Bachelor Thesis

Entwicklung einer Software zum Testen externer Systeme

an der FH Aachen, Campus Jülich

Autor: Heiko Joshua Jungen

Matr. Nr.: 4006608

Fachbereich 09: Medizintechnik und Technomathematik

Studiengang: Scientific Programming B.Sc.

1. Prüfer: Prof. Dr. rer. nat. Stephan Jacobs

2. Prüfer: Dennis Rollesbroich

Aachen, den 7. Juli 2016

Eigenständigkeitserklärung

Diese Arbeit ist von mir selbständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

Heiko Joshua Jungen

Zusammenfassung

Für die erfolgreiche Entwicklung von Hard- und Softwaresystemen ist ausgiebiges testen erforderlich. Dafür gibt es zahlreiche Testwerkzeuge in Form von eigenständiger Software und Software-Frameworks. Jedoch sind die üblichen Testwerkzeuge beim Testen von externen Systemen oftmals nicht einsetzbar, da die Werkzeuge nicht auf spezifische Geräte anzupassen sind. In dieser Arbeit wird eine Software zum Testen von externen Systemen entwickelt. Mit dieser werden sowohl Testfälle erstellt und verwaltet, als auch automatisiert durchgeführt und protokolliert. Das Programm ist spezialisiert auf das Testen von externen Geräten und für beliebige Geräte einsetzbar. Im Fokus steht die konkrete Entwicklung jener Software, welche die Anforderungserfassung, Konzeption, Zeitplanung und Realisierung des Projekts umfasst. Das Ergebnis dieser Arbeit ist das entwickelte Testwerkzeug, mit welchem externe Systeme getestet werden können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung

Die Bereiche Test und Testautomatisierung nehmen eine immer wichtiger werdende Rolle bei der Entwicklung von Hard- und Software ein. Vor allem die stetig wachsenden Anforderungen an Hard- und Softwareprodukte erfordern umfangreiche Tests, da die Anforderungserfüllung ein entscheidender Punkt der Produktentwicklung ist. Die Entwicklung von Hard- und Softwaresystemen ist ein komplexer und fehleranfälliger Prozess, denn „Komplexe Systeme sind analytisch in begrenzter Zeit nur unvollständig erfassbar. Fehler sind damit zwangsläufig die Folge.“[Vig10, S. 3]. Nicht erfüllte Anforderungen und unerwartetes Fehlverhalten der Produkte können diese unbrauchbar oder schädlich machen, weshalb Tests ein wichtiger Bestandteil der Entwicklung sind.

Im Bereich des Softwaretestens gibt es zahlreiche Testwerkzeuge, welche die Erstellung, Durchführung, Protokollierung und systematische Auswertung von Tests unterstützen und automatisieren. Bei der Entwicklung von Hard- und Softwaresystemen können jene Testwerkzeuge jedoch nicht verwendet werden, da sie die Interaktion mit den speziellen Geräten nicht unterstützen. Deswegen wird ein Testwerkzeug entwickelt, welche den Testprozess von externen Systemen bei der Erstellung, automatisierten Durchführung und Protokollierung unterstützt.

Zu Beginn dieser Arbeit wird das Unternehmen, für welches die Software entwickelt wird, vorgestellt und die bestehende Problematik geschildert. Nachdem der Kontext geklärt ist werden die für diese Arbeit notwendigen Grundlagen erläutert. Diese thematisieren sowohl den Bereich des Testens als auch den der Softwareentwicklung. Kapitel 4 widmet sich der konkreten Entwicklung der Arbeit. Dazu zählen die Anforderungsanalyse, Zeitplanung, Konzeption und Umsetzung des Softwareprojekts. Nach der abgeschlossenen Entwicklung folgt die Auswertung des Softwareprodukts. Diese wird anhand eines vom Unternehmen entwickelten Geräts durchgeführt. Abschließend werden die erarbeiteten Ergebnisse in Kapitel 6 zusammengefasst.

2. Kontext

Die ATESTEO GmbH ist Entwicklungspartner diverser, namhafter Automobilhersteller und ein Experte im Antriebsstrang-Testing. Das Unternehmen testet für seine Kunden beispielsweise Getriebetechnologien unter verschiedenen Bedingungen. Die zu diesem Zweck benötigte Drehmoment-Messtechnik wird vom Unternehmen selbst entwickelt und hergestellt. Jene Messtechnik besteht oftmals aus mehreren einzelnen Komponenten, die zusammen ein Messsystem formen. Entwickelt wird sowohl die verwendete Hardware, als auch die eingesetzte Software. Die Hardware wird bei externen Zulieferern produziert und vom Unternehmen selbst zusammengesetzt. Auf die verbaute Hardware wird anschließend die Software installiert.

Bei der Herstellung der Messgeräte entsteht ein hoher Testaufwand, weil jedes hergestellte Gerät getestet werden muss. Es ist nicht ausreichend wenn Hardware oder Software einmal vorab getestet und anschließend mit diesen eine Serie von Geräten hergestellt wird. Daran ist einerseits problematisch, dass bei der Produktion von Hardware Fehler möglich sind. Andererseits werden Messgeräte in verschiedenen Konfigurationen hergestellt, sodass diverse Kombinationsmöglichkeiten einzelner Hard- und Softwarekomponenten entstehen. Aus diesen Gründen muss jedes hergestellte Messgerät separat auf korrekte Funktionsfähigkeit getestet werden.

Zum Testen der Messgeräte ist oftmals eine umfangreiche Testumgebung aufzubauen. Diese besteht mindestens aus dem Testobjekt, aber auch weitere Geräte sind möglich. Denkbar sind weitere wie Spannungsmesser, Frequenzmesser oder ähnliche Messgeräte. Außerdem muss das Messgerät im Kontext des gesamten Messsystems getestet werden, sodass die übrigen Bestandteile des System ebenfalls in der Testumgebung vorhanden sein müssen.

Der Status Quo des Testprozesses ist, dass jedes Gerät einen händischen Test durchläuft. Wegen dem dadurch entstehenden hohen Aufwand ist der Testumfang reduziert, weshalb bestimmte Funktionalitäten des Geräts nicht ausreichend überprüft

werden können. Ein weiteres Problem des aktuellen Testverfahrens ist, dass nur wenige Personen qualifiziert genug sind Tests durchzuführen. Grund dafür ist die technisch komplizierte Testdurchführung, welche sowohl Fachwissen, als auch gute Kenntnisse des Geräts erfordern. Das hat zur Folge, dass es nicht möglich ist den Testaufwand auf mehrere Personen zu verteilen. Außerdem ist die Dokumentation von Tests, als auch Testergebnissen lückenhaft. Dadurch sind die Tests oftmals nur für die Personen nachvollziehbar, die jene selbst durchgeführt haben.

3. Grundlagen

3.1. Was bedeutet Test?

In einem Test wird ein Testobjekt verwendet und dabei überprüft. Dazu wird festgelegt wie sich das Testobjekt, auch *System Under Test* genannt, im Verlauf des Tests verhalten soll. Durch den Vergleich der Erwartungshaltung mit dem beobachteten Verhalten im Test wird das Testergebnis bestimmt [Vig05, S. 20]. Ein Test besteht aus einem Testablauf und den Testdaten. Der Testablauf wird unter Verwendung der Testdaten schrittweise ausgeführt. Jene Daten bestehen aus Eingabewerten und erwarteten Ausgabewerten, wobei dessen Vergleich zum Testergebnis führt [Vig05, S. 22]. Des Weiteren gehört zu einem vollständigen Testfall eine Vor- und Nachbedingung. Für manche Tests muss das Testobjekt zuerst in einen Zustand gebracht werden, in dem es getestet werden kann. Dementsprechend muss es anschließend in den Standardzustand zurück versetzt werden [Spi+09, S. 15]. Vor- und Nachbedingungen von Testfällen sind besonders wichtig, wenn mehrere Tests automatisiert nacheinander ausgeführt werden. Die Bedingungen stellen sicher, dass ein Test nur unter gültigen Testbedingungen ausgeführt wird beziehungsweise das nach einem Test das Testobjekt in einen Standardzustand zurück versetzt wird.

Für möglichst aussagekräftige Tests wird bei der Erstellung der destruktiver Ansatz verfolgt. Mit diesen Tests soll nicht gezeigt werden, dass ein *System Under Test* funktioniert, sondern dass dieses nicht funktioniert [Vig05, S. 20 f.]. Jedoch kann allgemein durch Tests nicht die absolute Fehlerfreiheit eines Testobjekts nachgewiesen, sondern lediglich dass es unter Testbedingungen funktionsfähig ist.

3.2. Darstellungsmöglichkeiten von Tests

Für die automatisierte Ausführung von Testfällen ist eine einheitliche Darstellung dieser notwendig, wobei es dafür verschiedene Herangehensweisen gibt. Eine Möglichkeit ist Tests aufzuzeichnen, wozu ein Testwerkzeug Nutzerinteraktionen, wie Mausbewegungen und Tastendrücke, aufnimmt und diese automatisiert wiederholt. Die Erstellung und Durchführung gestaltet sich dementsprechend simpel, jedoch haben Änderungen an graphischen Oberflächen des Testobjekt meistens eine vollständige Neuerfassung der Tests zur Folge [SBB12, S. 66].

Ein weitere Methode ist Tests in Programmcode zu speichern und diesen auszuführen. Der Programmcode kann Schnittstellen des Testobjekts direkt ansprechen und benötigt im Gegensatz zum Aufzeichnen keine weiteren Zwischenschritte. Bei diesem Ansatz ist es jedoch erforderlich, dass der Testersteller programmieren kann [SBB12, S. 66].

Die Testdarstellung anhand von sogenannten Schlüsselwörtern ist eine weitere Möglichkeit. Dabei werden Tests, wie beim programmatischen Ansatz in schriftlicher Form dokumentiert, jedoch mit einer für den Menschen leichter verständlichen Syntax. Bei der schlüsselwortgetriebenen Darstellung findet eine Trennung von Testdesign und Testimplementation statt. Dadurch ist für die Testerstellung keine Programmiererfahrung notwendig und ein flexibler Testablauf möglich [SBB12, S. 68 ff.].

3.3. Erfassung von Anforderung

Die Anforderungserfassung ist der erste Schritt bei der systematischen Entwicklung von Software. Anforderungen bilden die Basis eines Projekts, denn die nachfolgenden Schritte bauen auf diese auf und der Projekterfolg wird an ihnen gemessen. In dieser Phase der Entwicklung sind Fehler am schwerwiegendsten und verursachen umso mehr Kosten, desto später sie entdeckt werden [Gol11, S. 163]. Diese Fehler können nicht durch hervorragende Qualität gut gemacht werden [Kle09, S. 51], weswegen die Anforderungserfassung gründlich durchgeführt werden muss. In Zusammenarbeit mit dem Auftraggeber der Software werden Anforderungen erfasst, denn diese müssen die Wünsche des Kunden enthalten. Von diesem kann nicht erwartet

werden, dass er die Anforderungen an die Software alleine aufstellt, weswegen diese im Gespräch durch gezielte Fragen in Erfahrung gebracht werden müssen. Eine weitere Möglichkeit Anforderungen zu erfassen ist, wenn durch Software eine bestehender Vorgang verbessert werden soll. In diesem Fall hilft es den bestehenden Zustand zu analysieren und durch Optimierungspotential neue Anforderungen zu ermitteln [Gol11, S. 166]. Bei der Anforderungserfassung wird demnach die Frage „Was tut ein System?“geklärt, wobei die Details der Implementierung noch keine Rolle spielen [R.B+05, S. 75]. Des Weiteren werden Anforderungen formal in Form von Diagrammen oder in natürlicher Sprache festgehalten. Dabei hat die natürliche Sprache den Vorteil, dass die Anforderungen für jeden, also auch für den Auftraggeber verständlich sind. Im allgemeinen zeichnen sich gute Anforderungen dadurch aus, dass sie adäquat, vollständig, eindeutig, widerspruchsfrei und verständlich formuliert sind [Gol11, S. 164].

3.4. Konzeptentwurf von Softwareanwendungen

Bei der Entwicklung eines objektorientierten Konzepts werden gröbere Strukturen als Klassen benötigt. Klassen sind zu detailliert um komplexe Anwendungen strukturiert darzustellen, weswegen auf sogenannte Komponenten zurückgegriffen wird. Komponenten sind gröbere Einheiten die eine festgelegte Aufgabe haben oder einen bestimmten Dienst anbieten. Sie besitzen eine definierte Schnittstelle über die Informationen oder Dienste nach außen hin verfügbar gemacht werden. Des Weiteren können sie ineinander verschachtelt stehen, wodurch komplexe Aufgaben in mehrere Unterkomponenten aufgeteilt werden. Im Gegensatz zu Klassen sind Komponenten kein physisches Konstrukt, sondern vielmehr ein konzeptionelles. Eine Klassenstruktur wird zwar auf Basis der Komponenten entwickelt, jedoch können diese stark voneinander abweichen. [R.B+05, S. 89 ff.]

3.5. Planung von Softwareprojekten

Zu Beginn eines Softwareprojekts werden die zu erledigenden Aufgaben anhand der Anforderungen ermittelt und dokumentiert. Große und komplexe Aufgaben sind in Unteraufgaben zu unterteilen, damit deren Übersichtlichkeit und Kalkulierbarkeit gewährleistet bleibt. Die ermittelten Aufgaben und Unteraufgaben können beispielsweise in einer *Work-Breakdown-Structure* festgehalten werden, welche die Projektaufgaben in einer baumartigen Struktur abbildet [Kle09, S. 308]. Bevor ein konkreter Plan zur Abarbeitung der Projektaufgaben entwickelt werden kann müssen die Abhängigkeiten der einzelnen Aufgaben bestimmt werden. Unter Umständen werden für die Bearbeitung einer Aufgabe die Ergebnisse anderer Aufgaben benötigt, wodurch Abhängigkeiten zwischen diesen entstehen. Durch die Festlegung von klar definierten Aufgaben und deren Abhängigkeiten kann die benötigte Entwicklungszeit besser eingeschätzt werden [Kle09, S. 308 f.].

Außerdem ist die Aufwandsschätzung für das Entwicklungsvorgehen eines Softwareprojekts von Bedeutung. Jene ist nicht nur für die Kostenschätzung des Projekts erforderlich, sondern auch für die Projektplanung. Bei der Bearbeitung von Aufgaben sind sowohl die Abhängigkeiten, als auch die technischen und menschlichen Ressourcen zu beachten. Mit einer Zeitschätzung können diese im Projektplan berücksichtigt und Aufgaben idealerweise parallel bearbeitet werden. Für die Durchführung einer Aufwandsschätzung wird das Projekt, wie zuvor beschrieben, in Aufgaben und Unteraufgaben unterteilt. Zu jeder dieser Aufgaben wird der notwendige Zeitaufwand geschätzt. Aus der Summe aller einzelnen Schätzungen ergibt sich somit der Gesamtaufwand. Eine anerkannte Methode der Aufwandsschätzung basiert auf Erfahrungswerten aus vorangegangen Entwicklungsprojekten, dahingegen ist beispielsweise die sogenannte *Function-Point-Methode* ein systematischerer Ansatz. Jedoch ist festzuhalten, dass unabhängig von der gewählten Methode eine Aufwandsschätzung nicht immer zu richtigen Ergebnissen kommen kann. Die Genauigkeit einer Aufwandsschätzung hängt von der gewählten Methode, der Erfahrung des Schätzers und weiteren Faktoren ab [Kle09, S. 314 ff.].

4. Entwicklung

In diesem Kapitel wird die konkrete Entwicklung der Software thematisiert. Zunächst werden die Anforderung an die Software erarbeitet, anhand welcher ein Softwarekonzept sowie ein Zeitplan entwickelt wird. Schließlich werden die im Konzept erarbeiteten Inhalte im Detail betrachtet und realisiert.

4.1. Anforderungen

Es wird ein Testwerkzeug zum Testen von Messgeräten benötigt. Messgeräte, beziehungsweise allgemein externe Systeme sind eigenständige Hard- und Softwaresysteme, die aufgrund der fortlaufenden Entwicklung und Herstellung überprüft werden müssen. Die bisher händisch durchgeführten Tests sollen durch eine Software automatisiert werden. Mit dem Testwerkzeug werden Tests erstellt und bei Bedarf durchgeführt. Außerdem soll für die Testerstellung keine Programmiererfahrung notwendig sein. Die Erstellung wird ausschließlich von Fachpersonal vorgenommen, wohingegen die Durchführung einfach genug sein soll, sodass jeder dazu fähig ist. Beim händischen Testen der Geräte ist Fachpersonal erforderlich, weil die Interaktion mit den Geräten Fachwissen erfordert. Demnach muss die Software zum vereinfachen des Vorgangs die Interaktion mit den Geräten übernehmen. Mit Geräten sind sowohl das *System Under Test* gemeint, als auch mögliche weitere für den Test notwendige Geräte. Des Weiteren muss die Software zu jedem Test ein Protokoll schreiben, welches Informationen über den Ausgang des Tests enthält.

Aus den Anforderungen lässt sich das in Abb. 4.1 gezeigte *Use-Case* Diagramm ableiten. Das Testwerkzeug wird von den Akteuren Personal oder Fachpersonal verwendet. Beide Typen von Akteuren können Tests durchführen, jedoch ist lediglich das Fachpersonal befähigt Tests zu erstellen. Die Durchführung von Tests beinhaltet die Verwendung von externen Systemen sowie deren Protokollierung. Bei der

Testerstellung ist das dauerhafte Speichern von Tests mit enthalten. Nachfolgend wird eine Übersicht über die Anforderungen gegeben, wobei zwischen den Kategorien Akteure, externe Systeme, Test erstellen und Test durchführen unterschieden wird.

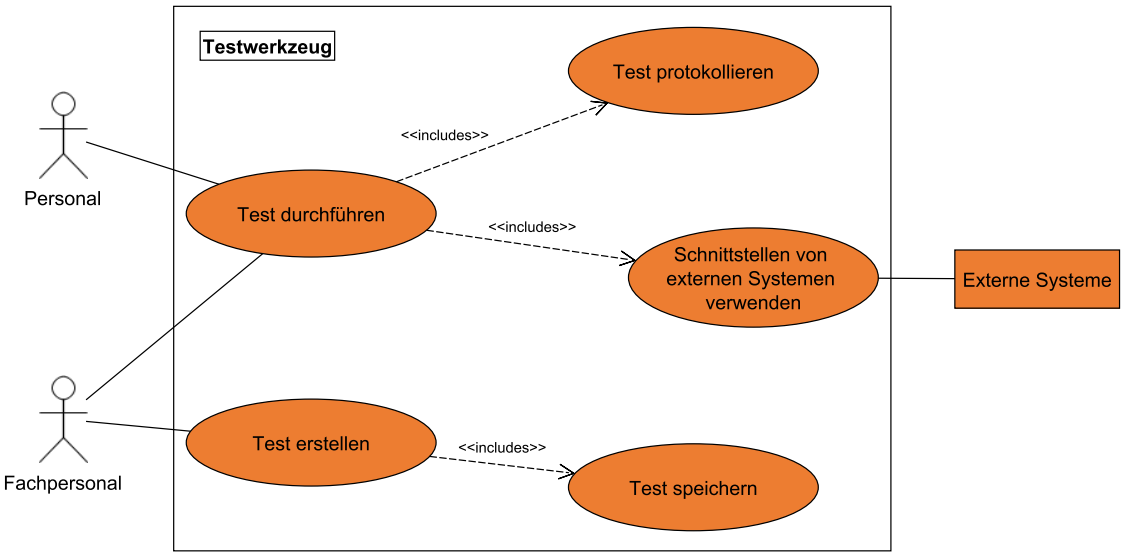

Abbildung 4.1.: *In diesem Use-Case Diagramm sind die Anwendungsfälle des Testwerkzeugs dargestellt. Die Akteure* Personal *und* Fachpersonal*, die mit dem System interagieren, sind links zu sehen. Die Anwendungsfälle* Test durchführen *und* Test erstellen *können von den Akteuren aufgerufen werden. Unter* Externe Systeme *auf der rechten Seite sind alle externen Systeme zusammengefasst, die in einem Test verwendet werden.*

Akteure: Die Software wird von zwei verschiedenen Personengruppen verwendet. Zum einen gibt es Fachpersonal, welches sich besonders gut mit dem zu testenden System auskennt. Zum anderen gibt es normales Personal, dass das System lediglich verwenden kann, jedoch wenig Wissen über die internen Abläufe hat. Beiden Personaltypen ist es möglich Tests durchzuführen, wohingegen das Fachpersonal zusätzlich Tests erstellen kann.

