

# Konzeptionierung automatisierter Tests für ein System zur Integration externer Massendaten

**Bachelorarbeit**

für die Prüfung zum

**Bachelor of Science**

des Studiengangs Informatik

an der Dualen Hochschule Baden-Württemberg Heidenheim

von

**Kathrin Ulmer**

September 2021

**Bearbeitungszeitraum**  
**Matrikelnummer, Kurs**  
**Ausbildungsbetrieb**  
**Erstgutachter**  
**Zweitgutachter**

12 Wochen  
8044845, TINF2018  
FNT GmbH, Ellwangen  
Dipl.-Ing. (FH) Marcus Fetzer  
Prof. Dr. Andreas Mahr

# Sperrvermerk

Die vorliegende Bachelorarbeit mit dem Titel *Konzeptionierung automatisierter Tests für ein System zur Integration externer Massendaten* enthält unternehmensinterne bzw. vertrauliche Informationen der FNT GmbH, ist deshalb mit einem Sperrvermerk versehen und wird ausschließlich zu Prüfungszwecken am Studiengang Informatik der Dualen Hochschule Baden-Württemberg Heidenheim vorgelegt. Sie ist ausschließlich zur Einsicht durch den zugeteilten Gutachter, die Leitung des Studiengangs und ggf. den Prüfungsausschuss des Studiengangs bestimmt. Es ist untersagt,

- den Inhalt dieser Arbeit (einschließlich Daten, Abbildungen, Tabellen, Zeichnungen usw.) als Ganzes oder auszugsweise weiterzugeben,
- Kopien oder Abschriften dieser Arbeit (einschließlich Daten, Abbildungen, Tabellen, Zeichnungen usw.) als Ganzes oder in Auszügen anzufertigen,
- diese Arbeit zu veröffentlichen bzw. digital, elektronisch oder virtuell zur Verfügung zu stellen.

Jede anderweitige Einsichtnahme und Veröffentlichung – auch von Teilen der Arbeit – bedarf der vorherigen Zustimmung durch den Verfasser und FNT GmbH.

Ellwangen, September 2021



---

Kathrin Ulmer

# Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema: *Konzeptionierung automatisierter Tests für ein System zur Integration externer Massendaten* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ellwangen, September 2021



---

Kathrin Ulmer

## **Abstract**

Mithilfe von Softwaretests kann die Qualität und Richtigkeit einer Software verifiziert werden. Eine solche Rückmeldung ist gerade im agilen Umfeld für das Team und bei der Kundengewinnung für die Vertrauensbildung wichtig. Mithilfe einer Automatisierung ist es möglich, die Tests regelmäßig und in kurzen Abständen durchzuführen, was manuelle Tester entlastet und für Flexibilität bei der Behebung auftretender Fehler sorgt.

Diese Arbeit befasst sich mit der Erstellung eines Konzepts für das automatisierte Testen der Funktionen des Command Integration Framework, welches ein System zur Integration externer Massendaten darstellt. Dazu wird zunächst der Ist-Zustand analysiert. Basierend auf den dort erarbeiteten Ergebnissen werden im Rahmen des Software Testing Life Cycles und der [ISO-Norm 29119](#) Methoden und Vorgehensweisen festgelegt, die zum definierten Soll-Zustand führen. Anschließend wird das Konzept prototypisch implementiert.

In der Evaluierung der Testergebnisse kann schließlich festgestellt werden, dass das gewählte Konzept die Anforderungen des Soll-Zustands erfüllt und nach vollständiger Implementierung trotz großem Aufwand einen hohen zeitlichen Vorteil und einen enormen Zugewinn an Qualitätssicherung erbringen wird.

## **Abstract**

Software tests can be used to verify the quality and correctness of software. Such feedback is important for the team, especially in an agile environment, and for building trust when acquiring customers. With the help of automation, it is possible to run the tests regularly and at short intervals, which relieves manual testers and provides flexibility in fixing any errors that occur.

This thesis deals with the creation of a concept for the automated testing of the functions of the Command Integration Framework, which is a system for the integration of external mass data. For this purpose, first the current state is analyzed. Based on the results obtained there, methods and procedures are determined within the framework of the Software Testing Life Cycle and the [ISO](#) standard 29119, which lead to the defined target state. The concept is then implemented as a prototype.

In the evaluation of the test results, it can finally be stated that the selected concept fulfills the requirements of the target state and, despite a great deal of effort, will yield a high time advantage and an enormous gain in quality assurance after full implementation.

# Inhaltsverzeichnis

<b>Abkürzungsverzeichnis</b>	<b>VI</b>
<b>Abbildungsverzeichnis</b>	<b>VII</b>
<b>Tabellenverzeichnis</b>	<b>VIII</b>
<b>Listings</b>	<b>IX</b>
<b>1 Einführung</b>	<b>1</b>
1.1 Firmenprofil . . . . .	2
1.2 Motivation . . . . .	2
1.3 Zielsetzung und Abgrenzung des Themas . . . . .	3
<b>2 Grundlagen</b>	<b>4</b>
2.1 FNT Command und das Command Integration Framework . . . . .	4
2.2 Qualitätssicherung durch Softwaretests . . . . .	8
<b>3 Ist-Analyse des Testens bei FNT</b>	<b>24</b>
3.1 Bestehende Richtlinien zum Testen von FNT Command . . . . .	24
3.2 Testen des <a href="#">CIF</a> . . . . .	28
<b>4 Soll-Zustand</b>	<b>30</b>
<b>5 Konzeptionierung</b>	<b>31</b>
5.1 Testplanung und -analyse . . . . .	31
5.2 Testentwurf . . . . .	57
5.3 Geplante Testdurchführung . . . . .	88
5.4 Geplanter Testabschluss . . . . .	90
<b>6 Umsetzung</b>	<b>91</b>
6.1 Einrichten der Entwicklungsumgebung . . . . .	91
6.2 Implementierung von <a href="#">API-Tests</a> . . . . .	97
6.3 Implementierung von <a href="#">GUI-Tests</a> . . . . .	103
6.4 Ergebnisdokumentation . . . . .	105
<b>7 Evaluierung</b>	<b>106</b>
<b>8 Fazit und Ausblick</b>	<b>112</b>
<b>Literatur</b>	<b>113</b>
<b>Anhang</b>	<b>125</b>

# Abkürzungsverzeichnis

<b>API</b>	Application Programming Interface
<b>BCG</b>	Boston Consulting Group
<b>BGE</b>	Business Gateway Entity
<b>BGW</b>	Business Gateway
<b>CI</b>	Continous Integration
<b>CIF</b>	Command Integration Framework
<b>CMDB</b>	Configuration Management Database
<b>CoP</b>	Community of Practice
<b>CSS</b>	Cascading Style Sheets
<b>E2E</b>	End-to-End
<b>GUI</b>	Graphical User Interface
<b>HTML</b>	Hypertext Markup Language
<b>HTTP</b>	Hypertext Transfer Protocol
<b>IEC</b>	International Electrotechnical Commission
<b>IEEE</b>	Institute of Electrical and Electronics Engineers
<b>ISO</b>	International Organization for Standardisation
<b>ISTQB</b>	International Software Testing Qualifications Board
<b>JAX-RS</b>	Jakarta RESTful Web Services
<b>JSON</b>	JavaScript Object Notation
<b>LOC</b>	Lines of Code
<b>NMS</b>	Network Management System
<b>PDCA</b>	Plan-Do-Check-Act
<b>POJO</b>	Plain Old Java Object
<b>REST</b>	Representational State Transfer
<b>STLC</b>	Software Testing Life Cycle
<b>UFT</b>	Unified Functional Testing
<b>URI</b>	Uniform Resource Identifier
<b>URL</b>	Uniform Resource Locator
<b>XLS</b>	Excel Spreadsheets
<b>XML</b>	Extensible Markup Language

# Abbildungsverzeichnis

2.1	Die drei Architekturschichten von <i>Command</i> . . . . .	4
2.2	Der allgemeine Aufbau des <i>CIF</i> . . . . .	5
2.3	Der <i>Software Development Life Cycle</i> . . . . .	10
2.4	Die Qualitätsmerkmale nach <i>ISO/IEC 25010</i> . . . . .	10
2.5	Der <i>Software Testing Life Cycle</i> . . . . .	14
2.6	Das <i>V-Modell</i> . . . . .	15
2.7	Der empfohlene Entwicklungskreislauf mit Smoke Tests. . . . .	20
2.8	Die agilen Testquadranten nach Crispin und Gregory. . . . .	21
2.9	Die Testdurchlauf-Kostenkurve für manuelle und automatisierte Tests. . . . .	23
5.1	Der Software-Produktlebenszyklus. . . . .	32
5.2	Die <i>BCG</i> Matrix. . . . .	33
5.3	Die ausgewählten Teststufen in der vorgesehenen Reihenfolge. . . . .	37
5.4	Die Testpyramide nach Mike Cohn. . . . .	42
5.5	Das Anwendungsfalldiagramm. . . . .	59
5.6	Das fachliche Log, dargestellt in der Oberfläche . . . . .	67
5.7	Das geplante Zusammenspiel der gewählten Testwerkzeuge. . . . .	89
6.1	Die erstellte Projektstruktur in einer Übersicht. . . . .	94
6.2	Die geplanten Stufen in der <i>Build-Pipeline</i> . . . . .	98
6.3	Die Oberfläche der <i>BGE</i> . . . . .	99
6.4	Die Details zu der gesendeten <i>REST</i> -Anfrage. . . . .	99
6.5	Die auf die Anfrage zurückgesendeten Daten. . . . .	100
6.6	Die erstellten Klassen. . . . .	101
6.7	Das Untersuchen der Login-Seite. . . . .	104
7.1	Der Implementierungsfortschritt graphisch dargestellt. . . . .	106
7.2	Die Durchführungszeiten graphisch dargestellt. . . . .	107
7.3	Die Statistik des Testdurchlaufs auf <i>TestRail</i> . . . . .	110



# Tabellenverzeichnis

5.1	Die Teststufe <i>Komponententests</i> . . . . .	38
5.2	Die Teststufe <i>Smoke Tests</i> . . . . .	38
5.3	Die Teststufe <i>Integrationstests</i> . . . . .	40
5.4	Die Teststufe <i>E2E Tests</i> . . . . .	40
5.5	Der Anwendungsfall <i>Einloggen</i> . . . . .	62
5.6	Der erste Testfall zum Anwendungsfall <i>Einloggen</i> über die <i>API</i> . . . . .	62
5.7	Der zweite Testfall zum Anwendungsfall <i>Einloggen</i> über die <i>GUI</i> . . . . .	63
5.8	Der Vorbedingungs-Testfall für Command . . . . .	64
5.9	Der Vorbedingungs-Testfall für das Datenlexikon . . . . .	65
5.10	Der Anwendungsfall <i>Job erstellen</i> . . . . .	66
5.11	Der Anwendungsfall <i>Job ausführen</i> . . . . .	69
5.12	Der Anwendungsfall <i>Adapter ausführen</i> . . . . .	70
5.13	Der Anwendungsfall <i>Delta berechnen</i> . . . . .	71
5.15	Der Anwendungsfall <i>Daten synchronisieren</i> . . . . .	74
5.16	Ein Testfall für <i>E2E</i> -Tests . . . . .	77

# Listings

6.1	Der grundlegende Aufbau einer Testfunktion in <i>JUnit</i> . . . . .	94
6.2	Ein <i>Gradle Task</i> , der nur <a href="#">API</a> Smoke Tests durchführen soll. . . . .	95
6.3	Der Aufbau einer <i>Stage</i> . . . . .	96
6.4	Der Aufbau der Klasse <i>QueryResponse.java</i> . . . . .	101
6.5	Ein Interface mit möglichen <a href="#">REST</a> -Anfragen. . . . .	102
6.6	Das Senden der <a href="#">REST</a> -Anfrage. . . . .	102
6.7	Der Login über <i>Selenide</i> . . . . .	105

# 1 Einführung

In einer immer schnelleren und digitaleren Welt nimmt die Größe und Anzahl von Softwarelösungen kontinuierlich zu. Hatte beispielsweise die Software *Photoshop* in ihrer ersten Version 1990 noch knapp 120000 Lines of Code (LOC), so wuchs deren Anzahl bis 2012 auf 450000 Zeilen an. Die LOC von *Microsoft Office* verdoppelten sich von 2001 bis 2013 von 25 Millionen LOC auf knapp 45 Millionen LOC. [McC15] 2015 beschäftigten sich einige Informatiker um Les Hatton mit diesem Wachstum und berechneten aus 2118 ihnen zugänglichen Projekten mit kontinuierlicher Code-Zunahme eine durchschnittliche jährliche Wachstumsrate für Software von 1,21. [HSv17] Dies bedeutet, dass die Anzahl der LOC eines Projektes sich innerhalb eines Jahres um 1,21 vervielfachen, sich die Software also in 48 Monaten verdoppelt. [vH12]

Dieses exponentielle Wachstum an LOC einer Software führt zu einem exponentiellen Wachstum an benötigten Ressourcen wie Hardware, Dokumentation und den Menschen, die an der Software und ihrer Pflege arbeiten. [HSv17] Zusätzlich werden gerade agile Vorgehensweisen immer beliebter, wodurch Arbeitszyklen kürzer getaktet werden. [OO19] Auch wenn die zunehmende Größe einer Software nicht unbedingt mit ihrer Komplexität korreliert, [Her+13][Dvo09] nimmt vor allem der Aufwand in den Bereichen des Management und der Strukturierung und Überwachung der Software zu. Dies birgt Risiken, welche sich schon früh in der Geschichte zeigen. 1996 explodierte die Rakete *Ariane 5* 37 Sekunden nach ihrem Start, nachdem sie ihre Flugbahn verlassen hatte. Der Grund waren Spezifikations- und Konstruktionsfehler in der Software. [Lio96] 2018 und 2019 stürzten zwei Flugzeuge der Firma Boeing aufgrund eines Fehlers im Steuerungsprogramm ab, was 346 Menschen das Leben und dem Maschinentyp 737-Max die Zulassung kostete. [Plu20] Ein aktuelles Beispiel sind die Softwarefehler, die sich seit ihrer Einführung in die *Corona Warn App*, einer App der deutschen Bundesregierung zur Bekämpfung der Ausbreitung von Sars-CoV-2, eingeschlichen hatten. [Fie20][Sil21] Diese Fehler und daraus resultierende Funktionsprobleme sehen viele Bürger unter anderem als einen Grund, die Wirksamkeit allgemein als eingeschränkt einzustufen. Dieses Misstrauen in die Qualität der Technik könnte Menschen davon abhalten, die App überhaupt zu nutzen. [Urb21] Dabei ist gerade für eine solche App eine hohe Teilnehmerzahl essenziell. [Dri20]

Wie hier deutlich wird, können durch kleine Fehler große Projekte, Menschenleben, der Ruf einer Software und unter Umständen das Ansehen des die Software produzierenden Unternehmens gefährdet werden. Ebenso wie gegebenenfalls große Schadenssummen entstehen.

Die Expertengruppe, die die Ursache der Explosion der *Ariane 5* erforschte, kam zu dem Schluss, dass der Fehler durch ausgiebige Analyse und strukturiertes Testen des Gesamtsystems unter realistischen Bedingungen hätte gefunden werden können. [Lio96] Das ausgiebige Testen einer Software ist ein wichtiger Faktor für die Qualitätssicherung eines Softwareprodukts und kann größere Fehler verhindern. Daher ist es wichtig, dem Testprozess genügend Zeit und Ressourcen zukommen zu lassen. Auch bei der Firma *FNT* wird deshalb Wert auf eine weitestgehend fehlerfreie und strukturiert getestete Software gelegt.

## 1.1 Firmenprofil

Die *FNT GmbH* ist ein mittelständisches Unternehmen mit rund 350 Mitarbeitern. Der Hauptsitz befindet sich in Ellwangen an der Jagst. Mit weiteren Standorten wie beispielsweise den USA, Singapur oder Russland ist *FNT* international tätig und hat weltweit insgesamt mehr als 500 Kunden, zu denen in Deutschland mehr als die Hälfte der DAX30-Konzerne gehören. Dabei ist *FNT* Marktführer in der Dokumentation und im Management von IT- und Telekommunikationsinfrastrukturen. [FNTtoJ] Das bekannteste und umfassendste Produkt ist die Software *FNT Command*. Diese ermöglicht es dem Nutzer, seine vorhandenen technischen Infrastrukturen vollständig und übersichtlich zu erfassen und zu optimieren. Mit ihren agilen Strukturen, der ausgezeichneten Ausbildung und ihren erfolgreichen Produkten wächst die *FNT GmbH* seit 1994 kontinuierlich und verspricht in Zukunft weitere Erfolge.

## 1.2 Motivation

*FNT* setzt auf eine vertrauensbasierte Kommunikation mit dem Kunden. Dessen Wünsche werden gehört und weitestgehend umgesetzt. Die korrekte Funktionalität und die hohe Qualität der dem Kunden zu liefernden Produkte hat einen hohen Stellenwert, der explizit in der Unternehmenspolitik festgehalten ist.

*FNT Command* besteht aus einem Standardmodul, das grundsätzliche Funktionen enthält, und vielen zusätzlichen Modulen mit erweiterten Funktionen, die über das Standardmodul hinaus erworben werden können. Eines dieser Zusatzmodule ist das Command Integration Framework (CIF). Dieses System bietet zentrale Funktionen für das Anbinden verschiedener externer Systeme und erleichtert dem Kunden die Migration seiner Daten in *Command*.

Im Rahmen einiger neuer Projekte und Anforderungen werden Änderungen am CIF vorgenommen, nachdem dieses länger ohne große Strukturänderungen auskam. Die Testabdeckung des gesamten Systems ist nicht gewährleistet, da lediglich vereinzelte Komponententests durch die Entwickler und manuelle Tests durch Tester auf der Benutzeroberfläche durchgeführt werden. Die zeitlichen Ressourcen reichen jeweils nur für die wichtigsten Funktionen. Häufig werden Fehler so erst bei der Releaseerstellung gefunden. Dies ist im Rahmen der steigenden Entwicklerzahl und des zunehmenden Ausbaus des CIF problematisch, vor allem in Hinblick auf die Qualitätssicherung, die trotz des hohen Aufwands der manuellen Tester nur unzureichend erfüllt wird.

## 1.3 Zielsetzung und Abgrenzung des Themas

Ziel der Arbeit ist die Entwicklung eines Konzepts für das flächendeckende automatisierte Testen der Funktionen des CIF, wo automatisierte Tests sinnvoll erscheinen. Dafür müssen die Funktionen des CIF zusammengetragen und analysiert werden. Die daraus abgeleiteten Testfälle sollen verschiedene Teststufen abdecken.

Die Testumgebung muss festgelegt, Testdaten und Testwerkzeuge ausgewählt werden. Tests sollen regelmäßig zu einem frühen Zeitpunkt und auf einer überprüften Datenbasis durchgeführt werden, sodass Fehler frühestmöglich entdeckt und behoben werden können. Die Testdurchführung sollte an das agile Umfeld angepasst werden.

Das daraus resultierende Testkonzept wird anschließend exemplarisch realisiert. Dafür werden einige der abgeleiteten Testfälle implementiert. So wird es möglich, konkrete Ergebnisse auszuwerten und das Konzept hinsichtlich Umsetzbarkeit und Nutzen zu evaluieren.

In der Arbeit wird aus Gründen der Lesbarkeit grundsätzlich die männliche Form verwendet. Es sind jederzeit auch weibliche Personen und andere Geschlechteridentitäten ausdrücklich mit einbezogen.

## 2 Grundlagen

Dieses Kapitel beschreibt Grundlagen, die für das Verständnis der Arbeit benötigt werden und kann von Lesenden mit bereits vorhandenem Hintergrundwissen bezüglich *FNT Command*, im Folgenden *Command* genannt, des *Command Integration Frameworks* und Softwaretests übersprungen werden.

### 2.1 FNT Command und das Command Integration Framework

Wie bereits in [Kapitel 1](#) erwähnt, ist *Command* eine Software für die Dokumentation und das Management von IT- und Telekommunikationsinfrastrukturen. Sie ermöglicht es, gesammelte Daten über die physikalische Ebene, beispielsweise in Form von Geräten, bis hin zur logischen Ebene, zum Beispiel in Form von Services, zu dokumentieren. Der Einsatz des jeweiligen Elements kann dabei direkt vermerkt oder zunächst geplant und anschließend realisiert werden. Aufgebaut ist *Command* aus drei Architekturschichten, wie in [Abbildung 2.1](#) dargestellt ist.

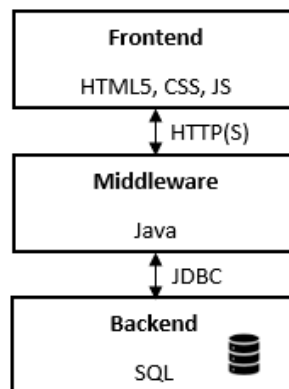


Abbildung 2.1: Die drei Architekturschichten von *Command*.

Die Validität und Verfügbarkeit von Daten ist in der heutigen Zeit ein entscheidender Faktor für Unternehmen. [Sou+19] Deshalb sollte für eine umfassende Transparenz und Flexibilität der Daten gesorgt werden, sodass diese jederzeit schnell und zuverlässig abrufbar sind und modifiziert werden können. In der Praxis ist es jedoch häufig der Fall, dass Unternehmen Daten in einer Vielzahl an Softwarelösungen speichern und so Daten dezentral abfragen müssen. [YYN11] Dies ist zeitaufwendig und kann zu Datenredundanz und einer niedrigen

Datenintegrität führen. Ein Zusammenführen der Daten, auch Datenintegration genannt, ist aber häufig gerade deshalb schwierig, weil die Daten in den verschiedenen Umgebungen verschiedene Formate aufweisen. [LS18]

*Command* ist eine Softwarelösung mitunter für genau dieses Problem. Daten sollen aus den Systemen des Kunden zentral in *Command* gespeichert werden können. Über eine webbasierte Bedienoberfläche kann der Nutzer die Daten anschließend übersichtlich verwalten. Für eben diese Datenintegration wurde das Command Integration Framework (CIF) als zusätzliches Modul implementiert. Wie in [Abbildung 2.2](#) dargestellt, kommt das CIF zwischen dem jeweiligen externen System und *Command* zum Einsatz und besteht aus mehreren Funktionsschichten.

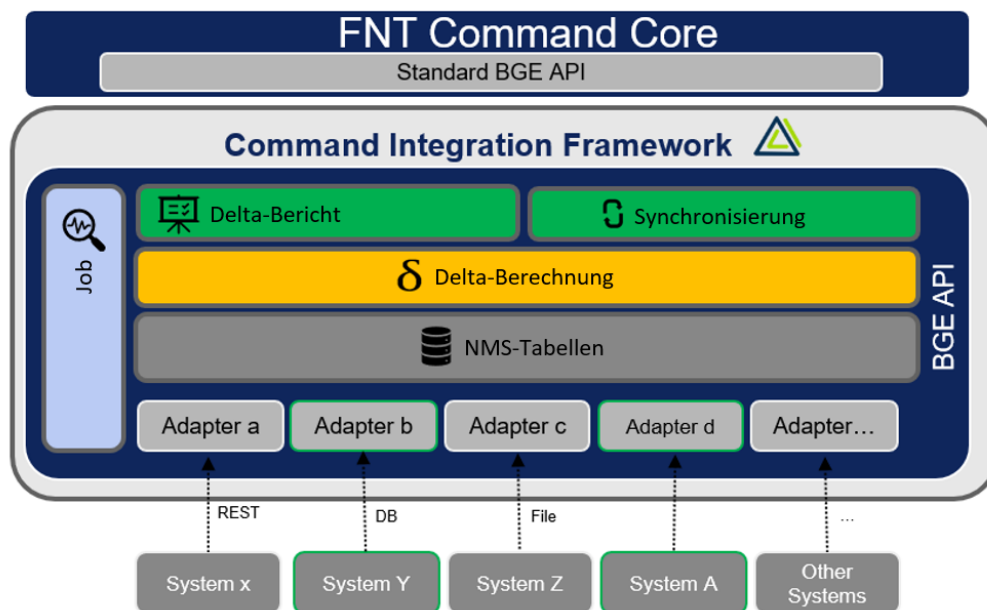


Abbildung 2.2: Der allgemeine Aufbau des CIF.

Die unterste Schicht ist der Adapter. Dieser führt den sogenannten *ETL*-Prozess durch: **E**xtrahieren, beziehungsweise Erhalten der Daten vom externen System, **T**ransformieren der Daten in das von *Command* vorgegebene Datenformat und **L**aden der Daten in die Datenbank. Adapter werden für jedes externe System neu geschrieben, da das Datenformat der externen Daten zunächst analysiert und der Transformationsschritt darauf angepasst werden muss. Es wird auf das Format der Daten eingegangen, so kann die Datenqualität gesichert werden. Hinzu kommt das Aufdecken von eventuell auftretenden logischen, syntaktischen und semantischen Fehlern in den Daten wie doppelt auftretende Einträge, fehlende Werte oder verwendete Synonyme für den gleichen Eintragungstypen. Dadurch wird die Homogenität der Daten innerhalb Commands gewährleistet. [Sou+19]

Die Daten werden nicht sofort in *Command* gespeichert, sondern in sogenannten *Staging-Tabellen*. Das hat den Grund, dass die Daten zunächst noch einmal bereinigt werden

müssen, denn trotz aufwendiger Implementierung des Adapters können einige Daten falsch importiert oder einige Datensätze bereits inhaltlich fehlerhaft aus dem externen System übernommen worden sein. Zudem ist es möglich, dass einige Daten bereits so in *Command* gespeichert sind, wie sie aus dem externen System geliefert werden, oder manche Datensätze nicht mit den bereits in *Command* vorhandenen Daten übereinstimmen. Die *Staging-Tabellen* werden im *CIF* und im Folgenden Network Management System (*NMS*)-Tabellen genannt. Dieser Name wurde den Tabellen gegeben, um zu unterstreichen, dass diese zum Verwalten der Objekte eines ganzen Netzwerks genutzt werden.

Im nächsten Schritt, der Delta-Berechnung, wird der in *Command* vermerkte Ist-Zustand mit dem Zustand der Daten aus dem externen System verglichen. Grundlage für diesen Vergleich sind dabei die ID und der Objekttyp des jeweiligen Datensatzes. Über diese werden Datensätze als neu, bereits in *Command* vorhanden, verändert oder gelöscht gekennzeichnet. Diese Kennzeichnung, auch *Delta-Fall* genannt, und die restlichen Ergebnisse des Vergleichs werden in Delta-Tabellen gespeichert und dem Benutzer zur Verfügung gestellt. Je nach Objekt kann es dabei bis zu 13 verschiedene Delta-Fälle geben, die verschiedene Aktionen der Objekte in den *NMS*-Tabellen mit den Objekten in *Command* beschreiben. Der Nutzer hat anschließend die Möglichkeit, die verglichenen Daten einzeln oder in Gruppen zu validieren und so fehlerhafte Daten auszuschließen, bevor diese in *Command* übernommen werden. So kann die Datenqualität ein weiteres Mal überprüft werden. [Sou+19]

Im Anschluss kann der Nutzer die Synchronisation starten, die die Daten den Delta-Fällen entsprechend endgültig nach *Command* speichert, in *Command* ändert oder aus *Command* löscht. Synchronisiert werden dabei nur die Daten, die im Vorhinein vom Nutzer validiert wurden. Über eine *Auto-Apply-Konfiguration* ist es zudem möglich, gewisse Daten automatisch zu validieren und sie ohne manuelle Bestätigung direkt nach *Command* zu laden.

Während der Adapter für jedes weitere System neu implementiert werden muss, laufen die Delta-Berechnung und die Synchronisation für alle Systeme automatisch ab. Auch der Start der Synchronisation kann automatisiert werden. Der Ablauf wird dabei von einem sogenannten Job gesteuert. Dieser ist über die Bedienoberfläche konfigurierbar und kann von dort gestartet werden. Ein Job hat dabei ein Set an bearbeitbaren Parametern, die einen Namen und einen Wert aufweisen. Durch diese kann der Job-Ablauf gesteuert, Pfade zu Dateien neu angegeben oder Daten mitgegeben werden. Der Job ruft dann je nach Parametern die Funktionen des *CIF* auf und liefert im Anschluss Informationen über den Ablauf der durchgeführten Aktivitäten. Dabei hat jeder Job verpflichtend folgende Parameter:

- *LOAD\_DATA*: Dieser Parameter ist vom Typ *Boolean* und gibt an, ob der Adapter aufgerufen werden soll.



- *CALCULATE\_DELTA*: Dieser Parameter ist ebenfalls vom Typ *Boolean* und gibt an, ob die Delta-Berechnung gestartet werden soll.
- *SYNCHRONIZE\_DATA*: Dieser Parameter, auch vom Typ *Boolean*, gibt an, ob die Synchronisation gestartet werden soll.
- *SOURCE\_SYSTEM*: In diesem Parameter wird der Name eines Quellsystems angegeben. Das ist meist der Name des externen Systems, aus dem die Daten integriert werden. Jeder Job arbeitet in einem Durchlauf für genau ein Quellsystem. Dieser Parameter existiert in *Command*, um die Daten der unterschiedlichen Kundensysteme noch voneinander unterscheiden zu können. Durch die Vergabe des Quellsystem-Parameters sind beispielsweise Massenoperationen auf Elemente dieses Systems möglich.

Es existieren meist weitere Parameter, die die Anwendung der Pflichtparameter auf jeweils eine oder eine Auswahl von folgenden Entitätsklassen beschränkt: Geräte, Services, Zonen, *Common Data*, welches unter anderem Personendaten beinhaltet, Datenkabel und Dynamische Daten.

Hinter *Command* und dem *CIF* steht ein durchgängiges Datenbankmodell, das auf dem Konzept einer Configuration Management Database (*CMDB*) beruht. Eine *CMDB* verwaltet Konfigurationseigenschaften und Beziehungen von Objekten im verwalteten System, ermöglicht eine Versionsverwaltung dieser Objekte und ist darauf ausgelegt, mit anderen Systemen fusionieren zu können. [YYN11]

Das *CIF* arbeitet dabei mit der *Command Business Gateway* (*BGW*) Application Programming Interface (*API*). Eine *API* ist eine Schnittstelle, über die Anwendungen und Programmteile miteinander in einem festgelegten Format kommunizieren können. [LA17] Die Business Gateway Entity (*BGE*) als *API* ist dabei *RESTful*. Das heißt, sie hält sich an die Architekturrichtlinien von Representational State Transfer (*REST*) für verteilte Systeme und Webservices, die vorgeben, wie diese zu entwickeln sind. *REST* nutzt hauptsächlich das Hypertext Transfer Protocol (*HTTP*) für Datenübertragungen. Jede Ressource hat eine Uniform Resource Identifier (*URI*), mit der sie eindeutig auffindbar ist. [Li11] Die *BGW* besteht aus *BGEs*, die die Tabellen der Datenbank widerspiegeln und die Middleware-Funktionen, die mit diesen arbeiten. Jede dieser Funktionen bekommt dabei einen eigenen Endpunkt der *URI* zugewiesen. Durch *HTTP*-Anfragen es möglich, Daten von der Datenbank zu erfragen, zu speichern oder weitere Funktionen auszuführen. Die in der *BGW* genutzten *HTTP*-Anfragen sind dabei *GET*, mit dem Ressourcen abgefragt werden können, und *POST*, mit dem Ressourcen unter anderem erstellt oder erweitert werden können. [Li11] Die ausgetauschten Daten sind dabei im JavaScript Object Notation (*JSON*)-Format. *JSON* ist ein schlankes Datenformat, in dem die Datensätze

so systematisch strukturiert sind, dass sie auch für den Menschen einfach zu lesen sind. [Mae12][Pez+16] Wie in [Abbildung 2.2](#) dargestellt, umfasst die [API](#) mit den BGEs das ganze [CIF](#). Die meisten Funktionalitäten des [CIF](#) können so direkt über die [BGW](#) erreicht und ausgeführt werden.

Das [CIF](#) ist momentan noch ein Zusatzmodul von *Command*, soll aber mit dem nächsten Release ein Standardmodul werden, das den Kunden zusammen mit den bisherigen Standardmodulen geliefert wird. Die Bedeutung des [CIF](#) wird entsprechend zunehmen.

## 2.2 Qualitätssicherung durch Softwaretests

Wie bereits aus [Kapitel 1](#) hervorgeht, ist das Testen einer Software für den Erfolg eines Produktes essenziell. Denn eine Software ist meist sehr komplex, was sie anfällig für Fehler und Fehlfunktionen macht, die auf den ersten Blick nicht auffallen und sich schnell häufen können. Als Fehler wird dabei ein Zustand oder ein Verhalten bezeichnet, der von dem erwarteten und als richtig betrachteten Zustand beziehungsweise Verhalten abweicht. [IEE93] Häufig wird zwischen einem „Fehlerzustand“ als falsch geschriebenem Code in der Software und einer „Fehlerwirkung“, die direkt oder indirekt aus einem Fehlerzustand entsteht und sich als Fehlverhalten in der Ausführung einer Software zeigt, unterschieden. [SL19] Dabei ist zu beachten, dass Softwaretests nur Fehlerwirkungen aufdecken. Die dahinterstehenden Fehlerzustände werden anschließend von einem Entwickler gesucht. [SL19] Diese Begriffsunterteilung wird im Folgenden größtenteils unterlassen. Der Begriff *Fehler* wird stellvertretend für Ursache und Wirkung stehen.

Fehler sind in verschiedene Klassen einzuteilen und können hohe Kosten verursachen:

- **Direkte Fehlerkosten** entstehen, wenn durch die Fehler in der Software beispielsweise Maschinen des Kunden ausfallen oder Daten verloren gehen.
- **Indirekte Fehlerkosten** entstehen, wenn der Ruf des softwareentwickelnden Unternehmens durch fehlerhafte Software geschädigt wird und so Kunden verloren gehen. Zudem können beispielsweise Vertragsstrafen anfallen.
- **Fehlerkorrekturkosten** entstehen dadurch, dass das Unternehmen die Fehler in der Software korrigieren und die Software neu ausliefern muss. Meist kommen Mitarbeiterschulungen hinzu.

[SL19]

Diese Kosten steigen exponentiell, je weiter die Entwicklung einer Software voranschreitet. Das liegt daran, dass ein früh entstandener Fehler meist weitere Folgefehler nach sich zieht,

die immer schwerer zu beheben sind, je weiter die Entwicklung gekommen ist. So kann es sein, dass sich die Entwicklungszeit verlängert und Arbeit aus früheren Entwicklungsphasen noch einmal überarbeitet werden muss, um Fehler zu beheben. Ist die Entwicklung schon soweit vorangeschritten, dass die Software bereits dem Kunden vorliegt, können zudem direkte und indirekte Fehlerkosten entstehen. [SL19]

Durch das Testen der Software sollen deren Funktionen deshalb möglichst früh vollständig überprüft und validiert werden. [ISO13a][KG13] Allgemein akzeptierte Grundsätze für das Testen von Software, an denen sich orientiert werden sollte, sind dabei:

- Testen soll nicht anzeigen, dass es keine Fehler gibt, sondern es sollen Fehler gefunden und auf diese aufmerksam gemacht werden. Werden keine Fehler gefunden, heißt das nicht automatisch, dass ein System fehlerfrei ist. [SBC12][SL19]
- Das vollständige Testen einer Software ist unmöglich. [SL19]
- Wo ein Fehler gefunden wird, werden meist weitere gefunden, Fehler sind nicht gleichmäßig verteilt. [SL19]
- Softwaretests werden unwirksam, wenn sie dauerhaft gleich auf einem ungeänderten System ausgeführt werden. Es muss also mit der Zeit das System geändert oder die Tests angepasst und erweitert werden. Sonst kann der Fehlschluss eintreten, dass keine Fehler mehr im System vorhanden sind. [SL19]
- Tests müssen immer an das System und das Projekt angepasst werden. Ein Test für ein System kann nicht automatisch für ein anderes System verwendet werden. [SBC12][SL19]

Das Testen ist dabei fester Bestandteil der Phasen des *Software Development Life Cycle*, wie in [Abbildung 2.3](#) dargestellt ist. [Wit19][Jam+16][RSS17] Dieser Lebenszyklus ist ein Modell aus aneinander grenzenden Phasen, die alle benötigten Aktivitäten für das erfolgreiche Entwickeln und Pflegen einer Software beinhalten sollen. Dieses Modell wird gerade in großen Projekten häufig angewandt. [SMG13]

### 2.2.1 Qualitätssicherung

Softwaretesten ist ein wichtiger Bestandteil der Qualitätssicherung einer Software. [KG13][Kle19] Neben dem Testen, also dem Erkennen von Fehlern in einer Software, gehören auch die Behebung der Fehler und die Verbesserung von Prozessen durch die gewonnenen Erkenntnisse zur Qualitätssicherung. [SL19][Kle19] Durchgeführt wird diese durch das Qualitätsmanagement. Hier werden organisatorische Aufgaben wie die Qualitätsplanung übernommen. Diese beinhaltet unter anderem das Festlegen der Qualitätsziele eines

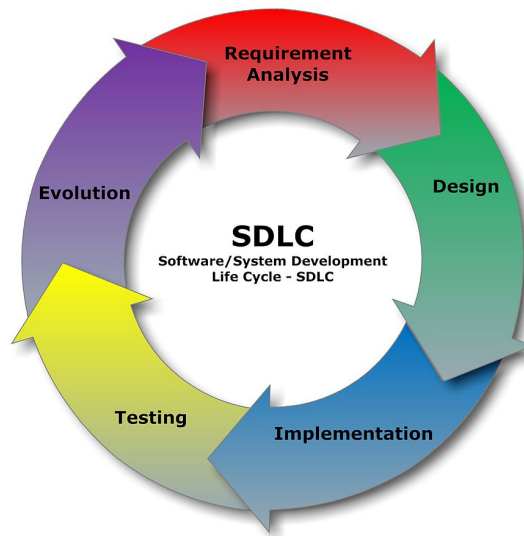


Abbildung 2.3: Der *Software Development Life Cycle*.

[Cli12]

Projekts und das Erstellen von Richtlinien zur Sicherung dieser. Ziel ist es, strukturierte Prozesse zu entwickeln, die bei Einhaltung der vorgegebenen Schritte ein Erreichen der Qualitätsanforderungen garantieren. Orientiert wird sich für das Qualitätsmanagement häufig an der Normenreihe um ISO 9000. [Kle19]

Wie Spillner und Linz in Kapitel 2 deutlich machen, umfasst die Qualitätssicherung darüber hinaus noch einen weiteren Raum:

*„Die Qualitätssicherung ist darauf ausgerichtet, Vertrauen in die Erfüllung der Qualitätsanforderungen zu erzeugen [...]“* [SL19]

Die Qualitätsmerkmale einer Software als Produkt werden zusätzlich in ISO/IEC 25010 festgelegt. Sie sind in Abbildung 2.4 dargestellt.

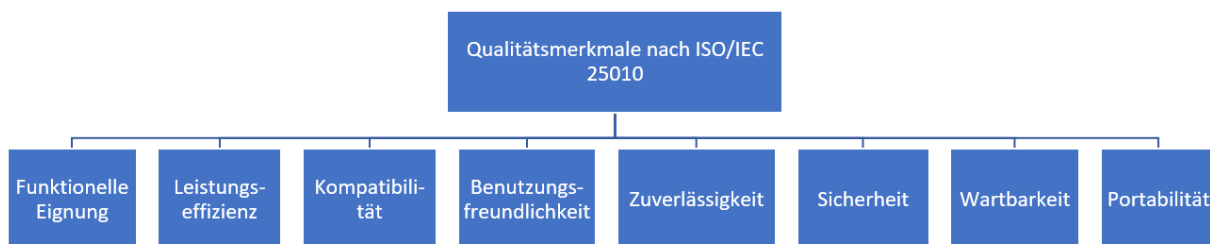


Abbildung 2.4: Die Qualitätsmerkmale nach ISO/IEC 25010.

Darstellung basierend auf [ISO13a]

Allgemein wird das Testen auf funktionelle Eignung als funktionelles Testen bezeichnet und das Testen der anderen sieben Qualitätsmerkmale als nicht-funktionelles Testen. [ISO13a] In Testprojekten sollten möglichst viele dieser Qualitätsmerkmale nachgewiesen werden, auf welche Merkmale jedoch Acht gegeben wird, kann sich von Projekt zu Projekt

unterscheiden und muss im Vorhinein festgelegt werden. [SL19][ISO13a] Die von Spillner und Linz genannte Erfüllung der Qualitätsanforderungen kann zudem nur ausreichend verifiziert werden, wenn eine vollständige Abdeckung der Anforderungen mit Testfällen vorliegt. [SIM11]

## 2.2.2 Testmanagement

Gerade weil das Testen einer Software so wichtig ist, ist hier eine geregelte Durchführung von großer Bedeutung. Der Testablauf und benötigte Ressourcen müssen effizient geplant werden. Testfälle sollen so erstellt werden, dass sie alle Anforderungen abdecken, [SIM11] müssen organisiert und durchgeführt werden. Testergebnisse und der Testfortschritt können mit den richtigen Mitteln so überwacht werden, dass sich der Testprozess immer weiter verbessern lässt. [Lou11] Diese Punkte müssen berücksichtigt werden, weshalb ein strukturiertes Testmanagement nötig ist.

