedenen Herausforderungen, da sich mit ihr gewisse Konstrukte wie zum Beispiel Interfaces nicht wie in andern objektorientierten Sprachen abbilden lassen. Durch diese Herausforderung konnten nicht alle architektonischen Probleme mit bekannten Entwurfsmustern gelöst werden. Nach der prototypischen Implementierung wurden im Rahmen der Validierung drei verschiedene Testfiles und die Auswirkung der Testsuite auf die Anzahl der benötigten Codezeilen untersucht. Trotz des Umstandes, dass es sich beim Proof of Concept lediglich um eine Teilimplementierung handelt, konnten hier bereits passable Ergebnisse erzielt werden. Weiterhin wurde das Ergebnis des Projekts mit den definierten Anforderungen abgeglichen, wobei diese weitestgehend erfüllt wurden.

8.2 Ausblick

Nachdem es sich im Rahmen dieser Arbeit nur um eine Proof of Concept Implementierung handelt, wird die tatsächliche Implementierung im Rahmen des Portierungsprojekt voraussichtlich fortgeführt werden. Hierbei kann davon ausgegangen werden, dass die in der Arbeit erstellten Konzepte und die darin enthaltene Architektur umgesetzt wird. Die bisher entwickelten Softwarekomponenten können hierfür weiterverwendet werden. Ein Refactoring, sowie ausführliche Tests sollten jedoch noch durchgeführt werden, um den Code zu verifizieren. Von Vorteil wäre eine enge Zusammenarbeit mit dem für die Portierung zuständigen Softwareentwickler. Nach Erstellung des konkreten Testplans durch den Entwickler müssen eventuell noch kleinere Änderungen an den "UserFunctions" vorgenommen werden. Jedoch ist dies aufgrund der

gewählten Architektur einfach möglich. Weiterhin muss die Logging Funktion, sowie die Ansteuerung der verschiedenen Netzteile implementiert werden. Durch die gewählte Architektur ist das Projekt auch für weitere Hardware, oder Softwareschnittstellen erweiterbar. So können zum Beispiel sehr schnell und komfortabel weitere Motoren eingebunden werden.

9 Literatur

Literatur

[1] SICK AG, "Die Unternehmensgeschichte von SICK," 2021, zugegriffen am 29.06.2021 08:25. [Online]. Available: https://www.sick.com/de/de/ueber-sick/die-unternehmensgeschichte-von-sick/w/the-history-of-sick/

[2] SICK AG, "SICK Unternehmenspräsentation," 2020, zugegriffen am 29.06.2021 08:30.

[3] mikrocontroller.net, "ST7," mikrocontroller.net, 2009, zugegriffen am 29.06.2021 11:00. [Online]. Available: https://www.mikrocontroller.net/articles/ST7

[4] Manju Khari, "Optimized test suites for automated testing using different optimization techniques," 2017, zugegriffen am 26.08.2021 14:00.

[5] Dipl.-Inform. Till Fischer, "Eine Technologe für das durchgängige und automatisierte testen eingebetteter Software," 2016, zugegriffen am 26.08.2021 14:00.

[6] Prof.Dr. Wolfgang Schramm, "Software-Architektur & Entwurf: Unterlagen Vorlesung Softwaretechnik," 2020.

[7] A. Spillner. *Praxiswissen Softwaretest - Testmanagement: Aus- und Weiterbildung zum Certified Tester - Advanced Level nach ISTQB-Standard*, 3. Aufl. Heidelberg: dpunkt-Verl., 2011.

[8] M. Broy und M. Kuhrmann. *Einführung in die Softwaretechnik*, 1. Aufl. ser. Xpert.press. Berlin and Heidelberg: Springer Berlin Heidelberg and Imprint: Springer Vieweg, 2021.