Externe Systeme: Beim Testen kommuniziert die Software mit einem oder mehreren externen Systemen, wozu in jedem Fall das *System Under Test* zählt. Außerdem ist es möglich weitere externe Geräte, wie beispielsweise ein Spannungsmessgerät, zu verwenden. Die Software muss die Möglichkeit bieten, dass sowohl beliebige, als auch beliebig viele externe Systeme verwendet werden können.

Test erstellen: Bei der Testerstellung wird der Testablauf festgelegt. Für die Testerstellung kann ein umfangreiches Wissen der verwendeten Systeme vorausgesetzt werden, jedoch soll keine Programmiererfahrung notwendig sein. Tests werden dauerhaft gespeichert um diese bei Bedarf durchzuführen.

Test durchführen: Für die Testdurchführung wählt der Benutzer beliebig viele Testdateien aus. Das Programm ermittelt für jeden Test den Testablauf und führt die einzelnen Schritte nacheinander aus. Die Abarbeitung der Einzelschritte verläuft normalerweise automatisch, jedoch sind manuelle Zwischenschritte möglich. Die Software muss dabei die Interaktion mit externen Geräten übernehmen. Jede Testdurchführung schließt mit einem Protokoll ab, welches Informationen zum Ablauf und Ausgang des Tests dokumentiert.

4.2. Konzept

Aus den ermittelten Anforderungen gehen zwei Hauptbereiche hervor, die bei der Entwicklung der Software im Fokus stehen. Der erste Bereich ist die Unterstützung des Testprozesses, der zweite ist die Integration von externen Systemen in diesen. Die beiden Bereiche werden in den nachfolgenden Abschnitten erläutert. Abschließend wird das darauf aufbauende Softwarekonzept vorgestellt.

4.2.1. Testprozess

Der Testprozess wird bei der Erstellung, Durchführung und Protokollierung von Tests durch das Testwerkzeug unterstützt. Für die automatisierte Ausführung wird eine formale Darstellung von Tests benötigt. Dabei ist es für den Erfolg der Software entscheidend, dass die Testerstellung keine Programmiererfahrung erfordert, weswegen eine geeignete Testdarstellung gewählt werden muss. Beim programmatischen Ansatz werden Skripte geschrieben, die direkt mit dem *System Under Test* interagieren. Dies erfordert jedoch Programmiererfahrung, weswegen dieser Ansatz zu verwerfen ist. Dahingegen werden bei der schlüsselwortgetriebenen Testdarstellung Tests mit sogenannten Schlüsselwörtern definiert. Schlüsselwörter sind für den Menschen verständliche Worte oder auch knappe Sätze. Der Testablauf wird nicht

durch ein Skript sondern durch Schlüsselwörter beschrieben, weshalb dies auch ohne Programmiererfahrung möglich ist. Diese werden vom Programm in eine Programmlogik übersetzt, welche dann ausgeführt wird. Diese Übersetzung geschieht mit Hilfe von sogenannten Bibliotheken, die eine Vielzahl von Schlüsselwörtern enthalten. Zu jedem Schlüsselwort gibt es eine Programmlogik, die vom Programm ausgeführt werden kann. Durch die Verwendung von Schlüsselwörtern findet eine Trennung der Testerstellung und der Testimplementierung statt. Diese Trennung ist notwendig, damit die Testerstellung ohne Programmierkenntnisse erfolgen kann. Aus diesen Gründen wird bei der Entwicklung des Testwerkzeug der schlüsselwortgetriebene Ansatz verfolgt.

Das Testwerkzeug arbeitet dabei mit Eingabe- und Ausgabedateien. Erstere werden vom Testersteller erstellt und enthalten den Testablauf in Form von Schlüsselwörtern sowie weitere für den Testablauf notwendige Informationen. Für die Durchführung eines Tests werden vom Benutzer Eingabedateien festgelegt, welche vom Testwerkzeug automatisiert durchgeführt werden. Nach der Durchführung wird eine Ausgabedatei erstellt. In dieser ist ein Protokoll der Testdurchführung gespeichert, welches die Ergebnisse des Testlaufs präsentiert. Das Protokoll unterstützt bei fehlgeschlagen Testfällen, durch entsprechende Fehlernachrichten, die Fehlerfindung im getesteten System.

4.2.2. Integration externer Systeme

Der zweite Hauptbereich des Testwerkzeug ist es externe Systeme in den zuvor beschriebenen Testprozess zu integrieren. Für eine erfolgreiche Integration eines externen Systems ist es notwendig, dass mit diesem kommuniziert und interagiert werden kann. Die Problemstellung dabei ist, dass die Geräte über unterschiedliche Kommunikationstechnologien mit der Software verbunden sind und darüber hinaus jedes Gerät eine eigene Schnittstelle anbietet. Die Programmlogik für die Verbindungsaufnahme zu einem Gerät sowie die Implementierung von Schnittstellen kann demnach nicht allgemeingültig gelöst werden, sondern muss spezifisch auf ein Gerät abgestimmt werden. Deswegen ist für jedes externe System ein einmaliger Programmieraufwand nötig. Dabei werden die Details der Verbindung und die vorhandenen Schnittstellen zu einem Gerät implementiert und dem Testwerkzeug zur Verfügung gestellt. Bei der Erstellung von Testdateien werden die Informationen

zu den Schnittstellen benötigt, damit im Testablauf die Verwendung eines externen Systems festgelegt werden kann. Daher orientiert sich die Integration eines externen Systems am schlüsselwortgetriebenen Ansatz, bei dem die Schnittstellen in Form von Schlüsselwörtern bereitgestellt werden. Die technischen Details der Kommunikation sind in der Programmlogik des Schlüsselworts hinterlegt und müssen dem Testersteller nicht bekannt sein. Bei der Durchführung wird die Programmlogik des Schlüsselworts ermittelt und die Kommunikation mit dem externen Gerät erfolgt gemäß der Implementierung.

4.2.3. Grobdesign der Software

Aus den zwei beschriebenen Hauptbereichen geht das Konzept aus Abb. 4.2 hervor. Innerhalb des schwarzen Kastens mit dem Titel *Testwerkzeug* sind die Softwarekomponenten des Testwerkzeugs zu sehen. Die Erstellung von Tests erfordert, dass der Testersteller die verfügbaren Schlüsselwörter kennt. Deswegen wird der Testersteller vom Testwerkzeug unterstützt, in dem ihm die verfügbaren Schlüsselwörter angezeigt werden. Dies wird in der Komponente *Testerstellung* umgesetzt. Die Eingabedateien werden als sogenannte Testdateien gespeichert, welche bei der Testdurchführung eingelesen werden. Die Komponente *Testdurchführung* liest Informationen und den Testablauf aus Testdateien ein und führt den Ablauf automatisiert durch. Nach einer Testausführung wird eine Ausgabedatei, auch Testprotokoll genannt, gespeichert. Beide Komponenten arbeiten mit der Komponente *Externe Systeme* zusammen. Diese befasst sich mit der Verarbeitung von Schlüsselwörtern und somit implizit für die Kommunikationen zu externen Systemen, welche in der Abbildung in einem Netzwerk zusammengefasst sind. Jene Komponente stellt zum einen die verfügbaren Schlüsselwörter für die Testerstellung bereit. Diese werden dem Testersteller mit zusätzlichen Informationen graphisch angezeigt. Zum anderen ermittelt sie die zu einem Schlüsselwort zugehörige Programmlogik, welche bei der Testdurchführung ausgeführt werden muss.

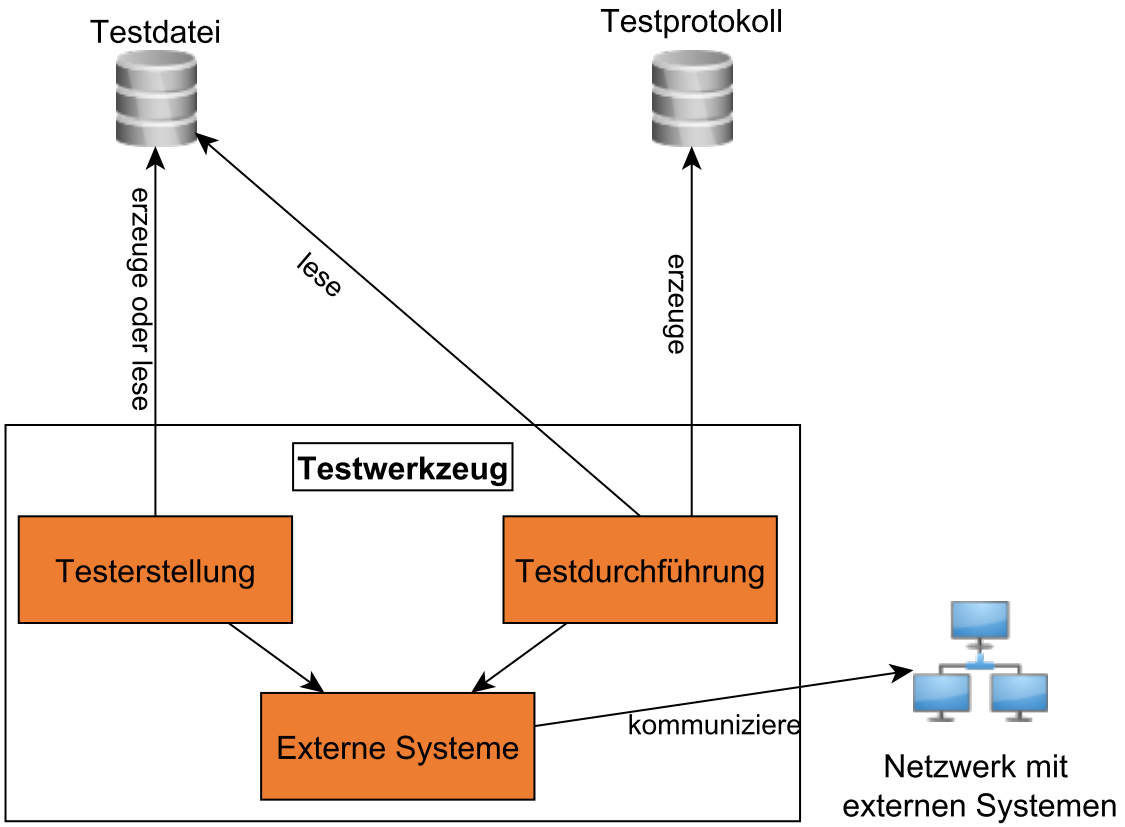

Abbildung 4.2.: *In dieser Abbildung ist das Softwarekonzept zu sehen. Dieses besteht aus den Komponenten Testerstellung, Testdurchführung und Externe Systeme. Des Weiteren sind die Dateien für Tests und Protokolle zu sehen, die von der Software erstellt oder gelesen werden. Externe Systeme sind in einem Netzwerk zusammengefasst, da es beliebig viele Systeme sein können.*

4.3. Zeit- und Projektplanung

Zu Beginn der Entwicklung werden die Aufgaben und Unteraufgaben des Projekts ermittelt. Diese sind in der *Work-Breakdown-Structure* in Abb. 4.3 dargestellt. Zunächst wird in die im Konzept festgelegten Bereiche Testerstellung, Testdurchführung und Integration externer Systeme unterschieden. Zu jeder dieser Hauptaufgaben gibt es eine Reihe Unteraufgaben, welche bei der Entwicklung umgesetzt werden. Für die Erstellung von Tests werden Testdateien geschrieben. Der Inhalt [Aufgabe 01] sowie der formale Aufbau [Aufgabe 02] der Dateien wird festgelegt. In einem Testablauf wird geprüft, ob das Testobjekt den erwarteten Eigenschaften entspricht. Für diese Vergleichsoperationen werden Standardbibliotheken [Aufga-

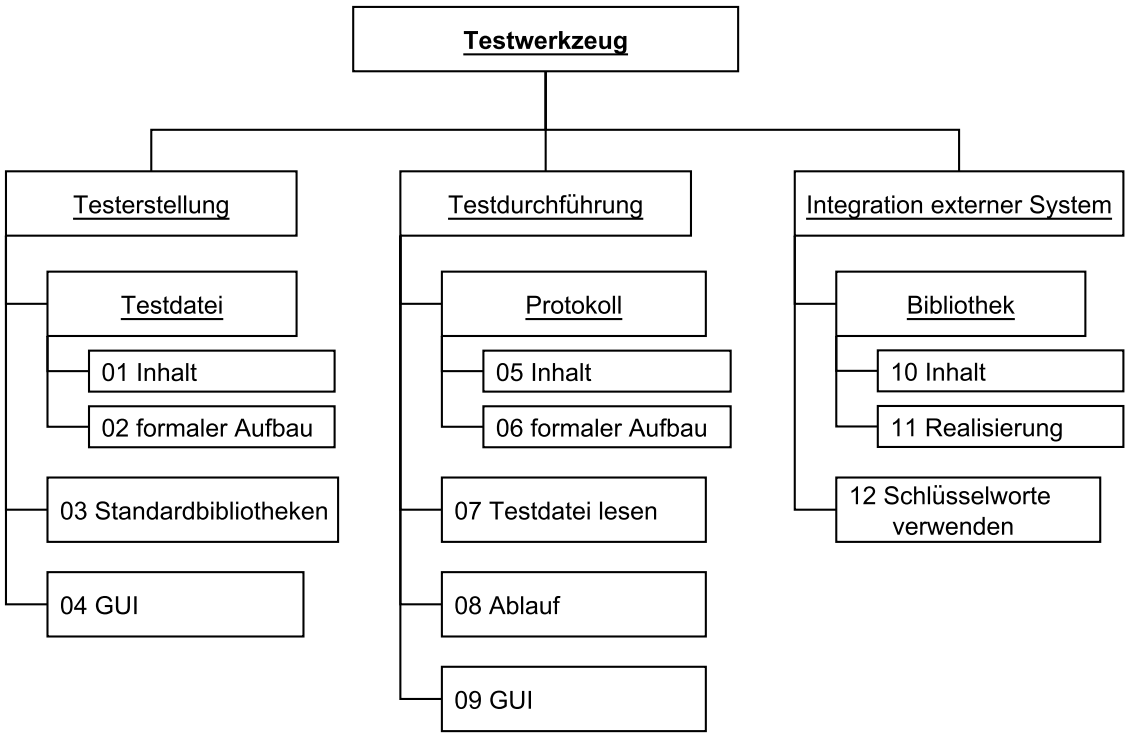

Abbildung 4.3.: *Die Abbildung zeigt eine* Work-Breakdown-Structure *mit den Projektaufgaben des Testwerkzeugs.*

be 03] bereitgestellt, die entsprechende Schlüsselwörter zum Vergleich von Daten enthalten. Des Weiteren wird eine graphische Oberfläche [Aufgabe 04] für die Testerstellung entwickelt. Das Ergebnis einer Testdurchführung ist ein Testprotokoll. Jenes enthält Informationen zum Testlauf [Aufgabe 05] und hat einen einheitlichen Aufbau [Aufgabe 06]. Für die Testdurchführung wird der Testablauf und weitere Informationen aus einer Testdatei eingelesen [Aufgabe 07]. Dabei werden nicht nur die Zeilen ermittelt und zurückgegeben, sondern es findet eine Prüfung auf syntaktische und lexikalische Korrektheit statt. Der Ablauf der Testdurchführung [Aufgabe 08] enthält den Prozess vom Einlesen einer Testdatei bis zur Ausgabe eines Testprotokolls. Zusätzlich wird eine graphische Benutzeroberfläche für die Testdurchführung entwickelt [Aufgabe 09]. Die externen Systeme werden über sogenannte Bibliotheken in den Testprozess integriert. Damit das Testwerkzeug diese Bibliotheken verwenden kann wird der Inhalt [Aufgabe 10] und die technische Realisierung [Aufgabe 11] festgelegt. Schließlich wird die konkrete Verwendung von den in Bibliotheken enthaltenen Schlüsselwörtern erarbeitet [Aufgabe 12].

Zu den Projektaufgaben werden die Prioritäten und mögliche Abhängigkeiten zu

anderen Aufgaben bestimmt. Die Prioritäten und Abhängigkeiten sind für die nachfolgende Zeitplanung des Projekts elementar. Die Priorität einer Aufgabe wird durch eine Zahl zwischen eins und drei definiert, wobei eine eins der höchsten und eine drei der niedrigsten Priorität entspricht. Bei der Prioritätensetzung wird die Kernfunktionalität des Testwerkzeugs am höchsten eingestuft. Dazu zählen Aufgaben der Erstellung, Ausführung und Protokollierung von Tests. Die Entwicklung graphischer Oberflächen wird mit der niedrigsten Priorität versehen. Benutzeroberflächen sind zwar bedeutsam für den Projekterfolg, jedoch sind sie nicht Teil der Kernfunktionalität, sondern verwenden diese lediglich. Daher werden die graphischen Oberflächen nach der Fertigstellung der Kernfunktionalität entwickelt und in das Programm integriert. Durch dieses Vorgehen bei der Prioritätensetzung wird sichergestellt, dass mindestens der Kern des Programms, also der technisch schwierige Bereich der Software, innerhalb der zur Verfügung stehenden Zeit fertig gestellt wird.

Die Abhängigkeiten zwischen Projektaufgaben ergeben sich aus dem logischen Zusammenhang von Projektaufgaben. Die meisten Aufgaben benötigen die Ergebnisse vorheriger Aufgaben, damit sie entsprechend bearbeitet werden können. Beispielsweise werden für die Festlegung eines formalen Aufbaus einer Testdatei [Aufgabe 02] die enthalten Informationen [Aufgabe 01] benötigt. Analog verhält es sich mit dem formalen Aufbau des Testprotokolls [Aufgabe 06] sowie der Realisierung von Bibliotheken [Aufgabe 11]. Das bei der Testdurchführung notwendige Einlesen von Testdateien [Aufgabe 07] ist ebenfalls vom formalen Aufbau der Datei abhängig. Die meisten Abhängigkeiten entstehen beim Ablauf der Testdurchführung [Aufgabe 08], denn dabei werden Testdateien eingelesen, Schlüsselwörter ausgeführt und Protokolle erstellt. Die Oberfläche für die Testerstellung [Aufgabe 04] ist abhängig vom formalen Aufbau der Testdateien und der Verwendung von Schlüsselwörtern. Schlüsselwörter werden zwar nicht ausgeführt, jedoch werden die Informationen dieser dem Ersteller zur Verfügung gestellt. Des Weiteren ist die graphische Oberfläche der Testdurchführung [Aufgabe 09] abhängig vom Durchführungsablauf. Die zuvor erläuterten Prioritäten und Abhängigkeiten sind in Tabelle 4.1 zusammengefasst.

Basierend auf den Prioritäten und Abhängigkeiten der Anforderungen wird ein Zeitplan für die Softwareentwicklung aufgestellt. Zunächst wird eine Bearbeitungsreihenfolge der Aufgaben ermittelt, wobei die Prioritäten und Abhängigkeiten zu an-

Aufgabe	Priorität	abhängig von
01	1	-
02	1	01
03	2	11
04	3	02, 12
05	1	-
06	1	05
07	1	02
08	1	06, 07, 12
09	3	08
10	1	-
11	1	10
12	1	11

Tabelle 4.1.: *Die Tabelle zeigt die Prioritäten und Abhängigkeiten der Projektaufgaben aus Abb. 4.3. Dabei steht 1 für die höchste und 3 für die geringste Priorität.*

deren Aufgaben berücksichtigt werden. Tabelle 4.2 zeigt den Zeitplan des Projekts. Zu Beginn wird der inhaltliche und formale Aufbau von Testdateien und Protokollen festgelegt. Nachdem der Aufbau von Testdateien feststeht wird das Einlesen dieser implementiert. Anschließend wird der Inhalt und die technische Realisierung von Bibliotheken erarbeitet. Auf Basis dieser Ergebnisse wird die konkrete Verwendung von Schlüsselwörtern umgesetzt. Mit dem Abschluss der vorhergehenden Aufgaben gibt es keine weitere Abhängigkeiten für die Hauptaufgabe des Testwerkzeugs, nämlich dem Ablauf der Testdurchführung. Demnach ist das Testwerkzeug nach 11 Tagen in einem grundsätzlich ausführbarem Zustand. Anschließend werden die für Tests benötigten Standardbibliotheken entwickelt und die graphischen Benutzeroberflächen dem Programm hinzugefügt. Insgesamt beträgt die Entwicklungszeit des Projekts somit 18 Arbeitstage.