Testmanagement wird ausführlich in der ISO-Norm 29119 dargestellt. Diese wurde in vier Teilen von der *International Organization for Standardisation (ISO)*, der *International Electrotechnical Commission (IEC)* und vom *Institute of Electrical and Electronics Engineers (IEEE)* als internationaler Standard für das Testen von Software verfasst. Es werden drei Testprozesse für das Testmanagement genannt:

1. Beim **betrieblichen Testprozess** ist es die Aufgabe des Unternehmens, allgemeingültige Vorgaben hinsichtlich des Testmanagements und des allgemeinen Testens zu erstellen. Dabei sollen eine betriebliche Testpolitik und eine betriebliche Teststrategie entstehen, die als Grundlage für alle Testprojekte im Betrieb gelten.
2. Der **Testmanagementprozess** wird in drei Schritte unterteilt:
  - a) Zuerst soll eine ausführliche **Testplanung** durchgeführt werden, in der ein Testplan und eine Teststrategie entstehen, die sich an den Ergebnissen des betrieblichen Testprozesses orientieren, aber an die Bedürfnisse und Eigenheiten des Projektes angepasst werden.
  - b) Als nächstes erfolgt das Festlegen der **Testüberwachung und -kontrolle**. Mit in der Teststrategie festgelegten Metriken soll es möglich gemacht werden, den Fortschritt des Testprozesses zu überwachen und rechtzeitig eingreifen zu können, falls etwas nicht nach Plan läuft.
  - c) Der dritte Schritt ist der **Testabschluss**. Hier wird festgelegt, was passiert, wenn die Tests das in der Testplanung festgelegte Abschlusskriterium erfüllen.

3. Die **dynamischen Testprozesse** finden ihre Anwendung in dynamischen Tests, also Tests, die eine Software durch Ausführung testen. Sie sind Teil der Implementierung der Teststrategie. Es wird das Testverfahren festgelegt und die Tests implementiert. Die Testumgebung wird aufgesetzt und die Tests anschließend ausgeführt. Die daraus entstehenden Testergebnisse bieten dann die Werte und Metriken, die für die Testüberwachung und für den Testabschluss benötigt werden.

[ISO13a]

Auch das International Software Testing Qualifications Board (**ISTQB**)® gibt klare Strukturen für das Testmanagement vor, die denen der ISO-Norm 29119 stark ähneln, diese jedoch weiter unterteilen. Das **ISTQB**® ist ein gemeinnütziger Verein, der weltweit angesehene Zertifikate für Tester ausstellt. Diese können beispielsweise ein *ISTQB® Certified Tester* werden. [IST20] Bei der **ISTQB**® ist der Testprozess in sieben Schritte aufgeteilt. Diese Schritte werden von einem Testmanagementprozess gesteuert, der an den Plan-Do-Check-Act (**PDCA**)-Zyklus angelehnt ist. [SL19] Der **PDCA**-Zyklus, oder Deming-Kreis, sieht eine ständige Optimierung des Prozesses vor, indem folgende vier Schritte periodisch durchgeführt und immer nach der letzten Phase abgesichert werden, sodass ein Prozess stetig verbessert wird und nicht zurückfallen kann: [Bre+20]

1. **Teststrategie festlegen:** Im **PDCA**-Zyklus die Phase *Plan*. Die hier erstellten Pläne müssen als klare Ziele formuliert und dokumentiert werden. Zu beachten sind dabei unter anderem Ressourcen und Zeitfenster. Eine erfolgreiche Planungsphase zeichnet sich durch das Festlegen der im nächsten Schritt folgenden Handlungsschritte aus. [Bre+20] Unter diese Phase fallen folgende Untertätigkeiten:
  - **Testplanung:** Es werden ein Testkonzept und eine Teststrategie entworfen, in denen Pläne zur Vorgehensweise und zu benötigten Ressourcen hinterlegt sind. [SL19]
  - **Testanalyse:** Es wird analysiert, was genau getestet werden soll. Testbedingungen werden zusammengestellt. [SL19]
2. **Testausführung planen:** Im **PDCA**-Zyklus die Phase *Do*. Diese Phase umfasst die konkrete Planung, Umsetzung, Steuerung, Ressourcenbereitstellung und Dokumentation der in der Planungsphase definierten Schritte. [Bre+20] Unter diese Phase fallen folgende Untertätigkeiten:
  - **Testentwurf:** Es wird festgelegt, wie getestet werden soll. Dafür werden Testumgebung, Testdaten und Testfälle festgelegt. [SL19]
  - **Testrealisierung:** Es werden alle Schritte vorgenommen, die noch nötig sind, um Tests auszuführen. Dazu gehört das Aufsetzen der Entwicklungsumgebung,

Strukturierung der Testfälle und Erstellung der Skripts für automatisierte Tests. [SL19]

3. **Testausführung initiieren und steuern:** Im PDCA-Zyklus die Phase *Check*. Der Fortschritt soll gemessen, geprüft und überwacht werden. Der Erfolg der Phase hängt deshalb maßgeblich mit der eindeutigen Zielsetzung der Planungsphase zusammen. [Bre+20] Unter diese Phase fällt folgende Untertätigkeit:

- **Testdurchführung:** Tests sollen ausgeführt werden. Dabei ist es wichtig, dass alle Schritte nachvollziehbar sind und aufgezeichnet werden. So können Fehler schnell identifiziert und Nachtests durchgeführt werden. Die Ergebnisse müssen ebenso ausgewertet werden. [SL19]

4. **Testergebnisse auswerten und berichten:** Im PDCA-Zyklus die Phase *Act*. Die Ergebnissen der vorherigen Phase sollen ausgewertet und Verbesserungsmaßnahmen abgeleitet werden. [Bre+20] Unter diese Phase fallen folgende Untertätigkeiten:

- **Testüberwachung und -steuerung:** Hier werden während des gesamten Prozesses der Fortschritt des Projektes und der Tests überwacht und festgehalten. [SL19]
- **Testabschluss:** Nach Ende der durchgeführten Testaktivitäten müssen die Ergebnisse ausgewertet und ein Abschlussbericht erstellt werden. [SL19]

Alle diese Schritte verlaufen im agilen Umfeld zyklisch, was sich wie in [Abbildung 2.5](#) darstellen lässt. Dieser Zyklus wird auch oft als Software Testing Life Cycle (STLC) bezeichnet. [OO19][Jam+16]

Betont wird in beiden vorgestellten Standards, dass sie lediglich als Grundlage und zur Orientierung dienen und projektspezifische Anpassungen unabkömmlich sind. [ISO13a][SL19]

### 2.2.3 Testarten

Testarten geben vor, von welchem Standpunkt aus Tests geplant werden können. In der Phase der Testplanung muss also entschieden werden, welche Testarten angewandt werden sollen. Sie bieten eine jeweils andere Sichtweise auf den Code und auf die zu testenden Funktionen. Je nach gewählter Testart können unterschiedliche Verfahren angewandt werden, mit denen sich Testfälle ableiten lassen.

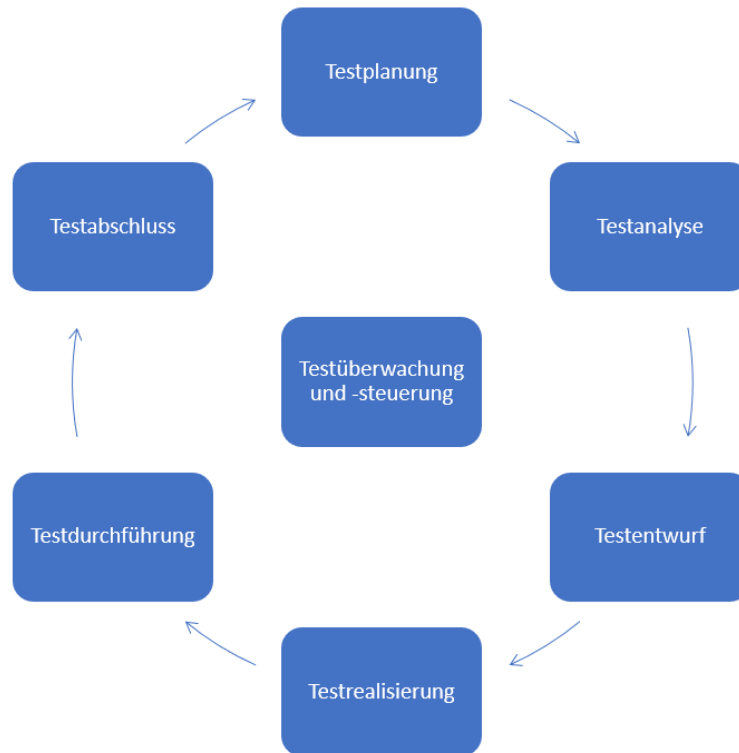


Abbildung 2.5: Der *Software Testing Life Cycle*.

Darstellung basierend auf [SL19]

### Blackbox-Tests

Eine Testart sind anforderungsbasierte Tests, die auch als *Blackbox-Tests* bezeichnet werden. Wie der Name andeutet, werden die Tests nach den Anforderungen an ein Projekt ausgerichtet. Die zu testenden Funktionen werden dabei als *Blackbox* gesehen. Es wird nur betrachtet, welches Ergebnis bei entsprechender Eingabe auftritt, dabei ist das interne Verhalten des Codes unerheblich. [ISO13a][SL19][KG13] Diese Testart wird hauptsächlich für funktionsorientierte Tests verwendet. Dabei ist es besonders wichtig, darauf zu achten, dass die Anforderungen an das Produkt vollständig und korrekt erfasst sind. Fehler können nur gefunden werden, wenn die Implementierung Fehler aufweist, die aus einem anderen Grund als aus falschen Vorgaben auftreten. [SL19]

### Whitebox-Tests

Eine weitere Testart sind strukturbasierte Tests, oder *Whitebox-Tests*. Hier wird die innere Struktur eines Systems oder Moduls getestet. Es zählt also nicht nur, was in das Testobjekt hineingeht und wieder herauskommt, sondern auch, wie das Testobjekt genau arbeitet. Whitebox-Tests benötigen Zugriff auf den Code eines Objekts. [ISO13a][SL19][KG13]



## Greybox-Tests

Zu erwähnen sind ebenso *Greybox-Tests*, die eine Mischung aus *Blackbox-* und *Whitebox-Tests* darstellen. In der Praxis kommen diese häufig vor, weil manche Testverfahren sich nicht eindeutig in eine der beiden Testarten einordnen lässt. [SL19]

## Erfahrungsbasierte Tests

Als vierte Testart wäre das erfahrungsbasierte Testen zu nennen. Diese Testart beschreibt das Testen nach Erfahrung und Intuition eines Testers. Da hier meist nicht strukturell vorgegangen wird, eignet sich erfahrungsbasiertes Testen vor allem als eine Erweiterung zu den oben genannten Testarten. Gerade wenn beispielsweise ein Fehler in den Anforderungen steckt, kann dies durch erfahrungsbasiertes Testen entdeckt werden. [ISO15][SL19]

### 2.2.4 Teststufen

Ebenfalls wichtig für den Testentwurf sind Teststufen. Sie werden bereits in der Planungsphase einbezogen. Die einzelnen Testfälle werden je nach Testobjekt in verschiedene Teststufen aufgeteilt. Die Stufen leiten sich ursprünglich aus dem *V-Modell* ab. Das *V-Modell* ist ein Referenzmodell für die strukturierte Planung eines Softwareprojekts. Obwohl dieses starre Modell nicht für das agile Umfeld gemacht ist, werden die darin dargestellten Teststufen auch im agilen Umfeld angewandt. [CDM18][HH08]

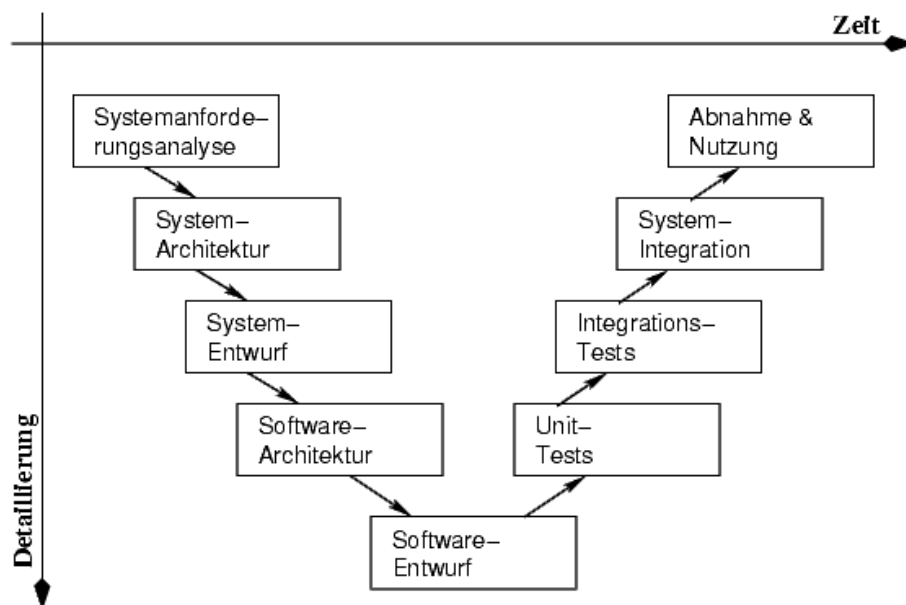


Abbildung 2.6: Das *V-Modell*.

[Sey06]

Jede Teststufe kann dabei eine eigene Testbasis und ein spezifischeres Testobjekt aufweisen, die das allgemeine Testziel der Teststufe entsprechend verfeinern, sodass Tests in dieser Stufe entsprechend ihres Zwecks ausgeführt werden können. Eine Teststufe wird erst dann ausgeführt, wenn die vorherige durchlaufen ist. [SL19][ISO13a]

### Komponententests

Komponententests, auch wie in [Abbildung 2.6](#) als Unit-Tests bezeichnet, testen die kleinsten Softwareeinheiten wie Klassen oder Module isoliert von anderen Einheiten auf ihre korrekte Funktion. Sie werden vor allen anderen Tests ausgeführt und bilden die Basis für erfolgreiche nachfolgende Tests. [SL19][Spi17] Komponententests sind zudem schnell und somit günstig zu implementieren, schnell in der Durchführung und verlieren selten ihre Funktionsfähigkeit bei Softwareänderungen. [Spi17][SBC12] Komponententests werden häufig stark unterschätzt und vernachlässigt, tragen aber nicht nur zu einer besseren Testbarkeit anderer Teststufen bei, sondern sorgen auch für ein klareres Code-Design, wie Klammer und Kern in ihrer Arbeit über Komponententests 2015 darlegen. [KK15] Komponententests sind meist Whitebox-Tests und werden vom jeweiligen Entwickler selbst oder von einem entwicklungsnahe Tester geschrieben. Dies hat den Grund, dass die neuen oder abgeänderten Klassen gekannt werden und die Tests sehr nah an den vorgenommenen Änderungen sein müssen. Es spart Zeit, wenn die Komponententests vom Entwickler entwickelt werden, der sich bereits mit dem Code und seinen Funktionen auskennt. Da hier aber auch Nachteile wie Befangenheit des Entwicklers bei seinem eigenen Code auftreten können, sollten die Komponententests bereits vor der eigentlichen Entwicklung entworfen werden. Das wird auch testgetriebene Entwicklung genannt. [KK15][SL19]

### Integrationstests

Integrationstests bilden die nächste Stufe nach den Komponententests und testen die Korrektheit von Kommunikation und Datenfluss zwischen den bereits einzeln getesteten Komponenten untereinander oder den Komponenten und verschiedenen Schnittstellen. [SL19][Win+13] Obwohl Integrationstests fest im V-Modell verankert sind, werden sie in der Praxis häufig übersprungen. Und das, obwohl sie gerade im agilen Kontext von großem Nutzen sind, wenn Schritt für Schritt neue Komponenten hinzugefügt und ihr Zusammenspiel mit anderen, bereits vorhandenen Komponenten getestet werden soll. [Win+13]

Integrationstests unterteilen sich hauptsächlich in drei Stufen:

1. **Klassenintegrationstests:** Das Zusammenspiel einzelner Klassen in einem Modul wird getestet. [Win+13] Es handelt sich also um *Whitebox-Tests*. [SL19]
2. **Komponentenintegrationstests:** Hier werden sowohl die Kommunikation unter einzelnen Modulen, als auch die Kommunikation von Modulen mit Datenbanken, Dateien oder Schnittstellen getestet. [Win+13] Sie sind als *Blackbox-Tests* umsetzbar. [SL19]
3. **Systemintegrationstests:** Diese Integrationstests testen die Integration von Modulen oder Teilsystemen in die Produktivumgebung oder eine Integration mit externen Systemen. Sie können auch als End-to-End (**E2E**)-Tests eines Teilsystems umgesetzt werden. [Win+13][SL19]

### Systemtests

Die Systemtests prüfen die Erfüllung von Anforderungen für ein ganzes System. Die Anforderungen werden hierbei aus Nutzersicht betrachtet. [SBC12][SL19] So kann direkt validiert werden, ob die Wünsche des Kunden hinsichtlich der Funktionen eines ganzen Systems erfüllt wurden. Die Testumgebung soll daher auch der Produktivumgebung möglichst ähnlich oder gleich sein. [SL19] Systemtests werden dabei häufig als **E2E**-Test realisiert. **E2E**-Tests beschreiben dabei einen Testvorgang, der eine bestimmte Funktion des Gesamtsystems von Beginn bis Ende vollständig testet. [Tsa+01][AV14]

### Akzeptanztests

Akzeptanztests, oder auch Abnahmetests, stellen die letzte Stufe im V-Modell dar. Akzeptanztests sind eine spezielle Form der Systemtests, werden im Gegensatz zu diesen aber nicht vom Entwickler oder Tester ausgeführt, sondern meist direkt vom Kunden, sodass dieser selbst prüfen kann, ob alle Vereinbarungen eingehalten wurden. [SL19][SBC12] Die Tests werden dabei nicht mehr auf einem Testsystem, sondern auf einem gesonderten Abnahme- oder direkt auf dem Produktivsystem getestet. [CG14]

### 2.2.5 Testmetriken

Um den Fortschritt und den Erfolg eines Projekts messen zu können, müssen Tests, Testergebnisse und der Testfortschritt objektiv messbar gemacht werden. Dafür gibt es die Testmetriken, mit deren Hilfe quantifizierte, vergleichbare Werte geschaffen werden können.

Sie sind essentiell und müssen bedacht festgelegt werden, sodass alle relevanten Aspekte zuverlässige Werte liefern. [SL19][Wit18] Deshalb sollte bei der Wahl der Metriken darauf geachtet werden, eine Datenerhebung möglichst einfach zu gestalten, eindeutige, nachvollziehbare Metriken auszuwählen und nur Daten zu erheben, deren Ergebnis verwendbar ist. Zusätzlich sollten Metriken reproduzierbar, objektiv und vergleichbar gehalten werden. [Wit18] Die Werte werden in der Phase der Testüberwachung und der Testdurchführung gesammelt und ausgewertet. Festgelegt, welche Werte genau gesammelt werden, wird bereits in der Testplanung. [ISO13a]

Die möglichen Messmethoden lassen sich hauptsächlich in folgende Kategorien einordnen:

- **Fehlerbasierte Metriken:** Hier kann die Anzahl der Fehler pro Testdurchlauf als Metrik dienen. Das kann verfeinert werden durch beispielsweise die Einteilung der Fehler in Fehlerklassen oder die Anzahl der Fehler kann relativ zu den Lines of Code oder der Testdauer gesetzt werden. [Wit18][SL19]
- **Testfallbasierte Metriken:** Nach einem Testdurchlauf weist jeder Testfall einen Status wie beispielsweise *Passed* oder *Failed* auf. Es kann gemessen werden, wie viele Testfälle sich in welchem Status befinden. [Wit18][SL19]
- **Testobjektbasierte Metriken:** Es kann gemessen werden, inwiefern die Anforderungen abgedeckt sind oder wie viel Prozent des Codes getestet wurde. [SL19]
- **Kostenbasierte Metriken:** Mit diesen Metriken kann gemessen werden wie viel der Testdurchlauf gekostet hat und wie viel der nächste vermutlich kosten wird. [Wit18][SL19]
- **Metriken zur Automatisierung** wie der Automatisierungsgrad oder Durchführungszeiten. [Wit18]

Damit Metriken für eine positive Verbesserung eines Projektes eingesetzt werden können, sollten sie regelmäßig erhoben werden. [Wit18]

## 2.2.6 Testen im agilen Umfeld

Der in [Abschnitt 2.2](#) dargestellte *Software Development Life Cycle* und das in [Unterabschnitt 2.2.4](#) erläuterte *V-Modell* wurden über Jahre erfolgreich eingesetzt. Ihre starre Vorgehensweise ist jedoch für heutige Prozesse nur bedingt einsetzbar. Denn diese fordern Flexibilität, Schnelligkeit und Effizienz. [AP18] Hinzu kommt die Tatsache, dass in beiden Modellen Tests erst gegen Ende des Projekts angesetzt sind, was wie in [Abschnitt 2.2](#) beschrieben, eigentlich zu spät ist. Im Jahr 2001 entwickelte eine Gruppe an Informatikern

agile Grundsätze, die eine iterativ-inkrementelle Herangehensweise an die Softwareentwicklung darstellten. Im agilen Umfeld sollte Fokus auf das Team und die Interaktionen der Teammitglieder gelegt werden. Hauptaugenmerk liegt auf den Anforderungen des Kunden und einer funktionierenden Software. Deshalb sollte bei einer agilen Entwicklung der Kunde regelmäßig mit einbezogen werden. So kann flexibel und schnell auf Veränderungen reagiert werden, was ebenfalls als Grundpfeiler der agilen Entwicklung gesetzt wurde. [Bec+01]

Es existieren verschiedene Frameworks, die diesen agilen Gedanken in Projekten umsetzen. Eine der am weitesten verbreiteten ist *Scrum*. Bei *Scrum* wird ein komplexes Projekt in Unteraufgaben zerlegt, die als *User Stories* im sogenannten *Product Backlog* hinterlegt werden. Diese Aufgaben werden in einem Sprint, der üblicherweise einen Monat lang ist, abgearbeitet. Ein Sprint kann aber auch sehr viel kürzer oder etwas länger sein. Dies muss an das jeweilige Projekt angepasst werden. Das *Scrum* Team wählt zu Beginn eines jeden Sprints ein Sprint-Ziel, das erreicht werden sollte. Dazu wird festgelegt, welche Unteraufgaben aus dem *Product Backlog* abgearbeitet werden. Der Fortschritt wird in täglichen Meetings, dem *Daily Scrum*, überwacht und am Ende eines Sprints wird in einem *Sprint Review* das Gesamtergebnis des Sprints bewertet. Die Sprints reihen sich aneinander und sollten aufeinander abgestimmt sein, sodass bestimmte Produktziele nach einer bestimmten Anzahl an Sprints erreicht werden können. [SS20]

Gerade in einem agilen Umfeld ist Testen von großer Bedeutung. In den unter Umständen kurzen Sprints soll die Funktionalität und die Qualität eines Produktes zuverlässig und früh verifiziert werden und das während des gesamten Sprints und nicht erst gegen Ende. [Nik20] Der in [Unterabschnitt 2.2.2](#) erläuterte [STLC](#) sollte bestenfalls in jedem Sprint erneut durchlaufen werden. Durch automatisiertes Testen wäre es möglich, vorhandene Funktionen wiederholt zu testen und damit in ihrer Funktion zu verifizieren, ohne dabei Zeit für Tests neuer Funktionen zu verlieren. Solche Tests werden auch Regressionstests genannt und sind essentiell für agile Projekte. Je kürzer der Iterationszyklus dabei ist, desto wichtiger werden automatisierte Regressionstests, um Ressourcen zu sparen und die exakte Ausführung sicherzustellen. [Coh10][SL19]

Ebenfalls häufig bei agilen Projekten eingesetzt werden Smoke Tests. Das sind End-to-End Tests, die die grundlegenden Funktionen eines Systems testen, ohne die das System nicht ordnungsgemäß funktionieren würde. Sie werden deshalb durchgeführt, bevor tiefergehende Tests verübt werden. Wenn Smoke Tests Fehler finden, werden nachfolgende Tests nicht ausgeführt, denn die Fehler sind so grundlegend, dass sie zuerst behoben werden müssen, wie [Abbildung 2.7](#) darstellt. Smoke Tests werden wegen ihrer Eigenschaften auch als *Build Verification Tests* bezeichnet. Gerade im agilen Umfeld sind Smoke Tests essenziell, denn Builds finden unter Umständen mehrmals am Tag statt. Durch Smoke Tests kann ein

Build so sehr schnell grundlegend verifiziert oder frühzeitig abgebrochen werden. Das spart Zeit und Kosten. [Cha14][McC96]

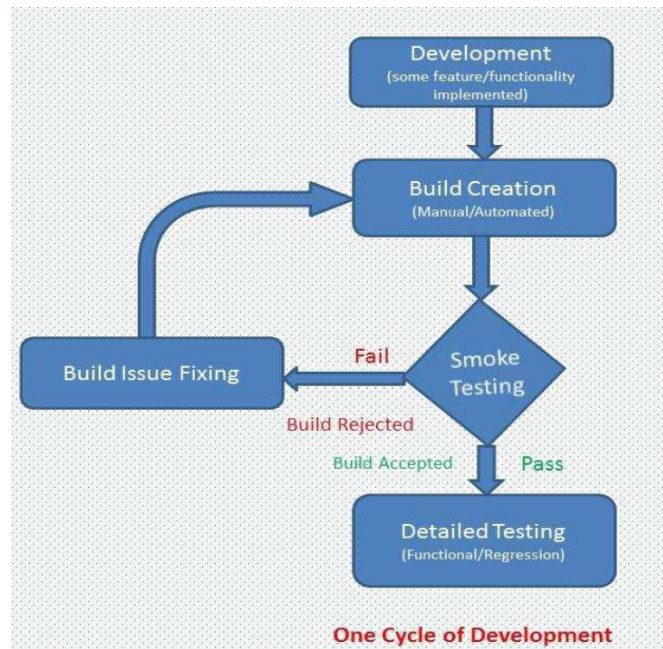


Abbildung 2.7: Der empfohlene Entwicklungskreislauf mit Smoke Tests.

[Cha14]

Während Testen vor der agilen Entwicklung lediglich als Mittel für das Aufdecken von Fehlern gesehen wurde, soll es im agilen Umfeld weitreichender betrachtet und eingesetzt werden. Dafür entwarfen Crispin und Gregory 2009 vier Testquadranten, die in [Abbildung 2.8](#) dargestellt sind.

Die rechten zwei Quadranten fokussieren sich dabei darauf, dass das Produkt überprüft und auf Fehler untersucht wird, während die beiden linken Quadranten den Fokus darauf setzen, das Team in ihrer Planung und ihrer Umsetzung wie der Implementierung abzusichern und zu unterstützen. Die beiden oberen Quadranten helfen zudem, zu überprüfen, ob das Team die Software so gestaltet, wie sie von Kunden gefordert ist, während die unteren beiden Quadranten die technische Umsetzung des Projektes verifizieren. Zusätzlich wird impliziert, dass das Testen einer Software nicht nur Aufgabe von Testern oder des Qualitätsmanagements ist, sondern das ganze Team involviert werden muss. So kann jedes Teammitglied seine Erfahrung und sein Einverständnis einbringen und lenkt seinen Blick weg vom reinen Implementieren der Software zur Sicherung der Qualität dieser. [CG14][Nik20][OO19]

Die Testquadranten müssen dabei nicht in einer gewissen Reihenfolge eingehalten werden. Ebenso dienen sie vor allem während der Testplanung als Stütze, um eine vollständige Testabdeckung eines Projektes zu überprüfen. Dabei ist es wichtig, auf die Anforderungen und Bedürfnisse des Projektes einzugehen und nicht zwanghaft alle in den Testquadranten aufgelisteten Testarten umzusetzen, wie Crispin und Gregory in Kapitel 6 ihres Buches hervorheben. [CG14]

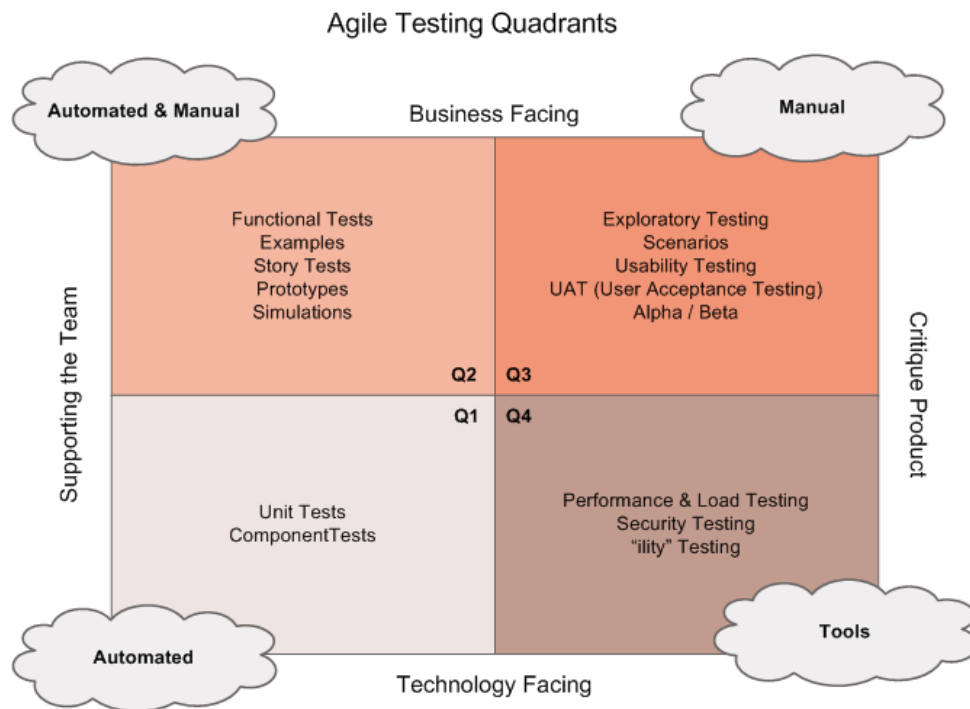


Abbildung 2.8: Die agilen Testquadranten nach Crispin und Gregory.

[Cri11]

Häufig im agilen Umfeld eingesetzt ist die Continuous Integration (CI). Diese beschreibt die regelmäßige und frühe Integration von neuem oder verändertem Code in das bereits bestehende System, was meist mithilfe eines *Build-Servers* stattfindet. Ein *Build-Server* baut dabei das System zusammen mit den Codeänderungen. Meist durchlaufen sie dabei eine sogenannte *Build-Pipeline*, die aus mehreren Schritten besteht. Tests werden häufig in diese Pipeline als einzelne Schritte eingebaut und erfolgen somit direkt mit dem *Build*, oder sie werden nach dem *Build* aufgerufen und durchlaufen. Letzteres ist auch durch manuelles Testen durchführbar, aber zeitaufwendiger. [Coh10][AP18]

## 2.2.7 Automatisiertes Testen

Beim Automatisierten Testen werden Tests durch bestimmte Werkzeuge ausgeführt und überwacht. [OO19][KG13] Es umfasst dabei nicht nur die Automatisierung von Testdurchläufen, sondern kann auch bei Bedarf und je nach Projekt die Testfallerstellung, Testüberwachung, Testdatengeneration, das Pflegen der Testumgebung und statische Tests automatisieren. [ISO13a][Wit19] Die Ausführung der einzelnen Testfälle übernimmt dabei ein sogenannter Testrahmen oder Testtreiber. Dieser ruft Methoden auf, versorgt sie mit Testdaten oder Platzhaltern und vergleicht meist auch direkt die resultierende Ausgabe mit einem Soll-Wert. [SL19][Win+13]



Das in agilen Projekten benötigte schnelle und regelmäßige Feedback über die Qualität der produzierten Software macht automatisiertes Testen fast unumgänglich. Tester müssen meist alte Testfälle immer wieder neu durchführen, um Änderungen zu verifizieren. Zusätzlich müssen für Neuerungen immer neue Testfälle erstellt und getestet werden. Die Masse nimmt weiter zu, sodass der Aufwand für einen manuellen Tester irgendwann so groß ist, dass die Tests zeitlich nicht mehr in einen Sprint passen. [Win+13][Wit19][Fui19] Auch dies ist ein eindeutiger Grund für die Einführung von automatisierten Tests. Eine solche Umstellung kann jedoch gerade bei bereits umfassenden Projekten wie dem CIF, die seit Jahren ausschließlich manuell getestet wurden, aufwendig sein. Bei solchen Projekten benötigt es eine genaue Evaluierung, ob der Aufwand sich schlussendlich lohnt. [Nik20]

Testfälle zu automatisieren ist nicht immer sinnvoll. Eine Automatisierung lohnt sich nur, wenn die Testfälle wiederholbar sind und auch in ihrer Praxis wiederholt werden sollen. Die Funktionen, die automatisiert getestet werden, sollten zudem regelmäßig vom System aufgerufen werden. Ein automatisierter Test für eine nebensächliche Funktion ist teurer als der Nutzen dafür. Die Testfälle und das zu testende System sollten zudem nicht zu häufig Änderungen unterliegen, sondern möglichst stabil bleiben, damit der Aufwand für Anpassungen nicht größer ist als der Aufwand für manuelle Tests. Zusätzlich ist eine Automatisierung dann sinnvoll, wenn für die Ausführung einer Funktion viele Kombinationen zur Verfügung stehen, die einen manuellen Tester viel Zeit kosten würden. [Wit19][OO19] Meist sind vor allem Testfälle, die mit Hardware zu tun haben oder Integrationen von externen System schwer oder unmöglich zu automatisieren. [Coh10]

Chancen, die sich mit der Automatisierung von Tests ergeben, sind unter anderem:

- Die wiederholte Ausführung mehrerer Testfälle in einem kürzeren Zeitraum ohne zusätzlichen Aufwand für Tester oder Entwickler. [SL19][OO19][Wit19]
- Mit Werkzeugen kann zum einen die kreative Arbeit wie das Erstellen von Testfällen, als auch die Testüberwachung unterstützt werden. Das fördert die Qualität und die Qualitätsüberwachung. [SL19]
- Der Testerfolg lässt sich sofort messen, indem das Ist-Ergebnis eines Testfalls direkt mit dem Soll-Ergebnis verglichen wird. [OO19]
- Die manuellen Tester können sich auf neue oder nicht-automatisierbare Testfälle konzentrieren, statt immer wieder die gleichen, meist stupiden Testfälle ausführen zu müssen. [OO19][Wit19]

Automatisierte Tests bergen aber auch Risiken, die vor allem dann auftreten, wenn sich nicht ausreichend mit dem Thema auseinandergesetzt wurde:



- Der Aufwand für die Einführung und Pflege automatisierter Tests wird häufig unterschätzt und zu wenig Ressourcen dafür eingeplant. Darunter leidet im Zweifelsfall die Qualität der Tests. [SL19][OO19]
- Auch Testfälle, die manuell günstiger umzusetzen sind, werden automatisiert. Das rechnet sich meist nicht. [SL19]
- Nach fertiger Implementierung aller geplanten Testfälle wird häufig davon ausgegangen, eine vollständige Testabdeckung erreicht zu haben und es wird nicht überprüft, ob Testfälle unter Umständen übersehen wurden. [OO19]
- Automatisierte Tests sind sehr anfällig für Änderungen in einem System. Je größer diese Änderungen, desto mehr Testfälle müssen angepasst und hinzugefügt werden. [OO19]
- Automatisierte Tests sind auf die geschriebenen Skripte festgelegt. Sonst durch manuelle Tests zufällig gefundene Fehler werden durch automatisierte Tests nicht mehr aufgedeckt. [OO19]

Der Aufwand für das Planen und Erstellen von automatisierten Tests ist zunächst sehr hoch. Dies liegt zum einen daran, dass häufig neue Testwerkzeuge eingeführt und Mitarbeiter geschult werden müssen. Zum anderen daran, dass vor allem Regressionstests so aufbereitet werden müssen, dass sie regelmäßig ausgeführt werden können. Dies ist auch sehr zeitintensiv. Automatisierte Tests rechnen sich demnach erst nach mehreren Testdurchläufen [SL19] und nur, wenn eine möglichst hohe Testabdeckung erreicht werden konnte. [Sin20][SIM11] Je häufiger dabei die automatisierten Testfälle ausgeführt werden, desto schneller rechnen sie sich. [Wit19] Dieser Sachverhalt ist auch in [Abbildung 2.9](#) abgebildet.

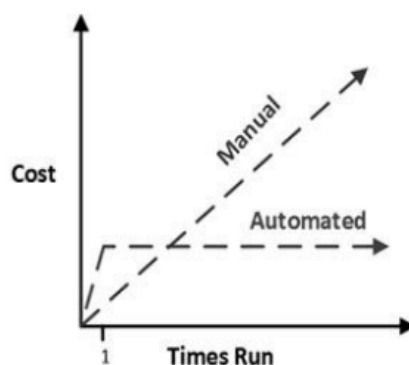


Abbildung 2.9: Die Testdurchlauf-Kostenkurve für manuelle und automatisierte Tests.

[OO19]

# 3 Ist-Analyse des Testens bei FNT

## 3.1 Bestehende Richtlinien zum Testen von FNT Command

Bei *FNT* gibt es sogenannte Communities of Practice (CoPs). Das sind Gruppierungen von Mitarbeitern, die sich für ein bestimmtes Thema interessieren und sich in dieser Gruppe sowohl weiterentwickeln, als auch aktiv an Projekten mitwirken möchten. In regelmäßigen Treffen findet Austausch von Wissen und das zentrale Festhalten von Vereinbarungen und Wissen statt. [PDE17] Ein CoP bei *FNT* ist dabei das *Quality Assurance CoP*. Dies ist eine Zusammenkunft hauptsächlich von ausgebildeten Testern und Qualitätsmanagern. In ihrer Zusammenarbeit entstanden zentral gültige Dokumente über die Richtlinien des Testens bei *FNT*. Das CoP hat damit den in der ISO-Norm 29119 beschriebenen betrieblichen Testprozess übernommen. Als Ergebnis des Prozesses stehen folgende Dokumente zur Orientierung bereit:

- Die *FNT* Test Politik
- Die Teststrategie von *FNT Command*
- Das Dokument „Best Practices and Testing Guidelines“

Da sich beim Testen eines Projekts in einem Unternehmen, wie in [Abschnitt 2.2](#) beschrieben, an den betrieblichen Vorgaben orientiert werden soll, sollen die groben Richtlinien der zur Verfügung gestellten Dokumente im Folgenden zusammengefasst wiedergegeben werden. Denn an diesen Richtlinien orientiert sich das Testen von Command und wird sich auch das Testkonzept für das [CIF](#) orientieren.

### 3.1.1 Qualitätsprinzipien von FNT

In der Testpolitik wird explizit aufgezählt, welche Qualitätsprinzipien bei *FNT* besonders beachtet werden und was Softwaretester darauf basierend beachten sollten:

- Der Fokus liegt auf der Kundenzufriedenheit. Produkte und Services sollen nach Kundenwünschen getestet und verbessert werden. Der Fokus des Testens soll auf der Funktion eines Produktes liegen.

- Testaktivitäten und -prozeduren sollen nachvollziehbar, wiederholbar, quantifizierbar, messbar und konsistent sein.
- Testprozesse müssen ausführlich definiert werden, denn redundanter Code und redundantes Testen sollen durch Dokumentation und Planung verhindert werden.
- Der Testprozess wird in einzelne Phasen unterteilt, die alle klar definierte Ziele mit Eingangs- und Endekriterien aufweisen.
- Die Testumgebung und Testdaten sollen einer Produktivumgebung gleichen.

[FNT20]

Die in [Abschnitt 2.2](#) aufgezählten allgemein akzeptierten Grundsätze für das Testen von Software werden in der Testpolitik noch einmal explizit aufgeführt. [FNT20]

#### 3.1.2 Formatvorgaben

Für das Schreiben von Testskripten gelten dieselben Code-Richtlinien, die für das allgemeine Coden bei *FNT* gelten. Das wären beispielsweise das Einhalten der englischen Sprache in der Namenvergabe bei Methoden und Klassen, sprechende Methodennamen und das Kommentieren im Code. Um die Einhaltung der Formatvorgaben überprüfen zu können, sollen Werkzeuge für eine statische Code-Analyse und regelmäßige Code-Reviews eingesetzt werden. [FNT21a]

Für Testfälle im Code wird eine erweiterte Namensgebung vorgeschrieben:

- Die Namen der Testmethoden sollten so sprechend sein, dass aus ihnen hervorgeht, was in diesem Testfall getestet wird und was als Ergebnis erwartet wird. Die ID des zugehörigen Testfalls sollte dieser Beschreibung vorangestellt sein und mit einem Unterstrich getrennt werden.
- Eine Testmethode sollte so kurz wie möglich gehalten werden, um den Test übersichtlich zu halten.
- Der Code sollte in *Packages* eingeordnet werden, die mit *com.fntsoftware.* beginnen.

[FNT21a]

### 3.1.3 Testmanagement

Der in [Abschnitt 2.2](#) beschriebene [STLC](#) soll für das Testmanagement der Tests bei *FNT* eingesetzt werden. [\[FNT20\]](#) *Scrum* wird als die Haupt-Softwareentwicklungsmethode genannt, an der sich die Entwicklung der Tests orientieren sollen. [\[FNT21b\]](#)

### 3.1.4 Testautomatisierung

Automatisierte Tests sollen nach ausgewählter Testart gebündelt werden und in diesen Bündeln ausgeführt werden. [\[FNT21a\]](#) Dabei sollten die Tests bestenfalls nicht auf dem Rechner des jeweiligen Testers, sondern in einer *Build-Pipeline* ausgeführt werden, wie bereits in [Unterabschnitt 2.2.6](#) beschrieben. Die Ausführung soll dabei in drei Hauptschritte unterteilt werden:

1. *Setup*: Testdaten werden kreiert, Vorbedingungen werden überprüft/erfüllt.
2. *Test*: Mit den erstellten Daten wird getestet.
3. *Teardown*: Die erstellten und bearbeiteten Testdaten und die Einstellungen in der Testumgebung werden auf einen definierten Anfangszustand zurückgesetzt.