[9] IT-Beauftragter der Bundesregierung, "Konzepte und Inhalte des V-Modell XT Bund: Systementwurf," 2005,

zugegriffen am 05.08.2021 10:20. [Online]. Available: http://download.gsb.bund.de/BundesCIO/V-Modell_XT_Bund/V-Modell%20XT%20Bund-2.0-HTML/ce1ff6e08414e9.html

[10] Wilhelm Hasselbring, "Software-Architektur," 2006, zugegriffen am 22.07.2021 08:35. [Online]. Available: https://gi.de/informatiklexikon/software-architektur

[11] Prof. Dr. Wolfgang Schramm, "Softwarearchitektur & Entwurf: Unterlagen zur Vorlesung Softwaretechnik an der HS Mannheim," 2009. [Online]. Available: https://services.informatik.hs-mannheim.de/~schramm/see/files/Kapitel04.pdf

[12] tagesschau, "Boeing 737 Max darf in Europa starten," 2021, zugegriffen am 28.08.2021 14:30. [Online]. Available: https://www.tagesschau.de/wirtschaft/technologie/boeing-737max-105.html

[13] Daniel Lindner, "Automatisierte Tests: Stützpfeiler der Langlebigkeit: Vorelsung Advanced Software Engeneering DHBW Karlsruhe," 2020, vorelsung Advanced Software Engeneering DHBW Karlsruhe.

[14] F. Witte. *Testmanagement und Softwaretest: Theoretische Grundlagen und praktische Umsetzung*, 1. Aufl. Wiesbaden: Springer Fachmedien Wiesbaden and Imprint: Springer Vieweg, 2016.

[15] S. Grünfelder, M. G. Harbour, R. Wilhelm, J. Bergsmann, und O. Alt. *Software-Test für Embedded Systems: Ein Praxishandbuch für Entwickler, Tester und technische Projektleiter*, 2. Aufl. Heidelberg: dpunkt.verlag, 2017.

[16] Marvin Böck, "Programmentwurf: Fridgeplaner-Kühlschrank und Vorratsverwaltung: Prüfungsleistung für die Vorlesung Advanced SW-Engeneering 2020/2021," zugegriffen am 05.07.2021 12:00.

[17] S. Basler. *Encoder und Motor-Feedback-Systeme: Winkellage- und Drehzahlerfassung in der industriellen Automation.* Wiesbaden: Springer Vieweg, 2016.

[18] ISTQB Glossary, "ISTQB Glossary," 2021, zugegriffen am 08.07.2021 08:15. [Online]. Available: https://istqb-glossary.page/de/testsuite/

[19] RedHat, "DEVOPS: Was versteht man unter CI/CD?" 2021, zugegriffen am 08.07.2021 12:30. [Online]. Available: https://www.redhat.com/de/topics/devops/what-is-ci-cd

[20] SICK AG, "Automated Testing: Intranet Eintrag Automated Testing," 2021, zugegriffen am 12.07.2021 11:20. [Online]. Available: https://mosaicplus.sick.com/pages/viewpage.action?pageId=489716689

[21] Dr. Martin Do, "Sensorik 1: Einführung und Interne Sensoren," 2020.

[22] SICK AG, "Produktübersicht MFB," 2021, zugegriffen am 21.06.2021 11:00. [Online]. Available: https://www.sick.com/de/de/motor-feedback-systeme/motor-feedback-systeme-rotativ-hiperface/sksskm36/c/g215684

[23] SICK AG, "Hiperface Schnittstellenbeschreibung," 2016, zugegriffen am 21.06.2021 4:50.

[24] STMicroelektronics, "Datenblatt ST7," 2008, zugegriffen am 30.06.2021 12:30.

[25] STMicroelektronics, "STM8 Reference Manual," 2017, zugegriffen am 26.08.2021 14:50. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm8af52a8.html#documentation

[26] Dr. Wolfgang Schröder, "SMARTe Ziele und wie sie formuliert werden," 2018, zugegriffen am 20.08.2021 11:25. [Online]. Available: https://www.business-wissen.de/hb/smarte-ziele-und-wie-sie-formuliert-werden/

[27] Clean Code Development, "Die Tugenden des CCC," 2015, zugegriffen am 23.08.2021 16:20. [Online]. Available: https://clean-code-developer.de/die-tugenden/

[28] E. Gamma und D. Riehle. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*, [nachdr.] Aufl. ser. Programmer's choice. München: Addison-Wesley, 2008.