Tag 1	Tag 2	Tag 3	Tag 4	Tag 5
01 02 05 06	07	07	07 10 11	12
Tag 6	Tag 7	Tag 8	Tag 9	Tag 10
12	12	08	08	08
Tag 11	Tag 12	Tag 13	Tag 14	Tag 15
08	03	04	04	04
Tag 16	Tag 17	Tag 18	Tag 19	Tag 20
09	09	09		

Tabelle 4.2.: *In der Tabelle ist der Zeitplan des Softwareprojekts abgebildet. Pro Tag wird mit insgesamt 8 Stunden Zeit gerechnet. Die Eingetragen Zahlen entsprechen den der Projektaufgaben aus Abb. 4.3*

4.4. Konkretisierung und Implementierung

Das in Abschnitt 4.2 entwickelte Konzept beschreibt den Lösungsansatz der Problemstellung vorerst nur oberflächlich, weshalb in diesem Abschnitt die Bestandteile des Konzept detaillierter betrachtet werden.

4.4.1. Erstellung von Testdateien

Tests werden in Dateien gespeichert, um bei Bedarf automatisiert durchgeführt zu werden. Die sogenannten Testdateien werden vom Testersteller erstellt und enthalten alle für die Durchführung wichtigen Informationen. Zu jedem Test wird ein Name und eine Beschreibung hinterlegt, sodass der Test nachvollziehbar ist. Zusätzlich wird der Ersteller der Datei gespeichert, damit dieser bei Unklarheiten

oder Fehlern zu Rate gezogen werden kann. Des Weiteren wird der Testablauf angegeben, in dem zur Aufbau-, Test- und Abbauphase Testschritte in der Testdatei definiert werden. In einer Testdatei müssen die Dateipfade zu den verwendeten Bibliotheken angegeben werden, damit das Testwerkzeug die Schlüsselwörter jener Bibliotheken zuordnen kann. Bei der Verwendung von mehreren Bibliotheken kann es passieren, dass es gleichnamige Schlüsselwörter gibt. Daher wird zu jeder Bibliothek ein Kurzbezeichner hinzugefügt. Dieser wird vom Testersteller bei gleichnamigen Schlüsselwörter vorangestellt, sodass das Testwerkzeug diese eindeutig einer Bibliothek zuordnen kann.

In einem Testablauf können ebenfalls Variablen definiert werden, welche einen absoluten Wert oder den Rückgabewert eines Schlüsselworts enthalten. Durch die Verwendung von Variablen wird Redundanz vermieden, denn mehrmals verwendete Werte werden einmal definiert und anschließend über den Variablenbezeichner referenziert. Außerdem können Werte die erst zur Testlaufzeit verfügbar sind in Variablen gespeichert werden. Eine Wertzuweisung einer Variablen besteht aus einem Variablenbezeichner und einem Variablenwert. In einem Test definierte Variablen bleiben für den gesamten Test, also für alle drei Testphasen, erhalten. Neben diesen lokalen Variablen gibt es globale Variablen. Diese werden in dafür vorgesehenen Dateien definiert und vom Programm bei der Testdurchführung eingelesen. Globale Variablen sind für die stapelweise Ausführung von Tests von Bedeutung, denn dadurch können Werte flexibel gehalten werden. Eine Wertänderung eines in mehreren Testdateien verwendeten Parameters findet daher nur an einer einzigen Stelle, nämlich in der Variablendatei, statt. Im Gegensatz zu lokalen Variablen werden globale ausschließlich absoluten Werten zugewiesen. Rückgabewerte von Schlüsselwörtern können sich abhängig vom Zeitpunkt der Testausführung unterscheiden, weswegen diese Werte auf globaler Ebene nicht verlässlich sind. Des Weiteren müssen in einem Test unter Umständen Berechnungen durchgeführt werden. Absolute oder auch zur Laufzeit ermittelte Werte müssen verrechnet werden, bevor sie beispielsweise als Übergabeparameter verwendet werden können. Deswegen ist es möglich mathematische Ausdrücke in Testdateien zu formulieren. Diese enthalten sowohl absolute Werte oder auch Variablen und werden zum Zeitpunkt der Testdurchführung ausgewertet.

Das Ergebnis eines Test wird durch den Vergleich eines beobachteten und erwartetem Verhalten bestimmt. Dem Testersteller muss bewusst sein, dass das beobachtete

Verhalten lediglich eine Momentaufnahme des Geräts ist. Vor allem Messgeräte reagieren sehr sensibel auf äußere Einflüsse, weswegen eine stichprobenartige Überprüfung unter Umständen keine besonders hohe Aussagekraft besitzt. Bei Tests für die dies zutrifft wird eine Wiederholungsanzahl des Test angegeben. Das Testwerkzeug führt einen Test entsprechend jener Zahl mehrmals hintereinander aus. Dieser wird nur dann als Bestanden gekennzeichnet wenn jede Ausführung fehlerfrei abschließt. Dadurch kann nach wie vor nicht auf Fehlerfreiheit geschlossen werden, jedoch ist das Testergebnis durch den höheren Stichprobenumfang aussagekräftiger.

Zusammengefasst sind in einer Testdatei ein Autor, ein Testname und eine Beschreibung zu finden. Des Weiteren werden die verwendeten Bibliotheken zusammen mit einem Kurzbezeichner benannt und Dateien mit globalen Variablen festgelegt. Außerdem werden die einzelnen Schritte des Testablauf in Form von Schlüsselwörtern oder Wertzuweisungen definiert. Zusätzlich kann eine Wiederholungsanzahl des Testablaufs festgelegt werden.

Format von Testdateien

Die zuvor genannten Inhalte einer Testdatei müssen in ein für den Menschen und Computer lesbares Format gebracht werden. Oftmals verwendete Formate für Eingabedateien sind beispielsweise CSV[1] oder XML[2]. Das erste der beiden Formate stellt Daten auf einfache Weise dar, weshalb es sich für einfache Informationen gut eignet. Die Werte stehen nacheinander und werden durch ein festgelegtes Trennzeichen unterschieden. Das Format ist wegen seiner trivialen Darstellungsweise jedoch nur für ein Programm leicht lesbar da die Aneinanderreihung von Werten für den Menschen bereits bei geringen Mengen unlesbar wird. Anders verhält es sich bei XML, denn mit diesem Format können selbst komplexe Informationen in einem für Mensch und Maschine lesbaren Format gespeichert werden. Dazu ist jedoch eine aufwändige Syntax notwendig, welche zusätzlich zu den Informationen weiteren Text erfordert. Durch den hohen Schreibaufwand würde die Testerstellung sowohl zeitintensiv, als auch anfällig für Tippfehler werden. Da Testdateien händisch erstellt werden, muss die Lesbarkeit gewährleistet sein und der Schreibaufwand gering

[1] *Comma-seperated values* oder *Character-seperated vaues*
[2] *Extensible Markup Language*

bleiben. Die hier vorgestellten Formate eignen sich nicht, da sie entweder für Menschen schlecht lesbar oder zu aufwändig sind. Aus diesen Gründen wird ein einfaches und zugleich lesbares Format für Testdateien entwickelt.

Die in einer Testdatei enthaltene Information müssen sowohl vom Testersteller, als auch vom Testwerkzeug in einen Kontext eingeordnet werden können. Dazu werden, ähnlich wie bei XML, sogenannte *Tags* verwendet. Diese stehen in eckigen Klammern und werden einer Information vorangestellt. Wenn eine Information vom Menschen oder Computer gelesen wird, dann erschließt sich der Kontext mit Hilfe des vorangestellten *Tags*. Der Name des *Tags*, also der in Klammern stehende Text, ist für den Menschen verständlich, jedoch muss er im Quellcode des Programms definiert werden, damit der Computer die Information einordnen kann. Nachfolgend wird anhand einer Beispieldatei in Listing 4.1 das Format erläutert.

In der ersten Zeile ist der Autor angegeben, wobei dessen Name ein AUTHOR-*Tag* vorangestellt wird. Die nächste Zeile enthält den Testnamen, welcher durch den TESTNAME-*Tag* identifiziert wird. Anschließend wird in der dritten Zeile die Beschreibung des Tests angegeben. Hierbei ist der Name des *Tag* eine Abkürzung für das englische Wort *Description*. In Zeile 5 wird ein Bibliothek referenziert, damit die in ihr enthaltenen Schlüsselwörter im Testablauf verwendet werden können. Dies wird durch einen LIB-*Tag* gekennzeichnet, welcher für das englische Wort *Library* oder zu deutsch *Bibliothek* steht. Hinter diesem wird der Dateipfad zur Bibliothek angegeben und zusätzlich der Kurzbezeichner *G1*. Dieser Kurzbezeichner kann, wie in Zeile 17 zu sehen, Schlüsselwörtern durch einen Punkt vorangestellt werden, um dieses eindeutig zuzuordnen. Dateien mit globalen Variablen werden mit einem VAR-*Tag* referenziert. Hinter diesem wird, wie in der sechsten Zeile, der Dateipfad angegeben. Eine Beispieldatei mit globalen Variablen ist in Kapitel B zu sehen. Die siebte Zeile enthält die Wiederholungszahl des Tests, welcher der REPEAT-*Tag* vorangestellt wird. Ab Zeile neun beginnt der Testablauf. Dieser ist in die Aufbau-, Test- und Abbauphase unterteilt, welche durch die *Tags* SETUP, TEST und TEARDOWN eingeleitet werden. Da zu jeder Phase mehrere Testschritte angegeben werden können, erwarten diese *Tags* die Testschritte nicht direkt dahinter, sondern in den nachfolgende Zeilen. Die Testschritte werden solange der entsprechenden Testphase zugeordnet, bis ein neuer *Tag* eingelesen wird. Wie in Zeile 16 zu sehen, können in einer Testdatei Kommentare eingefügt werden. Diese beginnen mit einem Rautezeichen und verlaufen über eine gesamte Zeile.

```
1  [AUTHOR]     Max Muster
2  [TESTNAME]   Wertänderung
3  [DESC]       Überprüft, ob Wertänderungen am Gerät
                    ↪funktionieren
4
5  [LIB]        "DeviceKeywordLibrary.jar" G1
6  [VAR]        "config.var"
7  [REPEAT]     1
8
9  [SETUP]
10     Baue Verbindung auf
11 [TEST]
12     Setze Wert auf       "20"
13     {val} =              Lese Wert
14     Ist gleich           {val}, "20"
15 [TEARDOWN]
16     # 'G1.' ordnet das Schlüsselwort der Bibliothek
                    ↪eindeutig zu
17     G1.Schließe Verbindung
```

Listing 4.1: *Das Listing zeigt eine Testdatei. Roter Text kennzeichnet* Tags*, blaue Schrift zeigt Werte oder Variablen und grüne zeigt Kommentare an.*

Testschritte

Testschritte werden in Testdateien im Kontext eines SETUP-, TEST- oder TEARDOWN-*Tag* angegeben. Dem jeweiligen *Tag* entsprechend werden sie vom Testwerkzeug einer Testphase zugeordnet. Ein Testschritt ist entweder ein Schlüsselwort oder eine Wertzuweisung. Da die *Tags* der Testphasen gleichartig funktionieren wird im folgenden lediglich die Testphase betrachtet. Zeile 12 zeigt die erste Art eines Testschritts, nämlich das Schlüsselwort *Setze Wert auf*. Aus dem Namen des Schlüsselworts geht hervor, dass durch ausführen des Schlüsselworts ein Wert gesetzt wird. Jener Wert wird hinter dem Schlüsselwort angegeben und der Programmlogik des Schlüsselworts beim Aufruf übergeben. Damit das Testwerkzeug einen Wert identifizieren kann, muss dieser grundsätzlich in doppelten Anführungszeichen stehen. In der Beispieldatei wird demnach der Parameter *20* übergeben. In der 13. Zeile steht eine Wertzuweisung, was anhand des Gleichheitszeichen zu erkennen ist. Auf der linken Seite des Zeichens steht ein Variablenbezeichner und auf der rechten der zuzuweisende Wert. Analog zu Werten müssen Variablenbezeichner in geschwungenen Klammern stehen. Gültige Werte für eine Wertzuweisungen sind reguläre

Werte, Variablen und Schlüsselwörter. Bei der Zuweisung eines Schlüsselworts wird nicht das Schlüsselwort gespeichert, sondern der Rückgabewert der ausgeführten Programmlogik. In Zeile 13 wird somit dem Bezeichner *val* der Rückgabewert des Schlüsselworts *Lese Wert* zugewiesen. Zeile 14 enthält erneut ein Schlüsselwort, welches den Namen *Ist gleich* trägt. Diesmal werden mehrere Parameter angegeben, welche durch ein Komma voneinander getrennt sind. Der Name des Schlüsselworts lässt erahnen, dass die angegeben Parameter miteinander verglichen werden. Hier wird somit überprüft, ob der in Zeile 13 gelesene Wert dem Wert *20* entspricht. Schlüsselwörter können Parameter in Form von Werten oder Variablen übergeben werden. Im Gegensatz zu einer Wertzuweisung ist es nicht möglich ein Schlüsselwort als Parameter zu verwenden. Theoretisch ist dies möglich, jedoch würde es zu schwer lesbaren Textkonstrukten führen. Damit Testdateien leicht lesbar und somit nachvollziehbar bleiben werden die Parameter für Schlüsselwörter auf Werte und Variablen beschränkt.

Zusätzlich zu den bisher genannten Parametern können sogenannte mathematische Ausdrücke formuliert werden. Unter Umständen ist es notwendig, dass numerische Werte berechnet werden müssen, weshalb der Testersteller mathematische Ausdrücke definieren muss. Diese werden vom Programm bei der Testdurchführung evaluiert und der dabei berechnete Wert als Parameter verwendet. Die mathematischen Ausdrücke werden ebenfalls in doppelten Anführungszeichen angegeben, jedoch werden sie zusätzlich von eckigen Klammern umschlossen. Dadurch werden sie von regulären Werten unterschieden und können mathematisch interpretiert werden. Da in der Beispieldatei kein solcher Ausdruck steht wird nachfolgend ein Beispiel gegeben:

$$\text{"[1.5*\{x\}\^{} 2 + 10]"} \quad \widehat{=} \quad 1.5 * x^2 + 10$$

Auf der linken Seite steht ein mathematischer Ausdruck, wie er als Parameter in einem Testschritt verwendet werden kann. In diesem steht der Variablenbezeichner x, welcher für die Berechnung zunächst durch den gespeicherten Wert ersetzt wird. Dies setzt voraus, dass dem Variablenbezeichner vorab ein Wert zugewiesen wurde. Anschließend wird der Ausdruck interpretiert und wie der Ausdruck auf der rechten Seite berechnet. In diesem wäre jedoch die Variable x durch ihren tatsächlichen Wert ersetzt.

4.4.2. Schlüsselwörter erstellen und bereitstellen

Schlüsselwörter werden bei der Erstellung von Testdateien eingesetzt, um den Testablauf zu beschreiben. Sie bestehen aus einem knappen Satz oder einem einzelnen Wort und sind sprechende Namen. Das bedeutet, dass durch lesen des Schlüsselworts die Funktionalität erklärt wird. Die technische Implementierung jener Funktionalität bleibt dem Testersteller jedoch verborgen, weswegen dieser keine Programmiererfahrung benötigt. Schlüsselwörter werden in sogenannten Bibliotheken implementiert und dem Testwerkzeug zur Verfügung gestellt. Eine Bibliothek enthält beliebig viele Schlüsselwörter und zu jedem Schlüsselwort die zugehörige Programmlogik. Obwohl Schlüsselwörter aussagekräftige Namen haben ist es sinnvoll, dass weitere Informationen bereitgestellt werden. Für den Testersteller sind eine Beschreibung des Schlüsselworts, sowie Hinweise zu den Übergabeparametern und dem Rückgabewert wichtig. Durch diese Informationen wird sichergestellt, dass der Testersteller ein Schlüsselwort geeignet einsetzt und die Übergabeparameter sowie den Rückgabewert richtig interpretiert. Außerdem werden Bibliotheken mit einer Beschreibung und dem Autoren versehen. Durch die Beschreibung wird dem Testersteller der Sinn und Zweck der Bibliothek dargelegt, sodass diese im richtigen Kontext verwendet wird. Bei Fehlern in der Bibliothek oder Unklarheiten kann der Autor der Bibliothek angesprochen werden.

Bibliotheken sind eigenständige Programmeinheiten und somit unabhängig vom Testwerkzeug funktionsfähig. Sie sind nicht Bestandteil des Testwerkzeugs, sondern werden nachträglich in dieses integriert. Wenn Bibliotheken fest in das Programm integriert wären, müsste das Testwerkzeug für jede neue oder geänderte Bibliothek kompiliert und beim Nutzer installiert werden. Der entstehende Aufwand wäre hoch und würde in zahlreichen Softwareaktualisierungen beim Nutzer enden. Deswegen werden Bibliotheken lose mit dem Testwerkzeug gekoppelt, in dem sie in Dateien gespeichert werden. Diese können zur Programmlaufzeit hinzugefügt, beziehungsweise ausgetauscht werden, was als sogenanntes *Hot Deployment* bezeichnet wird[Ull11, S. 580 ff.]. Dadurch ist die Integration von neuen oder geänderten Bibliotheken wesentlich flexibler und jederzeit möglich. Die konkrete Realisierung dieser Herangehensweise wird zu einem späteren Zeitpunkt in diesem Abschnitt vorgestellt.

Bei der technischen Umsetzung von Schlüsselwortbibliotheken ist die Abbildung von Schlüsselwörtern zu einer Programmlogik sowie das Lesen der allgemeinen Informationen zu berücksichtigen. Die Programmlogik von Schlüsselwörtern ist ausführbarer Programmcode, welcher demnach in Methoden implementiert wird, denn in der Programmiersprache Java existieren Methoden ausschließlich innerhalb von Klassen. Deswegen eignen sich Klassen als geeignete Repräsentanten für Schlüsselwortbibliotheken. In einer Klasse werden beliebig viele Methoden implementiert, welche die Programmlogik von Schlüsselwörtern beinhalten. Die Informationen zu Schlüsselwörtern werden über sogenannte *Annotations* bereitgestellt, welche es ermöglichen Meta-Daten zur Programmlaufzeit auszulesen. Methoden, die Programmlogik eines Schlüsselworts enthalten werden mit einer *Annotation* versehen. In dieser wird eine Beschreibung, sowie Informationen zu Übergabeparametern und dem Rückgabewert, als auch der Name des Schlüsselworts gespeichert. Dadurch können die Informationen zu Schlüsselwörtern vom Programm ermittelt werden. Methoden die mit der *Annotation* für ein Schlüsselwort versehen sind werden im folgenden auch Schlüsselwort-Methoden genannt. Analog dazu wird die Klasse von Schlüsselwort-Methoden ebenfalls mit einer *Annotation* versehen, welche eine Beschreibung und einen Namens des Autors enthält. Eine solche Klasse wird in dieser Arbeit auch Bibliotheks-Klasse genannt.

Die *Annotations* für Klassen und Methoden sind nicht nur Informationsträger, sondern sie kennzeichnen eine Klasse als Bibliothek, beziehungsweise eine Methode als Programmlogik eines Schlüsselworts. Das Testwerkzeug kann dadurch Bibliotheks-Klassen und Schlüsselwort-Methoden eindeutig identifizieren. Schlüsselwort-Methoden können somit von regulären Methoden unterschieden werden. Dies hat den Vorteil, dass mehrmals verwendete Programmlogik in Methoden ausgelagert werden kann, ohne dass diese als Schlüsselwort verfügbar sein müssen. Ein weiterer Vorteil ist, dass bereits in Java implementierte Schnittstellen nachträglich in eine Bibliothek umgewandelt werden können. Dazu werden die Klasse und die Methoden mit den entsprechenden *Annotations* versehen. Genauso können Bibliotheks-Klassen und enthaltene Methoden in anderen Projekten außerhalb des Testwerkzeugs verwendet werden.

In Listing 4.2 wird exemplarisch ein Ausschnitt der Bibliothek gezeigt, welche bereits in der Testdatei in Listing 4.1 auf Seite 21 eingebunden wurde. Zu Beginn des Listings ist die Klassendeklaration der Schlüsselwortbibliothek zu sehen. Die

```
1 @KeywordLibrary(Author = "...", Description = "...")
2 public class DeviceKeywordLibrary {
3
4     @Keyword(Description = "...", Name = "Setze Wert auf",
                  ↪Parameter = "...", Return = "...")
5     public void setValue(int value) { ... }
6
7 }
```

Listing 4.2: *Klassendeklaration einer Schlüsselwortbibliothek*

Annotation KeywordLibrary[3] in der ersten Zeile wird vor die Klasse gestellt, um den Autor und die Beschreibung anzugeben. Anschließend ist in Zeile vier die Definition des Schlüsselworts *Setze Wert auf* zu sehen. Die Methode *setValue* ist mit der *Annotation Keyword*[4] versehen, in welcher die Beschreibung und der Name des Schlüsselworts, sowie Informationen zu Übergabeparametern und dem Rückgabewert stehen. In beiden *Annotations* sind die Informationen aus Gründen der Übersichtlichkeit lediglich durch drei Punkte angedeutet.