[\[FNT21a\]](#)

Es wird empfohlen, schnelle Smoke Tests vor den anderen Tests wenn möglich einmal pro Tag auszuführen, mindestens aber bei jedem neuen *Build* des Projekts. Regressionstests sollen mindestens einmal pro Woche ausgeführt werden. Damit die Tests schneller durchlaufen werden, sollen die Testfälle entweder priorisiert oder eine parallele Ausführung ermöglicht werden. [\[FNT21a\]](#)

Die automatisierten Tests sollen, wie bereits erwähnt, kurz und übersichtlich gehalten werden. Zusätzlich soll kein redundanter Code erzeugt werden. Deshalb wird für einen Testfall folgende Struktur vorgeschlagen: Ein Testfall soll aus einzelnen Schritten bestehen. Diese Schritte sind Methoden, die außerhalb der Testklasse gehalten und gruppiert werden. Ein Testfall besteht somit möglichst nur aus sprechenden Methodenaufrufen und *Assertions*. *Assertions* sind dabei Verifikationen, meist am Ende eines Testfalls, mit denen überprüft wird, ob das Ergebnis des Falls dem erwarteten Ergebnis entspricht. Die einzelnen Methodenaufrufe sind dadurch als Schritte wiederverwendbar und die Testfälle übersichtlich. [\[FNT21a\]](#)

Die Dokumentation der automatisierten Tests soll über die Testfalldokumentation, Fehlertracking, eine festgelegte Teststrategie und einen Testplan erfolgen. Testdaten werden

während der Ausführung generiert oder statisch erstellt und zentral abgelegt. Die Testfallbeschreibungen sollen dabei auf die zugehörigen Testdaten verweisen. [FNT20]

### 3.1.5 Technischer Stand

Das CoP hat eine Sammlung an empfohlenen Technologien für das Testen von Command veröffentlicht. Die grundlegenden sind:

- Verwendete Testwerkzeuge:
  - *TestRail* für das Testfallmanagement
  - *JIRA* für das Projektmanagement und die Fehlerverfolgung
  - *Postman* und *SoapUI* für das Testen von APIs
  - *Bitbucket* als Versionskontrollsystem
  - *Eclipse* als Entwicklungsumgebung
- Verwendete Frameworks und Bibliotheken:
  - *Selenium* für die Browserautomatisierung
  - *RestEasy* für das Testen von APIs
  - *JUnit* als Testtreiber
  - *Gradle* als Projekt-Build-Tool
- Verwendete (Programmiersprachen):
  - *Java*
  - *Groovy* für die Gradle-Skripte
  - *SQL*
- CI-Build-Tool: *Jenkins*

[FNT21b]

## 3.2 Testen des CIF

Das CIF wird, abgesehen von automatisierten Komponenten- und Klassenintegrationstests, ausschließlich manuell von zwei ausgebildeten Testern getestet. Es existieren bisher 211 manuelle Testfälle, die in der von FNT empfohlenen webbasierten Testfallmanagementsoftware *TestRail* dokumentiert sind. Sie wurden auf Basis der Erfahrung der Tester und nach möglichen Aktionen auf der Bedienoberfläche erstellt und decken die Funktionen des CIF nicht vollständig ab. Gerade Funktionen, die mit vielen verschiedenen Kombinationen ausführbar sind, werden in diesen Testfällen nur mit wenigen Kombinationen aufgeführt.

Manuell werden die Tests auf verschiedene Weisen durchgeführt:

- Durch das Testen einer Funktion direkt auf der Oberfläche.
- Durch manuelle API-Tests, die über die in Abschnitt 3.1 genannten Werkzeuge *Postman* oder *SoapUI* durchgeführt werden.
- Durch direktes Verändern von Daten in der Datenbank selbst.

Die Testdaten sind dabei in Excel-Dateien hinterlegt, die vor jedem Testdurchgang über die Oberfläche in Command eingelesen und gespeichert werden müssen.

Als Testumgebung steht eine der Produktivumgebung sehr ähnliche Testinstanz bereit, die auf einem eigenen Server läuft und auf eine dafür angelegte Testdatenbank zugreift. Die Datenbank besteht aus denselben Tabellen wie die Datenbank der Produktivumgebung.

Wie Command wird auch das CIF im agilen Umfeld entwickelt. Konkret wird auch hier *Scrum* angewandt. Ist ein Entwickler fertig mit seiner Aufgabe, so fügt er den von ihm geschriebenen Code dem bereits bestehenden Code hinzu und lässt seine Änderung vom *Build-Tool Jenkins* bauen. In dieser *Build-Pipeline* wird der Code eingelesen, kompiliert, gebaut und einer statischen Code-Analyse unterzogen. Des Weiteren ist es üblich, dass der Entwickler für seine Änderung oder Neuerung am Code Komponententests verfasst, die die geschriebenen Funktionen validieren. Für die in Abbildung 2.2 dargestellten grundlegenden CIF-Funktionen existieren aktuell 67 Klassen. Die Anzahl an Klassen, die alle Funktionen und Entitäten des CIF abdecken, ist dreistellig. Aus dieser Menge an Klassen ergibt sich eine Anzahl im unteren vierstelligen Bereich an implementierten Komponententests für das CIF. Im Idealfall durchläuft der Code die zugehörigen Komponententests bereits lokal beim Entwickler, bevor dieser den *Build-Prozess* startet. Spätestens in der *Build-Pipeline* werden die Komponententests aber automatisiert durchlaufen. Nur wenn diese keine Fehler finden, wird der *Build* als erfolgreich angezeigt. Die manuellen Tester werden anschließend informiert und testen die Änderung wenn möglich sofort auf der Testinstanz. Zusätzlich zu den Komponententests wurden auch bereits Klassenintegrationstests mit einer Anzahl

im mittleren dreistelligen Bereich implementiert, wie sie in [Abschnitt 2.2](#) beschrieben sind. Auch diese werden in der *Build-Pipeline* durchlaufen. Jede Nacht wird zusätzlich zu den durch Änderungen angestoßenen *Builds* ein *Nightly-Build* gestartet, der den bestehenden Code so regelmäßig verifiziert.

Zusätzlich zu den funktionellen Tests werden regelmäßig *Performance Tests* durchgeführt. Das ist gerade bei einem System wie dem [CIF](#) essenziell, das im Regelfall mit über 200000 Datensätzen arbeitet. Für diese Tests wurde ein spezieller Job implementiert, der die Datensätze verarbeitet und die für die einzelnen Schritte benötigten Zeiten loggt. Der Job wird durch eine Zeitsteuerung gestartet, kann bei Bedarf aber auch manuell durchgeführt werden. Die Ergebnisse werden regelmäßig ausgewertet und besprochen.

## 4 Soll-Zustand

Folgender Zustand soll durch diese Arbeit für das automatisierte Testen des CIF erreicht werden:

- Die Funktionen des CIF sollen automatisiert getestet werden können. Dafür werden in der Konzeptionierung die Anforderungen des Kunden an das System und wie der Kunde das CIF nutzen wird gesammelt und daraus Testfälle mit möglichst hoher Abdeckung der Anforderungen abgeleitet. Dabei muss beachtet werden, dass unter Umständen nicht alle Testfälle automatisierbar sein werden. Diese werden entsprechend gekennzeichnet.
- Das Testkonzept ist an das agile Umfeld angepasst. Die Tests sollen dabei hauptsächlich als Regressionstests durchgeführt werden, um die bestehenden Funktionen des CIF regelmäßig verifizieren zu können.
- Die Tests sollen von einem zentralen Punkt aus aufgerufen werden können und automatisch durchlaufen werden. Bestenfalls im Rahmen der Continuous Integration.
- Die Tests und die Testplanung müssen ausreichend dokumentiert und die Dokumente an einem zentralen Ort zugänglich gemacht werden.
- Ein ausführlicher Bericht über die Testdurchläufe sollte jeweils erstellt und zentral abgerufen werden können. Ein Konzept über die regelmäßige Erstellung des Berichts sollte ebenfalls bestehen.
- Die in dieser Arbeit prototypische Umsetzung des Konzepts durch Implementierung einiger Testfälle sollte so erstellt werden, dass das Projekt jederzeit ohne allzu großen Aufwand von einem bisher nicht beteiligten Tester oder Entwickler übernommen und erweitert werden kann. Danach sollte auch das Konzept ausgerichtet werden.



# 5 Konzeptionierung

Das folgende Kapitel orientiert sich größtenteils an den in [Abschnitt 2.2](#) erläuterten Standards, da diese weithin anerkannt sind und eine solide Grundlage für ein vollständiges Konzept bieten. Wie vor allem in [ISO-Norm 29119](#) hervorgehoben wird, sollen neben diesen Standards vor allem auch die Ergebnisse des betrieblichen Testprozesses einbezogen werden. [\[ISO13a\]](#) In diesem Fall die drei in [Abschnitt 3.1](#) aufgezählten Dokumente, unter anderem die *FNT* Testpolitik.

Da in der *FNT* Testpolitik ausdrücklich auf den Software Testing Life Cycle verwiesen wird, soll das Konzept ebenfalls nach diesem strukturiert werden. Einige Konzepte der [ISO-Norm 29119](#) sollen zusätzlich einfließen und die Struktur verfeinern. Dieses Kapitel wird demnach zuerst mit der Testplanung und -analyse beginnen, wo festgelegt wird, was genau getestet wird, wie die Teststrategie aufgebaut ist und welche Ressourcen benötigt werden. Danach folgt der Testentwurf. Hier wird festgelegt, wie die Testfälle getestet werden sollen. Anschließend folgt in diesem Kapitel das Planen der Testdurchführung, der Testüberwachung und des Testabschlusses. [\[SL19\]](#)

## 5.1 Testplanung und -analyse

In der Testplanung und -analyse sollen die Rahmenbedingungen des Testkonzepts gesteckt werden. Dabei wird festgelegt, was auf welche Weise getestet werden soll und auf welcher Grundlage sich Testfälle erstellen und die Tests ausführen lassen. [\[SL19\]](#)[\[ISO13a\]](#)

### 5.1.1 Testobjekt

Zunächst sollte analysiert werden, was genau es zu testen gilt, um die Anforderungen und Funktionen richtig erfassen zu können. [\[ISO13a\]](#) Das zu testende Objekt ist das in [Abschnitt 2.1](#) beschriebene *Command Integration Framework*. Dieses ist bereits seit einigen Jahren im Einsatz und der Entwicklungsstand entsprechend fortgeschritten. Die Tests werden deshalb im Nachhinein die bereits implementierten Funktionen prüfen. Im Juli 2021 hat der Releasewechsel von Version 12 auf Version 13 stattgefunden. Während das *CIF* in der Version 12 bereits häufig verwendet und viele Fehler auf diese Weise gefunden wurden, ist die Version 13 noch fast nicht genutzt. Deshalb soll sich hauptsächlich auf das

CIF der Version 13 konzentriert werden. Eine mögliche Abwärtskompatibilität der Tests auf Version 12 sollte überprüft werden, ist aber nicht zwingend gefordert.

### 5.1.2 Wirtschaftlichkeit

Wie in [Unterabschnitt 2.2.7](#) bereits erwähnt, ist es gerade bei Softwareprodukten wie dem CIF, für die bisher fast ausschließlich manuelle Tests durchgeführt wurden, aufwendig, automatisierte Tests nachträglich einzuführen. Wie Nikolova in ihrem Artikel ausführt, muss daher zunächst evaluiert werden, ob sich das Hinzufügen von automatisierten Tests wirtschaftlich für ein Projekt lohnt. Dafür kann das Produkt zunächst hinsichtlich seiner Marktbedeutung für das Unternehmen bewertet werden. Dies ist unter anderem durch die Einordnung in eine Phase des Produktlebenszyklus der Software und in die Kategorien der Boston Consulting Group (BCG) Matrix durchführbar. [\[Nik20\]](#)

Da das CIF, wie bereits unter anderem in [Kapitel 1](#) und in [Abschnitt 2.1](#) dargelegt, gegenwärtig in seiner Funktion und seiner Bedeutung erweitert wird, befindet es sich in der in [Abbildung 5.1](#) dargestellten Phase des Wachstums. Es werden viele Ressourcen in die Entwicklung und Qualität des CIF gesteckt. Da das CIF zudem durch die Integration in das Standardprodukt einen höheren Marktanteil erlangt und die darin ausgeführte Datenintegration zunehmend an Bedeutung gewinnt, lässt sich das CIF zusätzlich als potentieller *Star* in der BCG Matrix einordnen. Diese ist in [Abbildung 5.2](#) dargestellt.

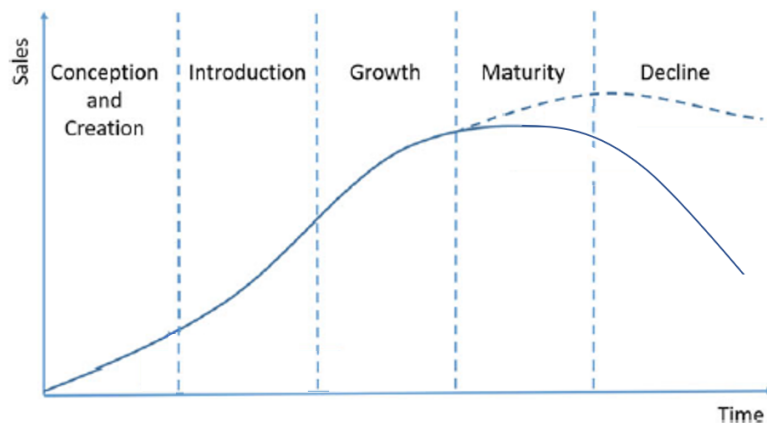


Abbildung 5.1: Der Software-Produktlebenszyklus.

[\[Cri11\]](#), Angepasste Darstellung

Zusätzlich kann eine Risikoanalyse durchgeführt werden, anhand derer die Notwendigkeit von automatisierten Tests evaluiert werden kann. Risiken werden dabei in Projekt- und Produktrisiken unterschieden. Als Projektrisiken werden Risiken bezeichnet, die mit dem Projektmanagement eines Projekts zusammenhängen und ein Projekt als ganzes scheitern lassen könnten. Produktrisiken sind Risiken, die aus Problemen mit der zu erstellenden

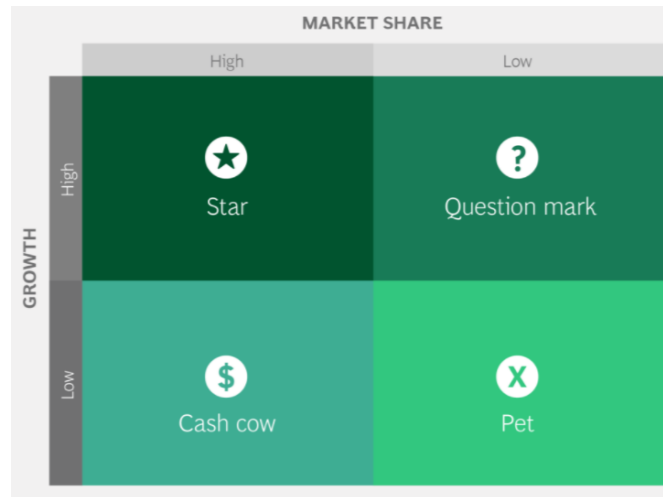


Abbildung 5.2: Die BCG Matrix.

[Bos21]

Software, in diesem Fall dem CIF, entstehen. [SL19][ISO13c] Es ergeben sich daraus folgende Projekt- und Produktrisiken, die die Funktionalität des CIF betreffen und aus dem aktuellen Zustand des manuellen Testens in Kombination mit dem Ausbau des CIF resultieren können:

### 1. Projektrisiken

- Fällt einer der beiden für das CIF zuständigen Tester aus, so kann es zu großen Verzögerungen beim Testen kommen. Das kann unter Umständen das ganze Projekt oder eine Auslieferung aufhalten.
- Manuelle Tests decken aufgrund des hohen zeitlichen Aufwands nicht alle Funktionen ab, die Testdokumentation ist daher unvollständig. Neue Tester können sich nicht daran orientieren und testen ebenfalls nicht vollständig.

[SL19][ISO13c]

### 2. Produktrisiken

- Das CIF funktioniert nicht, wie es versprochen wurde und erwartet wird und kann deshalb nicht für die Zwecke eingesetzt werden, für die es gedacht war.
- Die integrierten Daten sind fehler- oder lückenhaft.
- Bisher unentdeckte Fehler könnten bei einer Weiterentwicklung größere Fehlfunktionen verursachen und müssen unter Umständen zusammen mit der Weiterentwicklung aufwendig behoben werden.

[SL19][ISO13c]

Die hier aufgezählten Risiken nehmen mit einer Weiterentwicklung des CIF immer weiter zu. Da auch die Bedeutung des CIF angehoben wird, werden die durch die Risiken potentiell verursachten Schäden immer weitreichender. Die Risiken würden sich jedoch durch strukturiertes Testmanagement und organisierte, regelmäßig durchgeführte automatisierte Tests deutlich verringern lassen. [SL19]

Durch automatisierte Tests mit hoher Testabdeckung können hohe zukünftige Fehlerkosten vermieden werden. Da zudem weiterhin Ressourcen in das CIF und dessen Entwicklung investiert werden und das CIF zukünftig von großer Bedeutung für Command sein wird, sollten auch Ressourcen in die Qualitätssicherung und das Einführen von automatisierten Tests investiert werden. So kann langfristig für eine stabilere und schnellere Entwicklung des CIF gesorgt werden. [Nik20]

### 5.1.3 Testumfang

Da nun das Testobjekt eindeutig festgelegt und das Einführen von automatisierten Tests für dieses als wirtschaftlich sinnvoll erachtet wurde, kann festgelegt werden, wie umfangreich die Tests umgesetzt werden sollen. Wie auch aus Kapitel 4 hervorgeht, sollen die automatisierten Tests weitestgehend alle Funktionen des CIF abdecken und die Erfüllung der Anforderungen an diese nachweisen. Ausgeführt werden sollen also funktionelle Tests. [ISO13a]

Nicht-funktionelle Tests für folgende, in Unterabschnitt 2.2.1 genannte, Qualitätsmerkmale werden aus nachstehenden Gründen nicht mit einbezogen:

- Leistungseffizienz: Performance Tests, die dieses Qualitätsmerkmal überprüfen, werden, wie in Abschnitt 3.2 beschrieben, bereits regelmäßig mit großen Datenmengen durchgeführt und ausgewertet.
- Benutzungsfreundlichkeit: Die Oberfläche von Command wird nach einheitlich festgelegten Regeln entwickelt und designt, so auch die Oberfläche, mit der das CIF steuerbar ist. Die Benutzungsfreundlichkeit und Barrierefreiheit wird von bewusst für diese Tests aufgestellten *User Experience* Teams getestet.
- Zuverlässigkeit, Sicherheit, Wartbarkeit, Portabilität und Kompatibilität: Diese fünf Eigenschaften werden bereits in den Standardtests für Command überprüft. Sie decken dabei auch das CIF ab.

Damit beschränkt sich der Testumfang für dieses Projekt hauptsächlich auf die beiden linken Testquadranten von Crispin und Gregory, für die eine agile und automatisierte Herangehensweise vorgesehen ist. [CG14]

### 5.1.4 Testbasis

Um basierend auf dem festgelegten Testobjekt im vorgesehen Umfang Tests entwickeln zu können, werden Dokumente herangezogen, die die nötigen Informationen für die Testplanung und das Testdesign beinhalten. Diese werden auch als Testbasis bezeichnet. [ISO13a][SL19] Da die Funktionen des CIF überprüft werden sollen, sind vor allem Dokumente wichtig, die die einzelnen Module und Funktionalitäten des CIF beschreiben. Ein Lasten- oder Pflichtenheft, in dem solche Informationen üblicherweise zu finden sind, existiert für dieses Projekt nicht, da das CIF, wie bereits erwähnt, agil entwickelt wird. Die Anforderungen, Spezifikationen und einzelnen Funktionen sind hier über verschiedene *User Stories* verteilt. Zusammengefasst sind diese jedoch in der *CIF Spezifikation*, die in ihrem Aufbau und der Beschreibung der einzelnen Funktionen des CIF einem Lastenheft nahe kommt. Zusätzlich herangezogen werden kann das *CIF Trainingshandbuch*, das Kunden Schritt für Schritt durch alle Funktionalitäten des CIF führt. Aus diesen Dokumenten lassen sich die Anforderungen an das CIF vollständig ableiten und die Tests können darauf basierend entwickelt werden.

### 5.1.5 Testarten

Um Tests so entwickeln zu können, wie es für das Testobjekt und den festgelegten Testumfang benötigt wird, muss zunächst festgelegt werden, welche Testarten angewandt werden sollen. Denn jede Testart gibt eine andere Vorgehensweise für den Testentwurf vor. [SL19][ISO13a] Eine Festlegung auf eine oder mehrere Testarten wird also früh benötigt, um das Testkonzept daran ausrichten zu können.

Der Testumfang sieht ausschließlich funktionelle Tests vor. Deshalb soll hauptsächlich das in [Unterabschnitt 2.2.3](#) beschriebene *Blackbox-Testen* als Testart verwendet werden. Vor der Erstellung von Testfällen sollten die in der Testbasis beschriebenen Funktionen daher noch einmal auf Vollständigkeit untersucht werden. [SL19]

Die in [Abschnitt 3.2](#) beschriebenen, bereits implementierten Komponententests des CIF fallen unter die Kategorie der *Whitebox-Tests*. Für die noch zu erstellenden Testfälle werden *Whitebox-Tests* ausgeschlossen, da eine anforderungsbasierte Herangehensweise ausreichend ist, um die Funktionen des CIF abzudecken, ein Blick in den Code selbst ist dabei nicht

nötig. Aus diesem Grund wird auch die Mischung aus Blackbox- und Whitebox-Tests, die Greybox-Tests, außen vor gelassen.

Das erfahrungsbasierte Testen soll auch für das [CIF](#) als Testart eine Rolle spielen. Es wird vermutlich nicht möglich sein, alle Testfälle zu automatisieren, was dazu führt, dass weiterhin manuelle Tests durchgeführt werden. Die manuellen Tester weisen viel Erfahrung in dieser Testart auf und sind dazu angehalten, die von ihnen während manuellen Tests zufällig gefundenen Testfälle, zusätzlich zu den bereits durch die Anforderungen festgelegten Testfällen, zu dokumentieren. Diese sind anschließend zu ergänzen. Für das Herleiten der grundlegenden Testfälle für das automatisierte Testen wird das erfahrungsbasierte Testen aber nicht verwendet.

### 5.1.6 Teststufen

Die zu erstellenden Testfälle sollen wie in [Unterabschnitt 2.2.4](#) beschrieben in Teststufen eingeteilt werden. Wie bereits ausgeführt, basiert jede Teststufe auf einem anderen, spezifischen Testobjekt, woraus eine etwas andere Herangehensweise für das jeweilige Testen resultiert. Innerhalb dieser Teststufen können die Testfälle später gruppiert und gesammelt ausgeführt werden. [\[ISO13a\]](#)

Es wurden insgesamt fünf Teststufen festgelegt, die aufeinander folgen sollen. Eine Teststufe wird erst dann ausgeführt, wenn die vorherige durchlaufen ist. Sie sind in [Abbildung 5.3](#) dargestellt. Die Auswahl der für das [CIF](#) anzuwendenden Teststufen erfolgte dabei zum einen aus den traditionellen Teststufen, aber auch aus den erläuterten agilen Ansätzen. Zur Überprüfung der Vollständigkeit wurden auch die agilen Testquadranten nach Crispin und Gregory aus [Unterabschnitt 2.2.6](#) herangezogen. Die gewählten Teststufen werden im Folgenden erläutert und die Gründe für die Wahl der jeweiligen Teststufe dargelegt. Zur Verdeutlichung werden jeweils die Testbasis, das Testziel und das Testobjekt aufgeführt.

Sollten zukünftig weitere Funktionen und somit Testfälle hinzugefügt werden, müssen diese ebenfalls zunächst in diese Stufen eingeordnet und ebenso umgesetzt werden wie beschrieben.

### Komponententests

Komponententests testen, wie bereits in [Unterabschnitt 2.2.4](#) erläutert, die kleinsten Einheiten eines Projekts. Sie sind deshalb unumgänglich, denn wenn in den Einzelteilen bereits Fehler stecken, die nicht gefunden werden, kann nicht davon ausgegangen werden, dass das System als Ganzes fehlerfrei funktioniert. In den agilen Testquadranten werden sie

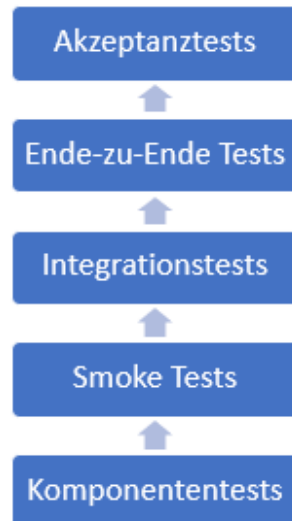


Abbildung 5.3: Die ausgewählten Teststufen in der vorgesehenen Reihenfolge.

den automatisierten Tests zugeordnet, die auch das Team dahingehend unterstützen sollen, die grundlegenden technischen Funktionen früh zu verifizieren und die Vorgehensweise der Entwickler zu unterstützen. [CG14]

Testbasis	Als Testbasis dient die in <a href="#">Unterabschnitt 5.1.4</a> genannte Testbasis.
Testziel	Es sollen Fehler in den Abläufen von Modulen gefunden werden, die von den Anforderungen an eine Klasse und ihre Funktionen abweichen. Es ist auch möglich, einzelne grobe Abweichungen von vorhandenen Designdokumenten zu finden. [SL19]
Testobjekt	Die erstellten Klassen sollen auf ihre Funktionalität und der zugehörige Sourcecode auf Korrektheit überprüft werden.
Verantwortung/ Entwicklung	Während der Entwicklung des <a href="#">CIF</a> wurden vom jeweiligen Entwickler bereits umfassende Komponententests erstellt, wie bereits in <a href="#">Abschnitt 3.2</a> beschrieben wurde. Die Entwicklung erfolgte dabei meist, wie in <a href="#">Unterabschnitt 2.2.4</a> empfohlen, testgetrieben. Daran wird auch weiterhin festgehalten und bei Neuerungen oder Änderungen Tests sukzessiv hinzugefügt.
Ausführung	Ausgeführt werden die Tests zum einen lokal beim Entwickler, zum anderen automatisiert auf dem <i>Build-Server</i> , wenn der Entwickler den erstellten Code in das Gesamtsystem integriert. [SL19] Für die Komponententests entspricht der Ist-Zustand also bereits dem Soll-Zustand und wird deshalb diesem Konzept ohne Änderungen hinzugefügt.
Eintrittskriterien	Da bereits alle vorhandenen Funktionen mit Testfällen abgedeckt sind, werden Komponententests dann erweitert und ausgeführt, wenn Änderungen am <a href="#">CIF</a> geplant sind.

Austrittskriterien	Alle Komponententests sind vollständig und fehlerfrei ausgeführt.
--------------------	---

Tabelle 5.1: Die Teststufe *Komponententests*

## Smoke Tests

Smoke Tests gehören nicht zu den erläuterten klassischen Teststufen. Da sie aber, wie in [Unterabschnitt 2.2.6](#) beschrieben, grundlegende Funktionen testen bevor ausladende und zeitraubende Tests durchgeführt werden, sollen sie auch bei den [CIF](#)-Tests zum Einsatz kommen. Da auch für Smoke Tests Eintritts- und Austrittskriterien gelten und diese eine abgeschlossene Einheit bilden, werden sie hier auch als Teststufe betrachtet.

Testbasis	Als Testbasis dient die in <a href="#">Unterabschnitt 5.1.4</a> genannte Testbasis.
Testziel	Fehler in den elementaren Funktionen des Systems sollen gefunden werden.
Testobjekt	Die grundlegenden Abläufe des Gesamtsystems.
Verantwortung/ Entwicklung	Die Tests werden von Testern entwickelt oder von Entwicklern, die nicht selbst am <a href="#">CIF</a> programmiert haben. Bei der Implementierung sollte darauf geachtet werden, ganze Durchläufe der zu testenden Funktionalität zu entwickeln, dabei aber nicht auf Details einzugehen. Diese werden zu einem späteren Zeitpunkt getestet. [ <a href="#">Cha14</a> ]
Ausführung	Ausgeführt werden sollen die Tests in der <i>Build Pipeline</i> , sobald Änderungen am <a href="#">CIF</a> in das Gesamtsystem integriert werden.
Eintrittskriterien	Die Komponententests wurden fehlerfrei durchlaufen und die zu den Smoke Tests gehörenden Testfälle wurden vollständig ausgearbeitet.
Austrittskriterien	Die Smoke Tests sind vollständig und fehlerfrei ausgeführt. Die Testergebnisse wurden gesammelt, ausgewertet und zur Verfügung gestellt.

Tabelle 5.2: Die Teststufe *Smoke Tests*

## Integrationstests

Integrationstests bilden die nächste Stufe nach den Smoke Tests und testen, wie bereits in [Abschnitt 2.2](#) beschrieben, die Kommunikation von Modulen untereinander. Auch dies ist eine wichtige Teststufe, denn der Sprung vom Testen von einzelnen Modulen zum Testen des Gesamtsystems wäre viel zu groß und birgt einige Risiken, die durch die schrittweise Integration der Module ausgeschlossen werden können. Bei der Zusammensetzung der Komponenten bieten sich verschiedene Strategien an, die angewandt werden können, zu nennen sind folgende:



1. *Ad-Hoc-Integration*: Tests werden in der Reihenfolge ausgeführt, in der Komponenten fertiggestellt wurden. [Win+13][SL19]

Diese Integrationsstrategie eignet sich zunächst nicht für dieses Projekt, da die Hauptkomponenten des CIF längst fertiggestellt sind und somit keine Umsetzungsreihenfolge mehr existiert. Bei einer späteren Erweiterung des CIF und damit der Tests kann diese Strategie jedoch angewandt werden.

2. *Big-Bang-Integration*: Zu dem Zeitpunkt, an dem das System fertig implementiert ist, werden alle Komponenten zusammengefügt und als Ganzes miteinander getestet. Diese Methode sollte möglichst vermieden werden, da bereits während der Entwicklung von Komponenten Zeit in die Entwicklung der Tests gesteckt werden sollte. Zudem werden alle Fehler durch diese Methode geballt und ungeordnet auftreten, was ein Beheben dieser erschwert. [Win+13][SL19]

De facto spiegelt aber der Ist-Zustand des CIF wie in Abschnitt 3.2 beschrieben eine solche Situation wieder: Das System existiert bereits und weist zahlreichen Komponenten auf, jedoch wurde noch keine Zeit in die Entwicklung von Tests gesteckt, was nun nachgeholt werden muss. Das Testen der Komponenten sollte dabei jedoch nicht durcheinander oder zufällig, sondern mit einer der nachfolgenden, inkrementellen Strategien geschehen.

3. *Top-Down-Integration*: Die Tests werden mit einer übergeordneten Komponente begonnen, die weitere Komponenten aufruft. Diese Komponenten werden Schritt für Schritt hinzugefügt und müssen anfangs unter Umständen mit Platzhaltern ersetzt werden. [SBC12][Win+13][SL19]

Diese Strategie wird hauptsächlich angewandt werden. Dies hat den Grund, dass die meisten Funktionen des CIF, wie in Abschnitt 2.1 beschrieben, mithilfe eines Jobs ausgeführt werden. Dieser muss dementsprechend zuerst implementiert und getestet werden und sukzessiv die von ihm aufgerufenen Funktionen.

4. *Bottom-Up-Integration*: Die Tests werden mit den Komponenten begonnen, die keine weiteren Komponenten aufrufen. Durch schrittweises Hinzufügen weiterer Komponenten wird das getestete System immer größer. [SBC12][Win+13][SL19]

Diese Integrationsstrategie wird bei den Komponenten angewandt, die nicht vom Job aufgerufen werden.

Testbasis	Anforderungen an Schnittstellen und das Systemdesign, die unter anderem beispielsweise in Workflow- oder Anwendungsfalldiagrammen dargestellt werden. [SL19]
Testziel	Schnittstellenfehler sollen aufgedeckt werden.

Testobjekt	Die zusammengeführten Komponenten werden im Zusammenhang getestet. Zusätzlich werden auch Fremdsysteme oder Datenbankanbindungen mit den Komponenten zusammen getestet.
Verantwortung/ Entwicklung	Die Tests werden von Testern entwickelt oder von Entwicklern, die nicht selbst am <b>CIF</b> programmiert haben.
Ausführung	Ausgeführt werden sollen die Tests in der <i>Build Pipeline</i> , sobald Änderungen am <b>CIF</b> in das Gesamtsystem integriert werden.
Eintrittskriterien	Die Komponenten- und Smoke Tests wurden fehlerfrei durchlaufen und die zu den Integrationstests gehörenden Testfälle wurden vollständig ausgearbeitet.
Austrittskriterien	Die Integrationstest sind vollständig ausgeführt. Die Testergebnisse wurden gesammelt, ausgewertet und zur Verfügung gestellt.

Tabelle 5.3: Die Teststufe *Integrationstests*

## Systemtests

Die Systemtests sollen als **E2E**-Tests erfolgen. Sie prüfen damit die Anforderungen aus Nutzersicht für gewisse Funktionen von Anfang bis Ende. [Tsa+01][AV14] Dadurch können die geforderten Funktionen des **CIF** eingehend geprüft werden.

Testbasis	Als Testbasis dient die in <a href="#">Unterabschnitt 5.1.4</a> genannte Testbasis.
Testziel	Fehler in den Abläufen im Gesamtsystem sollen aufgedeckt werden.
Testobjekt	Die vom Nutzer geforderten Abläufe im Gesamtsystem.
Verantwortung/ Entwicklung	Die Tests werden von Testern entwickelt oder von Entwicklern, die nicht selbst am <b>CIF</b> programmiert haben.
Ausführung	Ausgeführt werden sollen die Tests in der <i>Build Pipeline</i> , sobald Änderungen am <b>CIF</b> in das Gesamtsystem integriert werden.
Eintrittskriterien	Die Integrationstests wurden durchlaufen und die zu den Systemtests gehörenden Testfälle wurden ausgearbeitet.
Austrittskriterien	Die <b>E2E</b> Tests sind vollständig und fehlerfrei ausgeführt. Die Testergebnisse wurden gesammelt, ausgewertet und zur Verfügung gestellt.

Tabelle 5.4: Die Teststufe *E2E Tests*

## Akzeptanztests

Akzeptanztests stellen die letzte Stufe der geplanten Tests dar. In den agilen Testquadranten werden sie den Tests zugeordnet, die die Anforderungen an ein System und dessen

Funktionen testen. [CG14] Wie in [Abschnitt 2.2](#) erläutert, werden sie grundsätzlich vom Kunden ausgeführt. Da *FNT* intern ebenfalls das *CIF* nutzt, werden die Akzeptanztests bei Bedarf auch innerhalb von *FNT* ausgeführt. Dies geschieht nicht automatisiert, da aus der funktionellen Sicht mit den vorherigen Teststufen bereits alle geforderten Funktionen getestet wurden, die auf Erfüllung eines Vertrags mit dem Kunden hindeuten. [SL19] Die von *FNT* durchgeführten Tests sind dabei sogenannte *Alphatests*. Tests, die der Kunde vornimmt, werden *Betatests* genannt. [CG14][SBC12] Da die Akzeptanztests in diesem Konzept nicht weiter verfolgt werden, wird auf eine detaillierte Darstellung verzichtet.

### 5.1.7 Testautomatisierung

Wie bereits in [Abschnitt 2.2](#) beschrieben, ist eine Testautomatisierung in agilen Projekten wie dem des *CIF* besonders wichtig. Deshalb sollen in diesem Konzept, wie schon in [Kapitel 4](#) festgelegt, nur automatisierte Tests geplant und entwickelt werden. Diese sollen die bisher durchgeführten manuellen Tests weitestgehend ersetzen. Automatisierte Tests sind jedoch nicht immer sinnvoll oder für alle Fälle umsetzbar, dies wurde ebenfalls bereits in [Abschnitt 2.2](#) erläutert. Dementsprechend sollen vor der Ableitung von Testfällen alle Funktionalitäten des *CIF* auf die Durchführung automatisierter Tests geprüft werden.

#### Ansätze zur Automatisierung

Für die Testautomatisierung selbst stellt sich die Frage der Umsetzung. Die Tests müssen automatisiert aufgerufen und durchlaufen werden. Dafür wird zum einen ein Testtreiber benötigt und zum anderen müssen Testskripte erstellt werden. Da die Funktionen im Rahmen der Blackbox-Tests nicht direkt am Code getestet werden, bieten sich folgende zwei Ansätze an, die Funktionen ohne aktives Aufrufen von Methoden im Code selbst zu testen:

- **API-Tests:** Diese Möglichkeit bietet sich durch das Vorhandensein der in [Abschnitt 2.1](#) beschriebenen *BGW* an. Durch *REST*-Anfragen können jegliche in *Command* oder den *NMS*-Tabellen gespeicherten Daten aus den jeweiligen Tabellen abgefragt und die meisten der zu testenden Funktionen aufgerufen und ausgeführt werden. Wie in [Unterabschnitt 2.2.3](#) ausgeführt, ist bei Blackbox-Tests nur wichtig, was in die Tests eingegeben wird und was herauskommt. Dies kann durch die *API*-Tests erfüllt werden. [Spi17][Bel15]
- **Graphical User Interface (GUI)-Tests:** Da *Command* und somit auch das *CIF* vom Nutzer über die Benutzeroberfläche gesteuert werden kann, bieten sich zusätzlich

automatisierte **GUI**-Tests an. Dies kann beispielsweise durch ein Browserautomatisierungstools ermöglicht werden. [Spi17][Coh10]

Die Verteilung der Testfälle auf die Automatisierungsarten orientiert sich an der Testpyramide, die Mike Cohn 2009 entwickelt hat. Sie ist in [Abbildung 5.4](#) dargestellt. Nach Cohn sollen automatisierte Komponententests eine breite Basis bilden, auf die das Testen jeglicher Services, also in diesem Fall Funktionen des Testobjekts, aufgesetzt wird. Hier wird die Schicht durch die **API**-Tests vertreten. Die schmale Spitze stellen die **GUI**-Tests dar, denn sie sind langsam, aufwendig zu implementieren und anfällig für Änderungen. Um die Kosten so gering wie möglich zu halten, sollten **GUI**-Tests in einem vernünftigen Maß eingesetzt werden. [Coh10][Spi17]

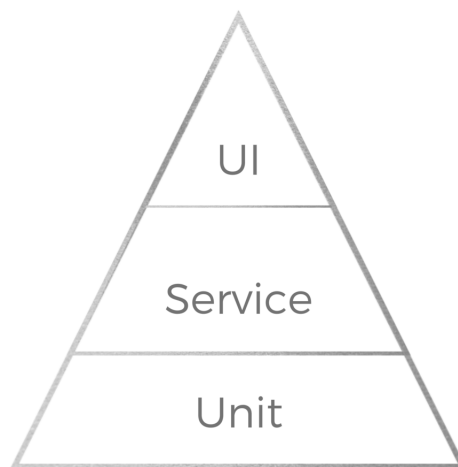


Abbildung 5.4: Die Testpyramide nach Mike Cohn.

[Cro20]

In der Praxis zeigt sich jedoch, dass die in dieser Pyramide dargestellte Verteilung häufig nicht eingehalten wird. Beispielsweise werden bei der Entwicklung von mobilen Anwendungen **GUI**-Tests deutlich öfter eingesetzt als **API**-Tests, da die Funktionalität der Benutzeroberfläche als wichtiger eingestuft wird. Die Pyramide nach Cohn muss also kritisch betrachtet und an das jeweilige Projekt angepasst werden. [CDM18]

Für das Testen des **CIF** bietet sich eine Struktur wie die der Pyramide jedoch gut an, denn es existieren bereits zahlreiche Komponententests, die die Basis darstellen. Durch die **BGE** sind umfassende **API**-Tests möglich, die schnell zu implementieren und auszuführen sind. Lediglich die Fälle, die nicht über die **BGE** testbar sind, sollen anschließend als **GUI**-Tests getestet werden. Das spart Zeit und Kosten.

## Ausführung im Kontext von Continuous Integration

CI, wie in [Unterabschnitt 2.2.6](#) beschrieben, soll bei der Ausführung der automatisierten Tests zum Zug kommen. Dabei werden alle Testfälle außer den Smoke Tests als Regressionstests angesehen. Das hat den Grund, dass alle Funktionen regelmäßig verifiziert werden müssen, um die stetige Weiterentwicklung des [CIF](#) abzusichern. Es soll daher keine Priorisierung der Regressionstestfälle stattfinden und so unterschiedliche Testdurchläufe produziert werden. Die Smoke Tests werden standardmäßig immer vor den Regressionstests ausgeführt.