10 Anhang

Test	Kat.	Inhalt
SW00001	General Requirements	Firmeware version format
SW00030	HW Architecture	Master IcDSL IcDSL resolution is set by EEPORIM parameter Part 1: SEY90
SW00031	HW Architecture	Master IcDSL IcDSL resolution is set by EEPORIM parameter Part 2: SEY70
SW00058	Functional Requirements	Boot-loader
SW00132	Functional Requirements	General user cmds W_EnterBootloader (70h) NO support of safety variants
SW00017	HW Architecture	Master mic. Supply Voltage Monitoring Part 1 (Accuracy)
SW00018	HW Architecture	Master mic. Supply Voltage Monitoring Part 2 (Lower limit)
SW00019	HW Architecture	Master mic. Supply Voltage Monitoring Part 3 (Higher limit)
SW00020	HW Architecture	Master mic. Supply Voltage Monitoring Part 4 (Voltage Correlation)
SW00021	HW Architecture	Master mic. Internal Voltage Rail 1.8V
SW00064	Functional Requirements	Master power supply check
Neu		LED Strom
SW00135	Calibration/Production	Internal temperature sensor
SW00036	HW Architecture	IcDSL Master IcDSL temperature read-out (Temperature coefficients)
SW00037	HW Architecture	IcDSL Master IcDSL temperature read-out (warning limits)
SW00038	HW Architecture	IcDSL Master IcDSL speed read-out
SW00060	Functional Requirements	Max. revolution speed at start-up
SW00127	Functional Requirements	General user cmds R_POS(42h)-SES/SEM70 Resolution (Single & multi tunr) & counter direction
SW00128	Functional Requirements	General user cmds R_POS(42h)-SES/SEM70 Binary position format
SW00130	Functional Requirements	General user cmds R_POS(42h)-Position actualization with first fallen edge of the answer
SW00075	Functional Requirements	SEY70/90 shall provide only Hiperface default values (9600baud)in normal mode
SW00076	Functional Requirements	SEY70/90 shall provide also 115200Bd/1xTo in Internal mode
SW00136	Non-Functional Requirements	Hiperface confliguration parameter are storedd in block 0 of the Master EEPROM
SW00138	Non-Functional Requirements	Master IcDSL configuration are stored in block 1of the Master EEPROM
SW00077	Functional Requirements	General user commands
SW00078	Functional Requirements	Error handlig referred to Hiperface specification (Three different kinds of error handling)
SW00089	Functional Requirements	Supported error codes (Master) 0Bh ERR_Unknown_CMD
SW00090	Functional Requirements	Supported error codes (Master) 0Ch ERR_NO_OF_TX_DATA
SW00091	Functional Requirements	Supported error codes (Master) 0Dh ERR_CMD_ARG
SW00092	Functional Requirements	Supported error codes (Master) 0Eh ERR_WRITE_NOT_Allowed
SW00093	Functional Requirements	Supported error codes (Master) 0Fh ERR_WRONG_ACCESS_CODE
SW00094	Functional Requirements	Supported error codes (Master) 10h ERR_BLOCK_SIZE_CANNOT_CHANGE
SW00095	Functional Requirements	Supported error codes (Master) 11h ERR_DATA_WORD_ADDR_OUT
SW00096	Functional Requirements	Supported error codes (Master) 12h ERR_DATA_NON_EXIST_DF
SW00102	Functional Requirements	Supported error codes (Master) 21h ERR_POS_MULTI_AMPL
SW00103	Functional Requirements	Supported error codes (Master) 22h ERR_POS_MULTI_SYNC