Bibliotheken werden wie zuvor erläutert in Java Klassen implementiert. Bei der Programmierung werden unter Umständen weitere Klassen von der Bibliotheks-Klasse benutzt, weshalb eine Bibliothek nicht als einzelne Datei, sondern als Java-Archiv gespeichert wird. Diese spezielle Art von ZIP-Archiven ist ein für Java Applikationen typisches Format zum exportieren von Programmcode. Dadurch ist die Klasse der Bibliothek mit allen notwendigen Klassen in einem Archiv gebündelt. Das Testwerkzeug kann Java-Archive laden, wodurch die Schlüsselwörter der enthaltenen Bibliotheks-Klasse bei der Testerstellung eingesetzt werden können. In Java werden Klassen durch sogenannte Pakete strukturiert und gekapselt, wobei Pakete im Prinzip reguläre Ordner sind. Damit das Testwerkzeug eine Bibliotheks-Klasse in den Paketen findet wird eine Namenskonvention für die Archiv-Dateien eingeführt. Der Dateiname muss dem sogenannten vollständigen Namen der Bibliotheks-Klasse entsprechen. Das bedeutet, dass die Pakete der Klasse, durch Punkte getrennt vor dem Klassennamen stehen. Sofern die Klasse in keinem Paket liegt genügt der Klassenname. Der Archivname *com.example.Bibliothek.jar* würde demnach zu einer Klasse namens *Bibliothek* führen, welche im Paket *example* ist. Dieses Paket ist wiederum im Paket namens *com* enthalten. Bibliotheken werden in einer Testda-

[3] Englischer Ausdruck für Schlüsselwortbibliothek
[4] Englischer Ausdruck für Schlüsselwort

tei über einen Dateipfad referenziert, wie es in Listing 4.1 in Zeile 5 zu sehen ist. Allgemein betrachtet werden Dateipfade entweder absolut oder relativ angegeben. Die erste Art der Pfadangabe beginnt auf unterster Ebene, also beim Laufwerkbuchstaben, wohingegen relative Pfade von einer festgelegten Position aus starten. Da die zweite Art der Pfadangabe deutlich flexibler ist werden Bibliotheken mittels relativen Pfaden referenziert. Dies erfordert, dass die Startposition der Pfade im Testwerkzeug definiert sein muss. Somit wird ein Verzeichnis als Startposition festgelegt, in welchem alle entwickelten Bibliotheken abgelegt werden. Die Pfadangabe in einer Testdatei ist somit relativ zu diesem Verzeichnis. Zusätzlich zu dieser expliziten Referenzierung können Bibliotheken implizit referenziert werden. Dafür wird im Verzeichnis für Bibliotheken ein Ordner mit dem Namen *std* erstellt, was die Abkürzung für *Standard* ist. Die in diesem Ordner abgelegten Bibliotheken können in jeder Testdatei verwendet werden. Das Testwerkzeug lädt somit automatisch jede Bibliothek aus diesem Verzeichnis, ohne dass sie in der Testdatei explizit referenziert ist. Deswegen ist es ratsam ausschließlich häufig benötigte Bibliotheken dort abzulegen.

Zusammengefasst lässt sich sagen, dass Schlüsselwörter mit sogenannten Bibliotheken in das Testwerkzeug integriert werden. Bibliotheken sind Java-Klassen, dessen Methoden die Logik zu Schlüsselwörtern bereitstellen. Die Identifikation von Bibliotheks-Klassen und Schlüsselwort-Methoden erfolgt durch *Annotations*. Da Bibliotheks-Klassen unter Umständen weitere Klassen verwenden werden Bibliotheken in Java-Archiven gespeichert. Diese Archive werden gemäß einer Namenskonvention benannt und in einem festgelegten Verzeichnis abgelegt. Bibliotheken werden entweder in einer Testdatei referenziert oder in einem speziellen Ordner abgelegt, sodass sie automatisch referenziert werden.

4.4.3. Durchführung von Tests

Die Testdurchführung erfolgt auf Anweisung des Benutzers der Software. Dazu werden die auszuführenden Testdateien angegeben, welche vom Programm eingelesen und ausgeführt werden. Zu jeder Testdatei wird ein Ergebnis gespeichert und nach Abschluss des Testlaufs in einem Protokoll vermerkt. Im folgenden wird der Prozess der Testdurchführung erörtert, wobei zu Beginn das Einlesen von Testdateien betrachtet wird. Daraufhin wird erläutert wie Bibliotheken in das Programm geladen

werden und im Anschluss daran wird die konkrete Ausführung von Schlüsselwörtern betrachtet. Abschließend wird die Protokollierung von Tests erläutert.

Einlesen von Testdateien

Die Testdateien müssen dem in Abschnitt 4.4.1 festgelegten Format entsprechen, damit sie vom Testwerkzeug verwendet werden können. Bevor der Inhalt einer Eingabedatei verwendet wird muss das Programm diesen auf lexikalischer und syntaktischer Ebene analysieren. Dazu wird sich an der lexikalischen und syntaktischen Analyse von Compilern orientiert, welche für Programmiersprachen benötigt werden um den Quellcode in ausführbaren Maschinencode zu übersetzen. Im Kontext des Testwerkzeugs wird aus den Eingabedateien zwar kein Maschinencode erzeugt, jedoch müssen die enthalten Anweisungen in entsprechende Befehle übersetzt werden. Nachfolgend wird die lexikalische und syntaktische Analyse erläutert. Eine detaillierte Betrachtung der Analysen übersteigt den Rahmen dieser Arbeit, kann jedoch in [CIAD12][5] nachgeschlagen werden. Anschließend wird die Verarbeitung und Speicherung der Inhalte von Eingabedateien betrachtet.

Bei der lexikalischen Analyse wird der Inhalt einer Eingabedatei in sogenannte *Tokens* zerlegt. *Tokens* sind kurze zusammenhängende Zeichenfolgen, die nicht weiter zu unterteilen sind. In einer Testdatei gibt es verschiedene Arten von *Tokens*, wie beispielsweise *Tags*, Werte oder Variablen. Wenn eine Zeichenfolge keinem *Token* zugeordnet werden kann wird die weitere Verarbeitung der Eingabedatei abgebrochen. Dies ist zum Beispiel der Fall wenn der Name eines *Tags*, also die in eckigen Klammern eingeschlossene Buchstabenfolge, nicht im Testwerkzeug definiert ist. Durch die fehlende Definition des *Tags* kann das Programm die zugehörigen Informationen nicht interpretieren, weswegen ein Fehler ausgelöst wird. Zusätzlich zur lexikalischen Analyse findet eine syntaktische Analyse statt. Diese Überprüft ob die Grammatik der Eingabedatei korrekt ist. Konkret bedeutet dies, dass die Anordnung von *Tokens* überprüft wird. Wenn zum Beispiel zwei Schlüsselwörter hintereinander stehen wird das hintere als Übergabeparameter interpretiert. Da dies aus Gründen der Lesbarkeit nicht erlaubt ist wird aufgrund der ungültigen Anordnung ein Fehler ausgelöst. Sofern die überprüfte Datei in Ordnung ist kann das Testwerkzeug die enthaltenen Informationen ermitteln.

[5] In englischer Literatur werden die Begriffe *Screener* und *Parser* für die lexikalische und syntaktische Analyse verwendet

Bei der Ermittlung von Informationen aus einer Testdatei haben *Tags* eine wichtige Rolle, da diese die Zuordnung der Information ermöglichen. Das Programm liest einen *Tag* und speichert die dazugehörige Informationen entsprechend ab. Dadurch ist der Testersteller bei der Testerstellung nicht an eine vorgegebene Reihenfolge der Informationen gebunden, sondern kann nach belieben die Inhalte der Testdatei aufbauen. Das Testwerkzeug erwartet keine Reihenfolge, jedoch müssen bestimmte Inhalte in einer Testdatei vorhanden sein. Mindestens festgelegt werden müssen der Autor, der Testname und der Ablauf der Testphase. Eine Beschreibung des Testfalls ist für die Nachvollziehbarkeit wichtig, jedoch liegt die Entscheidung der Notwendigkeit einer Beschreibung beim Testersteller. Die verwendeten Bibliotheken und Dateien mit globalen Variablen sind nur erforderlich wenn diese im Testablauf verwendet werden. Eine Aufbau- und Abbauphase ist nur notwendig wenn das *System Under Test* zu Beginn des Tests in einen anderen Zustand gebracht und anschließend in den Standardzustand versetzt werden muss. Standardmäßig wird ein Test einmal ausgeführt, weswegen eine Angabe der Wiederholungszahl eines Test nur bei Bedarf von mehrmaligen Ausführungen angegeben werden muss.

Die Informationen einer Testdatei werden in einem Objekt der Klasse *Testfile*, siehe Abb. 4.4, gespeichert. Die Klasse speichert den Autor, den Testnamen, die Beschreibung, die Wiederholungszahl, die Pfade zu referenzierten Bibliotheken und Dateien mit globalen Variablen, sowie die Testschritte der drei Testphasen. Die Testschritte werden in Objekten der Klasse *Testline* gespeichert. Diese Klasse enthält den Text und die Zeilennummer eines Testschritts. Dadurch kann zu einem Fehler bei der Ausführung eines Testschritts die Zeilennummer in der Fehlernachricht angegeben werden. Alle Informationen können über entsprechende *Getter*-Methoden abgefragt werden, jedoch besitzt die Klasse keine öffentlichen Methoden zum verändern der Informationen und keinen öffentlichen Konstruktor. Um ein *Testfile*-Objekt zu erzeugen wird die Klasse in Abb. 4.5 verwendet. Diese befindet sich im selben Paket und kann daher auf Methoden und Konstruktoren, die nur im Paket sichtbar sind, zugreifen. Die statische Methode *read* der Klasse *TestfileReader* erzeugt eine Instanz der Klasse *Testfile*. Dazu wird der Pfad einer Testdatei der Methode übergeben, welche diese daraufhin einliest und überprüft. Wenn die Datei auf lexikalischer und syntaktischer Ebene in Ordnung ist werden die Informationen gespeichert, ansonsten wird ein Fehler ausgegeben. Die Informationen werden anhand der festgelegten *Tags* ermittelt und entsprechend im *Testfile*-Objekt gespeichert. Somit ist die Klasse *Testfile* ausschließlich für die Datenhaltung zuständig, denn die Anwendungslogik

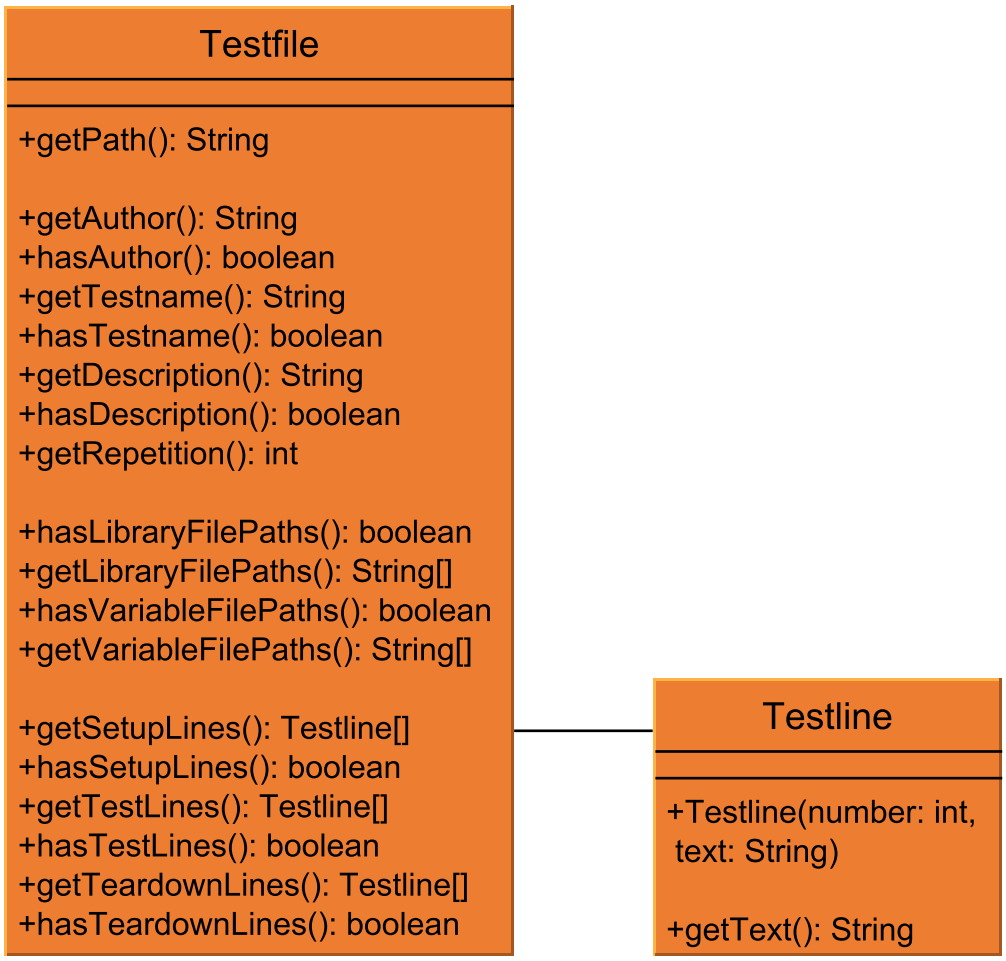

Abbildung 4.4.: *In diesem UML Klassendiagramm sind die Klassen* Testfile *und* Testline *zu sehen, wobei lediglich die öffentlichen Methoden der Klassen gezeigt werden.*

zum Einlesen einer Datei ist in die Klasse *TestfileReader* ausgelagert. Dadurch wird sowohl die Wartbarkeit der Software, als auch die Flexibilität gegenüber Änderungen erhöht.

Laden von Bibliotheken

Damit Schlüsselwörter einer Bibliothek verwendet werden können muss das Testwerkzeug das entsprechende Java-Archiv laden. Die bestehende Problematik ist, dass der Programmcode der im Archiv enthaltenen Klassen sich nicht in der sogenannten Laufzeitumgebung befindet, weshalb der Programmcode nicht verwendet werden kann. Bei Programmstart werden daher automatisch alle vom Programm

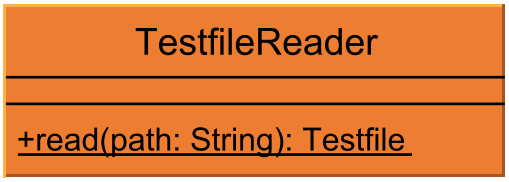

Abbildung 4.5.: *Dieses UML Klassendiagramm zeigt die Klasse* TestfileReader*, welche zum Einlesen von Testdateien verwendet wird. Des Weiteren speichert sie Informationen eingelesener Testdateien in einem Objekt der Klasse* Testfile*.*

benötigten Klassen in diese geladen. Da Bibliotheken jedoch erst zur Laufzeit verwendet werden müssen sie nachträglich der Laufzeitumgebung hinzugefügt werden, was in der Literatur als *Linking* bezeichnet wird S.575]JIAEI. Deswegen wird zunächst jede Klasse eines referenzierten Java-Archivs in die Laufzeitumgebung geladen. Anschließend ermittelt das Testwerkzeug die Bibliotheks-Klasse und die in ihr enthaltenen Informationen. Bei der Ermittlung der Klasse hilft die Namenskonvention von Archiv-Dateien, da durch diese der Pfad zur gesuchten Klasse angegeben wird. Zu diesem Zeitpunkt befindet sich das Programm nach wie vor zur Laufzeit, weswegen das sogenannte *Reflection*-Modell benötigt wird. *Reflection*, auch Introspektion genannt, ermöglicht es Klassen und Objekte in der Laufzeitumgebung zu analysieren, wodurch die enthaltenen Informationen ausgelesen werden können [Ull11, S. 1385]. Klassen, Methoden und *Annotations* werden durch *Reflection* in Form von Objekten repräsentiert. So werden Klassen in Objekten der Klasse *Class* gespeichert und Methoden in Objekten der Klasse *Method*. Diese Objekte stellen die Informationen durch Methoden bereit, wodurch die Information zur Laufzeit vom Programm abgefragt und verwendet werden können.

Mittels *Reflection* werden Bibliotheken und Schlüsselwörter in Objekte überführt. Abb. 4.6 zeigt die Klassen in der die jeweiligen Informationen gespeichert werden. Die Klasse *KeywordLibrary* sammelt die Informationen zu einer Schlüsselwortbibliothek. Dazu wird dem Konstruktor der Klasse ein *Class*-Objekt der Bibliotheks-Klasse, also der im Java-Archiv enthaltenen Klasse, übergeben. Anhand dieses Objekts wird die *Annotation* ermittelt, sodass der Autor und die Beschreibung der Bibliothek gespeichert werden können. Außerdem werden die Methoden der Klasse in Form von *Method*-Objekten abgefragt. Jede mit der *Annotation* für Schlüsselwörter versehene Methode wird in ein Objekt der Klasse *Keyword* überführt. Dem Konstruktor jener Klasse wird unter anderem ein *Method*-Objekt übergeben. Aus

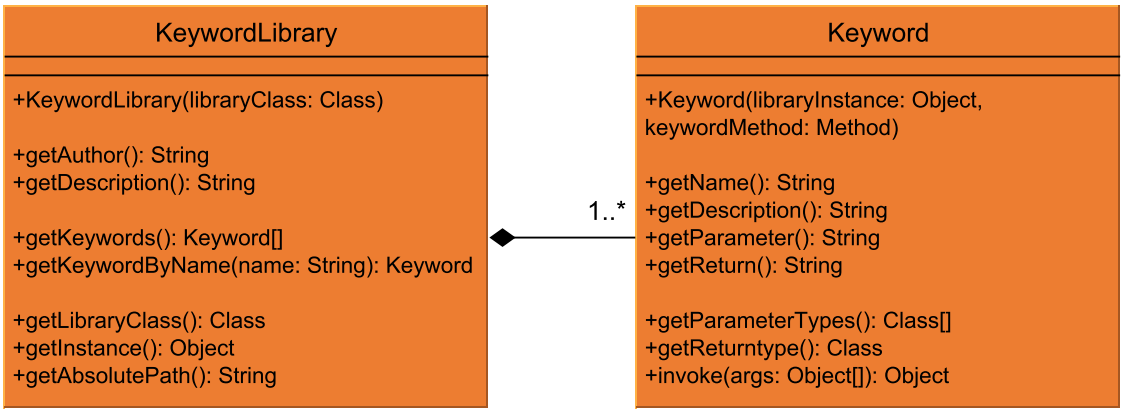

Abbildung 4.6.: *In diesem UML Klassendiagramm sind die Klassen* KeywordLibrary *und* Keyword *abgebildet. Die erste Klasse speichert Informationen zu Bibliotheken, die zweite zu Schlüsselwörtern.*

diesem Objekt können der Name des Schlüsselworts sowie die anderen in der *Annotation* enthaltenen Informationen gelesen werden. Außerdem können die von der Methode erwarteten Typen der Übergabeparameter und des Rückgabewerts gespeichert werden. Diese Informationen sind über die entsprechenden *Getter*-Methoden verfügbar. *Method*-Objekte besitzen die Methode *invoke*, welche aufgerufen wird, um die Programmlogik der repräsentierten Methode auszuführen. Diese Methode benötigt ein Objekt an dem die Programmlogik aufgerufen wird und eine Liste der Übergabeparameter. Das aufrufende Objekt muss eine Instanz der Klasse sein, in der diese Methode implementiert ist. Die Methode *invoke* der Klasse *Keyword* ist eine Stellvertreter-Methode, denn sie ruft intern eine gleichnamige Methode am *Method*-Objekt auf. Die Übergabeparameter werden an die intern aufgerufene *invoke*-Methode weitergereicht. Als aufrufendes Objekt der Programmlogik wird das Objekt *libraryInstance* aus dem Konstruktor der *Keyowrd*-Klasse verwendet. Demnach muss in der Klasse *KeywordLibrary* eine Instanz der im Java-Archiv enthaltenen Bibliotheks-Klasse erzeugt werden. Eine Referenz auf jene Instanz wird bei der Erzeugung der *Keyword*-Objekte, zusammen mit der jeweiligen Methode, dem Konstruktor übergeben. Die erzeugten *Keyword*-Objekte werden im erzeugenden *KeywordLibrary*-Objekt gespeichert, welches die Schlüsselwörter durch die Methoden *getKeywords* und *getKeywordByName* nach außen hin verfügbar macht. Diese Methoden werden vom Testwerkzeug für die Auflistung der Schlüsselwort-Dokumentation, beziehungsweise bei der Suche nach bestimmten Schlüsselwörtern verwendet.