Für die Ausführung muss auf dem *Build-Server* eine *Build-Pipeline* erstellt werden, die die einzelnen Schritte der Testausführung nacheinander ausführen soll. [\[Aks+15\]](#)[\[Pat17\]](#) Eine Ausführung soll dabei immer dann automatisch stattfinden, wenn eine Änderung am [CIF](#) vorgenommen und integriert wurde. Eine regelmäßige Ausführung mindestens einmal wöchentlich ist zusätzlich sinnvoll. Wenn die Rate an Änderungen weiterhin zunimmt, könnte auch ein *Nightly-Build* in Betracht gezogen werden, bei dem die Testfälle einmal pro Nacht durchlaufen werden. Von der *FNT Testpolitik* wird ein *Nightly Build* für Smoke Tests empfohlen. [\[FNT20\]](#) Dieser und ein wöchentlicher gesamter *Build* sollen umgesetzt werden.

### 5.1.8 Testansatz

In diesem Abschnitt sollen die möglichen Ansätze diskutiert werden, auf Basis derer die automatisierten Tests entwickelt werden können. Das Festlegen eines oder mehrerer Testansätze kann die darauf folgende Wahl von Testentwurfsverfahren erleichtern. Es gibt grundsätzlich folgende Testansätze, die einzeln oder in Kombination verfolgt werden können:

- **Kostenorientierter Ansatz:** Kosten und Zeitbedarf werden mit der Menge und Komplexität der Testfälle in Relation gesetzt. Testentwurfsverfahren werden so ausgewählt, dass teure Testfälle vermieden werden. [\[SL19\]](#)  
Dieser Ansatz wird für das [CIF](#) ausgeschlossen. Über den Kosten steht die weitgehend vollständige Testabdeckung zur Steigerung der Qualitätssicherung.
- **Risikobasierter Ansatz:** Die erarbeiteten Projekt- und Produktrisiken werden herangezogen und Bereiche mit hohem Risiko werden priorisiert. [\[SL19\]](#)[\[ISO13a\]](#)  
Da die angestrebte Testabdeckung alle Funktionen des [CIF](#) mit einbezieht und alle Tests als Smoke Tests oder gleichwertige Regressionstests geplant sind, wird dieser Ansatz ebenfalls ausgeschlossen.

- **Modellbasierter Ansatz:** Testfälle und die Testmethode werden anhand von abstrakten Modellen hergeleitet und ausgearbeitet. [ISO13a][SL19]

Das CIF ist in seiner Entwicklung schon weit fortgeschritten und das Erstellen eines Modells könnte Überblick über die bereits implementierten Funktionen verschaffen. Mit diesem Überblick sind Testfälle anschließend einfacher ableitbar und ihre Vollständigkeit kann besser überprüft werden.

- **Anforderungs- /Funktionsbasierter Ansatz:** Testfälle werden so entworfen, dass sie nachweisen können, dass alle Anforderungen an ein Produkt erfüllt sind. Dieser Ansatz benötigt deshalb eine detaillierte Planung der Testfälle bevor diese umgesetzt werden. [ISO13a]

Dieser Ansatz gibt den Hauptteil des in Kapitel 4 geschilderten Soll-Zustands wieder, denn jegliche Funktionen des CIF sollen überprüft werden. Die Schwierigkeit bei diesem Ansatz ist jedoch, die Anforderungen gänzlich abzudecken und dies nachzuweisen. Meist kommen hier weitere Ansätze zum Tragen. [ISO13a]

- **Methodisches Testen und prozess- oder standardkonformer Ansatz:** Bereits definierte Testfälle oder firmeninterne und branchenspezifische Standards werden für die Entwicklung der Teststrategie verwendet. [SL19]

Für das CIF liegen lediglich die für das manuelle Testen entworfenen Testfälle vor, die teilweise einen anderen Ansatz verfolgen, als für automatisiertes Testen möglich wäre. Einige von ihnen könnten aber als Basis für die GUI-Tests dienen. Als Basis der Herangehensweise an das Testen des CIF wird, wie bereits erläutert, die ISO-Norm 29119 und der STLC verwendet. Ebenso wird sich mit dem Entwurf der Tests an der in Abschnitt 3.1 erläuterten Testpolitik orientiert. Projektspezifische Anpassungen werden aber vorgenommen.

- **Wiederverwendungsorientierter Ansatz:** Fertige Tests aus anderen Projekten werden als Grundlage für die neuen Tests genommen. Ziel ist die schnelle und ressourcensparende Umsetzung. [SL19]

Es existieren zwar bereits fertige automatisierte Testsuiten in anderen Projekten, diese sind aber sehr spezifisch auf das jeweilige Projekt angelegt und stark verflochten, sodass ein Heraussuchen von ähnlichen Testfällen ein größerer Aufwand wäre, als Tests neu zu entwerfen.

- **Checklistenorientierter Ansatz:** Wenn bereits Tests durchgeführt wurden, können die dadurch gefundenen Fehler gesammelt und als Ausgangspunkt für Tests genutzt werden. Wie in Abschnitt 2.2 erwähnt, bündeln sich Fehler und kommen nicht gleichmäßig verteilt vor. So können erste Tests vor allem auf das Umfeld einer bereits gefundenen Fehlerquelle konzentriert werden. [SL19]

Die Tests für das CIF sollen alle Funktionen abdecken und gesammelt regelmäßig

ausgeführt werden. Eine Konzentration auf einzelne Fehlerquellen ist daher langfristig nicht nötig.

- **Erfahrungsorientierter Ansatz:** Für das Erstellen der Teststrategie wird Erfahrungswissen von sachkundigen Mitarbeitern, Stakeholdern oder Externen genutzt. Diese haben eine beratende Position. [SL19][ISO13a]  
Durch den standardkonformen Ansatz wird bereits niedergeschriebenes Expertenbeziehungsweise Erfahrungswissen genutzt. Um die Qualität des erstellten Konzepts und der darauf basierenden Umsetzung zu gewährleisten, wird zudem regelmäßig Feedback vom beteiligten Team oder anderen Mitarbeitern angefragt.
- **Leistungserhaltender Ansatz:** Die Erhaltung der Leistung der Produktes steht im Vordergrund. Es wird eine Wiederverwendung und Automatisierung möglichst vieler Testfälle und Testdaten angestrebt. Regressionstests sind erwünscht. [SL19]  
Dieser Ansatz spiegelt den Hauptgrund für die für das CIF zu entwickelnden Tests wieder. Die bisher entwickelten und genutzten Funktionen sollen verifiziert und gewahrt bleiben und bei Erweiterungen immer wieder validiert werden. Testautomatisierung und Regressionstest sind zudem ebenfalls umfangreich vorgesehen, wie bereits beschrieben.
- **Mathematikbasierter Ansatz:** Die Testfälle werden entweder zufällig priorisiert oder nach einem mathematischen Schema. [ISO13a]  
Dieser Ansatz wird für das CIF ausgeschlossen, da die Testfälle rein nach den Funktionen des CIF entwickelt werden soll.

Insgesamt wird sich bei der Entwicklung der Testfälle und Tests vor allem auf den leistungserhaltenden und den anforderungsbasierten Ansatz konzentriert. Der anforderungsbasierte Ansatz soll dabei vom modellbasierten Ansatz hinsichtlich Vollständigkeit und Übersicht unterstützt werden. Für die Entwicklung der allgemeinen Teststrategie und die Sicherung der Qualität dieser wird hauptsächlich der prozess- beziehungsweise standardkonforme, methodische Ansatz gewählt. Dieser wird mit Erstellung des Testkonzepts nach dem STLC und der ISO-Norm 29119 auch bereits verfolgt.

### 5.1.9 Testentwurfsverfahren

Testentwurfsverfahren sind Verfahren, mithilfe derer Tests und Testfälle systematisch entworfen werden können. Mithilfe der bereits gewählten Testansätze lassen sich geeignete Testentwurfsverfahren auswählen. Die Verfahren unterscheiden sich dabei nach Testziel und Testobjekt. Sie werden zudem in statische und dynamische Verfahren unterteilt. [ISO15]



## Statische Testentwurfsverfahren

Statische Tests testen nicht das ausführbare Programm, sondern statische Arbeitsergebnisse wie Dokumente, User Stories oder Sourcecode. Diese Überprüfungen sollten bestenfalls vor den dynamischen Tests stattfinden, um grundlegende Fehler wie Ungenauigkeiten, Inkonsistenzen, Widersprüche und Lücken zu finden und zu beheben, bevor die eigentlichen Funktionen getestet werden. [SL19][Fil+21] Da hier besonders Augenmerk auf die innere Struktur der Arbeitsergebnisse wie dem Code selbst gelegt werden, sind statische Tests meist *Whitebox-Tests*. [Fil+21]

Sowohl für das hier erarbeitete Konzept, als auch für das darauf basierende erstellte Projekt und das CIF selbst sollen folgende Ergebnisse statisch überprüft werden:

- Die erstellten Dokumente wie die Teststrategie, Anwendungsfälle oder Testfälle sollen mit dem Projektmanager und dem Integrationsarchitekten besprochen werden. Dabei soll vor allem Augenmerk auf Vollständigkeit, die korrekte Darstellung der Abläufe und Durchführbarkeit gelegt werden.
- Der Sourcecode der implementierten Tests, die Struktur des erstellten Testprojekts und die erstellten Testdaten sollen zunächst mit erfahrenen Entwicklern und Testern jeweils rezensiert werden mit Fokus auf die Einhaltung der Coderichtlinien, den testspezifischen Vorgaben der Testpolitik und Erfolgskonzepten, auch *Best Practices* genannt. Der Code soll zudem mithilfe von Werkzeugen regelmäßig automatisiert überprüft werden. So sollen vor allem Schwächen bezüglich der Sicherheit der Software und des Formats frühzeitig aus dem Code entfernt werden können. [Fil+21]
- Der Sourcecode des CIF soll ebenfalls automatisiert überprüft werden. Dies geschieht bereits in der *Build-Pipeline* des CIF-Projekts. Zusätzlich erfolgen persönliche Reviews nach Fertigstellung einer Änderung.

Die Reviews sollen dabei in Form von sogenannten *Walkthroughs* spätestens bei Fertigstellung des jeweiligen Dokuments beziehungsweise Quellcodes durchgeführt werden. Im Rahmen der agilen Vorgehensweise bietet sich auch ein regelmäßiges Review an. Ein *Walkthrough* beschreibt dabei das Durchsprechen des produzierten Objekts mit anderen Entwicklern, dem Projektmanagement, Qualitätsmanagement und bei Bedarf weiteren Mitarbeitern. Diese haben unterschiedliche Blickwinkel auf und Ansprüche an das Produkt. In einem *Walkthrough* wird dieses von Beginn bis Ende betrachtet und mögliche Unvollständigkeiten oder Fehler gefunden, die der Entwickler übersehen hat. Grundsätzlich existieren weitere Reviewarten als ein *Walkthrough*, diese sind jedoch meist sehr aufwendig und nicht für kurze Sprints oder kleine Teams ausgelegt wie das *Walkthrough*. [SL19]



## Dynamische Testentwurfsverfahren

Dynamische Tests testen den ausführbaren Programmcode. Die dynamischen Testentwurfsverfahren helfen vor allem bei den folgenden Schritten:

- Spezifisches Festlegen der Anforderungen, Voraussetzungen und Ziele der Tests
- Spezifizieren der Testfälle
- Planen, wie die Testfälle ausgeführt werden sollen

[ISO15][SL19]

Die dynamischen Verfahren teilen sich in die in [Abschnitt 2.2](#) erläuterten Testarten. Es existieren also Blackbox-Verfahren, Whitebox-Verfahren und erfahrungsbasierte Verfahren. [ISO15][SL19] Da bereits festgelegt wurde, dass für das Herleiten der automatisierten Testfälle lediglich Blackbox-Tests eingeführt werden, werden die anderen beiden Verfahren hier vernachlässigt.

Die ISO-Norm 29119, das ISTQB® und Liggesmeyer nennen hauptsächlich neun dynamische Verfahren, die für die Testgestaltung angewandt werden können. Bei der Auswahl helfen neben den gewählten Testansätzen [SL19] auch die Testbasis des Projektes, die ausschließlich Dokumente enthält, die die Funktionen des CIF beschreiben, wie in [Unterabschnitt 5.1.4](#) erläutert. [ISO13c] Dabei sollte das Ziel sein, mit möglichst wenig Aufwand genau so viele Testfälle herzuleiten, dass alle Kriterien ausreichend, aber nicht redundant getestet werden können. [SL19] Außerdem ist zu beachten, dass das Testen eines ganzen Systems wie des CIF komplex und nicht jedes Verfahren geeignet ist.

- **Äquivalenzklassenbildung:** Die möglichen Eingabewerte oder alternativ auch Ausgabewerte eines Testobjekts werden in Äquivalenzklassen aufgeteilt. Dabei werden alle Werte einer Gruppe zugeordnet, von denen zu erwarten ist, dass sie beim Objekt dieselbe Reaktion bewirken. Für jede Klasse kann anschließend einer der Eingabewerte getestet werden, es wird davon ausgegangen, dass dies ausreichend ist. So kann vermieden werden, redundante Testfälle zu erstellen. [SL19][ISO15][Lig09] Mit der **Klassifikationsbaummethode** ist es zudem möglich, die identifizierten Äquivalenzklassen graphisch in einer Baumstruktur darzustellen. Dies bietet sich bei einigen Projekten besonders für die Übersicht an. [ISO15] Da, wie bereits in [Abschnitt 2.1](#) erläutert, für die Delta-Berechnung verschiedene Delta-Fälle als Ausgabewerte möglich sind, bietet sich gerade beim Entwurf dieser Fälle eine Äquivalenzklassenbildung an.

- **Grenzwertanalyse:** Die Grenzwertanalyse stellt eine Erweiterung der Äquivalenzklassenbildung dar. Wie der Name andeutet werden die Grenzen der Äquivalenzklassen getestet. Diese sind fehleranfällig, weil hier eine klare Fallunterscheidung programmiert sein muss und dies nicht immer korrekt durchgeführt wird. Die Grenzwertanalyse ist aber nur einsetzbar, wenn die Eingabewerte zum Beispiel numerisch geordnet und so Grenzen ausgemacht werden können. [SL19][ISO15]  
Die Eingabewerte des CIF weisen keine Grenzübergänge auf. Deshalb wird dieses Verfahren nicht für die Tests des CIF verwendet.
- **Zustandsbasierter Test:** Dieses Verfahren spielt vor allem in objektorientierten Systemen eine Rolle, wo verschiedene Objektzustände auftreten. Meist ist die Basis ein Zustandsdiagramm und das Testziel, die Zustandsübergänge zu überwachen. [SL19][ISO15][Lig09]  
Innerhalb des CIF wird es keine Zustandsübergänge geben, weshalb dieses Verfahren ausgeschlossen werden kann.
- **Entscheidungstabellentests:** In einer Entscheidungstabelle wird die logische Beziehung zwischen Eingaben als Bedingungen und Ausgaben als Aktionen festgehalten. Das Ziel ist es, so viele Eingabekombinationen wie möglich mit jeweils einem Testfall abzudecken, um mögliche Fehler in Abhängigkeiten der Parameter untereinander zu finden. Dieses Verfahren findet vor allem dort Anwendung, wo verschiedene Kombinationen an Eingaben beziehungsweise Bedingungen ein System unterschiedlich beeinflussen. Bei sehr vielen Tabelleneinträgen kann dieses Verfahren mithilfe der Äquivalenzklassenbildung vereinfacht werden. [SL19][ISO15][Lig09] Wird keine Tabelle erstellt, sondern ein Graph, so wird von der **Ursache-Wirkung-Analyse** gesprochen. Durch diese können Beziehungen zusätzlich beispielsweise gewichtet werden. [SL19][ISO15][Lig09]  
Beim CIF gibt es zwar viele mögliche Eingabewerte, jedoch in wenigen Kombinationen. Deshalb wäre das Erstellen von Entscheidungstabellen oder Ursache-Wirkungs-Graphen ein hoher Aufwand, der durch andere Entwurfsverfahren ersetzt werden kann.
- **Kombinatorisches Testen:** Das kombinatorische Testen hängt eng mit dem mathematikbasierten Ansatz aus [Unterabschnitt 5.1.8](#) zusammen. Es beschreibt das Testen mit zufälligen oder mathematisch bestimmten Kombinationen aus Eingabewerten. Diese Eingabewerte basieren dabei häufig auf Ergebnissen von anderen Testentwurfsverfahren. Anders als Entscheidungstabellentests wird hier nicht auf vollständige, sondern sinnvoll reduzierte Testabdeckung gesetzt, häufig aus finanziellen oder zeitlichen Gründen. [SL19][ISO15]

Diese liegen bei den automatisierten CIF-Tests nicht vor, weshalb dieses Verfahren ausgeschlossen werden kann.

- **Syntaxtests:** Bei diesem wird eine Menge an Syntaxregeln erstellt, anhand derer die jeweiligen möglichen Eingabeparameter eines Testobjekts gebildet werden, ein Beispiel für die Umsetzung der Syntaxregeln ist die *Backus-Naur Form*, eine Metasprache, die unter anderem Programmiersprachen syntaktisch beschreibt. [MR03] Die Testfälle werden anschließend von den Syntaxregeln abgeleitet. [SL19][ISO15][Lig09] Dieses Verfahren ist für das Testen des CIF weniger geeignet, da es mit einer Menge an Syntaxregeln schwer ist, die Anforderungen entsprechend abzudecken.
- **Anwendungsfallbasierte Tests:** Testfälle werden aus Anwendungsfällen des Testsystems abgeleitet. Interaktionen von Nutzer und System können dadurch ebenso abgedeckt werden wie unterschiedliche Szenarien einer Funktion. [MWI17][SL19][ISO15] Dieses Verfahren hängt eng mit dem anforderungsbasierten Ansatz aus [Unterabschnitt 5.1.8](#) zusammen, der als einer der wichtigsten Ansätze identifiziert wurde. Die Funktionen des CIF lassen sich zudem gut in Anwendungsfälle formulieren. Dieses Verfahren wird hauptsächlich für die Testentwicklung auf allen Stufen genutzt werden.

Zusammengefasst werden also Testfälle auf Basis von Anwendungsfällen und der Äquivalenzklassenbildung entworfen. Diese Testentwurfsverfahren decken sich mit den gewählten Ansätzen.

### 5.1.10 Fehlermanagement

Wird durch einen Testfall ein Fehlverhalten aufgedeckt, so sollte dies so dokumentiert werden, dass der Fehler gefunden und behoben werden kann. Deshalb sollte mindestens die Fehlerursache mit dem Fehler gespeichert werden. [IEE93] Dafür sollten alle Fehler zentral gespeichert werden, bestenfalls in einem Projektmanagementtool, wo es möglich ist, die Fehler direkt mit der zugehörigen *User Story* zu verknüpfen. Die Meldungen sollten dabei wenn möglich automatisiert erstellt werden, sobald ein Fehler in einem Testdurchlauf auftritt.

Bei der Schwere von Fehlern kann es einen großen Unterschied machen, ob wegen eines Fehlers ein Programm nicht weiterlaufen kann oder ob durch einen Fehler ein kleiner Teil der Software nicht funktioniert, der nicht häufig genutzt wird. [Wit18] Deshalb sollten Fehler zusätzlich zu ihrer Aufzeichnung klassifiziert werden. Das ist schwer automatisiert umzusetzen. Zwar ist beispielsweise eine *NullPointerException* als Fehler häufig ernster als ein *AssertionFailedError*, jedoch ist dies nicht immer gegeben, weshalb eine endgültige

Klassifizierung von Fehlern manuell erfolgen sollte. Es ist beispielsweise üblich, Fehler nach ihrem Schweregrad in fünf Prioritätsklassen einzuteilen: Dringend, Hoch, Mittel, Niedrig, keine Priorität. [IEE93] Dies kann im entsprechenden Eintrag im Projektmanagementtool vermerkt werden und je nach Einstufung kann der Fehler dann in Rücksprache mit dem Team einem Entwickler zugewiesen werden, der sich um die Behebung des Fehlers kümmert.

### 5.1.11 Testmetriken

Wie in [Unterabschnitt 2.2.5](#) bereits erläutert wurde, ist das Festlegen von Testmetriken für eine objektive Auswertung der Testdurchläufe essenziell. Nach der ISO/IEC-Norm 15939, die Richtlinien für Software-Messprozesse enthält, werden vier Schritte beschrieben, mithilfe derer sich Metriken kontinuierlich richtig einsetzen lassen:

1. Ein- und später Weiterführen der Messungen: Das Einführen der Metriken muss mit dem Management abgesprochen und von diesem abgesegnet werden. Ressourcen für die Planung und Durchführung müssen gefunden und Verantwortungen verteilt werden. Nach einem erfolgreichen Messdurchgang müssen die Metriken weiter gepflegt und angepasst werden, um die bestmöglichen Informationen zu erhalten.
2. Vorbereitung der Messung: Es wird festgelegt, welche Informationen dem Unternehmen oder dem Projekt von Nutzen sind und gebraucht werden. Anschließend muss überlegt werden, mithilfe welcher Messmethoden benötigte Werte aus dem Projekt gezogen werden können. Zusätzlich wird festgelegt, wie und wann die Daten gesammelt und wo sie gespeichert werden sollen. Auch die jeweilige Darstellung der Messwerte wird in diesem Schritt festgelegt. Die Auswertung der Informationen soll ebenfalls festgelegt werden. Das ist beispielsweise über das Setzen von Grenzwerten möglich. Die Ergebnisse werden dem Management kommuniziert.
3. Durchführen der Messung: Die geplanten Messungen sollen zu den festgelegten Zeitpunkten und Orten mit den erarbeiteten Messmethoden stattfinden. Gemessene Werte werden wie vorgehabt festgehalten und gespeichert.
4. Auswerten der Messung: Die Werte werden nach den ausgewählten Kriterien bewertet. Verbesserungspotenzial sollte erkannt und mögliche Verbesserungen vorgeschlagen oder mit dem Management besprochen werden. Die Ergebnisse werden im Testabschlussbericht dargestellt.

[ISO17]

Es muss demnach festgelegt werden, welche Informationen dem Testprojekt von Nutzen sein könnten und wie diese Aspekte des Testprojekts überwacht beziehungsweise überprüft werden sollen.

- Die **Abdeckung der Anforderungen** an das **CIF** mit Testfällen sollte überprüft werden. [SL19] Es wird davon ausgegangen, dass die später erarbeiteten Anwendungsfälle auf Grundlage der Testbasis vollständig sind. Mithilfe einer sogenannten *Traceability Matrix* kann nachverfolgt werden, welche Testfälle welchen Anwendungsfällen zugeordnet werden. So ist nachvollziehbar, ob alle Anwendungsfälle von Testfällen abgedeckt sind. [Wit19][Lou11]

Die *Traceability Matrix* sollte entweder von Hand, beispielsweise in Excel, erstellt werden, oder wenn möglich über ein entsprechendes Werkzeug. Das muss in der Auswahl der Testwerkzeuge evaluiert werden. Wichtig ist eine Auflistung aller Anwendungsfälle, denen mehrere Testfälle übersichtlich zugeordnet werden können. [Jam+16]

- Im Rahmen der Neueinführung von automatisierten Tests ist eine Messung des **Implementierungsfortschritts** für das Projekt sinnvoll. [Wit18][SL19] So sind zwar alle Testfälle bereits beschrieben, eine Implementierung benötigt aber auch ihre Zeit und sollte überwacht werden. Testfälle, die bereits umgesetzt sind, sollen dementsprechend gekennzeichnet werden. Zusätzlich soll die Anzahl der implementierten Testfälle prozentual gegen die Anzahl der erfassten Testfälle gesetzt und diese Zahl im Kontext des Projekts bewertet werden. Da die Smoke Tests in ihrer Funktion höher priorisiert sind als die Regressionstests, sollten sie zuerst vollständig umgesetzt werden. Die **API**-Tests haben eine höhere Priorisierung als die **GUI**-Tests und sollten somit ebenfalls früher eine höhere Umsetzungsquote aufweisen.

Gesammelt werden können die Informationen ebenfalls in der *Traceability Matrix*. Durch farbliche Kennzeichnung wäre es möglich, die Testfälle hervorzuheben, die bereits implementiert wurden. So ist eine Übersicht und direkte Verknüpfung der Informationen möglich. Die berechneten Prozentzahlen könnten ebenfalls in diesem Dokument dargestellt werden. Ist dies zu unübersichtlich, bietet sich auch das Erstellen eines weiteren Dokuments an. Eine weitere Option wäre ein darauf ausgelegtes Werkzeug. Auch das muss bei der Auswahl der Testwerkzeuge noch einmal betrachtet werden.

- Zusätzlich zum Implementierungsfortschritt soll der **Testautomatisierungsgrad** gemessen werden. Dabei wird erhoben, wie viel Prozent der gesamten Testfälle automatisiert werden und wie viele noch manuell durchlaufen werden. [Wit18] Nach Erstellung der Testfälle wird deutlich werden, welche Testfälle nicht für automatisierte Tests geeignet sind und deshalb weiterhin manuell durchgeführt werden müssen.

Daraus ergibt sich der momentan maximale Automatisierungsgrad. Zusätzlich dazu soll gemessen werden, wie weit die Automatisierung bereits vorangeschritten ist. Da in diesem Projekt nur automatisierte Testfälle implementiert werden, ist dies bereits durch den gemessenen Implementierungsfortschritt abgedeckt.

- Die **Durchführungszeit** der einzelnen Teststufen, aber auch des gesamten Testverlaufs soll gemessen werden. Das wird in der Regel vom *Build-Tool* übernommen und übersichtlich gespeichert. Möglich wäre auch die Testproduktivität auszurechnen. Das wäre die Anzahl an Testfällen, die pro Stunde getestet werden. [Wit18] Gerade am Anfang des Projekts hilft eine solche Zahl bei einer Prognose zur Laufzeit, wenn alle Testfälle implementiert sind.
- Ein weiterer wichtiger Wert ist die durch die Automatisierung der Testfälle entstandene **Zeitersparnis**. Diese soll auf Schätzwerten beruhen, denn im Gegensatz zu den automatisierten Tests wird bei manuellen Tests meist kein genauer Zeitwert in der Durchführung aufgezeichnet. [Wit18] Ein erneutes manuelles Durchführen aller Testfälle zu Messzwecken würde mit einem hohen Zeitaufwand einhergehen. Deshalb sollen in einer Befragung eines der manuellen Tester des CIF Schätzwerte für die Dauer der manuellen Durchführung der Testfälle erfragt werden. Dabei können einige exemplarische Testfälle herangezogen werden, mithilfe derer sich die Durchführungszeit hochrechnen lässt. Die Schätzungen können dann im Anschluss mit den gemessenen Werten der automatisierten Testfälle verglichen werden.
- Der **Erfolg eines Testdurchlaufs** soll jeweils gemessen werden. Den Testfällen wird nach einem Testdurchlauf ein Status zugewiesen, möglich sind dabei die Status *Passed*, *Failed* oder *Blocked*. Die Verteilung der Status soll erfasst und ausgewertet werden. [Wit18][SL19] Das könnte durch den Testtreiber, die *Build-Pipeline* oder ein weiteres Werkzeug abgehandelt werden. Die jeweiligen Werte sollen zusätzlich in einem weiteren Dokument festgehalten werden, um die Änderungen zwischen Testläufen beurteilen zu können, falls dies kein Werkzeug übernehmen kann.
- Die **Fehlerverteilung** über Fehlerart beziehungsweise -schwere, Teststufe und Dauer der Behebung soll ebenfalls erfasst werden. Anhand dieser Daten lässt sich anschließend beispielsweise der Testfortschritt vorhersagen und die für die nächsten Testzyklen benötigten Ressourcen genauer einplanen. [Wit18] Das bereits in [Unterabschnitt 5.1.10](#) erläuterte Fehlermanagement sorgt bereits für die hier benötigten Werte, sodass die erstellten Fehlerdokumentationen direkt für die Metrik übernommen werden können.

Die während der Durchführung und Auswertung erstellten Dokumente sollen gesammelt an einem Ort gespeichert werden, der für alle Beteiligten zugänglich ist. Zusätzlich werden die Ergebnisse im Testabschlussbericht behandelt. [SL19]

Die gesammelten Metriken spiegeln den Bedarf des Testprojekts zum jetzigen Zeitpunkt wieder. Mit Fortschritt eines Projekts ist es üblich, dass sich Anforderungen ändern und manche Informationen in ihrer Bedeutung verlieren, während Informationen, die dann benötigt werden, von diesen Metriken noch nicht erfasst sind. Deshalb sollten die Metriken regelmäßig überarbeitet werden. [Wit18]

### 5.1.12 Anforderungen an Testdaten und Testumgebung

Damit Testdaten und Testumgebung richtig ausgesucht, entwickelt und gepflegt werden können, ist es wichtig, die Anforderungen an diese festzuhalten. [ISO13a]

#### Testdaten

Die Testdaten müssen strukturiert aufgebaut werden und akribisch aufeinander abgestimmt sein. Wie in [Abschnitt 2.1](#) beschrieben, werden Daten in den [NMS](#)-Tabellen benötigt und Daten in *Command* selbst. Um die Delta-Fälle zu berechnen, müssen die Daten in ihrer ID und ihrem Objekttyp übereinstimmen. Sie sollten gesammelt in die jeweilige Tabelle gespeichert werden können. Orientiert werden kann sich an den Daten, die auf dem Produktivsystem genutzt werden, sensible Kundendaten sollten aber wenn möglich nicht übernommen werden.

Wie auch in [Abschnitt 3.1](#) beschrieben, schlägt die *FNT Testpolitik* für den Umgang mit Testdaten drei Schritte vor, die auch die [ISO](#)-Norm 29119 unterstützt: Das Aufsetzen der Daten durch Speichern in die jeweiligen Tabellen, das Nutzen der Daten in den Tests und das abschließende Bereinigen der Tabellen von den Tests. [FNT20][ISO13c]

#### Testumgebung

Die [ISO](#)-Norm 29119 schlägt folgende Unterteilung für die Testumgebung vor: Hardware, Software, Sicherheit und Testwerkzeuge. [ISO13c]

- **Software:** Als Testumgebung wird eine Umgebung benötigt, auf der funktionelle [API](#)- und [GUI](#)-Tests ausgeführt werden können. Die Testumgebung sollte dabei der Produktivumgebung so ähnlich wie möglich sein und alle Funktionalitäten des [CIF](#) beinhalten.



- **Hardware:** Es wird ein Server benötigt, auf dem eine Testinstanz und eine Testdatenbank laufen können. Ein weiterer Server wird unter Umständen für das *Build-Tool* benötigt.
- **Sicherheit:** Es muss dafür gesorgt werden, dass die Testinstanz geschützt ist, beispielsweise durch passwortgeschützten Zugang.
- **Testwerkzeuge:** Es muss eine Auswahl an Werkzeugen und Bibliotheken für folgende Anwendungen erfolgen: Projektmanagement, Testfallmanagement, Versionskontrolle, [API-Test](#), [GUI-Test](#), [CI-Build-Tool](#), Test Reporting, Testtreiber, Bugtracking und ein Projekt-Build-Tool.

Die Testinstanz sollte, wie auch die Testdaten, nach jedem Testdurchlauf bereinigt werden. [\[ISO13c\]](#) Der Ausgangszustand soll eine Umgebung sein, in der sich keine Testdaten der automatisierten Tests befinden.

### 5.1.13 Personal

Neben benötigter Hardware und Software werden auch Menschen als Ressourcen benötigt, die am Projekt beteiligt sind oder sein werden. Wie bereits beschrieben, stehen für das [CIF](#) momentan zwei Tester zur Verfügung, von denen einer über das benötigte technische Wissen zur Implementierung von Tests verfügt. Für die Phase der Implementierung sollte entweder der zweite Tester ausreichend geschult werden oder ein weiterer Tester aus einem anderen Projekt hinzugezogen werden, um die Implementierung nicht in die Länge zu ziehen. [\[ISO13c\]](#) Zu beachten ist dabei die Tatsache, dass nicht zu viele Tester oder Entwickler an der Implementierung arbeiten, da dies mit einem steigenden Kommunikations- und Planungsaufwand verbunden ist und nicht automatisch zu einer schnelleren Umsetzung führt. [\[SBS07\]](#) Eine weitere Option wäre das Hinzuziehen eines Entwicklers, der bisher nicht selbst am [CIF](#) entwickelt hat, es aber bereits in groben Zügen kennt. Dies soll der möglichen Befangenheit eines Entwicklers bezüglich selbst geschriebenen Codes entgegenwirken. [\[SL19\]](#)[\[KK15\]](#) Zusätzlich erleichtert vorhandenes Wissen über ein System die Arbeit für das Entwickeln von Tests enorm. [\[BR08\]](#) Insgesamt sollte in der Implementierungsphase mehr als ein Tester oder Entwickler die Testfälle entwickeln, da dies häufig gerade bei Regressionstests sehr zeitaufwendig ist. [\[SL19\]](#)

Sind alle Tests implementiert, sollte es ausreichen, wenn ein Tester zuständig ist für die regelmäßige Überprüfung der Testergebnisse und dem Überarbeiten oder Hinzufügen von Testfällen. Falls größere Änderungen oder Erweiterungen vorgenommen werden, kann dieser wieder von weiteren Testern oder Entwicklern unterstützt werden. In der agilen Entwicklung kann gerade auf solche spontan benötigten Ressourcen schnell reagiert werden.



Für die Dokumentation von Fehlern wie in [Unterabschnitt 5.1.10](#) beschrieben, sollten wenn möglich mehrere Tester zuständig sein, sodass Fehler sehr früh ausreichend dokumentiert und eingestuft werden können. Erst dann kann ein Fehler auch behoben werden. [\[SL19\]](#)

Für die Gesamtverantwortung, den Überblick und die Leitung sollte ein Testmanager oder mindestens ein leitender Tester festgelegt werden. [\[SL19\]](#) Die Auswahl wird in der Teststrategie vermerkt. Diese Person, oder eine weitere Einzelperson, sollte zudem für das regelmäßige Auswerten der Testmetriken und das Erstellen des Testberichts verantwortlich sein. Das beteiligte Team sollte ebenfalls regelmäßig über Entscheidungen informiert und wenn möglich mit einbezogen werden. [\[CG14\]](#)

### 5.1.14 Testaufwand

Neben den menschlichen Ressourcen sind auch die aufzuwendende Zeit und Kosten als Ressourcen zu betrachten, die aufgewendet werden müssen. Dies kann anhand einer Schätzung geschehen, die auf Erfahrung oder auf Metriken beruht. Da eine erfahrungsbasierte Schätzung jedoch häufig subjektiv ausfällt und den Aufwand oft unterschätzt, soll hier mithilfe von Metriken eine Schätzung über die benötigten Ressourcen aufgestellt werden. [\[SL19\]](#) Eine erste Schätzung des zeitlichen Aufwands und der damit verbundenen Kosten kann also nach der Auswertung eines prototypischen Testdurchlaufs auf einer daraus resultierenden Datenbasis stattfinden.

### 5.1.15 Ergebnisdokumentation

Die Ergebnisse des Testkonzepts sollten in einigen Dokumenten festgehalten werden, die für alle Beteiligten und wenn möglich für das gesamte Unternehmen zugänglich sind und bestenfalls unter Versionskontrolle stehen. Dadurch soll gewährleistet werden, dass Außenstehende mithilfe der durchgängigen Dokumentation nachvollziehen können, wie vorgegangen wurde und dass neue Tester oder Entwickler eine fundierte Grundlage für ihre Einarbeitung und die Fortsetzung der Projekts haben. [\[ISO13a\]](#)[\[SL19\]](#)

Aus der Testplanungs- und -analysephase sollte eine **Teststrategie** entstehen, in der die Ergebnisse dieses Kapitels festgehalten sind. Aus der als nächstes durchzuführenden Testentwurfsphase sollten folgende Dokumente hervorgehen:

- Ein **Anwendungsfalldokument**, in dem die ausgearbeiteten Anwendungsfälle beschrieben sind.
- Ein **Testfalldokument**, in dem die aus den Anwendungsfällen abgeleiteten Testfälle dokumentiert sind.

- Ein **Testausführungsdokument**, das beschreibt, wie die Testfälle zu strukturieren und auszuführen sind.
- **Testdatendokumente** beziehungsweise -dateien, in denen die Testdaten hinterlegt sind.
- Ein Dokument, in dem die gewählten **Testwerkzeuge** erläutert werden.

Aus der Testumsetzung und -durchführung fließen Testprotokolle, Fehlerberichte und die ausgewerteten Metriken in einen **Testbericht** ein. [ISO13c] In jedem Dokument sollte eine Historie des Dokuments mit Anmerkungen zu Änderungen hinterlegt sein, ebenso wie die für das Dokument verantwortliche Person. Diese sollte mit der Erstellung und Pflege des Dokuments betraut werden. [SL19]

### 5.1.16 Abbruchs-, Wiederaufnahme- und Testabschlusskriterien

In bestimmten Situationen ist es sinnvoll, Tests an einer bestimmten Stelle abubrechen. Diese Situationen sollten vor einer Umsetzung und Ausführung festgelegt werden. [Wit20][SL19] Ein Fall wurde bereits bei den Smoke Tests in [Abbildung 5.3](#) beschrieben. Hier wurde erläutert, dass es wenig Sinn macht, weitere Tests durchzuführen, wenn in einer grundlegenden Funktion ein Fehler gefunden wurde. Deshalb ist einer solchen Stelle ein Testabbruch sinnvoll. Eine weitere Situation ist das Fehlschlagen der statischen Tests, die wie in [Unterabschnitt 5.1.9](#) beschrieben, bereits vor den Smoke Tests stattfinden. Es ist zudem möglich, dass in einer Testausführung Ressourcen zuneige gehen oder kurzfristig stärker belastet sind als üblich. Das betrifft einerseits beteiligte Tester, im Rahmen der Automatisierung aber vor allem das *Build-Tool*, das unter Umständen überlastet sein könnte und deshalb den *Build* abbricht oder pausiert. [Wit20]

Werden Tests nur pausiert, ist es möglich, sie zu einem späteren Zeitpunkt weiter auszuführen. Häufig haben sich zu diesem Zeitpunkt aber bereits die Umstände geändert wie durch eine hinzugefügte Änderung. Eine Wiederaufnahme nach einem Testabbruch ist meist nur dann sinnvoll, wenn die Fehler, wegen denen der Testdurchlauf abgebrochen wurde, behoben wurden. [Wit20] Da bei einem Beheben des Fehlers neue Fehler auftreten können, sollen die Testdurchläufe des **CIF** nach einer Pausierung wieder von Beginn an durchlaufen werden, um alle Funktionen zu verifizieren.

Die Tests gelten als abgeschlossen, wenn für jede Anforderung und somit jede Funktion des **CIF** eine vollständige Testabdeckung erreicht ist. Manuelle Tests werden hinzugezählt. Die Testfälle aller Teststufen müssen vollständig in diesem Testdurchgang durchlaufen worden sein.

### 5.1.17 Abweichungen von der FNT Testpolitik

Da die betriebliche Testpolitik als ein wichtiger Ausgangspunkt für das Erstellen des Testkonzepts gilt, sollte auf Abweichungen von ihr hingewiesen werden. [ISO13c] Das Konzept bewegt sich jedoch im Rahmen der Testpolitik, einige Punkte wurden mithilfe der ISO-Norm 29119 ausgebaut. Dadurch wurde das Konzept detaillierter, als in der Testpolitik oder vom Qualitätssicherungs-CoP festgelegt.

## 5.2 Testentwurf

Auf Grundlage der Ergebnisse aus der Planungsphase sollen nun im Testentwurf die Testfälle, Testdaten und die Testumgebung entsprechend der gewählten Rahmenbedingungen und Testentwurfsverfahren abgeleitet und ausgearbeitet werden. Neben der Phase des Testentwurfs ist auch der Schritt der Testrealisierung teilweise vertreten. [SL19]

### 5.2.1 Anwendungs- und Testfälle

Wie in [Unterabschnitt 5.1.9](#) festgelegt wurde, sollen die Testfälle hauptsächlich anwendungsfallbasiert entworfen werden. Dafür wird im ersten Schritt aus den in [Unterabschnitt 5.1.4](#) aufgezählten Dokumenten der Testbasis ein Anwendungsfalldiagramm entworfen, das alle dort beschriebenen Anwendungsfälle des CIF abdeckt. [MWI17][ISO15] Das Anwendungsfalldiagramm ist zum einen in [Abbildung 5.5](#) abgebildet, aber auch im Anhang [Abschnitt A](#) in größerer Erscheinungsform.

Ein Anwendungsfall, dargestellt als Ellipse, beschreibt dabei das Verhalten einer Aktion, die der Nutzer auf der Oberfläche ausführen kann. Eine solche Aktion steht häufig in Zusammenhang mit anderen Aktionen oder Akteuren. Die hier genutzten Beziehungen sind dabei folgende:

- Assoziation: Die Verbindung von Akteur, dargestellt als Strichmännchen, und Anwendungsfall. Da in diesem Diagramm immer ein Akteur mit genau einem Anwendungsfall interagiert, wurden die Multiplizitäten an den Enden der Striche weggelassen. [KG04][SW06]
- Extend: Anwendungsfälle werden von anderen Anwendungsfällen erweitert. Diese Erweiterungen können optional genutzt werden, hängen aber selbst in ihrer Existenz von dem Fall ab, den sie erweitern. [KG04][SW06]

- Include: Muss ein Anwendungsfall vor einem anderen Anwendungsfall ausgeführt werden, so wird auf diesen ein *include*-Pfeil gerichtet. Dadurch wird direkt das Abhängigkeitsverhältnis deutlich. [KG04][SW06]

Es wurden insgesamt 18 Anwendungsfälle identifiziert, die entweder das CIF betreffen oder mit ihm in direkter Verbindung stehen. Das ist auch anhand der beiden Systemgrenzen in [Abbildung 5.5](#) zu erkennen. Drei Anwendungsfälle liegen direkt im CIF, während die restlichen Anwendungsfälle in Command selbst aufzufinden sind. Gewählt wurden alle möglichen Anwendungsfälle, die aus den Dokumenten der Testbasis hervorgehen. Um deren Vollständigkeit, wie in [Unterabschnitt 5.1.5](#) festgelegt, sicherzustellen, wurden zusätzlich Mitarbeiter befragt und die Oberfläche auf weitere Funktionen untersucht. Es wurden keine weiteren Anwendungsfälle identifiziert, die hier dargestellten Anwendungsfälle sollten vollständig alle vorhandenen Funktionen des CIF abdecken. Allen Fällen wurde ein sprechender Name gegeben.

Zu den Anwendungsfällen wurden drei primäre und ein sekundärer Akteur identifiziert, die mit dem System kommunizieren. [KG04] Primäre Akteure sind dabei der *Benutzer*, die *BGE* und die *Zeit*, da alle drei als Akteur Funktionen direkt initiieren können. Ein sekundärer Akteur ist das *externe System* des Kunden, das auf Abfrage die Daten liefert, die der Kunde in Command integrieren möchte. Die *BGE* und der Benutzer haben hierbei Zugriff auf dieselben Anwendungsfälle und Methoden. Sie sind deshalb zur Vereinfachung als ein Ausgangspunkt dargestellt.

Für die Ausformulierung der Anwendungsfälle wird empfohlen, diese in einem einheitlichen Tabellenschema zu beschreiben. Dabei sollen die einzutragenden Werte projektspezifisch ausgewählt werden. [KG04] Es wird jeder Anwendungsfall zusätzlich zu seinem Namen eine eindeutige ID zugewiesen. Danach wird der Anwendungsfall grob beschrieben und der oder die beteiligten Akteure genannt. Da ein Anwendungsfall erst angestoßen werden muss, um ablaufen zu können, wird das auslösende Ereignis genannt. Ein Anwendungsfall hat zudem meist gewisse Vorbedingungen, die erfüllt sein müssen, damit er ausgeführt werden kann. Diese werden ebenfalls angegeben. [KG04][SL19] Im Anschluss wird zunächst ein Basisszenario geschildert, das meist aus mehreren Schritten besteht und das die Haupthandlung, also die am häufigsten oder wahrscheinlichsten auftretende Handlung, wiedergibt. Oft gibt es zudem noch abweichende Szenarien, die alle im gleichen Zustand enden sollten. Auch Fehlerfall-Szenarien werden geschildert. [KG04][SW06] Im Anschluss daran erweist sich ein Eintrag mit Regeln als sinnvoll, die die Handlung unterstützen und die bei Nichteinhaltung zu Fehlern führen. [ISO15] Für dieses Projekt wurde zudem schon bei der Ausarbeitung der Anwendungsfälle eine mögliche Realisierung von automatisierten Tests bedacht und vermerkt. Für die spätere Einteilung der abzuleitenden Testfälle ist zudem angegeben, ob der Anwendungsfall eine grundlegende CIF-Funktionalität darstellt,

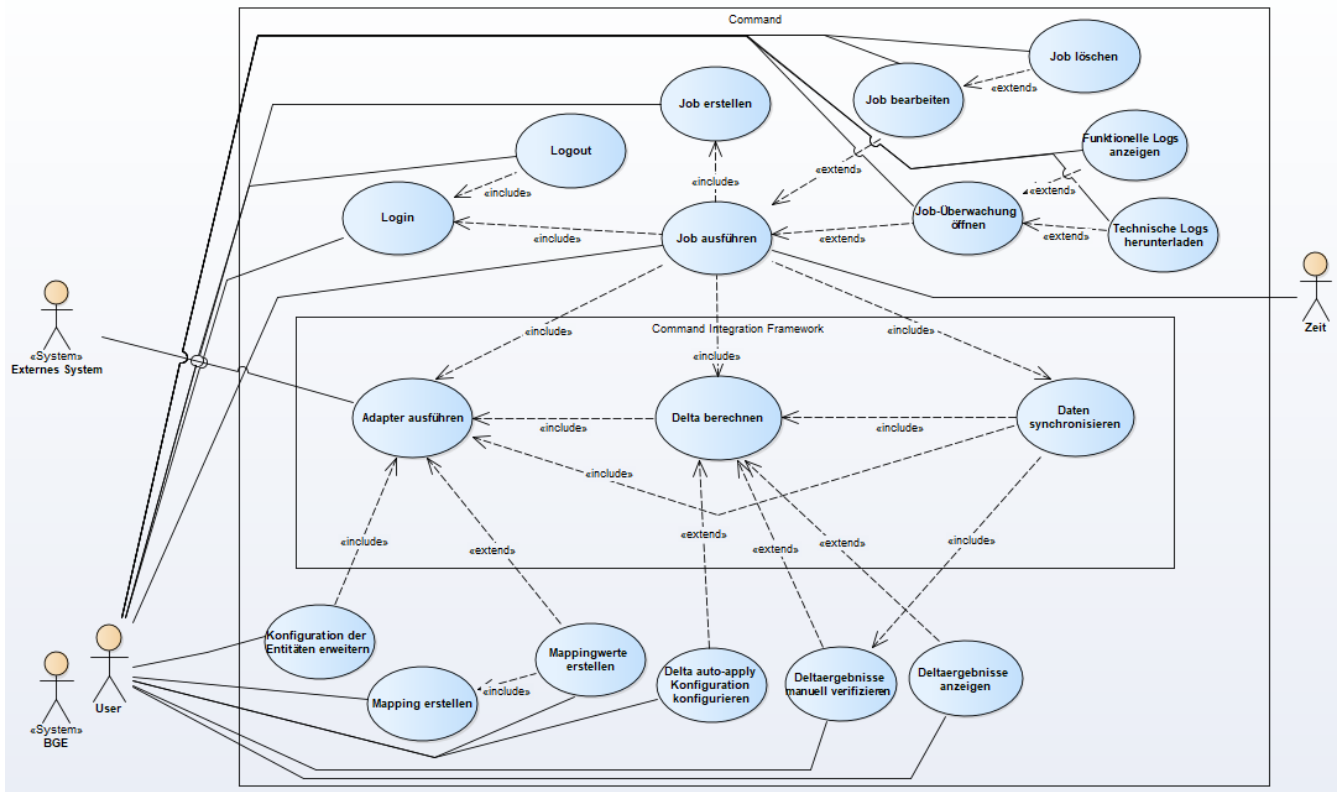


Abbildung 5.5: Das Anwendungsfalldiagramm.

hier also Smoke Tests durchgeführt werden sollten, oder ob die Hauptaufgaben des CIF auch ohne diese Funktionalität funktionieren würden. [Cha14][McC96] Um eine möglichst exakte Testabdeckung zu erreichen sollten die Anwendungsfälle so detailliert wie möglich beschrieben werden.

Das zusätzliche Nennen von Erweiterungspunkten oder der Nutzungsrate der Funktionen, wie in der Literatur teils empfohlen, wurde unterlassen. [KG04][ISO15] Die Information über Erweiterungspunkte liefert an dieser Stelle keinen Mehrwert und ist eindeutig im Diagramm vermerkt. Die Nutzungsrate ist schwer zu bestimmen und durch das regelmäßige Testen aller Funktionen, auch derer, die vermutlich selten benutzt werden, verfällt der Nutzen einer solchen Information an dieser Stelle.

Im Anschluss werden einige Anwendungsfälle exemplarisch erläutert. Anwendungsfälle, die automatisiert getestet werden sollen, aber nicht in diesem Kapitel beschrieben werden, sind im Anhang Abschnitt C zu finden. Zusätzlich werden die Anwendungsfälle in einem Anwendungsfalldokument für alle Mitarbeiter zugänglich festgehalten und dienen als Basis für das Erstellen der Testfälle. Da Anwendungsfälle eine abstrakte Außensicht auf das System darstellen, eignen sie sich besonders für System- und Akzeptanztests. Im Diagramm sind zudem alle Abhängigkeiten zwischen den Fällen gegeben, so können auch Testfälle für andere Stufen wie Integrationstests abgeleitet werden. [SL19]

Durch die detaillierte Ausführung der Anwendungsfälle können für die abzuleitenden Testfälle bereits Ausgangssituation, Vorbedingungen, Nachbedingungen und zu erwartende Resultate festgelegt werden. [SL19] Auch die Schritte der jeweiligen Szenarien bieten eine Grundlage für einzelne Testschritte. So können die meisten Testfälle aus den einzelnen Szenarien und Fehlerfällen abgeleitet werden, oft ergibt sich pro Szenario beziehungsweise Fehlerfall ein Testfall. Es wird somit nicht nur mit den Anforderungen getestet, sondern auch gegen diese. [ISO15][SBC12]

Die Testfälle selbst müssen ausgearbeitet und verfeinert werden. Auch dies sollte über ein einheitliches Template geschehen. Wie in [Abschnitt 3.1](#) bereits erwähnt, nutzt *FNT* die Software *TestRail* um Testfälle zu erstellen und zu managen. Es wurde entschieden, die Testfälle ebenfalls dort zu dokumentieren, um die Dokumentation an dem Ort zugänglich zu machen, wo sie von Testern erwartet wird. In *TestRail* werden folgende Felder ausgefüllt:

- Eine **ID**, welche automatisch von *TestRail* erstellt wird. Eine solche ID fängt mit einem C an, gefolgt von einer Zahlenfolge.
- Der **Titel**, der in einem Satz wiedergeben sollte, was in dem Testfall passiert und was als Ergebnis erwartet wird. Diese Vorgehensweise wird in der *FNT* Testpolitik vorgeschrieben und soll später die Übersichtlichkeit der implementierten Tests erhöhen.
- Die **Referenz** zum jeweiligen Anwendungsfall durch Angabe der Anwendungsfall-ID.
- Eine zusammenfassende **Beschreibung** der durchgeführten Aktion sollte im Normalfall ebenfalls aufgeführt werden. Da aber der sprechende Titel meist bereits alles beschreibt, wird hier nur eine Beschreibung hinzugefügt, wenn der Titel aufgrund der Länge nicht alle Einzelheiten enthält. Sonst würde eine doppelte Beschreibung stattfinden.
- Der **Automatisierungstyp** muss ebenso angegeben werden. Hier sind, wie in [Unterabschnitt 5.1.7](#) erläutert, **API**- und **GUI**-Tests vorgesehen.
- Die **Teststufe**, auf der der Testfall ausgeführt wird.
- Die **Vorbedingungen**, die erfüllt werden müssen, damit der Testfall durchgeführt werden kann. Das sind meist dieselben Vorbedingungen, die auch für den Anwendungsfall gelten, von dem der Testfall abgeleitet wurde.
- **Einzelschritte** des Testfalls und zu jedem Schritt das erwartete *Ergebnis*. Das letzte erwartete Ergebnis ist in dieser Darstellung auch die zu erfüllende Nachbedingung, ohne die der Testfall nicht als bestanden gilt.

Diese Felder sind auch in der [ISO-Norm 29119](#) und von [Spillner und Lutz](#) beschrieben. [\[ISO15\]](#)[\[SL19\]](#)[\[Lou11\]](#)

Die Beschreibung der Testfälle ist sehr detailliert und technisch, während die Anwendungsfälle sich rein auf die vom jeweiligen Akteur durchführbaren Aktionen konzentrieren und diese sehr allgemein beschreiben, wenn möglich ohne technische Details. [\[KG04\]](#)

Bei einer Erweiterung der Funktionen des [CIF](#) muss sukzessiv auch das Anwendungsfalldiagramm erweitert werden. Testfälle werden entsprechend der nachfolgend beschriebenen Vorgehensweise abgeleitet, ausformuliert und anschließend implementiert.

## Login und Logout

Der Login und das Logout sind grundlegende Anwendungsfälle, ohne deren korrekte Funktion keine anderen Fälle ausführbar sind. Sie werden aufgrund ihrer Bedeutung bereits in den Tests berücksichtigt, die Command selbst überprüfen. Dennoch sollten sie hier als Smoke Tests einbezogen werden, da eine weitere Ausführung der Tests bei einer auftretenden Fehlfunktion vor allem beim Login nicht sinnvoll ist.

Anwendungsfall	Einloggen
ID	UC001
Beschreibung	Der Nutzer loggt sich in Command ein.
Akteur	Benutzer
Auslösendes Ereignis	Der Benutzer ist auf der Startseite und wird aufgefordert, sich anzumelden.
Vorbedingungen	Command ist erreichbar.
Schritte Basisszenario	<ol style="list-style-type: none"><li>1. Der Benutzer gibt die erforderlichen Anmeldedaten ein.</li><li>2. Der Benutzer bestätigt die Eingabe.</li><li>3. Der Benutzer ist erfolgreich eingeloggt.</li></ol>
Fehlerfälle	<ul style="list-style-type: none"><li>• Eins oder mehrere der Anmeldedateneingaben sind fehlerhaft.</li><li>• Der Nutzer gibt nicht alle benötigten Daten an.</li></ul>
Regeln	<ul style="list-style-type: none"><li>• Drei Anmeldefelder sind verpflichtend auszufüllen:<ul style="list-style-type: none"><li>– Benutzername</li><li>– Passwort</li><li>– Benutzergruppe</li></ul></li><li>• Das Feld „manId“ ist freiwillig auszufüllen.</li></ul>



Automatisiert testen?	Ja, aber nicht die Fehlerfälle, da diese bereits in den Command-Tests enthalten sind.
Basis des CIF?	Ja.

Tabelle 5.5: Der Anwendungsfall *Einloggen*

Aus dem Anwendungsfall *Einloggen* ergeben sich für das CIF zwei Testfälle, die den Erfolg bei einer richtigen Eingabe überprüfen. Sie beschreiben dabei beide den Login-Vorgang, ein Testfall ist dabei für den Login bei API-Tests vorgesehen, ein Testfall für den Login bei GUI-Tests. Das ist nötig, weil API- und GUI-Tests zu unterschiedlichen Zeitpunkten ausgeführt werden und so für beide Testarten jeweils ein Login nötig ist. Da die Beschreibung des Testfälle für API- und GUI-Tests unterschiedlich ausfällt, sind deshalb zwei einzelne Testfälle erstellt worden. Für API-Tests werden in den einzelnen Schritten die jeweiligen Kommunikationsschritte mit der BGE beschrieben. Bei den GUI-Tests werden genau die Schritte wiedergegeben, die ein Benutzer auf der Oberfläche gehen würde. Der Ansatz der Ausführungen ist also grundlegend verschieden, wie aus den folgenden Testfallbeschreibungen hervorgeht. Das gilt nicht nur für diesen Anwendungsfall, sondern muss bei den anderen Anwendungs- und Testfällen ebenfalls berücksichtigt werden.

Testfall C19273067	Prüfe, dass Errorcode 0 und die Sitzungs-ID zurückgegeben werden, wenn der Login über die BGE durchgeführt wird und erfolgreich ist
Abgeleitet von	UC001
Automatisierungstyp	Der Login soll als API-Test durchgeführt werden, da bei einer Weiterarbeit mit der BGE eine gültige Sitzung nötig ist.
Teststufe	Smoke Test. Wie bereits beschrieben handelt es sich um eine grundlegende Funktion. [Cha14][Mcc96]
Vorbedingungen	Command ist erreichbar
Schritt	Erwartetes Ergebnis
Eine <i>POST</i> -Anfrage wird an den Endpunkt „rest/businessGateway/login“ geschickt. Ein JSON-String mit den entsprechenden Login-Daten wird angehängt.	Die Antwort ist im JSON-Format und beinhaltet einen Status mit Errorcode „0“ und die ID der durch den Login erstellten Sitzung.

Tabelle 5.6: Der erste Testfall zum Anwendungsfall *Einloggen* über die API



Testfall C19640882	Prüfe, dass die Startseite geöffnet wird, wenn der Login über die Oberfläche durchgeführt wird und erfolgreich ist
Abgeleitet von	UC001
Automatisierungstyp	Der Login soll als GUI-Test durchgeführt werden, da bei einer Weiterarbeit mit der Oberfläche in weiteren GUI-Tests eine gültige Sitzung nötig ist.
Teststufe	Smoke Test. Wie bereits beschrieben handelt es sich um eine grundlegende Funktion. [Cha14][Mcc96]
Vorbedingungen	Command ist erreichbar
Schritt	Erwartetes Ergebnis
Benutzername und Passwort werden in die entsprechenden Felder eingetragen und die Eingabe bestätigt.	Die Eingabe wird bestätigt und das nächste Dialogfenster geöffnet.
Die gewünschte Nutzergruppe und der entsprechende Mandant werden auf der Oberfläche ausgewählt und bestätigt.	Die Eingabe wird bestätigt und der Startbildschirm von Command öffnet sich.

Tabelle 5.7: Der zweite Testfall zum Anwendungsfall *Einloggen* über die GUI

Entsprechend existiert der Anwendungsfall *Ausloggen*, der mit der ID *UC001.1* versehen wird. Aus diesem werden, angelehnt an den Login, ebenfalls zwei Testfälle abgeleitet: *C19276348: Prüfe, dass Errorcode 0 zurückgegeben wird, wenn der Logout über die BGE erfolgreich ist* für API-Tests und *C19640882: Prüfe, dass die Login-Seite wieder angezeigt wird, wenn der Logout über die Oberfläche erfolgreich ist* für GUI-Tests. Die Logout-Testfälle werden nicht weiter ausgeführt, sie sollen der Vollständigkeit halber aber erwähnt werden.

Bei den bisherigen Testfällen ist die jeweilige Vorbedingung, dass Command erreichbar ist. Aus diesem Grund wird ein weiterer Testfall abgeleitet, der diese Bedingung prüfen soll. Dieser Testfall ist somit nicht direkt aus dem Anwendungsfalldiagramm abgeleitet, sondern direkt aus einer Vorbedingung entstanden. Erst wenn die Vorbedingungen, das Basisszenario, alle Alternativszenarien und die Fehlerfälle durch mindestens einen Testfall abgedeckt sind, kann ein Anwendungsfall als vollkommen testüberdeckt bezeichnet werden. Gerade deshalb ist auch die detaillierte und vollständige Ausführung der Anwendungsfälle essentiell. [ISO15][SL19]

Testfall C19276350	Prüfe, dass Command erreichbar ist
--------------------	------------------------------------

Abgeleitet von	Den Vorbedingungen von UC001 und UC001.1, wird aber auch für nachfolgende Anwendungs- und Testfälle eine Vorbedingung sein.
Automatisierungstyp	Es reicht ein <a href="#">API</a> -Test aus. Wenn Command erreichbar ist, dann auch automatisch die Oberfläche.
Teststufe	Smoke Test. Wie bereits beschrieben handelt es sich um eine Vorbedingung für alle grundlegenden Funktionen. [ <a href="#">Cha14</a> ][ <a href="#">Mcc96</a> ]
Vorbedingungen	-
Schritt	Erwartetes Ergebnis
Es wird eine <i>GET</i> -Anfrage an die Uniform Resource Locator ( <a href="#">URL</a> ) von Command geschickt	Der <a href="#">HTTP</a> -Status der Anfrage ist 200, also in Ordnung.

Tabelle 5.8: Der Vorbedingungs-Testfall für Command

## Job-Aktionen

Wie bereits in [Abschnitt 2.1](#) beschrieben, werden die Hauptfunktionen des [CIF](#) von einem sogenannten Job gesteuert. Dieser weist unter anderem den Parameter *SOURCE\_SYSTEM* auf, der den Daten ein Quellsystem zuordnet. Auch dies wurde bereits erläutert. Der Wert, der dort eingetragen wird, muss in einem sogenannten Datenlexikon hinterlegt werden, sonst ist er ungültig und kann nicht angewandt werden. Ob der entsprechende Eintrag im Datenlexikon besteht, muss entsprechend überprüft werden. Daraus ergibt sich direkt ein weiterer Testfall.

Testfall C19146924	Überprüfe, dass der den automatisierten Tests zugeordnete Eintrag im Datenlexikon vorhanden ist.
Abgeleitet von	Keinem Anwendungsfall, wird aber für nachfolgende Anwendungs- und Testfälle eine Vorbedingung sein.
Automatisierungstyp	Es reicht ein <a href="#">API</a> -Test aus. Wenn der Eintrag dadurch gefunden werden kann, steht er genauso für Aktionen auf der Oberfläche zur Verfügung, diese greift auf dieselbe Datenbank zu.
Teststufe	Smoke Test. Wie bereits beschrieben handelt es sich um eine Vorbedingung für grundlegende Funktionen. [ <a href="#">Cha14</a> ][ <a href="#">Mcc96</a> ]
Vorbedingungen	Command ist erreichbar und der Login war erfolgreich.
Schritt	Erwartetes Ergebnis
Es wird eine <i>POST</i> -Anfrage an den Endpunkt „entity/dataDictionary/“	Die Antwort ist ein <a href="#">JSON</a> -String, der zum einen den Status mit Errorcode 0

dataDictionary“ geschickt mit angehängter Sitzungs-ID, die beim Login erhalten wurde.	enthält, zum anderen eine Liste mit Datenlexikoneinträgen. In dieser muss sich der Eintrag „CIF_AUTOMATED_TEST“ befinden.
---	---

Tabelle 5.9: Der Vorbedingungs-Testfall für das Datenlexikon

Der Job muss zunächst erstellt werden. Das kann im ersten Schritt ohne jeglichen Zusätze geschehen. Die für die Ausführung des Jobs wichtigen Parameter, auch *Additional Settings* genannt, können alternativ auch später erst hinzugefügt werden. Ebenso wie die für den Job einstellbare Zeitsteuerung.

Name Anwendungsfall	Job erstellen
ID	UC002
Beschreibung	Ein Job wird erstellt.
Akteure	Benutzer, BGE
Auslösendes Ereignis	Der Akteur will einen Job erstellen.
Vorbedingungen	Command ist erreichbar und der Login war erfolgreich. Im Datenlexikon existiert ein Eintrag für das passende Quellsystem.
Schritte Basisszenario	<ol style="list-style-type: none"> <li>1. Der Akteur initiiert die Joberstellung.</li> <li>2. Der Akteur füllt mindestens alle acht benötigten Konfigurationsfelder aus.</li> <li>3. Der Akteur speichert seine Eingaben. Der Job ist erstellt.</li> </ol>
Schritte 1. Alternativszenario	<ol style="list-style-type: none"> <li>3. Der Akteur fügt dem Job die <i>Additional Settings</i> hinzu.</li> <li>4. Der Akteur speichert seine Eingaben. Der Job ist erstellt.</li> </ol>
Schritte 2. Alternativszenario	<ol style="list-style-type: none"> <li>3. Der Akteur stellt eine Zeit ein, zu der der Job regelmäßig gestartet werden soll.</li> <li>4. Der Akteur speichert seine Eingaben. Der Job ist erstellt.</li> </ol>
Fehlerfälle	<ul style="list-style-type: none"> <li>• Nicht alle benötigten Felder sind ausgefüllt.</li> <li>• Der Job existiert bereits.</li> </ul>
Regeln	<p>Folgende Felder sind mindestens auszufüllen:</p> <ul style="list-style-type: none"> <li>• <i>jobAction</i>: Gibt an, welche Aktionen der Job ausführt.</li> </ul>

	<ul style="list-style-type: none"> <li>• <i>active</i>: Gibt an, ob der Job aktiv genutzt wird.</li> <li>• <i>userName</i>: Wie der Login-Benutzername.</li> <li>• <i>jobName</i>: Der Name des Jobs.</li> <li>• <i>groupName</i>: Wie die Login-Benutzergruppe.</li> <li>• <i>type</i>: Der Typ des Jobs.</li> <li>• <i>factory</i>: Hier wird die Klasse hinterlegt, die beim Ausführen des Jobs ausgeführt werden soll.</li> <li>• <i>manId</i>: Wie der Login-Mandant.</li> <li>• <i>sourceSystem</i>: Der Name des Quellsystems, der im Datenlexikon hinterlegt ist.</li> </ul>
Automatisiert testen?	Ja

Tabelle 5.10: Der Anwendungsfall *Job erstellen*

Um die verschiedenen möglichen Szenarien abzudecken, wurden zunächst drei Testfälle erstellt: *Testfall C18943122: Überprüfe, dass der Job erfolgreich erstellt wurde*, der sich aus dem Basisszenario ableitet, *Testfall C19356883: Überprüfe, dass der Job erfolgreich mit Additional Settings erstellt wurde*, abgeleitet vom ersten Alternativszenario, und nach dem zweiten Alternativszenario *Testfall C18943122: Überprüfe, dass der Job erfolgreich mit Zeitsteuerung erstellt wurde*. Es wird hierbei nur getestet, ob die Zeitsteuerung wie vorgegeben dem Job hinzugefügt und aktiviert werden kann, ob sie funktioniert wird dabei nicht automatisiert getestet. Der Aufwand, zu überprüfen, ob ein Job beispielsweise regelmäßig um 12 Uhr läuft, ist manuell deutlich geringer.

Mit Blick auf die Fehlerfälle wurden diesen Testfällen zwei weitere hinzugefügt: *Testfall C19420491: Überprüfe, dass der Job nicht erstellt werden kann, wenn nicht alle Pflichtfelder ausgefüllt sind* und *Testfall C19420492: Überprüfe, dass der Job nicht erstellt wird, wenn er bereits existiert*. Die für die Vorbedingungen benötigten Testfälle sind bereits alle erfasst und beschrieben. Die meisten Testfälle sind als Smoke Tests vorgesehen, da das richtige Anlegen eines Jobs essentiell für die weitere Ausführung ist. Lediglich das zweite Alternativszenario mit der Zeitsteuerung wird als Integrationstest durchgeführt werden, da der Job ohne eine funktionierende Zeitsteuerung grundlegend arbeiten kann.

In [Abbildung 5.5](#) sind neben dem Anwendungsfall *Job erstellen* noch zwei weitere Anwendungsfälle für den Job vorgesehen: *Job bearbeiten* und *Job löschen*. Diese werden als Anwendungsfälle mit den IDs *UC003* und *UC004* erstellt. Die Anwendungsfälle decken dabei das Bearbeiten und Löschen des Jobs, der Additional Settings und der Zeitsteuerung ab. Zusätzlich wird überprüft, ob ein Job dupliziert und exportiert werden kann. Alle Szenarien können automatisiert getestet werden. Es ergeben sich daraus neun [API-Tests](#)

und acht GUI-Tests. Das Duplizieren des Jobs und jegliche Tests, die die Zeitsteuerung betreffen, sind nicht über die BGE erreichbar und müssen daher über die Oberfläche getestet werden. Die meisten der insgesamt 17 Testfälle sind als Integrations- oder E2E-Tests vorgesehen, da eine Änderung am Job nicht essentiell für die gesamte CIF-Funktionalität ist.

Für den Anwendungsfall *Job ausführen* fallen zunächst ausschließlich API-Smoke-Tests an. Dabei soll verifiziert werden, dass der Job grundlegend bei verschiedenen Parameterwerten die richtigen Funktionen aufruft und ausführt. Die Testfälle, die die jeweils vom Job aufgerufenen Funktionen wie die Delta-Berechnung oder die Synchronisation selbst überprüfen sollen, werden an anderer Stelle von den jeweiligen Anwendungsfällen abgeleitet und getestet.

Um überprüfen zu können, ob ein Job ordnungsgemäß durchgelaufen ist und alle Funktionen ausgeführt hat, gibt es das Logging. Dieses zeichnet alle Schritte des Jobs auf. Zu finden ist das Logging in der Job-Überwachung, die sich bei Doppelklick auf einen Job öffnet. Hier sind die einzelnen Testdurchläufe mit Ausführungszeit und Erfolg oder Misserfolg aufgeführt. Da das Öffnen der Job Überwachung die Voraussetzung für das Einsehen der Logs ist, wird sie als Anwendungsfall mit der ID *UC005* erstellt, aus der sich ein GUI-Testfall ergibt. Durch Rechtsklick auf einen Job-Durchlauf öffnet sich ein Kontextmenü. Hier kann zwischen technischem und fachlichen Log gewählt werden. Das fachliche Log ist für den Benutzer bestimmt und auf seinen Kenntnisstand angepasst. Es wird wie in *Abbildung 5.6* als Aufzeichnung der stattgefundenen Schritte in der Bedienoberfläche dargestellt. So kann der Benutzer in der Oberfläche alle Schritte nachvollziehen, die unternommen wurden. Hierbei wird unterschieden zwischen Info-Logs, Warnungs-Logs und Error-Logs, die bei jeweiligen Szenarien erfasst und dargestellt werden. Für das fachliche Log wird ein Anwendungsfall mit der ID *UC005.1* erstellt.






Result - Job log entires				
10 records				
Timestamp	State code		Action	Message
25.08.2021 06:50	10001		Schnittstelle gestartet	
25.08.2021 06:50	12140		Calculate delta	Start delta calculation
25.08.2021 06:50	12031		Calculate delta	Update for auto apply configuration Test_SK1-HARDWA
25.08.2021 06:50	12031		Calculate delta	Update for auto apply configuration AA-1032 executed
25.08.2021 06:50	12031		Calculate delta	Update for auto apply configuration Test_SK1-HARDWA
Close				

Abbildung 5.6: Das fachliche Log, dargestellt in der Oberfläche

Das technische Log zeichnet dagegen vor allem die Daten auf, mit denen innerhalb von Methoden gearbeitet wird. Dem Logger werden meist die Daten übergeben, die auch Methoden übergeben werden oder die diese zurückgeben. So kann nachvollzogen werden, ob beispielsweise alle benötigten Parameter richtig übergeben werden. Das technische Log richtet sich demnach eher an Programmierer oder technisch affine Nutzer. Sie werden während der Ausführung des Jobs in eine Logdatei geschrieben, die dann durch das Kontextmenü heruntergeladen werden kann. Das Herunterladen des technischen Logs wird ebenfalls ein Anwendungsfall mit der ID *UC005.2*.

Aus diesen drei Anwendungsfällen wurden zwei Smoke Tests abgeleitet: *Testfall C18943134: Überprüfe, dass Job-Durchläufe im fachlichen Log angezeigt werden* und *Testfall C19423968: Überprüfe, dass stattgefundenene Aktionen als Infos im fachlichen Log angezeigt werden*. Die Testfälle für das Anzeigen von Warnungen und Fehlermeldungen werden nicht als Smoke Tests eingestuft, da das grundlegende Logging der Info-Logs am Wichtigsten ist. Das Starten des Jobs beispielsweise oder das Aufrufen der Unteraufgaben wird immer von diesen geloggt, unabhängig davon, ob innerhalb des Job-Durchlaufs Fehler auftreten. Solche Rahmenlogs sind wichtig, um den Aufruf grundlegender Funktionen überprüfen zu können. Diese beiden Log-Möglichkeiten werden als Integrationstestfälle *Testfall C19146882: Überprüfe, dass stattgefundenene Aktionen als Warnungen im fachlichen Log angezeigt werden* und *Testfall C19146881: Überprüfe, dass fehlerhafte Aktionen als Fehlermeldungen im fachlichen Log angezeigt werden* erstellt. Zusätzlich entstand *Testfall C19146866: Überprüfe, dass die technische Logdatei für einen Job-Durchlauf heruntergeladen werden kann*. Da über die [BGE](#) nicht geprüft werden kann, ob ein Download der Logdatei stattgefunden hat, wird dieser Test als [GUI-Test](#) durchgeführt werden.

Es fehlt noch der Anwendungsfall *Job ausführen* selbst:

Name Anwendungsfall	Job ausführen
ID	UC006
Beschreibung	Der Job wird gestartet und ruft entsprechenden Funktionalitäten auf.
Akteure	Ein Benutzer, die Zeit oder die BGE.
Auslösendes Ereignis	Der Akteur öffnet den Job.
Vorbedingungen	Command ist erreichbar und der Login war erfolgreich, ein Job wurde mit <i>Additional Settings</i> erstellt. Das Loggen von Ereignissen funktioniert.

Schritte Basisszenario	<ol style="list-style-type: none"> <li>1. Der Akteur gibt für die angegebenen Parameter der Additional Settings entsprechende Werte ein.</li> <li>2. Der Akteur startet den Job.</li> <li>3. Der Akteur überprüft in den Logs, ob die richtigen Funktionen aufgerufen werden.</li> </ol>
Fehlerfälle	<ul style="list-style-type: none"> <li>• Der Adapter ruft eine Funktion nicht entsprechend auf.</li> </ul>
Regeln	<p>Zu folgenden Parametern können Werte angegeben werden:</p> <ul style="list-style-type: none"> <li>• LOAD_DATA</li> <li>• CALCULATE_DELTA</li> <li>• SYNCHRONIZE_DATA</li> <li>• SOURCE_SYSTEM</li> <li>• Mit Angaben zu folgenden Parametern wird festgelegt, in welche NMS-Tabellen die Daten geladen, mit welchen Daten sie verglichen und wohin sie in Command gespeichert werden: LOAD_HARDWARE_DATA, LOAD_ZONE_DATA, LOAD_CABLE_DATA, LOAD_COMMON_DATA, LOAD_DYNAMIC_DATA, LOAD_TELCO_SERVICE_DATA, LOAD_TELCO_LINKS.</li> </ul>
Automatisiert testen?	Ja.

Tabelle 5.11: Der Anwendungsfall *Job ausführen*

Durch Kombination der einzelnen Parameter aus den Regeln des Anwendungsfalls ergeben sich 21 Testfälle für die Smoke Tests. So wird für jede Entität jeweils einmal überprüft, ob die Daten geladen und die Delta-Berechnung und Synchronisation entsprechend ausgeführt wurden. Dabei finden die Test-Durchläufe als *Testrun* statt. Dies ist eine Einstellung, mit der festgelegt wird, dass keine Daten in die Tabellen gespeichert werden, nachdem die Funktion ausgeführt wurde. Es wird am Ende einer Transaktion in den Datenbanktabellen ein *Rollback* statt eines *Commits* ausgeführt. Es soll dadurch nur der Aufruf der Funktion verifiziert werden, das Testen der richtigen Funktionsweise erfolgt jeweils an anderer Stelle. Zusätzlich wird in jedem Testfall nur LOAD\_DATA, CALCULATE\_DELTA oder SYNCHRONIZE\_DATA ausgeführt. Ein durchgängiger Lauf durch alle drei Schichten des CIF wird hier noch nicht geprüft. Für die Systemtests ist jedoch ein solcher Durchlauf vorgesehen.

**Adapter ausführen**

Name Anwendungsfall	Adapter ausführen
ID	UC007
Beschreibung	Der Adapter wird ausgeführt, lädt Daten vom externen System, transformiert sie in das gewünschte Format und speichert sie in die <a href="#">NMS</a> -Tabellen.
Akteure	Benutzer, BGE
Auslösendes Ereignis	Der Akteur führt den Job aus, der den Adapter startet.
Vorbedingungen	Command ist erreichbar und der Login war erfolgreich; der Job wurde erstellt und der Adapter existiert.
Schritte Basisszenario	<ol style="list-style-type: none"> <li>1. Der Adapter lädt Daten vom entsprechenden externen System.</li> <li>2. Der Adapter transformiert die Daten in das passende Format.</li> <li>3. Der Adapter lädt die Daten in diesem Format in die entsprechende <a href="#">NMS</a>-Tabelle.</li> </ol>
Fehlerfälle	<ul style="list-style-type: none"> <li>• Das externe System ist nicht erreichbar.</li> <li>• Die Daten aus dem externen System haben ein anderes Format als erwartet.</li> <li>• Pflichtfelder des in die <a href="#">NMS</a>-Tabelle zu speichernden Objekts fehlen.</li> </ul>
Regeln	Je nach gewählten Parametern im Job kann der Adapter mit Daten aus den in <a href="#">Abschnitt 2.1</a> genannten Entitäten arbeiten.
Automatisiert testen?	Nein, da für jeden Kunden und jedes weitere System ein neuer Adapter implementiert werden muss. Allgemeine, für alle externen Systeme durchführbare automatisierte Tests sind nicht einführbar, da keine wiederholbaren Bedingungen gegeben sind. Neben den bereits vorhandenen Komponententests müssten automatisierte Tests für jeden einzelnen Adapter neu entwickelt werden. Zudem werden gerade an der Schnittstelle zum externen System und an der Beschaffenheit der externen Daten häufig Änderungen vorgenommen, die entsprechend in der Software und dann auch in den automatisierten Tests jedes Mal angepasst werden müssen. Der Aufwand hierfür ist größer als der Aufwand für die bereits durchgeführten strukturierten manuellen Tests für Adapter.

Tabelle 5.12: Der Anwendungsfall *Adapter ausführen*



**Delta berechnen**

Name Anwendungsfall	Delta berechnen
ID	UC008
Beschreibung	Die Unterschiede zwischen den Daten in den <a href="#">NMS</a> -Tabellen und den Daten in Command werden berechnet und in Delta-Tabellen gespeichert.
Akteure	Benutzer, BGE
Auslösendes Ereignis	Der Akteur führt den Job aus, der die Deltaberechnung startet.
Vorbedingungen	Command ist erreichbar und der Login war erfolgreich; der Job wurde erstellt. Es befinden sich entsprechend Daten in Command und den <a href="#">NMS</a> -Tabellen.
Schritte Basisszenario	<ol style="list-style-type: none"> <li>1. Die Delta-Berechnung wird, wie in <a href="#">Abschnitt 2.1</a> beschrieben, durchgeführt. Es ergibt sich je nach Datenlage ein Delta-Fall.</li> <li>2. Der Delta-Fall und weitere Informationen zu den Objekten und deren Delta-Berechnung werden in der jeweiligen Delta-Tabellen gespeichert.</li> </ol>
Fehlerfälle	<ul style="list-style-type: none"> <li>• Die Daten aus dem externen System sind fehlerhaft, sodass Attribute bei der Berechnung fehlen und eine Berechnung nicht möglich machen oder eine falsche Berechnung auslösen.</li> <li>• Die Daten in Command sind fehlerhaft, sodass Attribute bei der Berechnung fehlen und eine Berechnung nicht möglich machen oder eine falsche Berechnung auslösen.</li> </ul>
Regeln	Es ist zu beachten, dass für die verschiedenen Entitäten verschiedene Delta-Tabellen gefüllt werden.
Automatisiert testen?	Ja.

Tabelle 5.13: Der Anwendungsfall *Delta berechnen*

Wie in [Abschnitt 2.1](#) und im Basisszenario beschrieben, ergibt sich je nach Ergebnis der Delta-Berechnung ein anderer Delta-Fall. Um alle möglichen Testfälle für diesen Anwendungsfall abzudecken, wird die Äquivalenzklassenbildung angewandt. Dabei werden die Äquivalenzklassen nach möglichen Eingaben und Ausgaben der Delta-Berechnung gebildet. [\[SL19\]](#)[\[ISO15\]](#)

Die möglichen Eingabedaten sind dabei Objekte aus verschiedenen Entitäten. Die möglichen Ausgabedaten ist der jeweilige Delta-Fall. Es ergeben sich folgende Äquivalenzklassen, dargestellt als Zeilen:

	Entitätsklasse	Delta-Fall
1	Gerätedaten	CREATE
2	Gerätedaten	UPDATE
3	Gerätedaten	UPDATE_TYPE
4	Gerätedaten	NOP_CREATE (No operation)
5	Gerätedaten	NOP_DELETE (No operation)
6	Gerätedaten	NOP_OTHER (No operation)
7	Gerätedaten	PLANNED_CREATE
8	Gerätedaten	PLANNED_DELETE
9	Gerätedaten	PLANNED_DELETE_WITH_CREATE
10	Gerätedaten	PLANNED_DELETE_WITH_PLANNED_CREATE
11	Gerätedaten	PLANNED_CREATE_BUT_WITH_DIFFERENT_TYPE
12	Gerätedaten	PLANNED_DELETE_WITH_PLANNED_CREATE_BUT_WITH_DIFFERENT_TYPE
13	Servicedaten	CREATE
14	Servicedaten	UPDATE
15	Servicedaten	DELETE
<del>16</del>	<del>Servicedaten</del>	<del>ROUTING_MISMATCH</del>
<del>17</del>	<del>Servicedaten</del>	<del>STRUCTURE_AND_ROUTING_MISMATCH</del>
<del>18</del>	<del>Servicedaten</del>	<del>TYPE_AND_ROUTING_MISMATCH</del>
<del>19</del>	<del>Servicedaten</del>	<del>TYPE_AND_STRUCTURE_AND_ROUTING_MISMATCH</del>
20	Kabeldaten	CREATE
21	Kabeldaten	UPDATE
22	Kabeldaten	UPDATE_TYPE

	Entitätsklasse	Delta-Fall
23	Kabeldaten	DELETE
24	Kabeldaten	NOP_CREATE (No operation)
25	Kabeldaten	NOP_DELETE (No operation)
26	Kabeldaten	NOP_OTHER (No operation)
<del>27</del>	<del>Kabeldaten</del>	<del>REPATCH</del>
28	Kabeldaten	PLANNED_CREATE
29	Kabeldaten	PLANNED_DELETE
30	Kabeldaten	PLANNED_REPATCH
31	Zonendaten	CREATE
32	Zonendaten	UPDATE
33	Zonendaten	DELETE
34	Dynamische Daten	CREATE
35	Dynamische Daten	UPDATE
36	Dynamische Daten	DELETE
37	Allgemeine Daten	CREATE
38	Allgemeine Daten	UPDATE
39	Allgemeine Daten	DELETE

Aus jeder hier aufgeführten Äquivalenzklasse wird ein Testfall stellvertretend für diese Klasse abgeleitet, da so ein Fall pro Äquivalenzklasse existiert, gilt der Anwendungsfall als vollständig abgedeckt. [ISO15][Lig09]

Die Entitätsklassen lassen sich noch in weitere Klassen unterteilen. Diese Klassen sind zwar in unterschiedlichen NMS-Tabellen gespeichert, auf die jeweiligen Klassen wird aber die gleiche Delta-Berechnung durchgeführt und sie werden in eine Delta-Tabelle gespeichert. So existiert eine Geräte-Delta-Tabelle, für die eine Geräte-Delta-Berechnung durchgeführt wird, eine Service-Delta-Tabelle, für die eine Service-Delta-Berechnung durchgeführt wird, und so weiter. Für die hier stattfindenden Integrationstests wird deshalb nur eine zur

jeweiligen Entitätsklasse gehörende Klasse getestet, um die allgemeine Funktion der jeweiligen Delta-Berechnung und die Einträge in die Delta-Tabelle für jeden Delta-Fall zu verifizieren. Tests für alle Unterklassen werden an einem anderen Punkt stattfinden. Für die Fehlerfälle werden zusätzlich noch zwei weitere Testfälle abgeleitet, in denen jeweils ein fehlerhafter Datensatz aufgenommen wird. Für die Delta-Berechnung entstehen so 41 Testfälle für API-Tests.