Um ein Objekt der Klasse *KeywordLibrary* zu erzeugen, wird das *Class*-Objekt der im Archiv enthaltenen Bibliotheks-Klasse benötigt. Diese befindet sich jedoch nicht in der Laufzeitumgebung, sodass sie wie zuvor beschrieben hinzugefügt werden muss. Die dafür notwendige Logik wird in die Klasse *LibraryLoader*, siehe Abb. 4.7, ausgelagert. Die Klasse besitzt zwei öffentliche, statische Methoden namens *crea-*

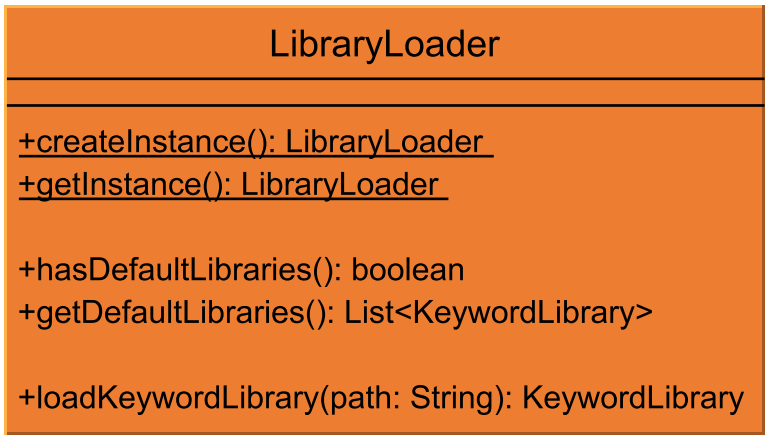

Abbildung 4.7.: *Die Abbildung zeigt ein UML Klassendiagramm der Klasse* LibraryLoader. *Jene Klasse wird zum Laden von Bibliotheken verwendet und realisiert das sogenannte* Hot Deployment.

teInstance und *getInstance*. Die erste Methode erzeugt eine neue Instanz der *LibraryLoader*-Klasse, die zweite gibt die zuletzt erzeugte Instanz zurück. Bei der Erzeugung der Klasse werden die Standardbibliotheken in die Laufzeitumgebung geladen. Jene liegen in dem Verzeichnis für Bibliotheken im Ordner *std*. Der Ordner wird vom Programm durchsucht und jedes Java-Archiv wird nacheinander in die Laufzeitumgebung geladen. Die geladenen Standardbibliothek sind anschließend über die Methode *getDefaultLibraries* verfügbar. In Testdateien werden Bibliotheken mit Hilfe des *LIB-Tags* referenziert. Um diese zu laden wird die Methode *loadKeywordLibrary* mit dem in der Datei angegebenen Pfad aufgerufen, welcher zu einem Java-Archiv führen muss. Die enthaltenen Klassen werden der Laufzeitumgebung hinzugefügt und mit der Bibliotheks-Klasse wird ein *KeywordLibrary*-Objekt erzeugt. Die Klasse realisiert somit das Prinzip des *Hot Deployment*, da der Programmcode der Java-Archive zur Laufzeit austauschbar ist. Somit können Bibliotheken ohne eine Aktualisierung des Programms hinzugefügt und ersetzt werden.

Automatisierte Ausführung von Testabläufen

Der in einer Testdatei enthaltene Testablauf besteht aus der Aufbau-, Test- und Abbauphase. Jede Testphase enthält beliebig viele Testschritte, welche entweder ein Schlüsselwort oder eine Wertzuweisung einer Variablen sind. Die beiden Arten von Testschritten müssen unterschiedlich verarbeitet werden. Anhand des Gleichheitszeichens unterscheidet das Testwerkzeug zwischen einer Wertzuweisung und einem Schlüsselwort. Demnach ist das Gleichheitszeichen ein besonderes Zeichen, welches ausschließlich für Wertzuweisungen verwendet werden kann.

Wertzuweisungen werden in den Variablenbezeichner und den zuzuweisenden Variablenwert aufgeteilt. Der Bezeichner wird zum Speichern und Laden des Variablenwerts verwendet, wobei dieser auf verschiedene Weise angegeben werden kann. Möglich sind absolute Werte, mathematische Ausdrücke, andere Variablen oder auch Schlüsselwörter. Absolute Werte können ohne Weiterverarbeitung gespeichert werden, dahingegen benötigen die übrigen Arten zusätzlichen Aufwand, um den konkreten Wert zu bestimmen. Mathematische Ausdrücke werden interpretiert und berechnet, damit das Ergebnis der Berechnung gespeichert werden kann. Wenn eine Variable einer anderen Variablen zugewiesen wird, dann findet eine Zuweisung des Wertes der zweiten Variable statt. Somit wird kein Alias der Variablen, sondern eine echte Kopie erzeugt. Bei einer Wertzuweisung mit einem Schlüsselwort als Parameter wird jenes zuerst ausgeführt und anschließend der Rückgabewert der Programmlogik als Variablenwert gesetzt. Das Testwerkzeug speichert den Bezeichner und den zugehörigen Wert in einem assoziativen Feld, wodurch der Variablenwert mit dem Variablenbezeichner abgefragt werden kann. Des Weiteren sind in einem Testlauf definierte Variablen für den restlichen Testlauf auf lokaler Ebene beständig. Demnach wird stets eine lokale Variable erzeugt, sodass Werte von globalen Variablen nicht überschrieben, sondern von möglicherweise gleichnamigen lokalen überdeckt werden. Daraus folgt, dass der Wert einer globalen Variable solange nicht referenzierbar ist, wie eine gleichnamige lokale Variable existiert.

Ein nicht als Wertzuweisung interpretierter Testschritt wird als Schlüsselwort betrachtet. In diesem Fall wird ein Testschritt in die Bestandteile Bibliotheksbezeichner, Name des Schlüsselworts und Liste der Übergabeparameter zerlegt. Der Bibliotheksbezeichner wird optional dem Namen voran gestellt, um das Schlüsselwort eindeutig einer Bibliothek zuzuordnen. Ohne einen Bezeichner wird das Schlüsselwort

in allen verfügbaren Bibliotheken, also den standardmäßig und den im Test geladenen, gesucht. Die verfügbaren Bibliotheken sind als Objekte der Klasse *KeywordLibrary* vorhanden. Die Suche nach einem Schlüsselwort erfolgt demnach mittels der Methode *getKeywordByName*, welche das Schlüsselwort mit dem entsprechenden Namen als Objekt der Klasse *Keyword* zurückgibt. Wenn das Schlüsselwort in keiner oder mehr als einer Bibliothek gefunden wird, so wird ein Fehler gemeldet, da keine eindeutige Zuordnung möglich ist. Sofern dem Schlüsselwort ein Bezeichner vorgestellt ist wird ausschließlich in der entsprechenden Bibliothek gesucht.

Das nach einer erfolgreichen Suche erhaltene *Keyword*-Objekt enthält die Informationen und die ausführbare Programmlogik. Zum Ausführen dieser müssen die Übergabeparameter in entsprechende Typen gewandelt werden, denn Werte, Variablen sowie mathematische Ausdrücke stehen als reiner Text in einem Testschritt. Variablen werden durch ihren gespeicherten Wert ersetzt und mathematische Ausdrücke evaluiert. Die Programmlogik benötigt festgelegte Typen von Übergabeparameter, welche mit der Methode *getParameterTypes* des *Keyword*-Objekts abgefragt werden. Das Testwerkzeug erkennt die primitiven Datentypen, also *boolean*, *char*, *byte*, *short*, *int*, *long*, *float* und *double* und als einzig nicht primitiven Datentyp *String*. Demnach können Übergabeparameter von Schlüsselwort-Methoden nur aus den genannten Typen gewählt werden. Die Übergabeparameter eines Testschritts werden in die von der Programmlogik benötigten Typen gewandelt. Wenn die Umwandlung von Text in den benötigten Datentyp nicht möglich ist wird ein Fehler ausgegeben. Dieser ist notwendig, da die Programmlogik nur mit gültigen Parametern ausgeführt werden kann.

Nachdem die Parameter erfolgreich in die entsprechenden Typen umgewandelt sind, kann das Schlüsselwort ausgeführt werden. Dazu wird die Methode *invoke* am *Keyword*-Objekt, dass bei der Suche ermittelt wurde, ausgeführt. Die Übergabeparameter des Schlüsselworts werden der Methode übergeben, welche diese intern der Programmlogik des Schlüsselworts weiterreicht. Das *Keyword*-Objekt speichert die Programmlogik als *Method*-Objekt und eine Instanz der Bibliotheks-Klasse als *Class*-Objekt. In der Methode *invoke* wird die gleichnamige Methode *invoke* des *Method*-Objekts mit der Instanz und den Parametern aufgerufen. In Folge des Methodenaufrufs wird der Rückgabewert der Programmlogik oder ein Fehler ausgegeben.

Die Fehlerbehandlung ist abhängig von der Testphase des Testschritts. Die Auf- und Abbauphase bringen das System in einen für den Test notwendigen Zustand und anschließend wieder in den Standardzustand zurück. Demnach kann nach einem Fehler in einer der beiden Phasen nicht garantiert werden, dass das *System Under Test* in einem testbaren Zustand ist. Deswegen wird der gesamte Testlauf abgebrochen, um Folgefehler zu vermeiden. Bei einem Fehler in der Testphase wird lediglich der einzelne Test abgebrochen. In diesem Fall wird der Fehler in einem Protokoll vermerkt und die Abbauphase ausgeführt. Dadurch wird das Testobjekt wieder in den Standardzustand versetzt und die nachfolgenden Tests können problemlos ausgeführt werden.

In Abb. 4.8 wird die Ausführung von Testschritten skizziert. Auf der linken Seite ist ein Ausschnitt des Testablaufs aus Listing 4.1 zu sehen. Auf der rechten Seite sind zwei Bibliotheken abgebildet. Die obere Bibliothek mit dem Namen *Device Bibliothek*, entspricht der in der Testdatei referenzierten. Sie enthält Schlüsselwörter, die die Schnittstellen zu einem externen Gerät implementieren und somit auch die Kommunikation zu diesem übernehmen. Die untere Bibliothek ist eine standardmäßig geladene Bibliothek und wird in der Testdatei nicht explizit referenziert. Diese enthält Schlüsselwörter für den Vergleich von Werten. Aus Gründen der Übersichtlichkeit werden in beiden Bibliotheken nur die für das Beispiel notwendigen Schlüsselwörter aufgeführt.

Der erste in der Abbildung aufgeführte Testschritt ist ein Schlüsselwort mit dem Übergabeparameter *20*. Das Testwerkzeug findet das Schlüsselwort in der *Device Bibliothek* und gelangt darüber an die Programmlogik, welche mit dem festgelegten Parameter ausgeführt wird. Als nächstes findet eine Wertzuweisung statt, die vom Programm in den Bezeichner und den zuzuweisenden Wert unterteilt wird. In diesem Fall ist der zuzuweisende Wert ein Rückgabewert des Schlüsselworts *Lese Wert*. Das Testwerkzeug führt das Schlüsselwort aus und speichert den Rückgabewert unter dem Variablenbezeichner *val* ab. Im letzten Testschritt wird das Testergebnis bestimmt. Dazu wird der gesetzte Wert mit dem gelesenen verglichen. Das Schlüsselwort *Ist gleich* enthält die Programmlogik für den Vergleich und wird mit den beiden Werten aufgerufen. Bei gleichen Werten wird es fehlerfrei ausgeführt, ansonsten wird ein Fehler mit einer entsprechenden Nachricht ausgegeben. Anhand der Abbildung ist zu erkennen, dass die Kommunikation mit externen Geräten ausschließlich in Bibliotheken realisiert ist. Das Testwerkzeug muss nach der

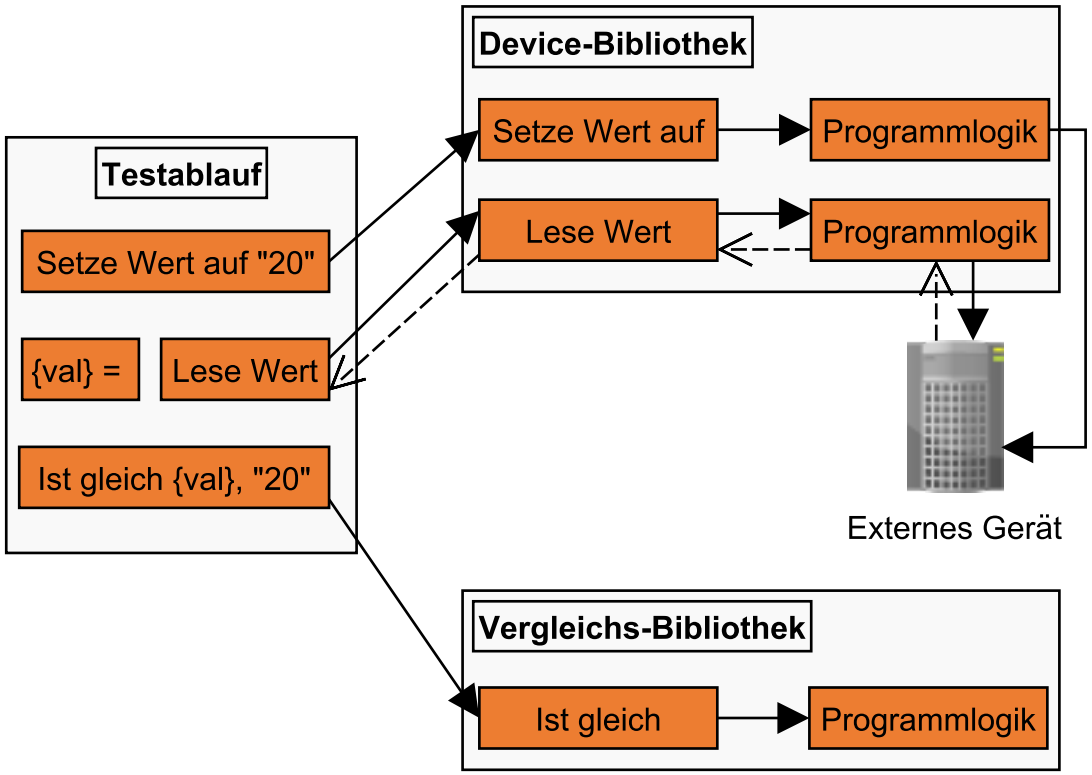

Abbildung 4.8.: *In dieser Abbildung wird die Ausführung von Testschritten skizziert. Links ist ein Testablauf mit Testschritten zu sehen. Rechts oben und rechts unten sind Bibliotheken abgebildet. Außerdem greift die obere der beiden auf ein externes Gerät zu.*

Ermittlung von Programmlogik nicht unterscheiden, ob oder ob nicht mit einem externen System kommuniziert wird.

Zusammengefasst wird bei der Ausführung von Testschritten zunächst zwischen Wertzuweisung und Schlüsselwort unterschieden. Bei ersterem wird eine lokale Variable angelegt, die den Wert hinter dem Gleichheitszeichen erhält. Variablen sind zu einem späteren Zeitpunkt mit dem Bezeichner referenzierbar, wobei lokale Variablen gleichnamige globale überdecken. Schlüsselwörter werden unter Beachtung des optional vorangestellten Bibliotheksbezeichner in der oder den vorhandenen Bibliotheken gesucht. Die Übergabeparameter werden in die von der Programmlogik erwarteten Typen übersetzt. Dabei muss man beachten, dass das Testwerkzeug ausschließlich die primitiven Datentypen und *String* kennt. Die Ausführung eines Schlüsselworts resultiert in einem Rückgabewert oder einem Fehler. Im Fehlerfall wird, je nachdem in welcher Testphase der Testschritt ist, der aktuelle Test oder der gesamte Testlauf beendet.

Protokollierung von Testläufen

Nachdem die Testabläufe der festgelegten Testdateien ausgeführt sind, wird ein Protokoll über den gesamten Testverlauf geschrieben. Das Protokoll listet die Testergebnisse aller eingelesenen Tests auf. Fehlerfrei ausgeführte Tests werden als bestanden, fehlerbehaftete als nicht bestanden gekennzeichnet. Im Fehlerfall wird zusätzlich zum Testergebnis eine Beschreibung des aufgetretenen Fehlers angegeben, welche bei der Fehlerfindung und Fehlerbehebung unterstützt. Die einzelnen Testergebnisse werden in einem Gesamtergebnis zusammengefasst, welches nur bei fehlerfreier Ausführung aller Tests bestanden ist. Dadurch kann nachvollzogen werden, ob in einem oder mehreren Tests Fehler aufgetreten sind. Des Weiteren wird der Name des Testers sowie der Zeitpunkt der Testdurchführung gespeichert.

In Listing 4.3 ist ein Protokoll eines Testlaufs zu sehen. Analog zu Eingabedateien werden ebenfalls *Tags* verwendet, um die Informationen darzustellen. Die ersten beiden Zeilen der Datei enthalten den Namen des Testers und den Zeitpunkt der Testdurchführung, welche mit den *Tags* AUTHOR beziehungsweise DATE gekennzeichnet sind. Der Name des Autors wird standardmäßig vom Testwerkzeug ermittelt, in dem der Benutzername des Betriebssystem ausgelesen wird, jedoch kann alternativ auch manuell ein Name angegeben werden. Dahingegen wird der Zeitpunkt der Testdurchführung ausschließlich vom Testwerkzeug festgelegt, wodurch ungenaue und falsche Zeitangaben vermieden werden. In der vierten Zeile steht das Gesamtergebnis der Testdurchführung. Dieses wird durch den STATE-*Tag* dargestellt und wird als PASS oder FAIL beschrieben, also als bestanden beziehungsweise nicht bestanden. In diesem Protokoll sind alle Tests fehlerfrei ausgeführt, weswegen das Gesamtergebnis bestanden ist. Im Anschluss an das Gesamtergebnis folgt eine Auflistung der einzelnen Tests, wobei zu jedem Test das Ergebnis als *Tag* vorangestellt wird. Anschließend folgt der Testname und der Dateipfad. Anhand des Testnamens und dem Testergebnis wird nachvollzogen, welche Funktionalität in Ordnung, beziehungsweise nicht in Ordnung ist. Besonders die fehlerhaft ausgeführten Tests sind für den Protokollleser interessant. Deswegen werden, wie in Listing 4.4 zu sehen, die nicht bestandenen Tests vor den bestandenen angezeigt, wodurch sie gruppiert und leicht zu finden sind. Zusätzlich zum Testergebnis, dem Testnamen und dem Dateipfad wird eine Fehlernachricht, wie in Zeile 6 zu sehen, angegeben. Zu Beginn der Nachricht steht die Testphase und Zeile des fehlerhaften Testschritts. Dahinter folgt eine Fehlerbeschreibung, welche in der Programmlogik des Schlüssel-

```
[AUTHOR]     Joshua Jungen
[DATE]       01.07.2016 12:15

[STATE]      PASS

[PASS]  Verändern von Werten        C:/werteÄndern.tst
[PASS]  Obere Grenzen einhalten     C:/obereGrenzen.tst
[PASS]  Untere Grenzen einhalten    C:/untereGrenzen.tst
```

Listing 4.3: *Protokoll über einen erfolgreichen Testablauf*

```
[STATE]      FAIL

[FAIL]  Obere Grenzen einhalten     C:/obereGrenzen.tst  Test
                  ↪: Zeile 11: Ungültiger Wert

[PASS]  Verändern von Werten        C:/werteÄndern.tst
[PASS]  Untere Grenzen einhalten    C:/untereGrenzen.tst
```

Listing 4.4: *Ausschnitt eines Protokoll über einen fehlgeschlagenen Testablauf*

worts festgelegt wird. Der von der Programmlogik ausgegebene Fehler enthält die Beschreibung, sodass das Testwerkzeug jene auslesen und als Fehlerbeschreibung verwenden kann. Demnach legt der Programmierer die Fehlerbeschreibung bei der Implementierung von Schlüsselwörtern fest. Des Weiteren werden Protokolldateien schreibgeschützt gespeichert, wodurch der Benutzer die Dateien nicht versehentlich überschreiben oder ändern kann.