## Synchronisierung

Die gerade beschriebenen Äquivalenzklassen sollen auch als Basis für die Testfälle der Synchronisation dienen. Für jede dort getestete Entitätsklasse wird nun getestet, ob die Daten auch wirklich so in Command gespeichert werden, wie das der jeweilige Delta-Fall in der entsprechenden Delta-Tabelle vorsieht.

Name Anwendungsfall	Daten synchronisieren
Anwendungsfall ID	UC009
Beschreibung	Die Daten aus den Deltatabellen werden entsprechend ihres Delta-Status in Command erstellt, bearbeitet oder gelöscht.
Akteure	Benutzer, BGE
Auslösendes Ereignis	Der Akteur führt den Job aus, welcher die Synchronisation startet.
Vorbedingungen	Command ist erreichbar und der Login war erfolgreich; der Job wurde erstellt. Es befinden sich entsprechend Daten in den Delta-Tabellen. Die Daten wurden zudem für die Synchronisation markiert.
Schritte Basisszenario	1. Die Daten werden entsprechend ihres Delta-Status in Command bearbeitet, erstellt oder gelöscht.
Fehlerfälle	<ul style="list-style-type: none"> <li>• Es werden Daten übernommen, die nicht für die Synchronisation gekennzeichnet waren.</li> <li>• Es werden nicht die Daten übernommen, die für die Synchronisation gekennzeichnet waren.</li> </ul>
Regeln	Die Synchronisation kann für die bestehenden Entitäten entsprechend ausgeführt werden.
Automatisiert testen?	Ja.

Tabelle 5.15: Der Anwendungsfall *Daten synchronisieren*

Es entstehen auch hier 39 Testfälle für [API](#)-Integrationstests aus den Äquivalenzklassen. Die Fehlerfälle werden nicht gesondert getestet, da dies Fehler sind, die nicht simuliert werden können, sondern die während der Durchführung der anderen Testfälle auftreten können.

Wie bereits erwähnt wurde, müssen die Daten, die in der Delta-Tabelle stehen, zunächst für eine Synchronisation gekennzeichnet werden. Dies dient der Verifizierung der Daten, wie in [Abschnitt 2.1](#) erläutert wurde. Das ist sowohl manuell, als auch automatisiert durchführbar.

### **Deltaergebnisse verifizieren**

Für das manuelle Verifizieren der Einträge in der Delta-Tabelle wurde der Anwendungsfall *UC010* erstellt. Die Vorgehensweise sieht das Markieren eines Eintrags durch eine Bejahung des Attributes *Approved* vor. Es wurde dementsprechend der Testfall *C19640880: Prüfe, dass ein ausgewählter Delta-Eintrag als Approved markiert wurde* abgeleitet. Dieser wird in die [API](#)-Smoke-Tests eingeordnet, da ohne eine solche Markierung keine Synchronisierung der Daten stattfinden kann.

Zusätzlich gibt es noch die *Delta auto-apply Konfiguration*, die *UC011* darstellt. In dieser Konfiguration kann eingestellt werden, für welche Delta-Tabelle welche Daten automatisch als *Approved* gekennzeichnet werden sollen. Für alle Delta-Tabellen soll dabei eine Konfiguration für einen Datensatz getestet werden. Da dies keine grundlegende Funktion des [CIF](#) darstellt, entsteht für jede der zwölf Delta-Tabelle ein [API](#)-Integrationstestfall.

### **Job End-to-End-Tests**

Die Systemtests sind, wie in [Unterabschnitt 2.2.4](#) beschrieben, besonders wichtig für die Verifikation der dem Kunden versprochenen Gesamtabläufe im System. Vorgesehen sind Testfälle, die einen kompletten Job-Durchlauf testen. Auch hier wird sich an den in [Abschnitt 5.2.1](#) erstellten Äquivalenzklassen orientiert. Für diese Tests werden die dort angegebenen Entitätsklassen jedoch weiter in ihre Unterklassen unterteilt, sodass jede beteiligte Tabelle und somit jede Kombinationsmöglichkeit mit berücksichtigt wird. Dabei ergeben sich folgende Testfälle:

- Die Geräte werden insgesamt in fünf Geräteklassen unterteilt. Die zwölf dargestellten Äquivalenzklassen werden also verfünffacht. Für Geräte werden somit 60 Testfälle abgeleitet.

- Die Services werden in neun Serviceklassen unterteilt. Mit sieben Äquivalenzklassen ergeben sich 63 Testfälle.
- Die Zonen bestehen aus vier Zonenklassen. Hier ergeben sich 12 Testfälle.
- Die dynamischen Daten bestehen aus zwei Klassen. Mit drei Äquivalenzklassen werden insgesamt sechs Testfälle abgeleitet.
- Die allgemeinen Daten bestehen aus zwei Klassen. Wie bei den dynamischen Daten entstehen sechs Testfälle.

Die Zahl der durchzuführenden Testfälle addiert sich für die [E2E](#)-Tests auf insgesamt 147. Ein solcher Testfall wird beispielsweise folgendermaßen beschrieben:

Testfall C19640285	Überprüfe, dass ein Chassisdatensatz aus der <a href="#">NMS</a> -Tabelle in Command erstellt wird.
Abgeleitet von	UC006
Automatisierungstyp	<a href="#">API</a> -Test.
Teststufe	<a href="#">E2E</a> -Test
Vorbedingungen	Command ist erreichbar, der Login war erfolgreich und das Quellsystem ist im Datenlexikon. Der Job existiert und ein Chassisdatensatz befindet sich in der <a href="#">NMS</a> -Tabelle, während sich in Command kein Datensatz mit dem gleichen <a href="#">NMS</a> -Typen oder gleicher ID befindet. Ein Chassis muss in eine Zone platziert werden, eine Zone muss also existieren.
Schritt	Erwartetes Ergebnis
Die <i>Additional Settings</i> werden über den <a href="#">BGE</a> -Endpunkt „rest/entity/toolFrameworkJob/update“ so gesetzt, dass der Job eine Deltaberechnung für Geräte durchführt, da ein Chassis zu der Entitätsklasse <i>Gerätedaten</i> gehört.	Die Antwort ist im <a href="#">JSON</a> -Format und enthält zum einen den Status mit Errorcode 0, zum anderen eine Liste mit den abgeänderten <i>Additional Settings</i> .
Der Job wird ausgeführt über den <a href="#">BGE</a> -Endpunkt „rest/entity/toolFrameworkJob/startJobSync“.	Die Antwort ist ein <a href="#">JSON</a> -String, in dem unter anderem der Status des Jobs als <i>FINISHED</i> aufgeführt ist.
Es wird über die <a href="#">BGE</a> überprüft, ob der Datensatz nun wie erwartet in der Delta-Tabelle aufgeführt ist.	Der Datensatz ist in der Delta-Tabelle aufgeführt mit dem Delta-Fall <i>CREATE</i> . Dies wurde erwartet, da das Gerät noch nicht in Command aufgeführt war, aber vom externen System vermutlich nach Command integriert werden soll.

Der Eintrag muss auf <i>Approved</i> gesetzt werden, damit er synchronisiert werden kann. Dies geschieht ebenfalls über einen <b>BGE</b> -Endpunkt.	Der Datensatz in der Delta-Tabelle wurde mit dieser Markierung gespeichert.
Die <i>Additional Settings</i> werden erneut über den <b>BGE</b> -Endpunkt so gesetzt, dass der Job nun die Synchronisation für Geräte durchführt.	Die Antwort ist erneut ein <b>JSON</b> -String mit den abgeänderten <i>Additional Settings</i> .
Der Job wird ein zweites Mal durchgeführt.	Die Antwort ist ein <b>JSON</b> -String, in dem unter anderem der Status des Jobs als <i>FINISHED</i> aufgeführt ist. Nun sollte auch das Chassis in der entsprechenden Command-Tabelle erscheinen und aus der Delta-Tabelle verschwunden sein.

Tabelle 5.16: Ein Testfall für **E2E**-Tests

Nach diesem Schema werden die übrigen 146 Testfälle ebenfalls entworfen. Je nach zu berechnendem Delta-Fall wird als Vorbedingung eine andere Kombination an Daten in der **NMS**-Tabelle und in Command benötigt.

## Mappings

In [Abbildung 5.5](#) werden zwei weitere Anwendungsfälle beschrieben, die die Funktion des **CIF** erweitern. Es soll ein Mapping und Mappingwerte erstellt werden. Ein Mapping ist dazu da, Werte, die vom externen System kommen, auf Werte zu mappen, die in Command vorkommen. So können Werte beispielsweise schnell automatisiert transformiert werden. Dazu muss ein Mapping erstellt und *From*-Werte mit *To*-Werten eingetragen werden. Dieses Mapping kann dann im Adapter angegeben, eingezogen und auf Werte angewandt werden. Das Erstellen und Löschen eines Mappings und von Mappingwerten wird automatisiert über die **BGE** getestet werden, es entstehen also vier Integrationstest-Testfälle. Die Anwendung des Mappings findet aber im Adapter statt, weswegen auch dieser Testfall nicht automatisiert wird.

## Konfiguration von Entitäten

Eine weitere Funktion ist die Erweiterung oder Abänderung von Entitäten des **CIF**. Hier kann eine ganze Entitätsklasse oder bei bestehenden Entitäten Attribute hinzugefügt werden. Und dies sowohl in **NMS**-Tabellen, den Delta-Tabellen oder den Command-Tabellen. Mögliche Delta-Fälle für eine solche Konfiguration sind dann aber lediglich

*CREATE*, *UPDATE* und *DELETE*. Da nach einer solchen Konfiguration an den **NMS**-Tabellen zunächst der Command-Applikationsserver einen Neustart benötigt, sind Tests in diesem Bereich manuell durchzuführen. Denn der Applikationsserver ist weder über die **BGE** noch über **GUI**-Tests erreichbar.

### 5.2.2 Gruppierung der Testfälle in Testsuiten

Testfälle sollten nicht zufällig einzeln hintereinander ausgeführt werden, sondern in logischen Gruppierungen. [SL19] Es wäre möglich, sie beispielsweise nach den Anwendungsfällen, von denen sich die Testfälle ableiten, zu gruppieren. Dann würden aber die bereits vorgestellten Teststufen durcheinander durchgeführt werden. Es empfiehlt sich daher, Testfälle nach Teststufen zu gruppieren. [ISO13a] Da zudem zwei unterschiedliche Automatisierungsverfahren angewandt werden, kann hier eine weitere Gruppierung stattfinden. Es werden also Testfälle in folgende Gruppen eingeteilt:

- **API**-Tests
  - Smoke Tests
  - Integrationstests
  - **E2E**-Tests
- **GUI**-Tests
  - Smoke Tests
  - **E2E**-Tests

### 5.2.3 Erstellen der benötigten Testdaten

Testdaten werden für die Ausführung des Jobs und der darin beinhalteten **CIF**-Funktionalitäten benötigt. Die restlichen, im Anwendungsfalldiagramm in Command liegenden, Anwendungsfälle benötigen keine zusätzlichen Testdaten. Wie in den Anforderungen an die Testdaten in **Unterabschnitt 5.1.12** festgehalten ist, werden Daten für die Command-Tabellen und Daten für die **NMS**-Tabellen benötigt. Diese müssen für jede Entitätsklasse einzeln genau aufeinander abgestimmt werden. Deshalb werden die Daten zunächst manuell und statisch erstellt. Eine Testdatengenerierung, wie sie in vielen Projekten angewandt wird, [SL19] soll hier zunächst nicht stattfinden, da keine alle Entitäten umfassenden Regeln existieren, auf deren Grundlage Daten automatisiert erstellt werden können. Regeln müssten für jede Entität einzeln festgelegt werden, was einen ebenso großen Aufwand



bedeutet wie das Erstellen der Testdaten manuell. [SL19] Da die Richtigkeit der Testdaten so essenziell ist, sollten diese regelmäßig von einem oder mehreren Mitarbeitern überprüft werden.

Befüllt werden können die Tabellen entweder im Excel Spreadsheets (XLS)-Format über die Oberfläche oder im JSON-Format über die BGW. Da eine automatisierte Aktion über eine API grundsätzlich schneller ist als über die Bedienoberfläche, [Coh10][Spi17] wurde sich dazu entschieden, die Testdaten in JSON-Dateien zu erstellen. Dabei entsteht jeweils eine Datei für die Command-Daten und eine Datei für die NMS-Daten. So können die in den Dateien hinterlegten Objekte gesammelt über die API in die jeweilige Tabelle gefüllt werden, ohne dass diese im Vorhinein sortiert werden müssen.

### 5.2.4 Aufsetzen der Testumgebung

In Abschnitt 3.2 wurde eine Testinstanz beschrieben, die der Command-Produktivumgebung sehr ähnlich ist und auf der momentan die manuellen Tests durchgeführt werden. Sie erfüllt die in Unterabschnitt 5.1.12 festgelegten Anforderungen an eine Testumgebung: Auf ihr sind sowohl API- als auch GUI-Tests durchführbar, der Aufbau der Testdatenbank entspricht dem der Produktivdatenbank und die Instanz wurde durch ein Passwort geschützt. Zusätzlich wird sie nächtlich gesichert. Auch ein Server für *Build-Prozesse* ist bereits vorhanden. Das Projekt kann an diesen angebunden werden. Die Auswahl der einzusetzenden Testwerkzeuge findet im Folgenden statt.

### 5.2.5 Auswahl der Testwerkzeuge

Wie in Unterabschnitt 5.1.12 und den Beschreibungen der Teststufen bereits festgehalten, werden Testwerkzeuge für verschiedene Bereiche benötigt. Die typische Werkzeugauswahl sieht dabei ein schrittweises Vorgehen vor. Zunächst sollten Anforderungen an die Werkzeuge gesammelt werden. Darauf basierend wird eine Vorauswahl an Werkzeugen getroffen werden, die sich auf dem Markt befinden. Die dabei eingegrenzte Menge an möglichen Werkzeugen kann dann genauer untersucht und Werkzeuge ausgeschlossen werden. Mit den schlussendlich gewählten Werkzeugen wird ein Probeprojekt aufgesetzt und die Machbarkeit und das Zusammenspiel der Werkzeuge überprüft. Ist dieses Projekt erfolgreich, kann eine Breitereinführung der Werkzeuge stattfinden. [SL19][KG13]

Zusätzlich zu den in Unterabschnitt 5.1.12 festgelegten Anforderungen an die Kategorien der Werkzeuge müssen folgende Rahmenbedingungen eingehalten werden:

- Das [CIF](#) wird in *Eclipse* und mit *Java* 11 programmiert. *Java* sollte weiterhin als Programmiersprache verwendet werden und die auszuwählenden Werkzeuge sollten in *Eclipse* integrierbar sein.
- Einfaches Einbinden von neuen Werkzeugen sollte aufwendigen Installationen vorgezogen werden.
- Langes Einarbeiten in den Entwicklern und Testern unbekannte Werkzeuge sollte vermieden werden. Die Werkzeuge sollten wenn möglich an Bekanntes grenzen, bereits bekannt oder schnell zu erfassen sein. [\[BR08\]](#) Wie Louridas in seinem Artikel ausführt, kommt es meist nicht nur auf die technischen Möglichkeiten eines Werkzeugs an, sondern auch darauf, wie effektiv das Werkzeug genutzt und in den Entwicklungs- und Testprozess integriert wird. Ein Ausbrechen aus bereits vertrauten und gut funktionierenden Werkzeugen sollte deshalb jeweils gut durchdacht werden. [\[Lou11\]](#) Dies ist auch in der letzten Anforderung in [Kapitel 4](#) festgehalten.
- Die Testwerkzeuge sollten in ihrer Funktionsweise und Kommunikation aufeinander abgestimmt sein und möglichst einfach und effizient miteinander arbeiten können.
- Die festgelegten Teststufen und Test- und Automatisierungsarten sollten realisierbar sein.
- Auch ein langfristiger und breiter Support und eine umfassende Dokumentation sollten für das jeweilige Werkzeug verfügbar sein. [\[SL19\]](#)

Die endgültige Auswahl der Testwerkzeuge und ihre geplante Kommunikation untereinander ist nach der Evaluierung dieser graphisch zusammengefasst in [Abbildung 5.7](#) dargestellt.

### Projektmanagement und Fehlertracking

Für das Projektmanagement wird bei *FNT* die Software *Jira* vom Unternehmen *Atlassian* verwendet. *Jira* ist weltweit eines der meistgenutzten Werkzeuge für das strukturierte Planen und Überwachen von Projekten im agilen Umfeld. Es bietet eine webbasierte Oberfläche, die Teams in ihrer täglichen Arbeit unterstützt. Hier werden Projekte in kleinere Aufgaben, den *User Stories* unterteilt, dokumentiert und gepflegt. Entdeckte Fehler, hier als *Bugs* bezeichnet, werden direkt bei der entsprechenden Story hinterlegt und können dort einem Entwickler zum Lösen zugewiesen werden. Eine Einteilung der Fehler findet wie in [Unterabschnitt 5.1.10](#) beschrieben in fünf Priorisierungsstufen statt. Zudem bietet es viele Integrationsmöglichkeiten mit anderen Werkzeugen und kann an unternehmensspezifische Anforderungen angepasst werden. [\[AtloJ\]](#)[\[ÖM19\]](#)

*Jira* wird seit fast zehn Jahren bei *FNT* für anfallende Projekte eingesetzt und soll für das Projektmanagement und das Festhalten der gefundenen Fehlern der *CIF*-Tests genutzt werden. Es ist bei Entwicklern und Testern bereits bekannt und das Nutzen eines weiteren Projektmanagementwerkzeugs für ein einziges Projekt ist nicht zweckmäßig. Durch die Wahl von *Jira* werden die aufgestellten Anforderungen erfüllt. Bekannte Alternativen wie *VersionOne*, *Microsoft Visual Studio Team Services*, *Trello* oder *SpiraTeam* sind durchaus auch sehr mächtig, in direkten Vergleichen bietet aber keines dieser Tools eine so große Funktionspalette wie *Jira*. [ÖM19][Mih17]

Daraus ergibt sich für viele der anderen Werkzeuge die zusätzliche Anforderung, mit *Jira* integrierbar zu sein.

### Testfallmanagement und Testberichte

Wie bereits bei der Ausarbeitung der Testfälle in [Unterabschnitt 5.2.1](#) erwähnt, wird für das Management von Testfällen *TestRail* verwendet. Für die Auswahl wurden zunächst ergänzende Anforderungen festgelegt. Das Werkzeug sollte mit *Jira* kommunizieren, eine Verbindung von Testfällen zu Anforderungen herstellen, Testfälle detailliert dokumentieren und Testergebnisse verwalteten und auswerten können. Tester, aber auch Entwickler, sollten Zugriff darauf haben. Darüber hinaus war zu beachten, wie aufwendig eine Inbetriebnahme eines solchen Systems ist und welche Kosten auftreten werden. [Lou11][SL19]

Da *FNT* *TestRail* bereits seit 2017 nutzt und inzwischen jegliche Testprojekte dort hinterlegt und dokumentiert werden, wurde dieses zuerst genauer betrachtet. *TestRail* ist eine Testmanagement Software des Unternehmens *Gurock*. Es bietet eine Anbindung an *Jira* und eine Referenzierung der jeweiligen Anwendungsfällen. *TestRail* ist eine Webanwendung, die eine detaillierte Beschreibung und Ordnung von Testfällen ermöglicht. [Lou11] *FNT* hat eine Lizenz für die Nutzung von *TestRail* erworben. Es läuft auf einem von *FNT* gehosteten Server. Die Entwickler und Tester sind mit dem Umgang vertraut und nutzen *TestRail* regelmäßig. Ein Umstieg auf eine andere Software wie die ebenfalls häufig genutzten Produkte *QMetry*, *TestLink* oder *Seapine TestTrack*, die grundsätzlich dieselben Aufgaben erfüllen, [Lou11][SG15] würde zusätzlich Zeit und Geld kosten, was nicht zielführend erscheint.

### Versionsverwaltung

Für die Versionsverwaltung des Quellcodes nutzt *FNT* bereits seit mehr als fünf Jahren *Bitbucket*. *Bitbucket* ist eine webbasierte Software von *Atlassian* und bietet über die Versionsverwaltung hinaus eine Anbindung über *Webhooks* an verschiedene *CI-Build-Tools*

und die Möglichkeit der Kollaboration zwischen Entwicklern. [Atl20][Dee+20] Alternative Werkzeuge wären beispielsweise *GIT* oder *Apache Subversion*, die in ihren Funktionen teilweise mehr bieten als *Bitbucket*. [Dee+20] Da sich aber alle Projekte und der gesamte *Command* Quellcode bereits in *Bitbucket* befinden, sollte auch das Testprojekt, das sich auf Code innerhalb von *Bitbucket* bezieht, wie die bereits bestehenden Testprojekte dort angelegt werden.

Die in [Unterabschnitt 5.1.15](#) beschriebenen erstellten oder zu erstellenden Dokumente werden in *SharePoint* hinterlegt. *SharePoint* ist eine Software von *Microsoft*, die unter anderem kollaboratives Arbeiten und Teilen von Wissen und Informationen in einem Unternehmen ermöglicht. [KM19] Bei *FNT* wird *SharePoint* bereits seit einigen Jahren als zentraler Speicherort für Projekt-, Entwicklungs- und Unternehmensinformationen verwendet. Da hier eine Versionsverwaltung der Dokumente integriert ist, [KM19] eignet es sich sehr gut auch für dieses Projekt.

## CI-Build-Tool

Das *Build-Tool* für die *Continuous Integration* ist bei *FNT* die Open-Source-Software *Jenkins*. *Jenkins* ist aufgrund seiner zahlreichen Funktionen die verbreitetste Lösung für CI-Server. [AP18][Aks+15] Mit *Jenkins* können Projekte automatisiert gebaut, getestet und bei Bedarf in eine Produktivumgebung integriert werden. Letztere Funktion wird auch *Continuous Deployment* genannt. [SAZ17] *Jenkins* bietet zudem die Möglichkeit, mit *Bitbucket* zu interagieren. [Atl21]

Weitere häufig genutzte und hoch funktionelle Alternativen sind *Bamboo*, das ebenso wie *Jira* und *Bitbucket* von *Atlassian* entwickelt wird und deshalb eine gute Integration beider Werkzeuge bieten würde, [Ebe+16] *Hudson* [SAZ17] und *TeamCity*, welches für eine flexible Nutzung bekannt ist. [Ebe+16]

Im Vergleich schneidet *Jenkins* aber gerade in den Bereichen Performance, Kosten, Nutzungsfreundlichkeit und guten Support-Möglichkeiten am besten ab. [Ebe+16] Zudem ist *Jenkins* flexibel an unternehmensspezifische Anforderungen anpassbar und mit sehr vielen Anwendungen und Sprachen kompatibel. [Aks+15] Aus diesem Grund und weil auch das *CIF* dort gebaut wird, wird *Jenkins* beim Testen des *CIF* verwendet werden.

## Werkzeuge für statische Tests

Für die Umsetzung der in [Unterabschnitt 5.1.9](#) erläuterten statischen Tests müssen ebenfalls Testwerkzeuge evaluiert werden. Um zu überprüfen, ob Coderichtlinien eingehalten werden, verwendet *FNT* für *Command* und auch für die bisherige Entwicklung des *CIF* die

Software *SonarQube*. *SonarQube* ist eine der am meisten genutzten Softwarelösungen für statische Codeanalyse. [Len+21] Es sucht anhand von gegebenen Regeln zu Zuverlässigkeit, Wartbarkeit und Sicherheit im Code nach Verletzungen dieser. [SAZ17][Len+21] Dafür gibt es bereits über 200 Standardregeln für *Java*, die direkt angewandt werden können. [Len+20] Lenarduzzi vergleicht 2021 die sechs Softwarelösungen *SonarQube*, *Better Code Hub*, *Coverity Scan*, *Findbugs*, *PMD*, und *CheckStyle*. Dabei schneiden die Standardregeln von *SonarQube* mit einer Genauigkeit von 18% als ungenauestes Werkzeug ab. Den Grund dafür sieht Lenarduzzi unter anderem darin, dass diese Standardregeln sehr detailliert sind und so zu vielen Falsch-positiven Ergebnissen führen. [Len+21] Deshalb sollten bei einer Nutzung von *SonarQube* die Regeln unternehmens- und projektspezifisch angepasst werden, um die Genauigkeit und Zuverlässigkeit zu erhöhen. [Len+20] Da *FNT SonarQube* seit Jahren nutzt und die Regeln über die Zeit so angepasst hat, dass *SonarQube* beim Finden der Fehler eine hohe Zuverlässigkeit aufweist, kann es auch für die Tests des *CIF* angewandt werden. Bei einer Einführung eines anderen Analysewerkzeugs müssten für dieses erst einmal die Regeln angepasst werden, was in einer doppelten Arbeit resultieren würde und nicht ressourcenschonend wäre.

Um Code in ein einheitliches Format zu bringen, verwendet *FNT* zusätzlich die Software *Spotless*. Das ist ein Werkzeug, das Abweichungen von Formatvorgaben nicht nur findet, sondern diese auch direkt behebt. So kann ein Projekt automatisiert in ein einheitliches Format gebracht werden. [Dif+21] *FNT* hat auch *Spotless* an unternehmensspezifische Formatvorgaben angepasst. *Spotless* zu integrieren ist sehr simpel und mit keinem weiteren Aufwand verbunden. Auch dieses Werkzeug wird bereits für den Code des *CIF* genutzt und soll auch auf den Quellcode der Tests angewandt werden.

## Testtreiber

Um die Ausführung von Testfällen konkurrieren hauptsächlich die zwei Frameworks *JUnit* und *TestNG*. [Mad+21][ZM17] *TestNG* wurde 2004 von Cédric Beust aus *JUnit* entwickelt, da dieser einige Funktionen bei *JUnit* vermisste. Lang galt *TestNG* *JUnit* deshalb in folgenden Aspekten überlegen: Die Menge an Annotationen, die Möglichkeiten zur Parametrisierung von Testfällen, das Ausführen von Testfällen in einer gewissen Reihenfolge und eine parallele Ausführung der Testfälle. Während *JUnit* zudem nur für Komponenten- und teilweise für Integrationstests geeignet war, konnte *TestNG* auch für höhere Teststufen eingesetzt werden. [Beu04] Mit der Einführung von *JUnit 5* wurden jedoch viele Änderungen eingeführt, die die bisherige Überlegenheit von *TestNG* ausglich. So existieren bei *JUnit 5* weitaus mehr Annotationen als in den vorherigen Versionen, eine parallele Testausführung ist möglich und das Festlegen der Testreihenfolge ebenso. Auch die Optionen für Parametrisierung von Testfällen wurden erweitert. [Bec+21] Dadurch

sind auch Tests auf höheren Teststufen mit *JUnit 5* durchführbar. Inzwischen sind *TestNG* und *JUnit 5* in ihrer Anwendung und ihren Funktionen nahezu gleich. In zwei Studien 2017 und 2021 wurde jedoch festgestellt, dass *TestNG* bedeutend seltener in Projekten verwendet wird als *JUnit*. [Mad+21][ZM17]

Für die Entscheidung für eines der beiden Frameworks wurden die Dokumentation und der mögliche Support verglichen. Während die Dokumentation von *JUnit 5* sehr aktuell, detailliert und optisch aufbereitet ist, [Bec+21] ist die Dokumentation von *TestNG* zwar auch sehr ausführlich, jedoch zuletzt 2012 aktualisiert und viele der Links in der Dokumentation führen nicht mehr zu der erwarteten Ressource. [Beu04][Beu12] *TestNG* wird zudem hauptsächlich von einer Person entwickelt und gepflegt, hinter *JUnit* steht eine Reihe an Entwicklern. Ein wichtiges Kriterium in den obigen Anforderungen war ein langfristiger und umfassender Support. Dieser scheint bei *JUnit 5* gesichert. Als letzten Grund für die Wahl von *JUnit 5* ist die Erfahrung zu nennen, die die Entwickler damit bei *FNT* haben. Die Komponententests für Command und das *CIF* werden mit *JUnit* geschrieben. Hier liegt bei den meisten Entwicklern also eine jahrelange Erfahrung vor.

## Projekt-Build-Tool

Das Bauen eines Projekts ist einer der wichtigsten Schritte bei der Erstellung eines Softwareprodukts. Ein *Build-Tool* kompiliert ein Projekt nicht nur, durch ein solches Werkzeug ist es auch möglich, Tests durchzuführen und *Dependencies* einzubinden. Deshalb ist es wichtig, ein passendes Werkzeug für das Projekt zu finden. [Dav20][GS17] Die drei bekanntesten Projekt-Build-Tools sind *Gradle*, *Maven* und *Ant*.

*Ant* war das erste Projekt-Build-Tool für *Java*. Damit Aufgaben ausgeführt werden können, werden diese als *Tasks* in Extensible Markup Language (*XML*)-Dateien verfasst. *Tasks* stellen dabei eine Art Arbeitsschritt dar oder eine Gruppe an auszuführenden, zueinander gehörenden Aufgaben. [Dav20] *Ant* gilt als sehr flexibel und leicht erweiterbar. [Aks+15] Das Schreiben des Build-Skripts in *XML* ist einfach, aber kann schnell sehr umfangreich werden. [Ebe+16]

*Maven* ist ein ebenfalls *XML*-basiertes Projektmanagement- und *Build-Tool*, das nicht nur *Java*-Projekte bauen kann. *Maven* bietet einen einfachen Weg, *Dependencies* in ein Projekt zu integrieren und zu managen. Es ist auch möglich, eine Bibliothek aus der *Maven Central*, einer Sammlung an Bibliotheken, in ein Projekt einzubeziehen. Es ist mit vielen anderen Werkzeugen und Sprachen vereinbar. [Dav20][Aks+15] Zudem sollten die im Rahmen von *Maven* verfassten *XML*-Skripte nicht so umfangreich werden wie die Skripte von *Ant*. Deshalb wird hier von *Tasks* abgesehen und mit dem *Project Object Model*

eine einheitliche Vorgehensweise für den Bau von Systemen beschrieben. Diese Vorgaben machen das Projekt aber für benutzerspezifische Anpassungen unflexibel. [Ebe+16]

*Gradle* ist ein Projekt-*Build-Tool*, dessen *Builds* auf Dokumenten beruhen, die in einer *Groovy*-basierten Sprache verfasst sind. [Dav20][BM11] Es baut auf die Techniken von *Ant* und *Maven* auf. [Aks+15] Wie in *Ant* werden die Aufgaben in *Tasks* verfasst. *Gradle*-Projekte sind über das Einbinden von *Plugins* und *Dependencies* erweiterbar. [BM11] Auch *Maven Central* kann eingezogen und genutzt werden. [Dav20] Durch Verfassen der Skripte in *Groovy* fällt der Umfang eines Skripts deutlich geringer aus als bei *Ant* oder *Maven*. [Ebe+16]

*Eclipse* unterstützt *Maven*- und *Gradle*-Projekte standardmäßig und bietet das automatisierte Erstellen des jeweiligen Projekts an. Auch hier wird die Entscheidung für ein Werkzeug hauptsächlich an die Erfahrung der Entwickler und Tester angelehnt. Command wird von *Gradle* gebaut, ebenso das CIF. Entwickler und Tester kennen *Gradle* also bereits und nutzen es häufig. Die Einführung eines *Build-Tools* nur für dieses Testprojekt scheint keinen Vorteil zu bringen. Entschieden wurde sich für die Ende Juli 2021 aktuellste Version von *Gradle*, Version 7.1.1. [Gra21]

## API-Tests

Um geeignete Werkzeuge für API-Tests auswählen zu können, muss zunächst analysiert werden, wie API-Tests grundlegend durchgeführt werden und welche Werkzeuge genau benötigt werden. Belorusetz beschreibt dafür in seinem Artikel vier Schritte, die sich aus dem natürlichen Verhalten einer API ergeben:

1. Es muss herausgefunden werden, wie das Format der geplanten REST-Anfrage aussieht und ob sie manuell wie gewünscht funktioniert. Das ist über einen REST-Client möglich.
2. Die zurückgegebene Antwort auf die REST-Anfrage sollte auf ein Objekt einer Java-Klasse gemappt werden. Eine solche Klasse wird auch Plain Old Java Object (POJO) genannt und muss, wenn nicht bereits vorhanden, so erstellt werden, dass sie die Struktur des zurückgegebenen JSON-Strings wiedergibt. Eine Variable der Klasse entspricht in einem POJO einem JSON-Attribut. So sind entweder ganze Objekte oder einzelne Objektattribute direkt vergleichbar. Denn Strings zu vergleichen führt oft zu Fehlern, da diese teilweise unterschiedlich von der API zurückgegeben werden. Es wird ein Werkzeug benötigt, mit dem die Antwort gelesen und in ein entsprechendes Objekt gespeichert werden kann.
3. Die REST-Abfrage muss mit einem geeigneten Werkzeug programmiert werden.



4. Der Testfall wird erstellt, in dem die [REST](#)-Abfrage gesendet und die Antwort mit einem erwarteten Wert verglichen wird. Das Testwerkzeug hierfür ist mit *JUnit 5* bereits vorhanden.

[\[Bel15\]](#)

Die [BGE](#) bietet einen [REST](#)-Client, über den der erste Schritt durchführbar ist.

Zwei häufig genutzte Werkzeuge für das Mappen von [JSON](#)-Strings auf *Java*-Objekte sind *Jackson* und *GSON*. *Jackson* ist eine Bibliothek, die verschiedene Möglichkeiten bietet, mit [JSON](#)-Strings zu arbeiten. Dabei setzt *Jackson* auf Schnelligkeit, Richtigkeit und Einfachheit. [\[Mae12\]\[Bal17\]](#) *GSON* ist eine schlanke, einfach gehaltene Bibliothek, die auch willkürliche [JSON](#)-Objekte verarbeitet. Die Umsetzung einer [REST](#)-Anfrage ist über das Aufrufen von nur zwei Methoden möglich. [\[Mae12\]](#) Komplizierte Abfragen sind jedoch dagegen nur schwer umzusetzen. [\[bae19\]](#) In ihrer grundlegenden Funktionsweise sind sich *Jackson* und *GSON* sehr ähnlich [\[bae19\]](#), in der Schnelligkeit schlägt *Jackson* *GSON* jedoch eindeutig. [\[Mae12\]](#) Das spricht für eine Verwendung von *Jackson*.

Für das Erstellen und Senden einer [REST](#)-Abfrage und das Empfangen der entsprechenden Antwort gibt es das Framework Jakarta RESTful Web Services ([JAX-RS](#)). Dies ist die Spezifikation einer Schnittstelle, durch die *Java*-Objekte mithilfe von Annotationen an URIs und [HTTP](#)-Anfragen geknüpft werden können. Dadurch wird das Senden und Empfangen von Anfragen standardisiert und vereinfacht. [\[Vel+18\]\[Li11\]](#) Für das [API](#)-Testen gibt es einige Implementierungen von [JAX-RS](#), die dessen Funktionen vereinfachen oder ausbauen. Zu nennen sind *Jersey* vom Unternehmen *Oracle* und *RestAssured*, die einen schlanken Ansatz bieten, und *Apache CXF* und *RESTEasy*, sie selbst als Framework agieren.

*Jersey* wird häufig eingesetzt und ermöglicht das Arbeiten mit [REST](#)-Anfragen auf Server- und auf Client-Seite. Es bietet eine solide Grundlage für [API](#)-Tests, ist jedoch langsam und bei zunehmender Anzahl an [REST](#)-Anfragen instabil. [\[Vel+18\]](#) *RestAssured* ist eine besonders schlanke Implementierung, die sich auf das Senden und Empfangen der wichtigsten [REST](#)-Anfragen beschränkt. Es basiert auf *Java* und bietet auch Lösungen zur Validierung der Antworten an. [\[Hal21\]](#) *RESTEasy* beinhaltet über das Senden und Empfangen von [REST](#)-Anfragen hinaus einige nützliche Bibliotheken und Werkzeuge, unter anderem ist *Jackson* standardmäßig enthalten. [\[Red21\]\[Li11\]](#) Auch *Apache CXF* beinhaltet eine Vielzahl an Funktionen und unterstützt nicht nur Webservices im [JSON](#)-, sondern auch im [XML](#)-Format. [\[Vel+18\]](#)

In einem direkten Vergleich der Schnelligkeit und Stabilität der Werkzeuge schneiden *Apache CXF* und *RESTEasy* am besten ab. [\[Vel+18\]](#) Da durch die Einbindung der *RESTEasy-Dependency* auch direkt *Jackson* in das Projekt integriert ist, wurde sich für dieses Framework entschieden. Ein weiterer Faktor war die Tatsache, dass für die



Entwicklung der **BGW** ebenfalls *RESTEasy* genutzt wird. Sollte ein Entwickler an der Implementierung der Tests beteiligt sein, spart dies Einarbeitungsaufwand.

## GUI-Tests

Für **GUI**-Tests gibt es grundlegend zwei Werkzeugarten: Erstens die sogenannten *Record-Playback*-Werkzeuge, mithilfe derer die Testfälle manuell auf der Oberfläche durchlaufen und währenddessen aufgezeichnet werden können. Bei einer erneuten Ausführung wiederholt das Werkzeug diese Schritte selbstständig. Zweitens kann die Vorgehensweise auf der Oberfläche mithilfe eines Werkzeugs auch Schritt für Schritt in einem Skript beschrieben werden. [KG13]

Für das **CIF** wurden einige Tools analysiert. *Selenium* ist die zurzeit meistgenutzte Open Source Software zur Browserautomatisierung. [KG13][SG17] Es ist möglich, Tests sowohl manuell zu verfassen, als auch über *Record-Playback*. Dabei werden acht Programmiersprachen unterstützt. Ausgeführt werden die Tests, indem *Selenium* eine Browserinstanz öffnet und an diese die entsprechenden Befehle sendet. *Selenium* ist mit vielen anderen Anwendungen und Browsern kompatibel, auch wenn die Anbindung meist sehr aufwendig zu programmieren ist. [SG17][Fui19]

*Quick Test Professional* ist ein proprietäres *Record-Playback*-Werkzeug der oberen Preisklasse. [Fui19] Es identifiziert Website-Bausteine über ihre ID oder ihren Namen. Es bietet zusätzlich die Möglichkeit beispielsweise Datenbanktests durchzuführen. [KG13] Es wurde inzwischen in *Unified Functional Testing (UFT)* umbenannt [SG17] und wird wegen seiner übersichtlichen Bedienoberfläche gerade für Einsteiger oder Tester mit wenig Programmiererfahrung empfohlen, ist jedoch sehr teuer und unterstützt nur drei eher unbekannte Sprachen. Über Plugins sind diese erweiterbar, Anbindungen von anderen Anwendungen sind jedoch nur eingeschränkt möglich. [Fui19] Sabev sieht zudem das Problem, dass die Ansprüche der agilen Entwicklung mit einem solch teuren Werkzeug unter Umständen nicht erreicht werden, da so nur wenige Menschen in einem Unternehmen einen Zugang zu der Software bekommen, während im agilen Umfeld eigentlich das ganze Team mit einbezogen werden sollte. [SG17]

*Test Complete* ist ebenfalls ein Werkzeug, mit dem sowohl manuell Skripte erstellt werden können oder über *Record-Playback*. Obwohl es sehr einfach gehalten ist und eine Anbindung an verschiedene andere Anwendungen möglich ist, [Fui19][KG13] kann es in Schnelligkeit und Zuverlässigkeit nicht mit den beiden bereits genannten Werkzeugen mithalten. [SG17]

*Ranorex* ist ein proprietäres Framework, das nicht nur die Ausführung von **GUI**-Tests für Desktop-, Web- und Mobile Anwendungen ermöglicht, sondern auch von Komponententests. Die Installation ist sehr einfach und zu *Ranorex* lässt sich sehr viel an ausführlicher

Dokumentation im Internet finden. Es unterstützt verschiedene gängige Programmiersprachen und die Tests sind einfach ausführbar. [Fui19] Ein Nachteil ist die unzuverlässige Ausführung. Zudem ist das Anbinden von Plugins nicht möglich. [SG17]

In einer Studie 2017 verglich Sabev die genannten Werkzeuge miteinander. *UFT* und *Selenium* schnitten dabei als die beliebtesten Produkte ab. Die Installation und Konfiguration von *Selenium* scheint aber komplexer als bei den anderen Werkzeugen. Die Benutzerfreundlichkeit nach einer Installation wird aber wieder höher eingestuft. *Ranorex* und *Selenium* sind die effizientesten, also schnellsten Werkzeuge. [SG17]

Insgesamt scheint *Selenium* das zuverlässigste Werkzeug zu sein. Da aber die Installation sehr aufwendig ist, wurde sich dafür entschieden, das Test-Framework *Selenide* zu verwenden. [Gar+20] *Selenide* vom Unternehmen *Codeborne* basiert auf Selenium, ist aber über eine Dependency schnell in ein Projekt einzubinden. Es erleichtert das Schreiben von Befehlen und ist ausschließlich auf GUI-Tests spezialisiert, während *Selenium* nicht nur für Tests genutzt werden kann und deshalb weitaus umfangreicher ist als benötigt. [OF16]

### Weitere Werkzeuge für Testmetriken

Für das Messen der Abdeckung der Anforderungen wurde eine *Traceability Matrix* vorgesehen. [Wit19][Lou11] Für die Verbindung zwischen Test- und Anwendungsfällen ist es möglich, in *TestRail* zu jedem Testfall den zugehörigen Anwendungsfall zu verlinken. Jedoch bietet *TestRail* nicht die Möglichkeit, aus diesen Daten eine *Traceability Matrix* zu erstellen. Deshalb soll diese in *Excel* erstellt werden. Hier ist auch, wie in [Unterabschnitt 5.1.11](#) vorgesehen, der Implementierungsfortschritt mit zu vermerken.

Die Durchführungszeit der Tests wird von *JUnit*, *Gradle* und *Jenkins* jeweils gemessen. [Bec+21][BM11] Es liegen demnach genügend Daten vor, aus denen sich auch eine grobe Zeitersparnis berechnen lässt. Der Erfolg eines Testdurchlaufs wird in *TestRail*, aber auch von *JUnit* gemessen und graphisch dargestellt. [Bec+21] Ebenfalls aufgeführt werden in *TestRail* die gefundenen Fehler. Diese werden anschließend in *Jira* vermerkt, hier wird also kein weiteres Werkzeug benötigt.

## 5.3 Geplante Testdurchführung

Die Testdurchführung stützt sich auf die in [Abschnitt 5.1](#) und [Abschnitt 5.2](#) festgelegten Testarten und Vorgehensweisen. [SL19] Sie findet in der ISO-Norm 29119 Platz in den dynamischen Testprozessen. [ISO13a]

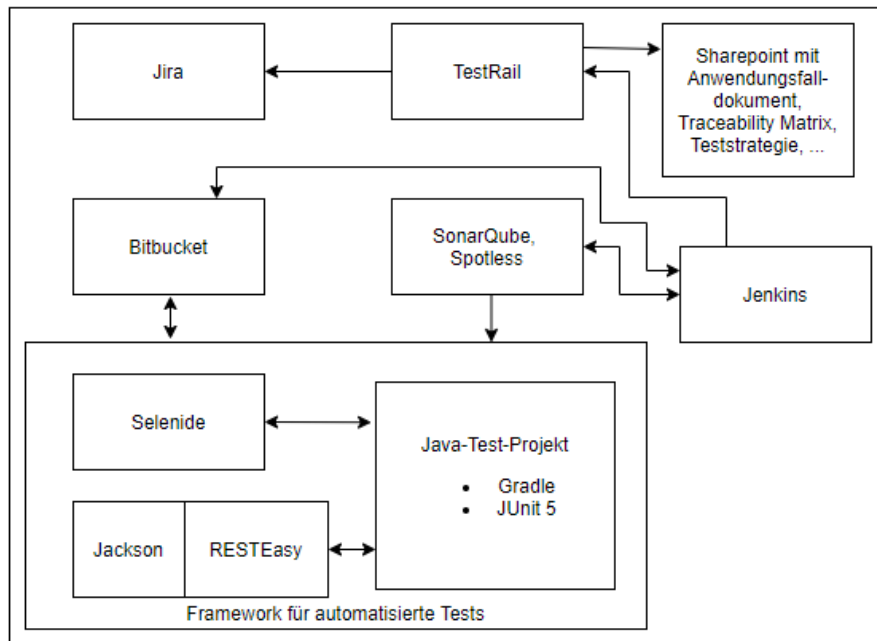


Abbildung 5.7: Das geplante Zusammenspiel der gewählten Testwerkzeuge.

Die Durchführung soll im Kontext von Continuous Integration auf einem *Build-Server* stattfinden. Die Testfälle sollen hier in ihrer jeweiligen Testgruppe nacheinander getestet werden. Gestartet wird der *Build* automatisch, sobald eine Codeänderung ins *CIF* integriert wurde. Wie in [Unterabschnitt 5.1.9](#) bereits erläutert, sollten zunächst statische Tests durchgeführt werden. Erst wenn diese durchlaufen sind, können die dynamischen Tests durchlaufen werden.

Da alle Tests auf der Testinstanz stattfinden, muss zunächst geprüft werden, ob diese verfügbar ist. [\[Wit20\]](#) Ist dies nicht der Fall, kann die *Build-Pipeline* abgebrochen werden, denn ohne Testinstanz wird kein Testfall fehlerfrei durchlaufen. Ist die Testinstanz aber erreichbar, kann im nächsten Schritt die Testumgebung mit den Testdaten wie in [Unterabschnitt 5.1.12](#) beschrieben aufgesetzt werden. Im Anschluss folgen die in [Unterabschnitt 5.2.2](#) festgelegten Testsuiten hintereinander. Begonnen wird bei den grundlegenden Smoke Tests, die von den *API*-Integrationstests, den *API-E2E*-Tests und anschließend den *GUI-E2E*-Tests gefolgt werden. Die Überprüfung der jeweiligen Ergebnisse findet dabei in *JUnit 5* über die *Assertions* statt. Informationen für die Testmetriken werden während des gesamten Durchlaufs kontinuierlich gesammelt.

Nach der letzten Testsuite soll die Testinstanz auf den in [Unterabschnitt 5.1.12](#) definierten Anfangszustand gebracht werden. Zusätzlich sollte der Testbericht in *TestRail* erstellt und die gefundenen Fehler in *Jira* vermerkt werden.

## 5.4 Geplanter Testabschluss

Beim Testabschluss werden die Ergebnisse der Testausführung ausgewertet und ein Abschlussbericht erstellt. [SL19][ISO13a]

Am Ende eines jeden Testdurchlaufs sollte ein Teststatusbericht, oder Testabschlussbericht, erstellt werden. In diesem wird zusammengefasst darüber berichtet, wie sich das Testprojekt entwickelt und wie der Testdurchlauf insgesamt ausgefallen ist. Dies soll dabei helfen, Schwachstellen des Testprojekts oder auch schwerwiegende Fehler in der Software früh zu erkennen und zu beheben. [Wit18] In dem Bericht sind folgende Themen aufzuführen:

- Welche Tests wann ausgeführt wurden.
- Darstellen der durch die Metriken gesammelten und ausgewerteten Daten. Zusätzlich überarbeiten der Metriken hinsichtlich Aktualität und Vollständigkeit. [ISO17]
- Überarbeiten der Risikoanalyse.
- Ausblick auf die nächsten Testdurchläufe und Testaktivitäten.
- Es erfolgt am Schluss eine Gesamtbewertung des Testdurchlaufs, die häufig auf Expertenwissen basiert.

[Wit18]

Im agilen Rahmen ist eine Testberichterstattung nach jedem Testdurchlauf nicht umsetzbar. Änderungen erfolgen häufig in nahen Abständen, unter Umständen mehrmals täglich. Das Erfassen der festgelegten Testmetriken erfolgt, wie bereits beschrieben, teils automatisiert, teils manuell. Auch das Auswerten der erfassten Daten, die Risikoanalyse und der Ausblick benötigen ihre Zeit. Zu große Abstände zwischen Testberichten stehen wiederum einem der Hauptgründe für diese Tests entgegen, nämlich der frühen Reaktionsmöglichkeit bei auftretenden Problemen. Deshalb wird es als sinnvoll erachtet, einen wöchentlichen Teststatusbericht zu verfassen. Empfohlen wird dabei, die Daten an einem Freitag zu erheben, am folgenden Montag aufzuarbeiten und spätestens am Dienstag den Teststatusbericht im Team vorzustellen. So sind die Daten noch nicht allzu veraltet und es kann noch rechtzeitig innerhalb der laufenden Woche auf Probleme reagiert werden, wie Witte in Kapitel 28 darlegt. [Wit18]

## 6 Umsetzung

In diesem Kapitel wird die exemplarische Umsetzung des in [Kapitel 5](#) erstellten Konzepts erläutert. Dabei wird vor allem auf den Einsatz und das Zusammenspiel der ausgewählten Testwerkzeuge und den Aufbau des Testprojekts eingegangen. Die Umsetzung stellt die Testrealisierung und -durchführung im [STLC](#) dar [\[SL19\]](#) und ist Teil der dynamischen Testprozesse in [ISO](#)-Norm 29119. [\[ISO13b\]](#) Vor der Umsetzung wurde das erstellte Konzept teamintern vorgestellt und die Vollständigkeit hinsichtlich der kommenden Implementierung verifiziert. Während der Umsetzung des Konzepts wurden die Teammitglieder regelmäßig über den Fortschritt, die Ergebnisse der Metriken und die Bewertung der Testwerkzeugauswahl informiert. Dies wird bei einer Weiterführung des Projekts beibehalten.

### 6.1 Einrichten der Entwicklungsumgebung

#### 6.1.1 Jira

In *Jira* wird zunächst eine *User Story* erstellt. Diese beinhaltet die einzelnen Schritte der auf dem Konzept basierenden Umsetzung der Tests. Eine *User Story* ist gerade deshalb wichtig, um den Projektfortschritt festzuhalten. Für das Speichern der gefundenen Fehler existiert bereits die *User Story* „CIF Bugs Story“. Diese wird identifiziert und ihre ID kann den Testfällen in *TestRail* hinzugefügt werden.

#### 6.1.2 Bitbucket und SharePoint

Das Projekt sollte möglichst von Beginn an versionsüberwacht sein. [\[ISO13a\]](#)[\[SL19\]](#) Deshalb wird zunächst ein *Repository* für das Projekt in *Bitbucket* und ein Ordner für die Dokumentation in *SharePoint* erstellt. Dabei wird das *Repository* in einem Verzeichnis angelegt, in dem sich alle Testprojekte befinden, die bei *FNT* geschrieben wurden und werden. Der Ordner für die Dokumente wird dort hinterlegt, wo sich allgemeine Informationen über das [CIF](#) befinden. Aus dem Namen des *Repositories* sollte eindeutig hervorgehen, welches System getestet wird. Um die einheitliche Benennung der anderen Testprojekte hier weiterzuführen, wurde das Projekt *command-integration-framework* genannt.

Dem *Repository* kann über das Hinzufügen eines *Webhooks* eine *Build-Pipeline* in *Jenkins* an das Projekt angebunden werden. Ein *Webhook* ist eine Nachricht oder ein Befehl, die bei einem bestimmten Ereignis an einen Endpunkt, beispielsweise einen Server, geschickt wird. [Bis21] Der *Webhook* kann in *Bitbucket* in den Einstellungen des *Repository*s erstellt werden, indem die URL zum *FNT-Jenkins*-Server angegeben wird. Zusätzlich wird festgelegt, durch welche Ereignisse jeweils ein *Build* in *Jenkins* ausgeführt werden soll. [Atl21][Aks+15] Es wird angegeben, dass ein *Build* dann nötig ist, wenn ein *Merge* im *CIF*-Projekt ausgeführt wurde. Zudem wird festgelegt, dass ein *Build* einmal pro Woche automatisiert gestartet werden soll. Als Tag wird hier Montag gewählt, da so im Laufe der Woche auf gefundene Fehler reagiert werden kann. In *Jenkins* wird durch den *Webhook* automatisch ein entsprechendes Projekt angelegt, in dem nach der Ausführung eines *Builds* die *Build-Pipeline* zu sehen ist.

Zusätzlich wurde zur Dokumentation noch eine *README*-Datei erstellt. Diese ist nach einer Studie von Prana das Aushängeschild eines Projekts und sollte so gepflegt werden, dass aus ihr hervorgeht, was in dem zugehörigen Projekt wie umgesetzt wurde, wie es lokal in *Eclipse* in Betrieb genommen werden kann und wie der Status des Projekts ist. [Pra+19] Diese Infos wurden für das Testprojekt hinzugefügt, ebenso wie die theoretischen Hintergründe und Links zu den in [Unterabschnitt 5.1.15](#) aufgezählten zu erstellenden Dokumenten. [Pra+19] So sind alle wichtigen Informationen an dem Ort festgehalten, wo sich auch der Quellcode befindet.

### 6.1.3 Eclipse und Gradle

Als nächstes wird ein neues *Gradle*-Projekt in *Eclipse* angelegt und mit dem erstellten *Bitbucket Repository* verknüpft. Dabei werden eine *build.gradle*-Datei und eine *settings.gradle*-Datei generiert. Die *build.gradle* ist die Datei, in der die benötigten *Dependencies*, *Plugins* und *Tasks* eingebunden beziehungsweise erstellt werden. [BM11] In der *settings.gradle* wird der Name des Projekts festgelegt, [BM11] unter dem es gebaut werden soll. Es heißt hier wie das angelegte *Repository* „*command-integration-framework*“.

Alle in [Abbildung 5.7](#) ausgewählten Testwerkzeuge für statische, *API*- und *GUI*-Tests lassen sich über *Dependencies* einbinden und sind sofort verfügbar. So wurden *Dependencies* von *SonarQube*, *Spotless*, *JUnit 5*, *TestRail*, *RESTEasy* und *Selenide* eingebunden.

Für jede geplante Testsuite wurden anschließend *Tasks* erstellt, die separat ausführbar sind und später von *Jenkins* nacheinander aufgerufen werden sollen. [Aks+15]

### 6.1.4 Implementierung der statischen Überprüfung

Wie bereits erläutert, werden *SonarQube* und *Spotless* über *Dependencies* in das Projekt eingebunden. Die *Dependency* von *SonarQube* ist dabei nicht die des Herstellers, sondern eine *FNT*-interne *Dependency*. Um *SonarQube* nutzen zu können, muss zusätzlich zum Einbinden der *Dependency* ein Login auf die *FNT*-interne Instanz von *SonarQube* stattfinden. Dafür wird dort ein Projekt angelegt oder ein bereits existierendes Projekt ausgewählt, welches entsprechende Regeln zur Überprüfung des Codes liefert. Für das Testprojekt wurde das bereits existierende Projekt der Command-Standardprodukte verwendet. Die dort angelegten Regeln sind auch für das *CIF* gültig und bereits erprobt. Der Login-Vorgang wird in einem *Gradle Task* vorgenommen. Auch für *Spotless* wird ein Start-Befehl in einem *Gradle Task* festgehalten. Die beiden *Tasks* werden später in die *Build-Pipeline* eingebaut und ausgeführt.

### 6.1.5 Strukturierung der Klassen des Projekts

Eine geordnete Struktur ist für die Qualität, die Übersichtlichkeit und die Möglichkeit der Fortführung eines Softwareprojekts essenziell. Deshalb muss die Strukturierung des Projekts geplant, stringent eingeführt und konsequent fortgesetzt werden. [TT21] Die *Java*-Klassen sind durch *Gradle* bereits standardmäßig in zwei Zweigen organisiert. Ein Zweig, *src/main/java*, beinhaltet die Geschäftslogik beziehungsweise alle funktionalen Klassen und Methoden. Der zweite Zweig, *src/test/java*, beinhaltet ausschließlich die Testklassen und -methoden. [GS17][Dav20] Diese klare Struktur wird beibehalten.

Darauf basierend werden alle Testfunktionen in *src/test/java* implementiert. Wie in [Abschnitt 3.1](#) beschrieben wurde, wird vom *FNT* Qualitätssicherungs-CoP vorgesehen, dass diese Testfunktionen simpel gehalten werden. Sie sollen nur Funktionsaufrufe und *Assertions* beinhalten, die das Ergebnis bewerten. Die aufzurufenden Funktionen sind dabei so zu verfassen, dass sie nur jeweils einen Schritt einer Vorgehensweise ausführen, um so mehrfach in unterschiedlichen Situationen verwendbar zu sein. [FNT21a] Eine solche Wiederverwendung spart bei einer kontinuierlichen Entwicklung viel Aufwand und dadurch Zeit und Kosten. Bei Abänderungen im *CIF* müssen unter Umständen nur einzelne Schritte angepasst werden, statt ganze Testfälle. [SVP18] Aus diesem Grund werden die Klassen, die diese Funktionen beinhalten auch *Step*-Klassen genannt. [FNT21a] Die *Step*-Klassen werden, obwohl sie direkt von den Testklassen aufgerufen werden, in *src/main/java* gespeichert. Das fördert die Übersicht und lässt im Testpfad ausschließlich von *JUnit* ausführbare Funktionen zu.



Zu den beiden Zweigen werden im Laufe der Implementierung weitere Klassen hinzugefügt, die in *Packages* sortiert werden. Die *Packages* sind, wie in der *FNT* Testpolitik vermerkt, mit dem Präfix *com.fntsoftware* versehen. Darauf folgt eine weitere Unterteilung der Klassen in *API*- oder *GUI*-Testklassen. So ist das Projekt grundlegend strukturiert. Eine Übersicht darüber ist in *Abbildung 6.1* abgebildet.

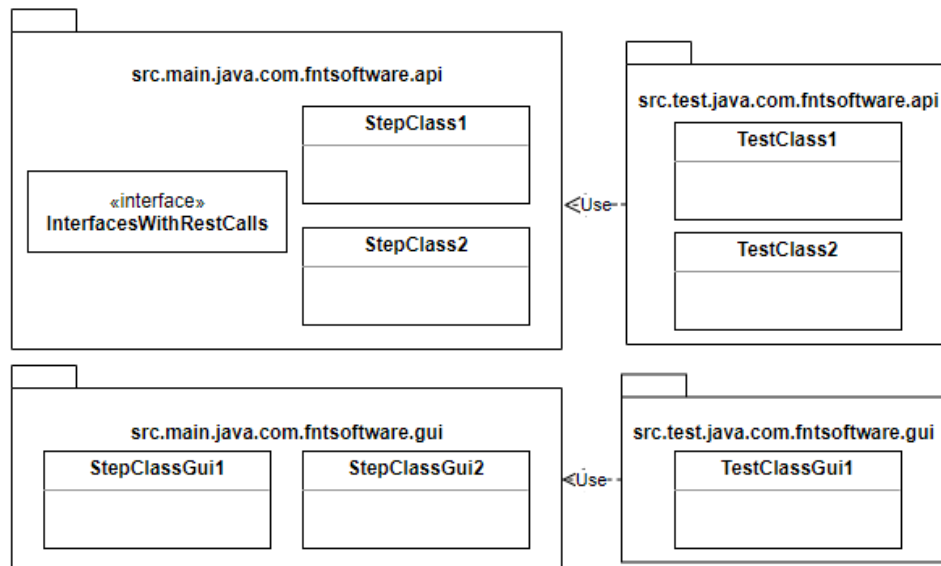


Abbildung 6.1: Die erstellte Projektstruktur in einer Übersicht.

### 6.1.6 Implementierung des Testtreibers

Als Testtreiber ist *JUnit 5* eines der grundlegenden Werkzeuge des Projekts. Es wird in *src/test/java* für die Ausführung der Testfunktionen eingesetzt. Eine Funktion wird dabei mit der Annotation *@Test* als Testfunktion ausgewiesen. [Bec+21] Diese Funktion ist dann typischerweise so aufgebaut, dass eine oder mehrere Funktionen der *Step*-Klassen ausgeführt werden und das Ergebnis der letzten Methode mit einem Soll-Wert verglichen wird. Das geschieht in *JUnit* mit *Assertions*. [Bec+21] Ein simpler Testfall sieht beispielsweise aus wie in *Listing 6.1*. Hier ist ebenfalls dargestellt, dass alle Testfälle mit der ID anfangen, die sie in *TestRail* zugewiesen bekommen haben. Das ist wichtig, um den Testfällen in *TestRail* die Ergebnisse aus Testdurchläufen richtig zuordnen zu können.

```

1 class SimpleTestClass {
2     private final StepKlasse step = new StepKlasse();
3
4     @Test
5     void C23907876_subtractionOfTwoNumbers() {
6         // Aufruf von einzelnen Schritten
7         int one = step.getNumber1();

```



```

8         int result = step.subtract(3, one);
9         // Assertion
10        assertEquals(2, result);
11    }
12}

```

Listing 6.1: Der grundlegende Aufbau einer Testfunktion in *JUnit*.

Um die Tests auch mit *Gradle* ausführen zu können, wird den Testfällen eine weitere Annotation vorangestellt. Mit `@Tag("suite")` lassen sich die Testfälle im Code den Testsuiten zuordnen, denen sie im Konzept zugeordnet wurden. Je nach Suite wird dem *Tag* ein anderer Parameter übergeben. So sind die Testfälle gebündelt je nach Testsuite ausführbar. [Bec+21] In *Gradle* kann ein Aufruf eines Testfalls anhand seines *Tags* in einem *Task* erfolgen. [BM11] So ist mit einem *Task* eine Testsuite abgedeckt. In diesem *Task* ist angegeben, welche Tests mit welchen *Tags* aufgerufen werden sollen und auch, aus welchen *Packages*. So lassen sich beispielsweise *API*- und *GUI*-Tests getrennt als Smoke Tests ausführen, ohne dass die Bezeichnung `@Tag("suite:smoke")` für die jeweilige Testart geändert werden muss. [GS17] Der Aufbau einer solchen *Task* ist in Listing 6.2 dargestellt. [BM11]

```

1 task apiSmoke(type: Test) {
2     description = 'ApiSmokeTests mit Annotation @Tag(suite:smoke)'
3
4     // Angeben, dass JUnit fuer die Ausfuehrung verwendet wird
5     useJUnitPlatform() {
6         includeTags 'suite:smoke'
7     }
8     // Nur API-Testfaelle werden in diesem Task beruecksichtigt
9     filter {
10         includeTestsMatching "*com.fntsoftware.api.*"
11     }
12 }

```

Listing 6.2: Ein *Gradle Task*, der nur *API* Smoke Tests durchführen soll.

*JUnit* erstellt nach Beendigung eines Testdurchlaufs einen lokalen Hypertext Markup Language (*HTML*)-Testbericht, der die Ergebnisse graphisch darstellt. [GS17][BM11] Ein Beispiel ist im Anhang in Abschnitt E dargestellt.

### 6.1.7 Integration der Testläufe in TestRail und Jira

*TestRail* bietet eine *API*, an die die Testergebnisse von *JUnit* gesendet werden können. [GuroJb] Damit *TestRail* diese auswerten kann, muss jeder Testfall mit den Ergebnissen des Testlaufs erfasst und in einen *JSON*-String gemappt werden. Für diesen Schritt stellt

*FNT* ein firmeninternes Projekt bereit, das über eine *Dependency* in das Testprojekt einbindbar ist. Dieses Projekt erfasst auf Basis der erstellten *JUnit*-Testprotokolle, welche Testfälle in einem Testlauf ausgeführt wurden, wie ihr Status ist und den Inhalt der eventuell auftretenden Fehlermeldung. Die Namen der Testmethoden beginnen, wie bereits beschrieben, mit der jeweiligen Testfall-ID des zugehörigen Testfalls. Diese wird erfasst und dem *JSON*-String beigelegt. So kann *TestRail* die eingehenden Daten direkt auf die bestehenden Testfälle mappen. Daraus entsteht dann ein Testbericht in *TestRail* mit einer Übersicht über die durchgeführten Testfälle und die Erfolgsrate. Der beschriebene Vorgang wird in einer *Gradle Task* ausgeführt. [GuroJb]

Sollte bei einem der Testfälle ein Fehler auftreten, so kann dies automatisiert in *Jira* in der in *TestRail* verlinkten *User Story* vermerkt werden. In *TestRail* selbst wird dafür das *Jira-Plugin* aktiviert. Diesem muss zunächst ein Link zum *Jira*-Server übergeben werden. *Jira* auf der anderen Seite muss eine Kommunikation mit *TestRail* zulassen. Das kann in den Einstellungen des *Jira*-Servers vorgenommen werden, [GuroJa] wurde aber für dieses Projekt aus zeitlichen Gründen noch nicht umgesetzt.

### 6.1.8 Ausführung in Jenkins

Damit die Tests in *Jenkins* durchlaufen werden können, muss zunächst eine *Build-Pipeline* in einem *Jenkinsfile* erstellt werden. Das ist eine Datei, die alle durchzuführenden Schritte und benötigten Daten beinhaltet. Sie wird vom *Jenkins*-Server eingelesen und entsprechend der dort beschriebenen Schritte werden die Aufgaben ausgeführt. [Pat17][Las18]

Im *Jenkinsfile* selbst werden die einzelnen Schritte als *Stages* bezeichnet. [Pat17] Nach dem Festlegen einiger für den *Build* wichtiger Parameter wie der Name des *Builds* oder die dem *Build* zuzuweisenden *CPU*- und Speicherkapazitäten wird die *Pipeline* erstellt. Einer *Stage* wird ein in *Jenkins* später angezeigter Name gegeben und anschließend wird die entsprechende *Gradle-Task* aufgerufen. [Aks+15][Las18] Standardmäßig wird der *Build* immer abgebrochen, wenn bei der Ausführung einer *Stage* ein Fehler aufgetreten ist. [Las18] Da dies, wie festgelegt, nicht bei allen Stufen erwünscht ist, wird bei diesen ein *Try-Catch-Block* hinzugefügt, sodass der *Build* weiterlaufen kann. Eine *Stage* sieht dabei aus wie in Listing 6.3. [Pat17]

```
1  try {
2      // Der Name der Stage wird direkt als Parameter angegeben
3      stage('Execute_API_Tests_-_Suite_Environment_Setup') {
4          callGradle 'apiSetUpEnvironment', gradleProperties+[]
5      }
6  } catch (Exception e) {
```

```
7     publishJUnitTestReport()
8 }
```

Listing 6.3: Der Aufbau einer *Stage*.

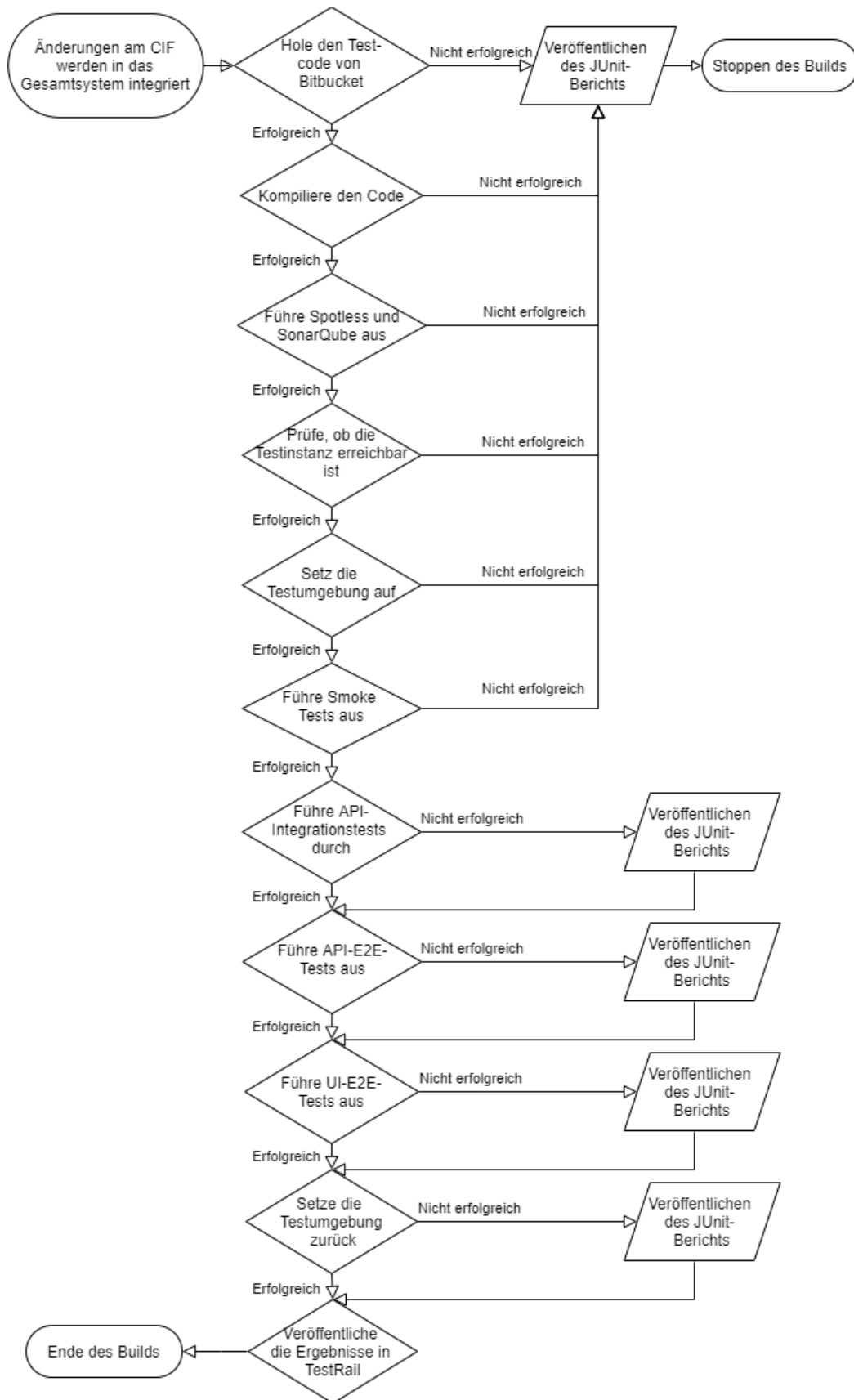
Orientiert an der in [Unterabschnitt 5.2.2](#) und [Abschnitt 5.3](#) festgelegten Reihenfolge der Testsuiten, der statischen Testwerkzeuge und des Aufsetzens und Zurücksetzens der Testumgebung und Testdaten ergibt sich eine geplante Ausführungsreihenfolge wie in [Abbildung 6.2](#). Hier wird auch dargestellt, bei welchen Schritten Fehler gefunden werden dürfen, ohne dass der *Build* abgebrochen wird und bei welchen dies nicht der Fall ist. [\[Aks+15\]](#) Bei einem Abbruch und am Ende der Pipeline werden die von *JUnit* verfassten Testberichte dem Projekt in *Jenkins* angehängt. Diese sind dort abrufbar. Zusätzlich werden die Testergebnisse an *TestRail* gesendet.

## 6.2 Implementierung von **API**-Tests

Obwohl **API**-Tests in jeder Teststufe vorkommen, können die in [Abschnitt 5.2.5](#) geschilderten Schritte für das Erstellen von **API**-Tests für alle Teststufen angewandt werden. Das liegt daran, dass sie grundlegend aus mindestens einer **REST**-Abfrage bestehen, unabhängig vom jeweiligen Testobjekt. Dementsprechend kann eine grundsätzlich gleiche Vorgehensweise gewählt werden.

Der erste Schritt, den Belorusetz beschreibt, ist die Analyse der Details der jeweiligen **REST**-Abfrage. [\[Bel15\]](#) Wie in [Abschnitt 2.1](#) beschrieben, werden **REST**-Anfragen an eine bestimmte **URL** gesendet. Zunächst muss deshalb für jeden Fall herausgefunden werden, an welche **URL** die zugehörige **REST**-Anfrage geschickt werden muss. Diese besteht immer aus der Basis-**URL** `http://sapp-pse-02:1206` der Testinstanz und angehängten Pfadangaben, die sich je nach Entität oder Aktion unterscheiden. Die Pfadangaben werden auch als Endpunkte bezeichnet. [\[Bal17\]](#) Da die Basis-**URL** immer dieselbe bleibt, wird sie in einer Datei im Projekt vermerkt und zu Beginn dort ausgelesen.

Für die **REST**-Anfragen müssen nun die jeweiligen Endpunkte herausgefunden werden. [\[Bel15\]](#) Hier hilft der **REST**-Client der **BGE**. Dieser ist in [Abbildung 6.3](#) abgebildet. Hier ist beispielsweise die Entität `toolFrameworkJob` geöffnet, mithilfe derer sich Jobs unter anderem erstellen oder bearbeiten lassen. Beispielsweise sollen nun bestimmte Jobs abgefragt werden. Dafür wird, wie in [Abbildung 6.3](#), die Schaltfläche `query` angeklickt, woraufhin sich rechts ein Bereich öffnet, in dem Attribute eingetragen werden können, nach denen die Abfrage gefiltert wird.

Abbildung 6.2: Die geplanten Stufen in der *Build-Pipeline*.

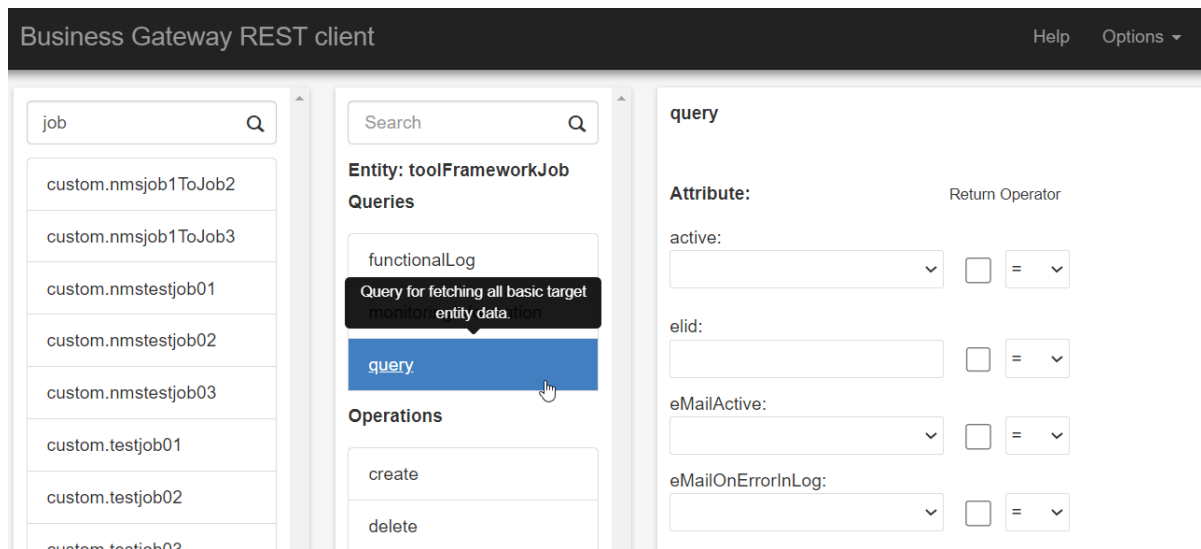


Abbildung 6.3: Die Oberfläche der BGE

Es wird beispielsweise das Feld *active* mit *true* befüllt und die Abfrage anschließend abgesendet. Nun öffnet sich ein weiterer Bereich, in dem die Details der gerade abgesendeten REST-Abfrage eingesehen werden können. Wie in [Abbildung 6.4](#) ersichtlich, wird hier zum einen der entsprechende Endpunkt angezeigt, der an die Basis-URL angehängt wird, und zum anderen der JSON-String, der der POST-Anfrage als *Body* angehängt wird. In diesem Fall befinden sich in dem String die eingegebenen Suchbeschränkungen.

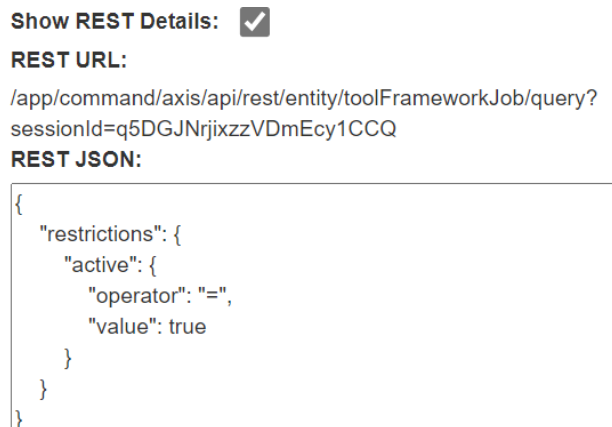


Abbildung 6.4: Die Details zu der gesendeten REST-Anfrage.

Neben den Details zur Abfrage wird auch die Antwort abgebildet. Diese besteht immer aus dem Objekt *status* und dem Array *returnData*. Das ist auch in [Abbildung 6.5](#) ersichtlich. Damit sind alle Details vorhanden, die für eine REST-Anfrage benötigt werden.

Die Daten der Antwort sollen im nächsten Schritt auf eine **POJO**-Klasse gemappt werden. [Bel15] Ein JSON-String besteht dabei im Wesentlichen aus Schlüssel-Werte-Paaren und Objekten, die diese beinhalten. [Pez+16] Grundsätzlich wird so verfahren, dass jedes Objekt im JSON-String, das diese Paare oder andere Objekte beinhaltet, als eine Klasse

**Result Data:**

```

{
  "status": {
    "errorCode": 0,
    "subErrorCode": null,
    "message": null,
    "success": true
  },
  "returnData": [
    {
      "jobAction": "IMPORT",
      "elid": "NO1AGBU3Y1KZ3K",
      "eMailOnErrorInLog": false,
      "active": true,
      "technicalLogPath": null,
      "eMailActive": false,
      "additionalSettings": [
        {
          "defaultValue": null,
          "required": "N",
          "fixed": "N",
          "value": "N",
          "encrypted": "N",
          "name": "VALIDATE_CIF_CONFIGURATION"
        }
      ]
    }
  ]
}

```

Abbildung 6.5: Die auf die Anfrage zurückgesendeten Daten.

erstellt wird. [Bal17] Bezogen auf die Antwort in [Abbildung 6.5](#) ergibt sich zunächst die Klasse *QueryResponse.java*, die ein Objekt der Klasse *Status.java* und die generische Liste *returnData* enthält. Die Liste ist generisch, da so für die beinhalteten Objekte jeweils eine Klasse erstellt und der Liste hinzugefügt werden kann, ohne dass für jedes neue Rückgabeobjekt eine neue *returnData*-Liste erstellt werden muss. In diesem Fall wird die Klasse *JobProperties.java* erstellt, die neben einigen Attributen ein Objekt der ebenfalls erstellten Klasse *AdditionalSettings.java* enthält. [Pez+16] Die Klassen im Zusammenhang sind in [Abbildung 6.6](#) zu sehen.

Durch Hinzufügen von *@JsonProperty* vor jedes Attribut einer Klasse kann dieses eindeutig auf den JSON-String gemappt werden. *Jackson* unterstützt zwar auch ein Mapping ohne *@JsonProperty*, in diesem Fall müssen aber die Attribute genau so heißen wie die Felder im JSON-String und das ist fehleranfällig bei Änderungen im Code. [Bal17] *Jackson* fordert zusätzlich für jedes Attribut *Getter* und *Setter*. Diese werden ebenfalls erstellt. So sieht eine POJO-Klasse nach erfolgreichem Erstellen beispielsweise aus wie in [Listing 6.4](#). [Bal17]

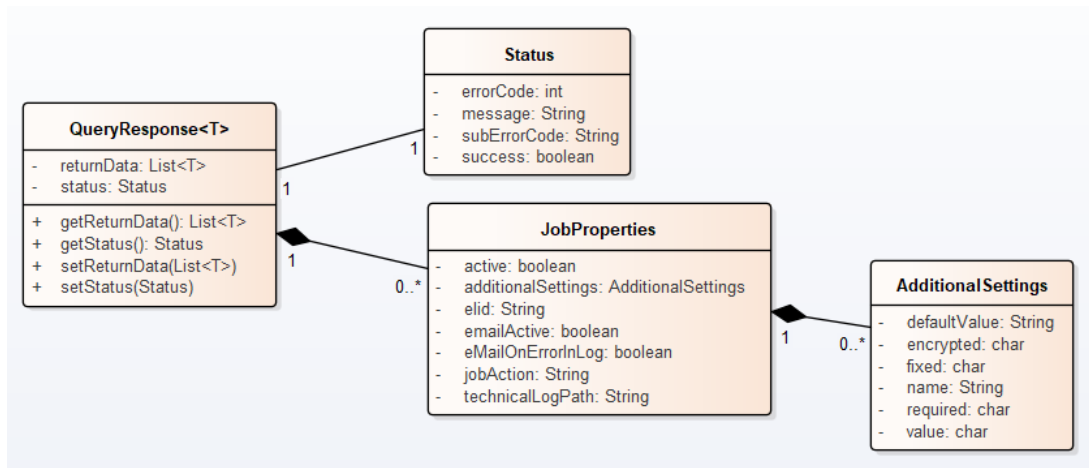


Abbildung 6.6: Die erstellten Klassen.

```

1 public class QueryResponse<T> {
2
3     @JsonProperty("status")
4     private Status status;
5
6     @JsonProperty("returnData")
7     private List<T> returnData;
8
9     public Status getStatus() {
10         return this.status;
11     }
12     public void setStatus(Status status) {
13         this.status = status;
14     }
15
16     // Getter und Setter von returnData werden der Einfachheit halber
17     // weggelassen
18 }

```

Listing 6.4: Der Aufbau der Klasse *QueryResponse.java*.