4.4.4. Gestaltung der graphischen Oberfläche

Das Testwerkzeug wird vom Benutzer über eine graphische Oberfläche bedient. Mit dieser werden einerseits Tests erstellt und andererseits Tests durchgeführt. In den Anforderungen ist festgelegt, dass die Erstellung ausschließlich von Fachpersonal getätigt werden darf. Deswegen werden zwei separate Anwendungen entwickelt, die sich auf die Testerstellung beziehungsweise Testdurchführung spezialisieren. Aufgrund der Separation können die Anwendungen entsprechend der Zielgruppe freigegeben werden. Das Fachpersonal bekommt beide Anwendungen, da sie sowohl Tests erstellen als auch durchführen können. Dahingegen erhält normales Personal ausschließlich die Anwendung für die Testdurchführung.

Oberfläche für die Testerstellung

Für die Testerstellung benötigt der Testersteller einen Texteditor und die Dokumentation der Schlüsselwörter. Jedes Betriebssystem hat einen Standard-Texteditor und darüber hinaus gibt es zahlreiche kommerzielle und nicht-kommerzielle Editoren. Deswegen wird auf bestehende Editoren gesetzt, sodass der Testersteller einen beliebigen Texteditor verwenden kann. Dadurch kann er in einer gewohnten Anwendung Tests erstellen und auf die ihm bekannte Funktionalität des Editors, wie zum Beispiel *Syntax Highlighting*, zurückgreifen. Demnach fokussiert sich die Oberfläche für die Testerstellung auf die Darstellung der Dokumentation von Bibliotheken und Schlüsselwörtern. Die Dokumentation zu Bibliotheken und Methoden wird über die *Getter*-Methoden der Klassen *KeywordLibrary* und *Keyword*, siehe Abb. 4.6 auf Seite 31, ermittelt. Abb. 4.9 zeigt die entwickelte Oberfläche. Die Anwendung

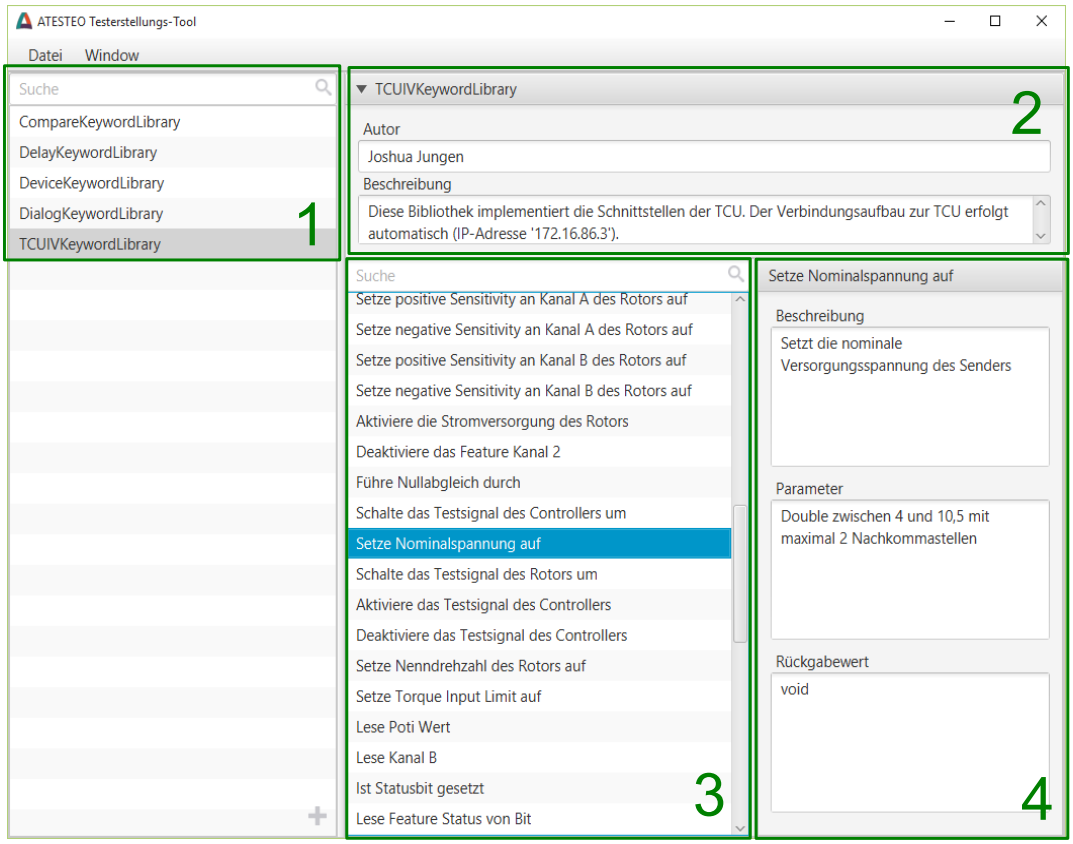

Abbildung 4.9.: *Die Abbildung zeigt die graphische Oberfläche für die Testerstellung.*

lädt beim Start alle verfügbaren Bibliotheken aus dem Verzeichnis für Bibliotheken und führt diese in einer Liste, siehe Bereich 1, auf. Durch Selektion eines Listenein-

trags werden die Informationen der ausgewählten Bibliothek in Bereich 2 angezeigt. Der Name des Autors und die Beschreibung werden in die entsprechenden Textfelder geschrieben. In Bereich 3 sind die Schlüsselwörter der ausgewählten Bibliothek angezeigt. Durch Auswahl eines der Schlüsselwörter werden die zugehörigen Informationen in Bereich 4 aktualisiert. Der Name des Schlüsselworts, die Beschreibung, sowie die Informationen zu den Parametern und dem Rückgabewert sind in den entsprechend benannten Textfeldern angezeigt. Die Listen in Bereich 1 und 3 haben zusätzlich ein Suchfeld. Besonders im Bereich 3 ist zu sehen, dass bei einer hohen Anzahl von Schlüsselwörtern schnell der Überblick verloren geht. Das Suchfeld hilft dem Testersteller die Bibliotheken und Schlüsselwörter zügiger zu finden. Über das Dateimenü können Testdateien erstellt oder geöffnet werden. Der Nutzer kann seinen bevorzugten Texteditor in den Einstellungen der Anwendung angeben, welcher für die Bearbeitung von Testdateien verwendet wird. Der Benutzer kann somit aus der Anwendung heraus den bevorzugten Texteditor starten und eine ausgewählte Datei öffnen.

Des Weiteren können die aufgelisteten Bibliotheken manuell entfernt und hinzugefügt werden. Die aufgelisteten Bibliotheken können gespeichert und zu einem späteren Zeitpunkt geladen werden. Der Testersteller kann demnach bestimmen, welche Inhalte er sehen, beziehungsweise nicht sehen möchte. Dadurch kann beispielsweise für jedes Testobjekt eine eigene Konfiguration von Bibliotheken angelegt werden, sodass unnötige Bibliotheken verborgen bleiben. Dadurch bleibt die Oberfläche vor allem bei einer Vielzahl von Bibliotheken übersichtlich.

Oberfläche für die Testdurchführung

Für die Testdurchführung werden Testdateien vom Benutzer festgelegt. Zusätzlich benötigt das Testwerkzeug den Namen des Benutzers, um diesen zusammen mit dem Zeitpunkt der Ausführung im Protokoll zu vermerken. Das Protokoll wird in eine vom Nutzer angegebene Datei gespeichert. Die Abb. 4.10 zeigt die Oberfläche der Testdurchführung. Im Bereich 1 wird der Benutzer festgelegt. Dabei wird standardmäßig der aktuelle Benutzer des Betriebssystems eingetragen, dies kann jedoch manuell geändert werden. Als nächste werden Testdateien festgelegt, wobei der Benutzer einzelne Dateien oder gesamte Ordner der Liste in Bereich 2 hinzufügt. Danach wird im Textfeld im fünften Bereich die Protokolldatei angegeben, welche

nach der Testdurchführung vom Testwerkzeug erzeugt wird. Anschließend werden die aufgelisteten Testdateien durch Betätigen des *Button* mit der Aufschrift *Ausführen*, siehe Bereich 6, ausgeführt. Während der Ausführung wird der Fortschritt des Testlaufs mit den beiden Fortschrittsbalken im vierten Bereich angezeigt. Der obere Balken zeigt den Fortschritt der aktuell ausgeführten Testdatei an, wohingegen der untere Balken nach jeder ausgeführten Datei erhöht wird und somit den Gesamtfortschritt anzeigt. Wenn der Testlauf während der Ausführung abgebrochen werden soll wird der *Button* mit der Aufschrift *Abbrechen* gedrückt. Nachdem die Ausführung beendet ist wird das Protokoll in der angegebenen Datei abgespeichert und zusätzlich im Textfeld in Bereich 3 angezeigt.

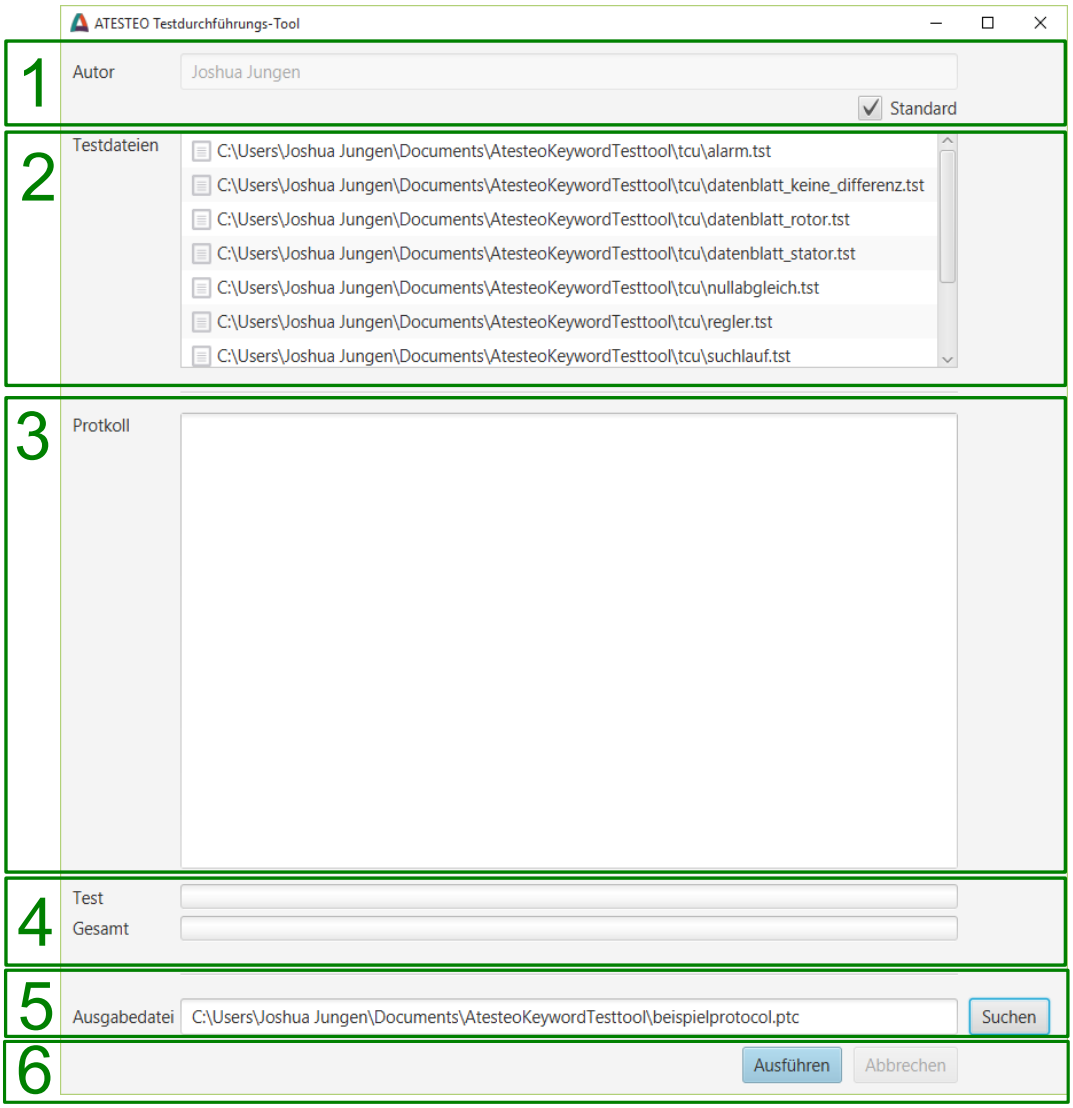

Abbildung 4.10.: *Die Abbildung zeigt die graphische Oberfläche für die Testdurchführung.*

5. Auswertung

5.1. System Under Test

Ein Geschäftsfeld der ATESTEO GmbH ist die Entwicklung von Drehmoment-Messtechnik, welche primär beim Testen von Antriebssträngen eingesetzt wird. Jene Messtechnik erhebt für die Produktentwicklung von Automobilzulieferern wichtige Daten, die beispielsweise bei der Bestimmung von Belastbarkeits- und Verschleißgrenzen verwendet werden. Zu dieser Art von Messtechnik wird unter anderem die sogenannte DF-Serie gezählt. Das System besteht aus insgesamt drei Komponenten, die in Abb. 5.1 abgebildet sind. Ein zu untersuchender Antriebsstrang wird an den sogenannten Rotor montiert, welcher im obereren Bereich von Abb. 5.1 zu sehen ist. Dieser ist ein zylindrischer Metallkörper und enthält die für die Messungen benötigten Sensoren. Er dreht sich zusammen mit dem Antriebsstrang, weswegen die Messdaten kabellos zur Empfangseinheit, dem sogenannten Stator, gesendet werden. Außerdem hat der Rotor keine eigene Spannungsversorgung, weshalb er vom Stator induktiv mit Spannung versorgt wird. Jenes Gerät befindet sich in der Abbildung unmittelbar unter dem Rotor. Anschließend werden die empfangenen Messdaten vom Stator zur Auswerteeinheit, TCU[1] genannt, über ein Kabel weitergeleitet. Über das Kabel werden sowohl Messdaten gesendet, als auch der Stator mit Spannung versorgt. Somit ist die TCU für die Versorgung des Stators zuständig, welcher wiederum den Rotor versorgt. Außerdem empfängt die TCU die Messdaten und ist für die Aufbereitung dieser zuständig. Anhand verschiedener Konfigurationen der Auswerteeinheit werden die Daten auf unterschiedliche Art und Weise verarbeitet, sodass sie die Bedürfnisse des Kunden decken. Schließlich werden die aufbereiteten Messdaten von der TCU an weitere Programme gesendet, welche für die Speicherung und Analyse zuständig sind. Im Rahmen der Auswertung ist die

[1] *Torque Control Unit*

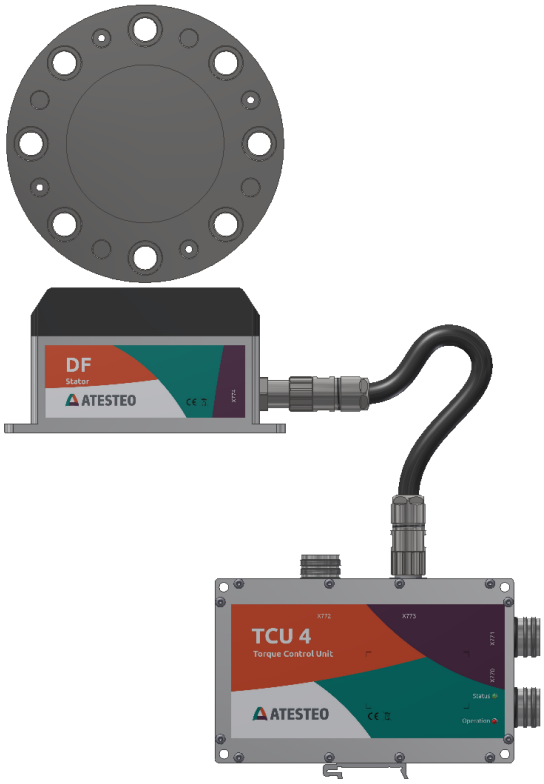

Abbildung 5.1.: *In dieser Abbildung ist die DF-Serie zu sehen. Es sind der Rotor, der Stator und die TCU (von oben nach unten) abgebildet.*

TCU das *System Under Test* und wird nachfolgend mit dem entwickelten Testwerkzeug untersucht.

5.2. Testwerkzeug in der Praxis

In diesem Abschnitt wird die TCU mit dem Testwerkzeug getestet. Dafür werden zunächst die Schnittstellen des Messgeräts in einer Schlüsselwortbibliothek implementiert. Danach werden mit der entwickelten Bibliothek Tests erstellt, welche anschließend mit dem Testwerkzeug ausgeführt werden.

5.2.1. Implementierung der Schnittstellen

Die zu untersuchende TCU wird mit einem Netzwerkkabel an einen Computer angeschlossen und nimmt die Funktion eines Servers ein. Dieser Server stellt einerseits die Messdaten bereit und ermöglicht außerdem die Konfiguration des Messgeräts. Die Datenübertragung findet mittels HTTP[2] statt, wobei die Daten im XML-Format angegeben werden. In der Dokumentation des Messgeräts wird jede Schnittstelle beschrieben, welche von einem Programmierer in einzelnen Methoden implementiert werden. Für die Wertabfrage wird ein HTTP-*Request* an die TCU gesendet, woraufhin ein XML-String zurückgegeben wird. Aus diesem wird der Wert ausgelesen und von der Methode zurückgegeben. Analog dazu werden mittels eines HTTP-*Request* Werte an die TCU gesendet, welche diese dann in den Konfiguration übernehmen kann. Die resultierende Klasse enthält Rund 150 Methoden, die jeweils eine einzelne Schnittstelle repräsentieren. Als nächste wird der Autor sowie eine Beschreibung der Bibliothek mit der *Annotation* für Bibliotheken angegeben. Zu diesem Zeitpunkt ist die Klasse für das Testwerkzeug als Bibliothek verwendbar, jedoch sind noch keine Schlüsselwörter enthalten. Nun wird für jede Methode einer Schnittstelle entschieden, ob diese als Schlüsselwort tauglich ist. Einige Schnittstellen müssen aufgrund ihrer Komplexität in mehrere Schlüsselwörter aufgeteilt werden, sodass sie für den Testersteller verständlich und leicht zu verwenden sind. Generell ist es ratsam, dass die Programmlogik von Schlüsselwörtern möglichst kompakt ist, denn durch zu komplexe Programmlogik wird die Testerstellung unflexibel. Andererseits sind zu kleinschrittige Schlüsselwörter ebenfalls schlecht, weil durch diese die Anzahl der Testschritte gesteigert wird. Demnach muss für jede Schnittstelle entschieden werden, ob sie als Schlüsselwort tauglich ist oder ob sie durch mehrere Methoden vereinfacht wird. Die Methoden für Schlüsselwörter werden mit der entsprechenden *Annotation* versehen und die geforderten Information in dieser angegeben. An dieser Stelle zeigt sich, dass die Verwendung von *Annotations* gut funktioniert, da nicht jede Methode der Klasse ein Schlüsselwort repräsentieren muss. Dadurch wird redundanter Programmcode vermieden und komplexe Schnittstellen können vereinfacht werden.

[2] *Hypertext Transfer Protocol*

5.2.2. Erstellung und Ausführung von Tests

Mit der entwickelten Bibliothek können nun Tests erstellt werden. Dazu wird die graphische Oberfläche für die Testerstellung verwendet, welche die Dokumentation der Bibliothek und der Schlüsselwörter darstellt. Tests für die TCU sind bereits dokumentiert, sodass sie händisch durchgeführt werden können. Daher werden für die Auswertung des Testwerkzeugs einige der bereits dokumentierten Tests ausgewählt und in Testdateien umgesetzt. Nachfolgend werden die Tests erläutert und anschließend die automatisierte Ausführung dieser betrachtet.

In der TCU können sogenannte Alarmschwellen eingestellt werden. Wenn Messwerte einen festgelegten Wertebereich verlassen teilt die TCU dies visuell mit. Dazu wird ein Test namens *Alarm* erstellt, der die korrekte Funktionsweise der Alarmschwellen überprüft. Außerdem können bei der Aufbereitung von Messdaten diverse Parameter eingestellt werden, wodurch die Messwerte unterschiedlich verarbeitet werden. Die Parameter werden über das sogenannte Datenblatt festgelegt. Im Messsystem haben sowohl der Rotor, als auch die TCU ein solches Datenblatt. Da es verschiedene Rotoren gibt wird ein Standard-Datenblatt im Rotor hinterlegt, welches von der TCU übernommen werden kann. Es ist jedoch möglich benutzerdefinierte Werte zu verwenden, weswegen die TCU ebenfalls ein Datenblatt besitzt. Wenn die beiden Datenblätter unterschiedliche Parameter haben muss ein Datenblatt ausgewählt werden. Sind die Datenblätter identisch so muss die TCU dies erkennen, jedoch muss kein Datenblatt explizit bestimmt werden. Dafür werden drei Tests namens *Datenblatt (Rotor)*, *Datenblatt (TCU)* und *Datenblatt (identisch)* erstellt, welche diese Funktionalität überprüfen. In den ersten beiden Tests werden zwei unterschiedliche Datenblätter erzeugt, woraufhin die TCU abfragt, welches davon verwendet werden soll. Daraufhin wird im ersten Test das Datenblatt des Rotors gewählt und im zweiten Test das der TCU. Im Anschluss wird geprüft, ob die entsprechenden Werte übernommen wurden und im dritten Test werden zwei identische Datenblätter erzeugt, was von der TCU erkannt werden muss. Des Weiteren wird bei der ersten Inbetriebnahme der TCU die benötigte Speisespannung durch den sogenannten automatischen Suchlauf ermittelt. Bei diesem wird Stück für Stück die Speisespannung erhöht bis der Rotor mit genügend Spannung versorgt wird und betriebsbereit ist. Dies wird durch den Test *Automatischer Suchlauf* überprüft. Während die TCU Messdaten verarbeitet gibt es betriebsbedingte Schwankungen der Speisespannung, weswegen diese nachträglich geregelt werden muss. Im Tests

namens *Langsamer Regler* wird der Regler geprüft, der jene Spannung automatisch anpasst. Dazu wird die Spannung bewusst manipuliert, woraufhin der Regler innerhalb eines bestimmten Zeitraums die Spannung in den optimalen Bereich regeln muss. Des Weiteren gibt es zwei verschiedene Testsignale namens *Testsignal TCU* und *Testsignal Rotor*. Durch die Aktivierung eines dieser Testsignale werden bestimmte Messwerte simuliert, was zur Überprüfung von Signalausgängen des Messgeräts notwendig ist. Für die Testsignale werden zwei weitere Testdateien erstellt.

Zu den zuvor erläuterten Tests werden Testdateien verfasst und anschließend mit dem Testwerkzeug ausgeführt. Die konkreten Inhalte der Testdateien sind in Kapitel A aufgeführt. Um die Testergebnisse zu validieren wurden die Testabläufe noch einmal händisch durchgeführt und mit den Ergebnissen des Testwerkzeugs verglichen. Es hat sich gezeigt, dass das Testwerkzeug die angegeben Testdateien erfolgreich einliest und die Testabläufe wie beschrieben durchführt. Die Ergebnisse des Testwerkzeugs stimmen mit denen der händischen Ausführung überein, womit die automatisierte Testausführung korrekt funktioniert hat. Dabei fiel vor allem eine deutliche Verbesserung der Testausführungszeit auf, welche durch die Automatisierung auf einen Bruchteil der vorherigen reduziert wird. Die gemessenen Zeiten sind in Abb. 5.2 abgebildet, wobei die manuelle Ausführungszeit in ausgefüllten und die automatisierte in schraffierten Balken dargestellt ist. Wie im Diagramm zu sehen ist die Zeitersparnis zwischen Tests unterschiedlich groß. Dabei gibt es eine maximale Zeitersparnis von 90% beim Test *Alarm* und eine minimale Ersparnis von 63% beim Test *Langsamer Regler*. Die großen Unterschiede bei der Zeitersparnis sind auf betriebsbedingte Verzögerung zurückzuführen. Im Fall des Tests für den langsamen Regler muss das Programm nach Aktivierung des Reglers eine festgelegte Zeit warten, bis es den nächsten Testschritt ausführt. Verzögerungen dieser Art verlangsamen die Ausführung, jedoch sind sie betriebsbedingt nicht zu vermeiden, da das Messgerät für die Verarbeitung einiger Befehle mehrere Sekunden in Anspruch nimmt. Insgesamt wird durch die Ausführung der acht automatisierten Testfälle eine Zeitersparnis von ca. 77% erreicht. Dieser Wert basiert auf einer kleinen Stichprobe von gemessenen Werten, weswegen er differenziert betrachtet werden muss. Wenn neu hinzukommende Tests ähnliche Laufzeiten wie der Test *Alarm* haben wird mehr Zeit eingespart. Andernfalls wird die Zeitersparnis geringer, wenn die Laufzeiten sich an der des Tests *Langsamer Regler* orientieren. Dennoch wird in beiden Fällen der Testprozess deutlich effektiver als er bisher ist.

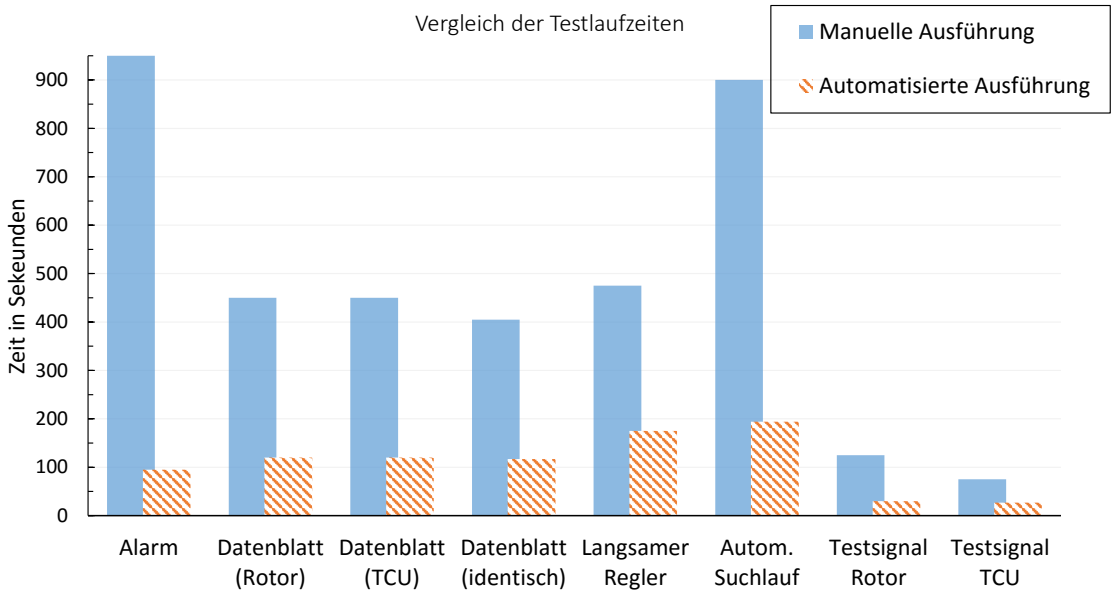

Abbildung 5.2.: *Das Diagramm bildet die Ausführungszeiten der Testfälle ab. Zu jedem Testfall wird die Zeit für die manuelle und die automatisierte Ausführung, in ausgefüllten beziehungsweise schraffierten Balken, dargestellt.*

Die Zeitersparnis bei der Testdurchführung ist nicht die tatsächliche Zeitersparnis, da zuvor die Bibliothek und Testdateien entwickelt werden mussten. Ohne diese wäre die automatisierte Ausführung nicht möglich, weswegen die dafür benötigte Zeit nicht außer Acht gelassen werden darf. Im Fall der TCU hat die Entwicklung der Bibliothek vier Arbeitstage und somit 32 Stunden benötigt. Die Programmierung einzelner Schnittstellen ist nicht besonders zeitaufwändig, jedoch entsteht aufgrund der Vielzahl von Schnittstellen der entsprechende Aufwand. Im Anschluss daran wurden die vorgestellten Testdateien erstellt, wofür ein weiterer Arbeitstag notwendig war. Zusammengerechnet wurden demnach 40 Stunden in die Automatisierung investiert bevor die Testfälle zum ersten Mal ausgeführt werden konnten. Hinzu kommt die Entwicklungszeit des Testwerkzeug, welche nach Abschnitt 3.5 18 Arbeitstage, beziehungsweise 144 Stunden beträgt. In Abb. 5.3 wird die zuvor berechnete Zeitersparnis dem Zeitaufwand gegenübergestellt. Die Zeitersparnis durch die automatisierte Ausführungen der erstellten Tests wird durch die durchgezogene, der Aufwand für die TCU durch die gestrichelte und der Gesamtaufwand durch die gepunktete Linie dargestellt. Unter der Anzahl der Testdurchführung versteht sich die stapelweise Ausführung der insgesamt acht vorgestellten Tests. Dabei ist

zu erkennen, dass der Zeitaufwand für die Entwicklung der Bibliothek und der Testdateien nach circa 50 Ausführungen durch die Zeitersparnis amortisiert wird. Dahingegen wird der Gesamtaufwand der für das Testwerkzeug und die TCU notwendig war nach 239 Ausführung ausgeglichen. Dazu sei gesagt, dass das Unternehmen beabsichtigt ungefähr 250 der getesteten Geräte pro Jahr verkaufen möchte. Durch das Diagramm wird gezeigt, dass der Zeitaufwand für die Automatisierung

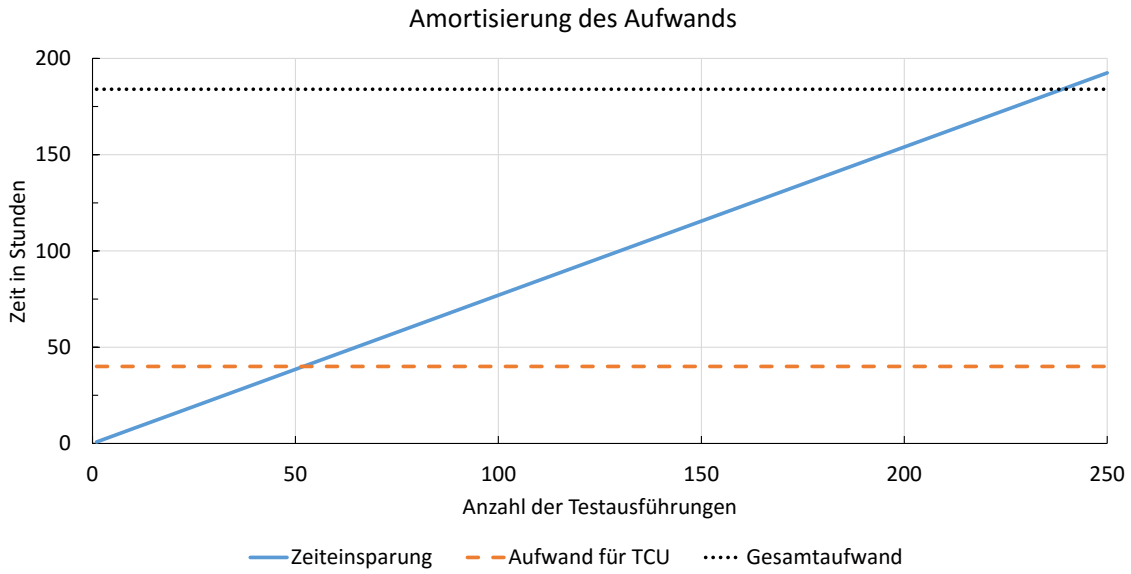

Abbildung 5.3.: *Das Diagramm zeigt den Zeitaufwand des gesamten Projekts sowie die Zeitkosten für die Testautomatisierung der TCU. Dem gegenüber steht die, durch automatisierte Durchführung der erstellten Tests, eingesparte Zeit.*

bei entsprechender Anzahl von Testausführungen wirtschaftlich wird. Dabei muss beachtet werden, dass in der Realität weitaus mehr als acht Tests erstellt werden müssen sowie weitere Bibliotheken für umfangreichere Tests des Gerät notwendig sind. Somit würde auch der für die Automatisierung notwendige Aufwand steigen. Dennoch wird durch die insgesamt deutliche Zeitersparnis der benötigte Aufwand langfristig gerechtfertigt, da der Testprozess in gesteigertem Umfang davon profitiert.

5.3. Bewertung

Der Testprozess wird durch das entwickelte Testwerkzeug in verschiedenen Bereichen verbessert. Aufgrund des einheitlichen Formats von Testdateien werden Tests standardisiert dokumentiert, wodurch mit diesen Testabläufe nachvollzogen werden können. Dabei ist die Erstellung der Testdateien wegen dem schlüsselwortgetriebenen Ansatz ohne Programmiererfahrung möglich, womit die Anforderungen an die Testerstellung erfüllt sind. Des Weiteren wurde gefordert, dass beliebige externe Systeme in den Testprozess integriert werden können. Dies wird vom Testwerkzeug mit sogenannten Bibliotheken umgesetzt, welche von einem Programmierer erstellt und dem Testwerkzeug hinzugefügt werden. Jene Bibliotheken enthalten die für die Kommunikation notwendige Programmlogik, welche somit spezifisch auf die Geräte angepasst werden kann. Dadurch können sowohl beliebige, als auch beliebig viele externe Systeme in den Testprozess integriert werden. Durch das sogenannte *Hot Deployment* ist die Integration von Bibliotheken zur Programmlaufzeit möglich, sodass keine Aktualisierung der Software notwendig ist. Dies erleichtert die Integration von neuen Systemen, da lediglich die entwickelten Bibliotheken einem festgelegten Verzeichnis hinzugefügt werden müssen.

Außerdem ist die Testdurchführung mit dem Testwerkzeug nun für einen größeren Personenkreis möglich, da das Testwerkzeug die Kommunikation und Interaktion zu externen Geräten, unter Verwendung der Bibliotheken, übernimmt. Bei der Testdurchführung werden durch eine graphischen Oberfläche Testdateien ausgewählt, welche anschließend vom Testwerkzeug automatisiert durchgeführt und protokolliert werden. Dadurch wird das Fachpersonal entlastet, weil die Testdurchführung mit dem Testwerkzeug kein qualifiziertes Personal mehr erfordert. Zusätzlich wird durch die Automatisierung die Ausführungszeit reduziert und die Qualität der Testergebnisse verbessert. Die Qualitätssteigerung erklärt sich durch die vom Testwerkzeug stets gleichartig ausgeführten Testabläufe, denn deren händische Ausführung ist stark von der Aufmerksamkeit und Präzision des Testers abhängig. Die vom Testwerkzeug dokumentierten Testergebnisse haben somit eine bessere und konstantere Qualität. Darüber hinaus werden die Protokolle in einem einheitlichen Format gespeichert, wobei präzise Fehlermeldungen bei der Fehlerfindung und -behebung von nicht bestandenen Tests unterstützen.

5.4. Ausblick

Das Testwerkzeug erfüllt in seinem jetzigen Zustand die Anforderung, jedoch sind verschiedene Möglichkeiten für Erweiterungen und Optimierungen vorhanden. Im Rahmen dieser Arbeit wurde eine einfache Oberfläche entwickelt, die die grundlegenden Funktionen des Testwerkzeugs graphisch bedienbar macht. Die rudimentär gehaltene Oberfläche könnte in den Bereichen Benutzerfreundlichkeit und *Corporate Design* optimiert werden. Im Praxiseinsatz des Testwerkzeugs hat sich gezeigt, dass eine Kategorisierung von Tests hilfreich sein kann. Durch Kategorien könnte der Nutzer beliebige Testdateien angeben und anschließend nur ausgewählte Kategorien von Tests ausführen. Dadurch würde sich die Auswahl von Testdateien vereinfachen, da der Nutzer die tatsächlich ausgeführten Tests durch Kategorien einschränkt. Dies würde Änderungen am Eingabeformat der Testdateien erfordern, weil die Kategorie eines Test in der Testdatei angegeben werden muss. Außerdem müsste die Testdurchführung angepasst werden, damit Tests entsprechend der angegebenen Kategorien ausgesondert oder ausgeführt werden. Des Weiteren könnte dem Testwerkzeug ein Befehlssatz für die Kommandozeile hinzugefügt werden, da die Software bisher nur über die graphischen Oberflächen bedient werden kann. Durch die Steuerung per Kommandozeile können wiederkehrende Abläufe mittels Skripten automatisiert werden, wodurch die Testdurchführung einfacher und effizienter gestaltet werden kann.

Darüber hinaus könnte die Protokollierung umfangreich erweitert werden. Bisher werden Protokolldateien erstellt und in einem vom Benutzer angegebenen Verzeichnis abgelegt. Bei einer Vielzahl von Protokollen geht, vor allem von gleichartigen Geräten, unter Umständen die Übersicht verloren. Daher wäre die Speicherung der Protokolle in einer Datenbank sinnvoll, die die Protokolle zusammen mit einer eindeutigen Identifikationsnummer speichert. Somit könnten zum Beispiel anhand der Seriennummer eines Testobjekts die Testergebnisse abgefragt und bei mehrmals ausgeführten Tests ein Vergleich der Ergebnisse vorgenommen werden. Mit einer entsprechenden Datenbank wäre die erweiterte Auswertung von Testergebnissen möglich. Es könnten die Testergebnisse von gleichartigen Testobjekten statistisch untersucht werden, wodurch Aussagen zu Fehlerquoten oder ähnlichem getroffen werden können.

6. Fazit

Im Fokus dieser Arbeit stand die Entwicklung eines Testwerkzeugs zum Testen von externen Geräten. Dafür wurde der Entwicklungsprozess von der Anforderungserfassung über die Konzeption und Zeitplanung der Software bis hin zur konkreten Realisierung betrachtet.

Bei der Entwicklung sollte beachtet werden, dass die Testerstellung und Testdurchführung ohne Programmiererfahrung erfolgen soll. Deswegen wurde ein schlüsselwortgetriebener Testansatz gewählt, welcher zusätzlich auf die Integration von externen Systemen in den Testprozess abgestimmt wurde. Das resultierende Testwerkzeug erfüllt somit die Erwartungen und zeichnet sich gegenüber anderen Testwerkzeugen dadurch aus, dass es den Fokus auf externe Geräte legt. Es unterstützt den Testprozess bei der Testerstellung und automatisiert sowohl die Testdurchführung als auch die Testprotokollierung. Ein vorteilhaftes Nebenerzeugnis des Testwerkzeugs ist eine standardisierte Testdokumention, welche durch die einheitliche Art der Testdarstellung und der programmgesteuerten Protokollierung entsteht. In der Praxis hat sich gezeigt, dass das Testwerkzeug die Anforderungen erfüllt und Tests an externen Geräten erfolgreich durchgeführt werden konnten. Für die Automatisierung von Tests unter Verwendung des Testwerkzeugs ist ein nicht zu unterschätzender Zeitaufwand notwendig, jedoch überwiegen die Vorteile der automatisierten Testdurchführung, sodass der Aufwand langfristig wirtschaftlich vertretbar ist. Des Weiteren wird die Effizienz und Qualität durch den Einsatz des Testwerkzeugs erhöht, wie es bei Testautomatisierung oftmals der Fall ist.

Abschließend ist festzuhalten, dass die erfolgreiche Entwicklung der Software den Grundstein für andere Erweiterungen und Optimierungen des Testprozess legt. Denkbar wäre beispielsweise eine Erweiterung der Testprotokollierung durch eine Datenbankanbindung, wodurch tiefgreifende Analysen und statistische Auswertungen von getesteten Geräten möglich werden können.

Literatur

[CIAD12] Bill Campbell, Swami Iyer und Bahar Akbal-Delibas. *Introduction to Compiler Construction in a Java World*. CRC Press, 2012.

[Gol11] Joachim Goll. *Methoden und Architekturen der Sofwaretechnik*. 1. Aufl. Vieweg+Teubner Verlag, 2011.

[Kle09] Stephan Kleuker. *Grundkurs Software-Engineering mit UML*. 1. Aufl. Vieweg+Teubner, 2009.

[R.B+05] R.Breu u. a. *Software Engineering. Objektorientierte Techniken, Methoden und Prozesse in der Praxis*. Oldenburg, 2005.

[SBB12] Richard Seidl, Manfred Baumgartner und Thomas Bucsics. *Basiswissen Testautomatisierung: Konzepte, Methoden und Techniken*. 1. Aufl. dpunkt.verlag, 2012.

[Spi+09] Andreas Spillner u. a. *Praxiswissen Softwaretest. Testmanagement*. 3. Aufl. dpunkt.verlag, 2009.

[Ull11] Christian Ullenboom. *Java ist auch eine Insel. Das umfassende Handbuch*. 9. Aufl. Galileo Press, 2011.

[Vig05] Uwe Vigenshow. *Objektorientiertes Testen und Testautomatisierung in der Praxis: Konzepte, Techniken und Verfahren*. dpunkt.verlag, 2005.

[Vig10] Uwe Vigenshow. *Testsen von Software und Embedded Systems. Professionelles Vorgehen mit modellbasierten und objektorientierten Ansätzen*. 2. Aufl. dpunkt.verlag, 2010.

A. Testdateien

```
1  [AUTHOR]     Joshua Jungen
2  [TESTNAME]   Überprüfung der Alarmschwellen
3
4  [LIB]        "TCUIVKeywordLibrary.jar" tcu
5  [VAR]        "config.var"
6
7  [REPEAT] 10
8  [SETUP]
9      Ist Torque Alarm inaktiv
10     {status_ch2} = Lese Feature Status von Bit  "4"
11     Deaktiviere das Feature Kanal 2
12     Warte für {delay_short}
13
14 [TEST]
15     {ratedTorqueA} =    Lese Rated Torque von Kanal A
16     Setze Torque Input Limit auf    "[0.25*{ratedTorqueA}]"
17
18     Schalte das Testsignal des Rotors um
19     Warte für {delay_short}
20     Ist Torque Alarm aktiv
21 [TEARDOWN]
22     Schalte das Testsignal des Rotors um
23     Setze Torque Input Limit auf {ratedTorqueA}
24     Setze Alarmausgänge zurück
25     Setze Feature Kanal 2 auf       {status_ch2}
26     Warte für {delay_short}
27     {rtA} = Lese Kanal A
28     Ist ungefähr gleich {rtA}, "0", "10"
```

Listing A.1: *Test: Alarm*

```
[AUTHOR] Joshua Jungen
[TESTNAME] Sende/Empfange elektronisches Datenblatt (Keine differenz)

[LIB]       "TCUIVKeywordLibrary.jar" tcu
[VAR]       "config.var"

[SETUP]
    {status_ch2} = Lese Feature Status von Bit  "4"

    Aktiviere das Feature Kanal 2
    Aktiviere das Feature Speed

    # TCU Werte speichern
    {tcu_rtA} =         Lese Rated Torque von Kanal A
    {tcu_psA} =         Lese positive Sensitivity an Kanal A
    {tcu_nsA} =         Lese negative Sensitivity an Kanal A
    {tcu_rtB} =         Lese Rated Torque von Kanal B
    {tcu_psB} =         Lese positive Sensitivity an Kanal B
    {tcu_nsB} =         Lese negative Sensitivity an Kanal B
    {tcu_nenn} =        Lese Nenndrehzahl der TCU
    {tcu_inc} =         Lese TCU Inkrement

    # Rotor Werte speichern
    {rot_rtA} =         Lese Rated Torque A aus dem Rotor
    {rot_psA} =         Lese positive Sensitivity an Kanal A
                    ↪ aus dem Rotor
    {rot_nsA} =         Lese negative Sensitivity an Kanal A
                    ↪ aus dem Rotor
    {rot_rtB} =         Lese Rated Torque B aus dem Rotor
    {rot_psB} =         Lese positive Sensitivity an Kanal B
                    ↪ aus dem Rotor
    {rot_nsB} =         Lese negative Sensitivity an Kanal B
                    ↪ aus dem Rotor
    {rot_nenn} =        Lese Nenndrehzahl des Rotors
    {rot_inc} =         Lese Rotor Inkrement

```

```
33      # TCU Werte setzen
34      Setze Rated Torque von Kanal A auf          {rot_rtA}
35      Setze positive Sensitivity an Kanal A auf   "9000.0"
36      Setze negative Sensitivity an Kanal A auf   "9000.0"
37      Setze Rated Torque von Kanal B auf          {rot_rtB}
38      Setze positive Sensitivity an Kanal B auf   "9000.0"
39      Setze negative Sensitivity an Kanal B auf   "9000.0"
40      Setze Nenndrehzahl der TCU auf              "25000"
41      Setze Inkremente der TCU auf                "60"
42
43      # Rotor
44      Sende Datenblatt an Rotor   "9000.0", "9000.0", "9000.0"
                ↪, "9000.0", "25000", "60"
45      Warte für   {delay_datasheet}
46
47      Deaktiviere die Stromversorgung des Rotors
48      Warte für   {delay_short}
49
50  [TEST]
51      Aktiviere die Stromversorgung des Rotors
52      Warte für   {delay_extra_long}
53      Ist Statusbit nicht gesetzt "16"
54
55  [TEARDOWN]
56      Setze Rated Torque von Kanal A auf          {tcu_rtA}
57      Setze positive Sensitivity an Kanal A auf   {tcu_psA}
58      Setze negative Sensitivity an Kanal A auf   {tcu_nsA}
59      Setze Rated Torque von Kanal B auf          {tcu_rtB}
60      Setze positive Sensitivity an Kanal B auf   {tcu_psB}
61      Setze negative Sensitivity an Kanal B auf   {tcu_nsB}
62      Setze Nenndrehzahl der TCU auf              {tcu_nenn}
63      Setze Inkremente der TCU auf                {tcu_inc}
64
65      Sende Datenblatt an Rotor   {rot_psA}, {rot_nsA}, {
                ↪rot_psB}, {rot_nsB}, {rot_nenn}, {rot_inc
                ↪}
66      Warte für                   {delay_datasheet}
```

```
67      Deaktiviere die Stromversorgung des Rotors
68      Warte für                       {delay_short}
69      Aktiviere die Stromversorgung des Rotors
70      Warte für                       {delay_extra_long}
71
72      Ist Statusbit gesetzt           "16"
73      Behalte Stator Werte
74
75      Setze Feature Kanal 2 auf       {status_ch2}
```

Listing A.2: *Test: Datenblatt(identisch)*

```
1  [AUTHOR] Joshua Jungen
2  [TESTNAME] Sende/Empfange elektronisches Datenblatt (Apply
                  ↪Rotor)
3
4  [LIB]        "TCUIVKeywordLibrary.jar" tcu
5  [VAR]        "config.var"
6
7  [SETUP]
8      {status_ch2} = Lese Feature Status von Bit  "4"
9
10     Aktiviere das Feature Kanal 2
11     Aktiviere das Feature Speed
12
13     # TCU Werte speichern
14     {tcu_rtA} =          Lese Rated Torque von Kanal A
15     {tcu_psA} =          Lese positive Sensitivity an Kanal A
16     {tcu_nsA} =          Lese negative Sensitivity an Kanal A
17     {tcu_rtB} =          Lese Rated Torque von Kanal B
18     {tcu_psB} =          Lese positive Sensitivity an Kanal B
19     {tcu_nsB} =          Lese negative Sensitivity an Kanal B
20     {tcu_nenn} =         Lese Nenndrehzahl der TCU
21     {tcu_inc} =          Lese TCU Inkrement
22
23     # Rotor Werte speichern
24     {rot_rtA} =          Lese Rated Torque A aus dem Rotor
25     {rot_psA} =          Lese positive Sensitivity an Kanal A
```

```
                    ↪ aus dem Rotor
26      {rot_nsA} =         Lese negative Sensitivity an Kanal A
                    ↪ aus dem Rotor
27      {rot_rtB} =         Lese Rated Torque B aus dem Rotor
28      {rot_psB} =         Lese positive Sensitivity an Kanal B
                    ↪ aus dem Rotor
29      {rot_nsB} =         Lese negative Sensitivity an Kanal B
                    ↪ aus dem Rotor
30      {rot_nenn}=         Lese Nenndrehzahl des Rotors
31      {rot_inc} =         Lese Rotor Inkrement
32
33      # TCU Werte setzen
34      Setze Rated Torque von Kanal A auf          "500"
35      Setze positive Sensitivity an Kanal A auf   "9000.0"
36      Setze negative Sensitivity an Kanal A auf   "9000.0"
37      Setze Rated Torque von Kanal B auf          "500"
38      Setze positive Sensitivity an Kanal B auf   "9000.0"
39      Setze negative Sensitivity an Kanal B auf   "9000.0"
40      Setze Nenndrehzahl der TCU auf              "25000"
41      Setze Inkremente der TCU auf                "60"
42
43      Sende Datenblatt an Rotor   "450.0", "450.0", "450.0", "
                    ↪450.0", "14000", "680"
44      Warte für   {delay_datasheet}
45
46      Deaktiviere die Stromversorgung des Rotors
47      Warte für               {delay_short}
48  [TEST]
49      Aktiviere die Stromversorgung des Rotors
50      Warte für               {delay_extra_long}
51      Ist Statusbit gesetzt   "16"
52      Behalte Rotor Werte
53
54      # Vergleiche die aktuelle TCU Werte mit den Rotor Werten
55      {new_tcu_rtA} = Lese Rated Torque von Kanal A
56      Ist gleich      {new_tcu_rtA}, {rot_rtA}
57
```

```
58      {new_tcu_psA} = Lese positive Sensitivity an Kanal A
59      Ist gleich      {new_tcu_psA}, "450.0"
60
61      {new_tcu_nsA} = Lese negative Sensitivity an Kanal A
62      Ist gleich      {new_tcu_nsA}, "450.0"
63
64      {new_tcu_rtB} = Lese Rated Torque von Kanal B
65      Ist gleich      {new_tcu_rtB}, {rot_rtB}
66
67      {new_tcu_psB} = Lese positive Sensitivity an Kanal B
68      Ist gleich      {new_tcu_psB}, "450.0"
69
70      {new_tcu_nsB} = Lese negative Sensitivity an Kanal B
71      Ist gleich      {new_tcu_nsB}, "450.0"
72
73      {new_tcu_nenn}= Lese Nenndrehzahl der TCU
74      Ist gleich      {new_tcu_nenn}, "14000"
75
76      {new_tcu_inc} = Lese TCU Inkrement
77      Ist gleich      {new_tcu_inc}, "680"
78
79      [TEARDOWN]
80      Setze Rated Torque von Kanal A auf          {tcu_rtA}
81      Setze positive Sensitivity an Kanal A auf   {tcu_psA}
82      Setze negative Sensitivity an Kanal A auf   {tcu_nsA}
83      Setze Rated Torque von Kanal B auf          {tcu_rtB}
84      Setze positive Sensitivity an Kanal B auf   {tcu_psB}
85      Setze negative Sensitivity an Kanal B auf   {tcu_nsB}
86      Setze Nenndrehzahl der TCU auf              {tcu_nenn}
87      Setze Inkremente der TCU auf                {tcu_inc}
88
89      Sende Datenblatt an Rotor {rot_psA}, {rot_nsA}, {rot_psB
            ↪}, {rot_nsB}, {rot_nenn}, {rot_inc}
90      Warte für               {delay_datasheet}
91      Deaktiviere die Stromversorgung des Rotors
92      Warte für               {delay_short}
93      Aktiviere die Stromversorgung des Rotors
```

```
94      Warte für                {delay_extra_long}
95
96      Ist Statusbit gesetzt    "16"
97      Behalte Stator Werte
98
99      Setze Feature Kanal 2 auf    {status_ch2}
```

Listing A.3: *Test: Datenblatt(Rotor)*

```
1  [AUTHOR] Joshua Jungen
2  [TESTNAME] Sende/Empfange elektronisches Datenblatt (Apply
                  ↪Stator)
3
4  [LIB]        "TCUIVKeywordLibrary.jar" tcu
5  [VAR]        "config.var"
6
7  [SETUP]
8      {status_ch2} = Lese Feature Status von Bit  "4"
9
10     Aktiviere das Feature Kanal 2
11     Aktiviere das Feature Speed
12
13     # TCU Werte speichern
14     {tcu_rtA} =          Lese Rated Torque von Kanal A
15     {tcu_psA} =          Lese positive Sensitivity an Kanal A
16     {tcu_nsA} =          Lese negative Sensitivity an Kanal A
17     {tcu_rtB} =          Lese Rated Torque von Kanal B
18     {tcu_psB} =          Lese positive Sensitivity an Kanal B
19     {tcu_nsB} =          Lese negative Sensitivity an Kanal B
20     {tcu_nenn} =         Lese Nenndrehzahl der TCU
21     {tcu_inc} =          Lese TCU Inkrement
22
23     # Rotor Werte speichern
24     {rot_rtA} =          Lese Rated Torque A aus dem Rotor
25     {rot_psA} =          Lese positive Sensitivity an Kanal A
                  ↪ aus dem Rotor
26     {rot_nsA} =          Lese negative Sensitivity an Kanal A
                  ↪ aus dem Rotor
```

```
27      {rot_rtB} =          Lese Rated Torque B aus dem Rotor
28      {rot_psB} =          Lese positive Sensitivity an Kanal B
                  ↪ aus dem Rotor
29      {rot_nsB} =          Lese negative Sensitivity an Kanal B
                  ↪ aus dem Rotor
30      {rot_nenn}=          Lese Nenndrehzahl des Rotors
31      {rot_inc} =          Lese Rotor Inkrement
32
33      # TCU Werte setzen
34      Setze Rated Torque von Kanal A auf           "500"
35      Setze positive Sensitivity an Kanal A auf    "9000.0"
36      Setze negative Sensitivity an Kanal A auf    "9000.0"
37      Setze Rated Torque von Kanal B auf           "500"
38      Setze positive Sensitivity an Kanal B auf    "9000.0"
39      Setze negative Sensitivity an Kanal B auf    "9000.0"
40      Setze Nenndrehzahl der TCU auf               "25000"
41      Setze Inkremente der TCU auf                 "60"
42
43      Sende Datenblatt an Rotor   "450.0", "450.0", "450.0", "
                  ↪450.0", "14000", "680"
44      Warte für   {delay_datasheet}
45
46      Deaktiviere die Stromversorgung des Rotors
47      Warte für                {delay_short}
48  [TEST]
49      Aktiviere die Stromversorgung des Rotors
50      Warte für                {delay_extra_long}
51
52      Ist Statusbit gesetzt    "16"
53      Behalte Stator Werte
54
55      # Vergleiche die aktuelle TCU Werte mit den alten
56      {new_tcu_rtA} = Lese Rated Torque von Kanal A
57      Ist gleich      {new_tcu_rtA}, "500"
58
59      {new_tcu_psA} = Lese positive Sensitivity an Kanal A
60      Ist gleich      {new_tcu_psA}, "9000.0"
```

```
61
62      {new_tcu_nsA} = Lese negative Sensitivity an Kanal A
63      Ist gleich      {new_tcu_nsA}, "9000.0"
64
65      {new_tcu_rtB} = Lese Rated Torque von Kanal B
66      Ist gleich      {new_tcu_rtB}, "500"
67
68      {new_tcu_psB} = Lese positive Sensitivity an Kanal B
69      Ist gleich          {new_tcu_psB}, "9000.0"
70
71      {new_tcu_nsB} = Lese negative Sensitivity an Kanal B
72      Ist gleich      {new_tcu_nsB}, "9000.0"
73
74      {new_tcu_nenn}= Lese Nenndrehzahl der TCU
75      Ist gleich      {new_tcu_nenn}, "25000"
76
77      {new_tcu_inc} = Lese TCU Inkrement
78      Ist gleich      {new_tcu_inc}, "60"
79
80  [TEARDOWN]
81      Setze Rated Torque von Kanal A auf          {tcu_rtA}
82      Setze positive Sensitivity an Kanal A auf   {tcu_psA}
83      Setze negative Sensitivity an Kanal A auf   {tcu_nsA}
84      Setze Rated Torque von Kanal B auf          {tcu_rtB}
85      Setze positive Sensitivity an Kanal B auf   {tcu_psB}
86      Setze negative Sensitivity an Kanal B auf   {tcu_nsB}
87      Setze Nenndrehzahl der TCU auf              {tcu_nenn}
88      Setze Inkremente der TCU auf                {tcu_inc}
89
90
91
92      Sende Datenblatt an Rotor {rot_psA}, {rot_nsA}, {rot_psB
                ↪}, {rot_nsB}, {rot_nenn}, {rot_inc}
93      Warte für               {delay_datasheet}
94      Deaktiviere die Stromversorgung des Rotors
95      Warte für               {delay_short}
96      Aktiviere die Stromversorgung des Rotors
```

```
97      Warte für                {delay_extra_long}
98      Ist Statusbit gesetzt    "16"
99      Behalte Stator Werte
100
101     Setze Feature Kanal 2 auf    {status_ch2}
```

Listing A.4: *Test: Datenblatt(TCU)*

```
1  [AUTHOR] Joshua Jungen
2  [TESTNAME] Überprüfe langsamen Regler
3
4  [LIB]       "TCUIVKeywordLibrary.jar" tcu
5  [VAR]       "config.var"
6
7  [SETUP]
8      {status_reg} = Lese Status von Bit      "18"
9      Verwende langsamen Regler   "false"
10
11 [TEST]
12     {poti} =                    Lese Poti Wert
13     Setze Versorgungsspannung auf    "[{poti}-1.5]"
14     Verwende langsamen Regler   "true"
15     Warte für                   {delay_extra_long}
16     {rotor} =                   Lese Versorgungsspannung am
                 ↪Rotor
17     {nom}   =                   Lese Nominalspannung
18     Ist ungefähr gleich         {rotor}, {nom}, "0.3"
19
20     Verwende langsamen Regler   "false"
21
22     {poti} =                    Lese Poti Wert
23     Setze Versorgungsspannung auf    "[{poti}+1.5]"
24     Verwende langsamen Regler   "true"
25     Warte für                   {delay_extra_long}
26     {rotor} =                   Lese Versorgungsspannung am
                 ↪Rotor
27     {nom}   =                   Lese Nominalspannung
28     Ist ungefähr gleich         {rotor}, {nom}, "0.3"
```

```
29
30 [TEARDOWN]
31     Verwende langsamen Regler    {status_reg}
```

Listing A.5: *Test: langsamer Regler*

```
1  [AUTHOR]     Joshua Jungen
2  [TESTNAME]   Überprüfe automatischen Suchlauf
3
4  [LIB]        "TCUIVKeywordLibrary.jar" tcu
5  [VAR]        "config.var"
6
7  [SETUP]
8      Setze Versorgungsspannung auf    "0.0"
9      {poti} =                         Lese Poti Wert
10     Ist gleich                       {poti}, "0.0"
11
12     {std_nom} = Lese Nominalspannung
13     Setze Nominalspannung auf        "9.50"
14     {nom} =                          Lese Nominalspannung
15     Ist ungefähr gleich                          {nom}, "9.50
                 ↪", "0.0"
16 [TEST]
17     Aktiviere automatische Speisespannungs Suchlauf
18
19     Ist Statusbit gesetzt        "10"
20     Warte für                    {delay_long}
21     Ist Statusbit nicht gesetzt "10"
22
23     {sp} =                       Lese Versorgungsspannung am
                 ↪Rotor
24     Ist ungefähr gleich          {sp}, {nom}, "0.25"
25 [TEARDOWN]
26     Setze Nominalspannung auf    {std_nom}
```

Listing A.6: *Test: automatischer Suchlauf*

```
1  [AUTHOR]     Joshua Jungen
2  [TESTNAME]   Überprüfung des TCU Testsignals
```

```
3
4  [LIB]        "TCUIVKeywordLibrary.jar" tcu
5  [VAR]        "config.var"
6
7  [SETUP]
8      Aktiviere das Feature Speed
9      Aktiviere das Testsignal des Controllers
10 [TEST]
11     Ist Statusbit gesetzt        "7"
12 [TEARDOWN]
13     Deaktiviere das Testsignal des Controllers
14     Deaktiviere das Feature Speed
```

Listing A.7: *Test: Testsignal TCU*

```
1  [AUTHOR]     Joshua Jungen
2  [TESTNAME]   Überprüfung des Rotor Testsignals
3
4  [LIB]        "TCUIVKeywordLibrary.jar" tcu
5  [VAR]        "config.var"
6
7  [REPEAT] 5
8  [SETUP]
9      {chA} = Lese Kanal A
10     {chB} = Lese Kanal B
11     Ist ungefähr gleich {chA}, "0", "5"
12     Ist ungefähr gleich {chB}, "0", "5"
13 [TEST]
14     Schalte das Testsignal des Rotors um
15     {ratedTorqueA} =    Lese Rated Torque von Kanal A
16     {chA} = Lese Kanal A
17 #   {chB} = Lese Kanal B
18     Ist ungefähr gleich {chA}, "[0.5*{ratedTorqueA}]", "
                  ↪[0.01*{ratedTorqueA}]"
19 #   Ist ungefähr gleich {chB}, "[0.5*{ratedTorque}]", "5"
20 [TEARDOWN]
21     Schalte das Testsignal des Rotors um
```

Listing A.8: *Test: Testsignal Rotor*

B. Globale Variablen

```
1 {delay_short} =          "500"
2 {delay_long}  =          "6000"
3 {delay_extra_long} =     "15000"
4 {delay_datasheet}  =     "35000"
```

Listing B.1: *Eine Beispieldatei mit globalen Variablen.*