Dieselbe Vorgehensweise sollte auch für die der **REST**-Anfrage angehängten **JSON**-String gelten. So kann der **REST**-Anfrage ein Objekt angehängt werden. Das ist leichter zu erstellen und weniger fehleranfällig als das Erstellen eines **JSON**-Strings.

Als nächsten Schritt definiert Beloruset das Erstellen der **REST**-Abfrage selbst. [Bel15] Dafür gibt *RESTEasy* bereits einen Rahmen vor. Es bietet nämlich das *RESTEasy Proxy Framework*, mit dessen Hilfe eine **HTTP**-Anfrage zum Aufrufen eines entfernten Web Services möglich gemacht wird. [JB013] Dafür wird zunächst ein Interface erstellt, in dem **REST**-Anfragen als Methoden definiert werden. Hier wird mit **JAX-RS**-Annotationen

festgelegt, welche [HTTP](#)-Methode für eine Anfrage verwendet werden soll, an welchen Endpunkt die Anfrage geschickt wird und von welchem Typ der Rückgabewert sein soll. [Bal17][JBo13] Über Parameter können die Endpunkte variabel gestaltet werden und auch festgelegt werden, was der [REST](#)-Anfrage mitgegeben werden soll. So lässt sich mit `@PathParam` ein im *Path* mit geschweiften Klammern markierter Teil durch einen Parameterwert ersetzen oder mit `@QueryParam` kann an den *Path* ein *Query* Parameter angehängt werden. [bae20][JBo13] Ein Beispiel ist in [Listing 6.5](#) zu sehen.

```

1 @Produces(MediaType.APPLICATION_JSON)
2 @Consumes(MediaType.APPLICATION_JSON)
3 public interface ICommandPreconditionRestOperations {
4 // Simple POST-Anfrage an Basis-URL mit angehaengtem Path
5     @POST
6     @Path("/axis/api/rest/businessGateway/login/")
7     Response loginCommandViaBge(final BgeLoginRequestModel bgeLoginModel);
8
9 // Simple GET-Anfrage an Basis-URL ohne angehaengtem Path
10    @GET
11    Response validateBgeIsRunning();
12
13 // POST-Anfrage, bei der die protocolElid durch den mitgegebenen
14 // Parameterwert ersetzt wird und ein Query-Parameter mit der
15 // mitgegebenen Sitzungs-ID den Pfad erweitert
16    @POST
17    @Path("/axis/api/rest/entity/planningProtocol/{protocolElid}/activate"
18         )
19    Response activatePlanningProtocol(@PathParam("protocolElid") final
20                                     String protocolElid,
21                                     @QueryParam("sessionId") final String sessionId, final String
22                                     restrictions);
23 }

```

Listing 6.5: Ein Interface mit möglichen [REST](#)-Anfragen.

Durch das Interface sind die [REST](#)-Anfragen vom restlichen Code getrennt. Das Senden der [REST](#)-Anfragen findet in den *Step*-Klassen statt, da meist eine [REST](#)-Anfrage als ein Schritt betrachtet werden kann. Mithilfe des *Proxy-Frameworks* werden die Anfragen aus dem Interface geholt, an die Basis-URL angehängt und anschließend abgesendet. [JBo13] Diese Vorgehensweise ist in [Listing 6.6](#) dargestellt. Dabei werden die ersten beiden Zeilen in jeder *Step*-Klasse nur einmal ausgeführt und die im Interface erstellten Funktionen bei Bedarf über die Instanz der Interfaces aufgerufen. [JBo13]

```

1 ResteasyWebTarget target = this.base.getRestClient().target(getBaseUrl()
2     );

```



```

2 ICommandPreconditionRestOperations proxy = target.proxy(
    ICommandPreconditionRestOperations.class);
3 // Aufrufen einer der oben aufgeführten REST-Abfragen mit
    Parameteruebergabe
4 proxy.activatePlanningProtocol(planningProtocolElid, sessionId, "{}");

```

Listing 6.6: Das Senden der REST-Anfrage.

Die Antwort auf die gesendete REST-Anfrage wird anschließend mithilfe der Klasse *ObjectMapper* von *Jackson* auf die entsprechende *Java*-Klasse gemappt. So entsteht ein Objekt dieser Klasse mit den im JSON-String gelieferten Werten. [Bal17]

Die für dieses Projekt exemplarisch implementierten Testfälle sind hauptsächlich Smoke-Testfälle, die in Unterabschnitt 5.2.1 als Vorbedingungen festgelegt wurden und die Smoke-Testfälle, die die grundlegenden Funktionen des Jobs, der Delta-Berechnung und der Synchronisation testen. Als Integrationstests wurden ebenfalls einige den Job betreffende Testfälle umgesetzt. Wie in Abbildung 5.3 beschrieben, sollte hier eine *Top-Down-Integration* stattfinden, weshalb mit den Integrationstests für den Job angefangen wurde. Als nächstes sollten dementsprechend die Integrationstests der Delta-Berechnung und danach die der Synchronisation umgesetzt werden. Des Weiteren sind für die Geräte-Entität einige E2E-Testfälle implementiert worden. Es sind also alle vorgesehenen Teststufen für API-Tests exemplarisch implementiert. So muss bei einer Weiterführung der Implementierung nur noch das bereits Umgesetzte erweitert und kein zusätzliches Konzept ausgearbeitet werden.

## 6.3 Implementierung von GUI-Tests

Für die Implementierung der GUI-Tests wurde sich für *Selenide* entschieden. Auch bei diesem Werkzeug gibt es eine vorgegebene Struktur, nach denen Testfälle geschrieben werden sollen. So sieht die Dokumentation von *Selenide* drei grundlegende Schritte vor:

1. Öffne die Seite, auf der die Tests durchgeführt werden sollen.
2. Führe die Schritte aus, die im Testfall beschrieben werden.
3. Vergleiche den Ist-Zustand des letzten Schritts mit dem Soll-Zustand.

[codoJ]

Der erste Schritt ist dabei mit dem Befehl *open(URL)* direkt umsetzbar. Da für jeden Testfall eine neue Instanz des Browsers geöffnet wird, muss dies vor jedem Testfall erneut durchgeführt werden. [codoJ] Dafür gibt es in JUnit die Annotation *@BeforeEach*. Eine

Methode, der diese Annotation vorgesetzt wird, wird vor jedem einzelnen Testfall ausgeführt. Da bei einem Login jeweils eine neue Sitzung in Command geöffnet wird, sollte diese auch wieder nach jedem Testfall geschlossen werden, indem ein Logout stattfindet. Dafür wird eine zusätzliche Methode mit der Annotation `@AfterEach` erstellt, die entsprechend wie `@BeforeEach` arbeitet, nur, wie der Name sagt, nach jedem Testfall. [Bec+21]

Im Zuge des zweiten Schritts wird schrittweise der Login vollzogen. Dafür muss zunächst die Oberfläche untersucht und herausgefunden werden, wie beispielsweise das Feld für den Benutzernamen identifizierbar ist. Das ist durch die Entwickleransicht des Browsers schnell umzusetzen, die in [Abbildung 6.7](#) dargestellt ist. *Selenide* unterstützt die Identifizierung von Bestandteilen einer Website über deren ID, den Namen, den Cascading Style Sheets (CSS)-Selektor oder *XPath*-Angaben. [codoJ]

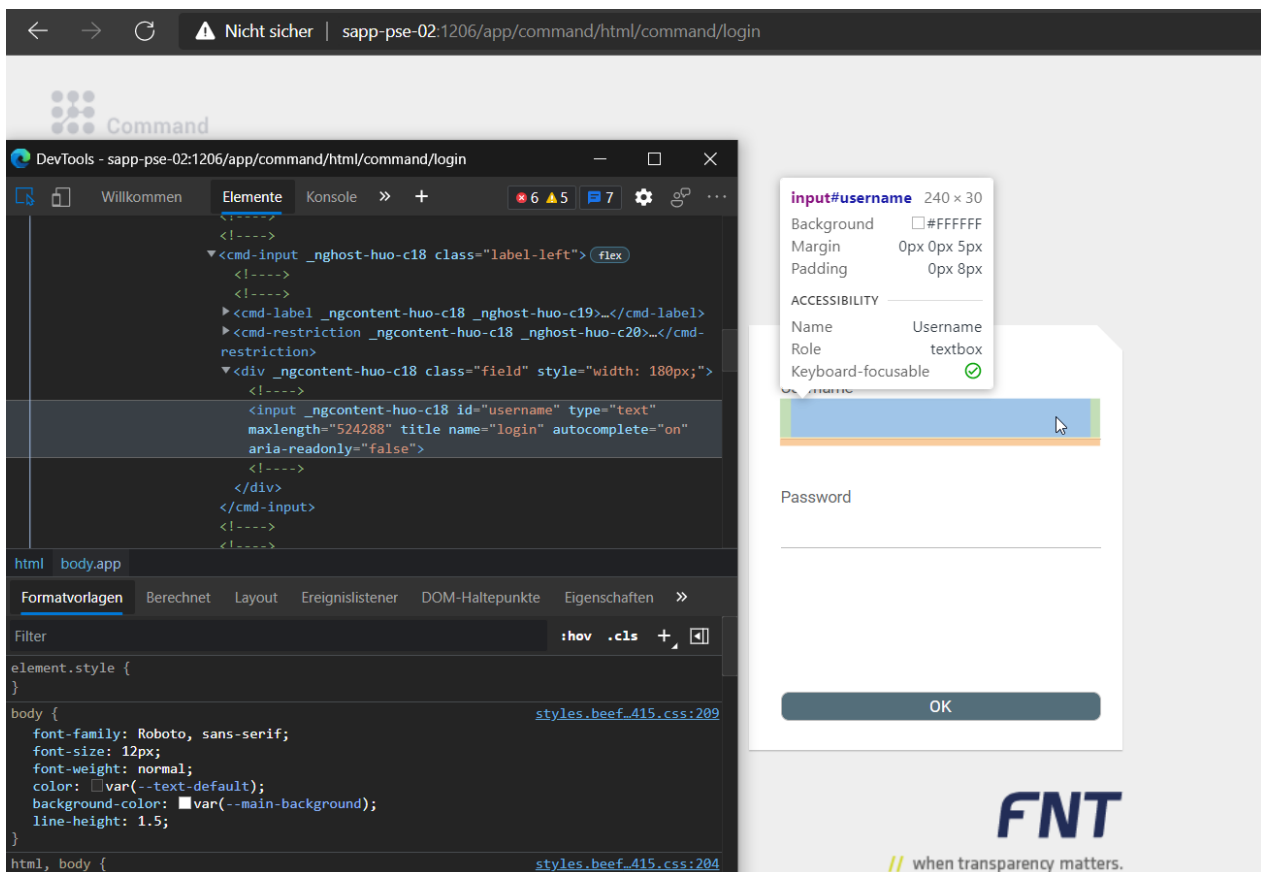


Abbildung 6.7: Das Untersuchen der Login-Seite.

Der schrittweise Login sieht im Code entsprechend aus wie in [Listing 6.7](#) ersichtlich.

Für die anderen GUI-Testfälle werden wieder wie in [Unterabschnitt 6.1.6](#) beschrieben, Methoden mit `@Test` und `@Tag` markiert. Der Login hat bereits stattgefunden, was bedeutet, dass als nächstes von der Startseite aus durch Command bis hin zur eigentlichen Testausführung navigiert werden muss. Von dort aus wird der Testfall Schritt für Schritt wie in *TestRail* beschrieben ausgeführt.

```
1 public void login() {  
2     open(baseUrl + "html/command/login");  
3     $(By.name("login")).setValue(loginData.getUser());  
4     $(By.name("password")).setValue(loginData.getPassword());  
5     $("#ok").click();  
6 }
```

Listing 6.7: Der Login über *Selenide*.

*Selenide* bietet, wie der dritte Schritt in der Dokumentation andeutet, eine eigene Vergleichsmöglichkeit des Ergebnisses mit einem Soll-Wert. Diese ist den *Assertions* von *JUnit* sehr ähnlich. [Bec+21][codoJ] Da *JUnit* für jede Testfunktion eine *Assertion* erwartet [Bec+21] und das Ergebnis von *Selenide* nicht in seine Auswertung übernehmen kann, werden hier weiterhin die *Assertions* von *JUnit* verwendet.

Umgesetzt wurden auch hier einige GUI-Testfälle exemplarisch für die vorgesehen Teststufen. Es entstanden zwei Smoke-Testfälle und ein E2E-Testfall. Dies erscheint sehr wenig, ist aber dem Umstand geschuldet, dass *Selenide* zwar einfache Befehle zur Umsetzung von GUI-Tests liefert, die Implementierung aber dennoch zeitlich deutlich länger benötigt als die Umsetzung der API-Tests. Diese Tatsache deckt sich mit dem in Unterabschnitt 5.1.7 bereits geschilderten höheren Aufwand und den daraus resultierenden erhöhten Kosten bei der Umsetzung von GUI-Tests. [Coh10]

## 6.4 Ergebnisdokumentation

Die Dokumentation erfolgt wie in Unterabschnitt 5.1.15 festgelegt. Testergebnisse und Testfälle beziehungsweise Testsuiten werden in *TestRail* und *Jenkins* festgehalten. Die Ergebnisse des Konzepts sind in *SharePoint* als Teststrategie, in einem Anwendungsfall-dokument und einem Testausführungsdokument hinterlegt, der Code, die Auswahl an Testwerkzeugen und wie das Testprojekt zu strukturieren ist, in *Bitbucket*. Die Testdaten sind im Projekt als Ressourcen angelegt.

Zusätzlich entsteht der Testabschlussbericht mit den Auswertungen der Testmetriken, der regelmäßig aktualisiert werden muss. Die Testmetriken werden dabei durch *JUnit*, *Jenkins* und *TestRail* gesammelt. Weitere Informationen können aus der erstellten *Traceability Matrix* entnommen werden, die auch im Anhang in Abschnitt D abgebildet ist.

## 7 Evaluierung

In diesem Kapitel soll der Erfolg des entwickelten Konzepts anhand der darauf basierten implementierten Testfälle bewertet werden. Hierfür können unter anderem die in [Unterabschnitt 5.1.11](#) entwickelten Testmetriken herangezogen werden, deren Aufgabe eine objektive und zahlenbasierte Evaluierung des Testprojekts vorsieht. Für die Testmetriken wurde ein Testlauf und der Stand der Testfälle und der Implementierung vom 14.09.2021 zur Auswertung herangezogen.

Die **Abdeckung der Anforderungen** ist durch eine *Traceability Matrix* im *xls*-Format festgehalten. Aus dieser geht hervor, dass die 16 Anwendungsfälle mit insgesamt 354 Testfällen abgedeckt sind. Es gibt keine Anforderung, zu der nicht mindestens ein Testfall besteht.

Ebenfalls in der *Traceability Matrix* festgehalten ist der **Implementierungsfortschritt**. Aus dem Diagramm in [Abbildung 7.1](#) ergibt sich eine Umsetzungsquote von 50% für Smoke Tests, 1,3% für Integrationstests, 8,2% für [E2E-API-Tests](#) und 11,5% für die [GUI-Tests](#). Diese Zahlen sind noch nicht zufriedenstellend. Zwar ist die Quote bei den grundlegenden Smoke-Testfällen wegen ihrer hohen Relevanz absichtlich sehr viel höher als bei den restlichen Testsuiten, doch gerade bei den Integrationstests ist ein Ausbau als nächstes nötig.

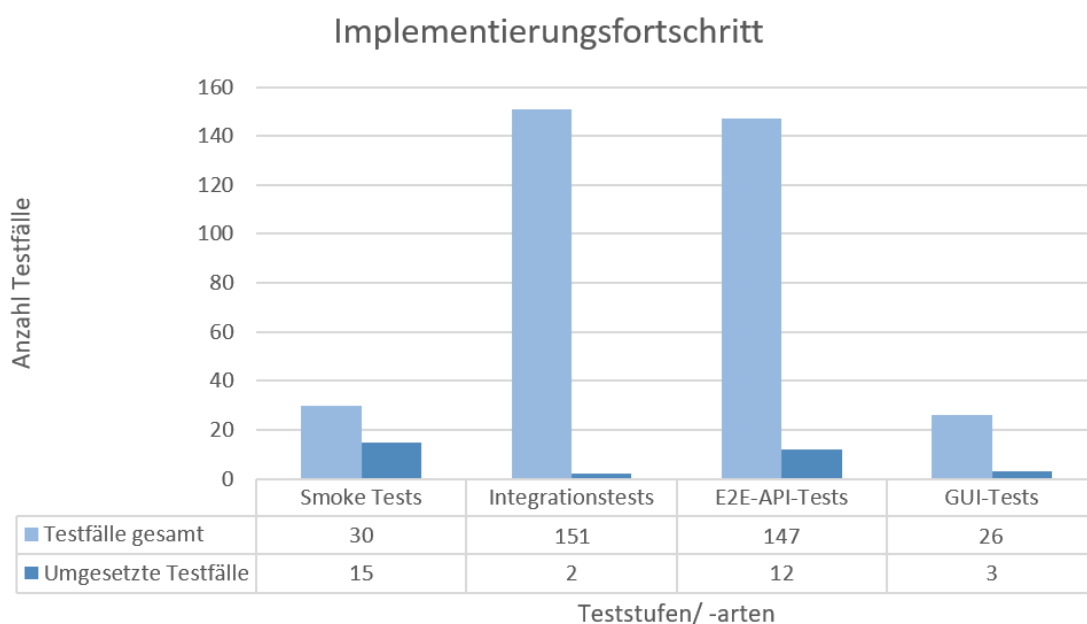


Abbildung 7.1: Der Implementierungsfortschritt graphisch dargestellt.

Die Anzahl an Testfällen konnte von 211 manuellen Testfällen auf 354 automatisierte Testfälle übernommen und erweitert werden. Die hinzugekommenen Testfälle sind dabei vor allem die, die jede Variante der möglichen Delta-Fälle testen. Dies war manuell bisher zu aufwendig. Da einige Anwendungsfälle jedoch für die automatisierten Tests ausgeschlossen wurden, befinden sich im manuellen Testprojekt noch 37 Testfälle, die weiterhin manuell ausgeführt werden müssen. Die Gesamtzahl an Testfällen für das [CIF](#) beträgt somit 391. Damit ist ein geplanter maximaler **Automatisierungsgrad** von 90,5% möglich. Mit dem momentanen Implementierungsfortschritt liegt der Automatisierungsgrad bei 8,2%. Das sollte zeitnah erweitert werden. Die Ausführung der Testfälle erfolgt wie festgelegt automatisiert einmal wöchentlich und immer dann, wenn Änderungen am [CIF](#) vorgenommen werden. Der *Nightly-Build* mit Smoke Tests konnte aus zeitlichen Gründen noch nicht umgesetzt werden.

Die **Durchführungszeit** der Teststufen wurde von *Jenkins* aufgezeichnet. Insgesamt wurden in diesem Testlauf für die festgelegten Teststufen 126 Sekunden benötigt. Die Dauer der Durchführung der einzelnen Stufen sind in [Abbildung 7.2](#) dargestellt.

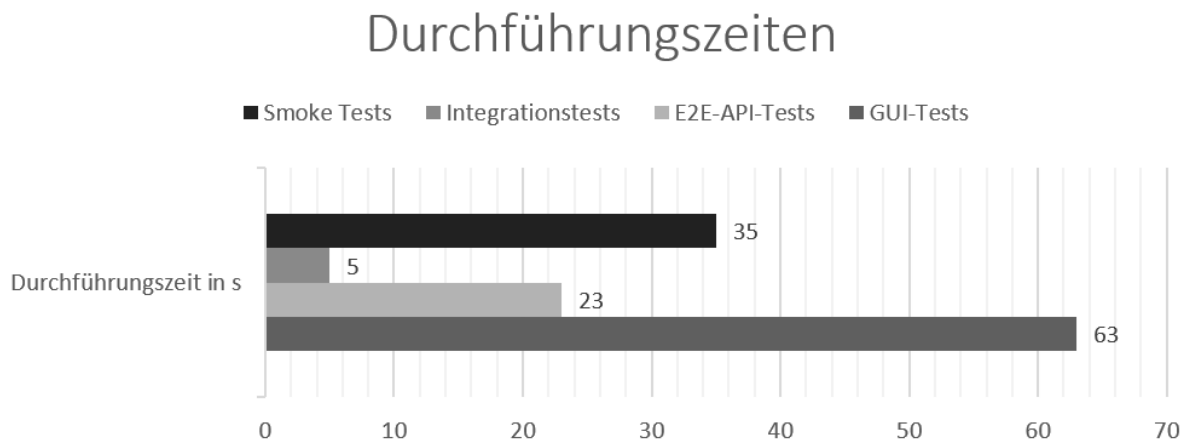


Abbildung 7.2: Die Durchführungszeiten graphisch dargestellt.

Die Berechnung der Dauer der automatisierten Tests nach einer Umsetzung aller Testfälle ist mit diesen Zahlen möglich: Da 15 Smoke-Testfälle umgesetzt wurden, ergibt sich für die Durchführung eines einzelnen Smoke-Testfalls eine durchschnittliche Durchführungszeit von 2,33 Sekunden. Eine Durchführung aller 30 Smoke-Testfälle würde hochgerechnet ungefähr 70 Sekunden dauern. Mit einer Einzeldurchführungszeit von 2,5 Sekunden ergibt sich für alle Integrationstests schätzungsweise eine Durchführungszeit von 377,5 Sekunden. Ein einzelner [E2E-API](#)-Testfall benötigt rund 1,92 Sekunden, das Durchführen aller 147 Testfälle würde also grob 282 Sekunden dauern und mit 21 Sekunden für einen [GUI](#)-Testfall würden rund 546 Sekunden für eine Durchführung aller Fälle benötigt.

Insgesamt würde ein Durchlauf aller 354 Testfälle also um die 1276 Sekunden benötigt, was ungefähr 22 Minuten entspricht. Das ist aber eine grobe Hochrechnung und in der Realität ist mit einer durchaus längeren Testdurchführungszeit zu rechnen. Auch für die statischen Tests, die vor den dynamischen Tests durchgeführt werden, sind die Durchführungszeiten aufzuführen. Das Kompilieren des Codes dauert momentan durchschnittlich 50 Sekunden, die Überprüfung des Formats mit *Spotless* 25 Sekunden und die Überprüfung des Codes mit *SonarQube* 72 Sekunden. Insgesamt ergibt sich hier momentan eine Gesamttestzeit von 147 Sekunden. Mit Zunahme der **LOC** werden diese Tests ebenfalls in ihrer Dauer zunehmen.

Für das Berechnen der **Zeitersparnis** durch automatisierte Tests wurde ein *Test Engineer* befragt, der das **CIF** regelmäßig manuell testet. Er gab Schätzungen zu einigen exemplarischen Testfällen ab, mithilfe derer einige zeitliche Hochrechnungen möglich sind. [Ulm21]

- Testfall 1: Für das manuelle Erstellen eines neuen Jobs mit 13 *Additional Settings* und das anschließende Überprüfen auf Richtigkeit braucht der Tester geschätzte 30 bis 45 Minuten. Wird davon ausgegangen, dass alle Integrationstestfälle manuell in durchschnittlich 30 Minuten durchgeführt werden, so würden manuelle Tests für die 151 Testfälle hochgerechnet 75,5 Stunden oder 4530 Minuten dauern. Mit der bereits berechneten zu erwartenden Durchführungszeit von 6,3 Minuten bei einem automatisierten Testdurchlauf ergibt sich eine grobe Zeitersparnis von 75,4 Stunden.
- Testfall 2 und 3: Zonendaten werden über die Oberfläche in Command erstellt. Für diesen Schritt und die anschließende Überprüfung der Daten in Command benötigt der *Test Engineer* ungefähr 15 Minuten. Um zu überprüfen, ob der Job „CIF\_AUTOMATED\_TEST\_JOB“ existiert, benötigt der Tester ungefähr 5 Minuten. Angenommen, alle Smoke-Testfälle außer den Testfällen, die die Testdaten in die jeweiligen Tabellen speichern, würden den Tester im Schnitt 10 Minuten kosten, dann würde für die 29 entsprechenden Smoke-Testfälle eine Durchführungszeit von 290 Minuten entstehen. Die automatisierten Testfälle würden rund 50 Sekunden benötigen. Die Zeitersparnis beträgt somit rund 289 Minuten oder 4,82 Stunden.
- Testfälle 4 und 5: Das Erstellen der Testdaten in den **NMS**-Tabellen für eine Geräteklasse kostet den Tester schätzungsweise 20 Minuten, um entsprechende Testdaten in Command zu erstellen rechnet er mindestens 25 Minuten ein. Wie in [Abschnitt 5.2.1](#) beschrieben, existieren insgesamt fünf Geräteklassen, neun Serviceklassen, vier Zonenklassen und jeweils zwei Klassen für dynamische und allgemeine Daten. Das sind insgesamt 22 Entitätsklassen, die vom Tester bei Bedarf gespeichert werden müssen. Würde der Tester also an einem Tag alle Entitätsklassen durchtesten wollen, so müsste er geschätzte 990 Minuten allein mit dem Speichern der Testdaten verbringen.

Für diesen Vorgang bräuchten die automatisierten Tests hochgerechnet um die 4 Minuten. Der Zeitunterschied ist hier mit rund 986 Minuten, also umgerechnet 16,43 Stunden, ebenfalls beträchtlich.

- Testfall 6: Das Durchführen eines der in [Abschnitt 5.2.1](#) beschriebenen [E2E](#)-Tests für eine Geräteklasse und das anschließende Überprüfen der Daten in Command kostet den Tester schätzungsweise 20 Minuten. Die 147 [E2E](#)-Tests würden einen manuellen Tester demnach ungefähr 2940 Minuten, also 49 Stunden kosten. Dies ist der Grund, weshalb hier nie ausführliche Tests stattfanden. Wie bereits hochgerechnet benötigen automatisierte Tests hier rund 4,7 Minuten für alle [E2E](#)-Tests, wobei bei dieser Zahl sicherlich noch ein paar Minuten addiert werden müssen, da die Entitätsklassen nicht äquivalent zu betrachten sind. Doch selbst bei einer Durchführungszeit von beispielsweise 15 Minuten ist eine Zeitersparnis von 2925 Minuten vorhanden.

Insgesamt lässt sich die Zeitersparnis auf rund 8724 Minuten addieren. Für die 26 [GUI](#)-Tests kann zudem mit einer Zeitersparnis von ungefähr 250 Minuten ausgegangen werden, wenn der Tester durchschnittlich 10 Minuten für jeden Testfall benötigt. Aus den daraus resultierenden 8974 Minuten ergeben sich 149,57 Stunden Zeitersparnis, was rund 6,23 Tagen gleichkommt. Bei einem Arbeitstag von acht Stunden entspricht dies 18,7 Arbeitstagen Zeitersparnis. Diese Zahlen sind kritisch zu betrachten, denn wie bereits beschrieben wurden aufgrund der knappen Zeit einige Tests manuell nicht durchgeführt, die Zeit, die hier mit eingerechnet wurde, also nie wirklich investiert. Da aber für eine ausreichende Qualitätssicherung alle erstellten Testfälle getestet werden müssten, wurden diese Zahlen mit eingerechnet. Darüber hinaus zu beachten ist, dass die Zahlen lediglich eine Hochrechnung sind, hier also deutliche Abweichungen auftreten können. Sie geben aber eine grobe Richtung der zu erwartenden theoretischen Zeitersparnis wieder.

Der **Testaufwand** lässt sich ebenfalls schätzen. In rund 20 Arbeitstagen von durchschnittlich acht Stunden wurde das Konzept für die automatisierten Tests erstellt, in weiteren 20 Arbeitstagen wurde das Testprojekt aufgesetzt, 32 Testfälle implementiert und das Projekt dokumentiert. Für das reine Implementieren kann davon mit 15 Arbeitstagen gerechnet werden. Das ergibt 2,13 implementierte Testfälle pro Tag als **Testproduktivität**. [[SBS07](#)] Würde diese Zahl konstant bleiben, würden für die Umsetzung der übrigen 322 Testfälle 151 Arbeitstage benötigt werden. Zu beachten ist jedoch, dass sich zunächst in die jeweilige Arbeitsweise und die Werkzeuge eingearbeitet werden musste. Nun besteht für jede Testart und jedes Werkzeug mindestens ein exemplarischer Testfall und eine ausführliche Dokumentation, die die Implementierung erleichtern. Die Testumgebung ist eingerichtet und die Werkzeuge kommunizieren miteinander wie geplant. Auch die Struktur des Testprojekts ist vorgegeben und muss nur weitergeführt werden. Mit zunehmender Expertise wird die Zahl an Testfällen pro Tag weiter steigen. Bei vier Testfällen, die pro Tag durchschnittlich



implementiert werden, würden noch 80 Arbeitstage, also ungefähr vier Monate benötigt. Für eine eventuell benötigte Einarbeitung und Wartung oder Anpassung der Testfälle sollten mindestens weitere 15 Arbeitstage eingeplant werden. Der Gesamtaufwand kann damit auf mindestens 135 Arbeitstage geschätzt werden. Kosten, die daraus entstehen sind zum einen der Lohn des Testers oder Entwicklers und die Kosten, die daraus entstehen, dass diese Person in diesem Zeitraum an keinem anderen Projekt oder nur sehr wenig an weiteren Projekten arbeiten kann.

Wie in [Unterabschnitt 2.2.7](#) beschrieben, ist es durchaus normal, dass das Einführen von automatisierten Tests zunächst mehr Zeit und Geld in Anspruch nimmt als einige manuelle Durchläufe. Die Umsetzung der automatisierten Tests wird sich erst nach einigen Testdurchläufen rechnen. Bei einem Aufwand von 135 Arbeitstagen und einer Zeitersparnis von 18,7 Arbeitstagen lohnen sich automatisierte Tests nach rund acht Durchläufen. Werden die Tests regelmäßig einmal wöchentlich ausgeführt, würde sich das Projekt rund zwei Monaten nach Fertigstellung bereits rechnen. Diese Zahlen sind eine sehr grobe Hochrechnung und sollten kritisch betrachtet werden. Die Richtung, die die Zahlen einschlagen, sollte aber dennoch als Wegweiser dienen können. Selbst bei einem Aufwand von 150 Tagen und einer Zeitersparnis von 15 Arbeitstagen würden sich automatisierte Tests früh, also nach 2,5 Monaten beziehungsweise zehn Durchläufen lohnen. Bei jeder Änderung am CIF und damit einem Hinzufügen oder Abändern von Testfällen steigt der Aufwand, das muss beachtet werden. Vor allem bei größeren Änderungen und bei einem weiteren Implementierungsfortschritt sollte eine erneute Berechnung der hier dargestellten Zahlen erfolgen.

Der **Erfolg eines Testdurchlaufs** wird von *TestRail* gemessen. Bei diesem Testdurchlauf liefen alle Testfälle fehlerfrei durch, wie in [Abbildung 7.3](#) abgebildet ist.

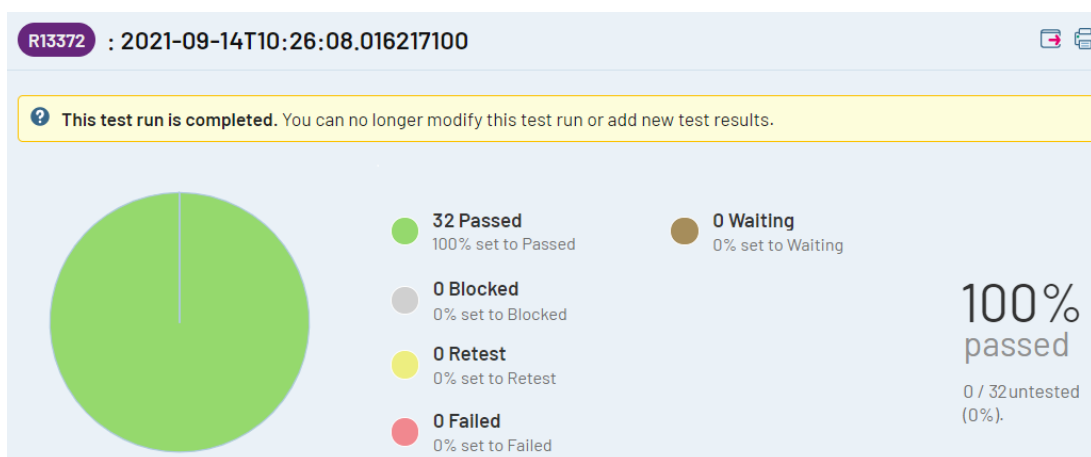


Abbildung 7.3: Die Statistik des Testdurchlaufs auf *TestRail*.



In den bisher implementierten Testfällen wurden keine **Fehler** gefunden, weshalb keine Daten über diese erhoben werden können.

Insgesamt arbeiten die **Testwerkzeuge** schnell und zielführend miteinander. Alle erstellten Testfälle lassen sich mit der in [Kapitel 6](#) beschriebenen Vorgehensweise umsetzen. Für die Implementierung der **GUI-Test** wäre vielleicht ein *Record-Playback*-Werkzeug schneller in der Umsetzung der einzelnen Testfälle. Da aber nun eine Einarbeitung in *Selenide* stattgefunden hat, muss evaluiert werden, ob der Zeitaufwand für ein neues Werkzeug den Zeitaufwand für die manuelle Implementierung der **GUI-Testfälle** schlägt.

Was bisher funktioniert ist die **Abwärtskompatibilität** zum **CIF** der Version 12. Die **BGW** und die Oberfläche unterscheiden sich aktuell in beiden Versionen nicht voneinander. Endpunkte und Tabellennamen sind gleich und auf der Oberfläche sind alle Bestandteile gleich benannt. Lediglich die Basis-URL unterscheidet sich. Sie kann aber im Projekt flexibel ausgetauscht werden. Diese Kompatibilität muss aber nicht zwingend länger gegeben sein. Sobald eine Änderung im **CIF** der Version 13 auftritt und Testfälle an diese angepasst werden müssen, wird das System aufgrund abweichender **API-Endpunkte** oder Benennungen der Bestandteile der Oberfläche nicht mehr oder nur noch teilweise abwärtskompatibel testbar sein.

Insgesamt lässt sich sagen, dass die in [Kapitel 4](#) genannten Punkte des **Soll-Zustands** mit diesem Konzept vollständig erfüllt werden konnten. In regelmäßigen Reviews konnte die Qualität des erstellten Testprojekts hinsichtlich Vollständigkeit, korrekter Funktionsweise und Einhalten der Code- und Testrichtlinien sichergestellt und Grundlagen für eine nahtlose Fortsetzung der Umsetzung gelegt werden. Neben der großen Zeitersparnis, die sich aus den automatisierten Tests ergibt, ist vor allem der Zugewinn an Qualitätssicherung für das **CIF** beträchtlich.

## 8 Fazit und Ausblick

Das Ziel der Arbeit war die Entwicklung eines Konzepts für das automatisierte Testen der Funktionen des [CIF](#). Dabei sollte eine möglichst vollständige Testabdeckung und ein hoher Automatisierungsgrad ermöglicht werden. Wie in [Kapitel 7](#) ausgeführt, ist durch das entwickelte Konzept eine passende Umsetzung geplant, prototypisch implementiert und für zielführend befunden worden. Das Konzept selbst ist umfangreich und umfasst Bedingungen wie die Einteilung in Teststufen, Einrichten von Testumgebung und Testdaten und die Auswahl von passenden Testwerkzeugen. Durch die regelmäßige Ausführung auf einem [CI-Server](#), die Fülle an Regressionstest und des voraussichtlich wöchentlichen Testberichts konnte dem agilen Umfeld gerecht werden. Zusätzlich wird eine frühe Aufdeckung von Fehlern gefördert. So kann die Qualität des [CIF](#) als Software nachhaltig gesteigert und bei ausreichender Pflege gehalten werden.

Auch wenn das Ergebnis der Arbeit zufriedenstellend ist, so ist das Projekt immer noch erweiterbar und kann verbessert werden. So wäre das Einbeziehen der Performance Tests in die *Build-Pipeline* denkbar. Das hätte den Vorteil, dass so alle speziell für das [CIF](#) durchgeführten Tests außer den Komponententests an einem Ort hinterlegt und ausführbar sind.

Da einige der [E2E](#)-Testfälle, abgesehen von den zugehörigen Tabellen, in ihrem Aufbau identisch sein werden, könnte sich gerade bei den noch umzusetzenden 135 Testfällen eine automatisierte Codegenerierung lohnen. Ebenso wäre eine Testdatengenerierung unter Umständen denkbar. Beide Vorschläge sind aber vermutlich aufwendiger in der Planung, als die Fortsetzung der hier vorgestellten Umsetzung Ressourcen kosten würde, denn wie beschrieben müssen gerade die Testdaten akribisch aufeinander abgestimmt sein.

Um die Schnelligkeit der Ausführung weiter zu steigern wäre auch eine parallele Ausführung der Tests umsetzbar. Alle ausgewählten Werkzeuge unterstützen das. Weitere Werkzeuge für das Sammeln der Anwendungsfälle oder das Erstellen einer *Traceability Matrix* könnten ebenfalls hinzugezogen werden.

Zukünftig wird die Implementierung zunächst von einem [CIF](#)-nahen Entwickler übernommen, wie in [Unterabschnitt 5.1.13](#) vorgeschlagen. Je nach Fortschritt ist der Einsatz eines weiteren Mitarbeiters sinnvoll. Mit dem hier erstellten Konzept kann insgesamt aber auch von einer Person ein solides Testprojekt umgesetzt werden, das das Team unterstützt und bei Anwenden des [CIF](#) für Vertrauen sorgen kann. Manuelle Tester werden entlastet und die Softwarequalität nachhaltig gesichert.

# Literatur

- [Aks+15] H. L. Akshaya u. a. „A Basic Introduction to DevOps Tools“. In: *International Journal of Computer Science and Information Technologies* 2015.6 (2015), S. 2349–2353. URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.735.2925&rep=rep1&type=pdf> (besucht am 18.09.2021).
- [AP18] S.A.I.B.S. Arachchi und Indika Perera. *Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management: 4th International Multidisciplinary Engineering Research Conference : May 30-June 1, 2018, Civil Engineering Complex, University of Moratuwa, Sri Lanka*. Piscataway, NJ: IEEE, 2018. ISBN: 978-1-5386-4417-1. URL: <http://ieeexplore.ieee.org/servlet/opac?punumber=8410204> (besucht am 15.09.2021).
- [Atl20] Atlassian. *Get started with Bitbucket Cloud*. 2020. URL: <https://support.atlassian.com/bitbucket-cloud/docs/get-started-with-bitbucket-cloud/> (besucht am 18.09.2021).
- [Atl21] Atlassian. *Manage webhooks*. 2021. URL: <https://support.atlassian.com/bitbucket-cloud/docs/manage-webhooks/> (besucht am 19.09.2021).
- [AtloJ] Atlassian. *A brief overview of Jira*. o.J. URL: <https://www.atlassian.com/software/jira/guides/getting-started/overview> (besucht am 11.09.2021).
- [AV14] A. Apostolov und B. Vandiver. „End to End testing - What should you know?“. In: *2014 67th Annual Conference for Protective Relay Engineers*. IEEE, 2014, S. 125–131. ISBN: 978-1-4799-4739-3. DOI: [10.1109/CPRE.2014.6798999](https://doi.org/10.1109/CPRE.2014.6798999). (Besucht am 25.09.2021).
- [bae19] baeldung. *Jackson vs Gson*. 2019. URL: <https://www.baeldung.com/jackson-vs-gson> (besucht am 13.09.2021).
- [bae20] baeldung. *REStEasy Client API*. 2020. URL: <https://www.baeldung.com/resteasy-client-tutorial> (besucht am 13.09.2021).
- [Bal17] Bogunova Mohanram Balachandar. *RESTful Java web services: A pragmatic guide to designing and building RESTful APIs using Java*. Third edition. Birmingham, UK: Packt Publishing, 2017. ISBN: 9781788294966.
- [Bec+01] Kent Beck u. a. *Manifest für Agile Softwareentwicklung*. 2001. URL: <https://agilemanifesto.org/iso/de/manifesto.html> (besucht am 15.09.2021).

- [Bec+21] Stefan Bechtold u. a. *JUnit 5 User Guide*. 2021. URL: <https://junit.org/junit5/docs/current/user-guide/> (besucht am 18.09.2021).
- [Bel15] Vladimir Belorusetz. „REST API Test Automation with Open Source Tools“. In: *Methods & Tools*. Bd. Volume 23 - number 4. 2015, S. 16–28. URL: <https://www.methodsandtools.com/PDF/mt201504.pdf#page=16> (besucht am 13.09.2021).
- [Beu04] Cédric Beust. *TestNG*. 2004. URL: <https://testng.org/doc/> (besucht am 18.09.2021).
- [Beu12] Cédric Beust. *TestNG*. 2012. URL: <https://testng.org/doc/documentation-main.html> (besucht am 18.09.2021).
- [Bis21] Nabendu Biswas. „Using Webhooks at the Site“. In: *Advanced Gatsby Projects*. Hrsg. von Nabendu Biswas. Berkeley, CA: Apress, 2021, S. 133–147. ISBN: 978-1-4842-6639-7. DOI: [10.1007/978-1-4842-6640-3\\_4](https://doi.org/10.1007/978-1-4842-6640-3_4). (Besucht am 25.09.2021).
- [BM11] Tim Berglund und Matthew McCullough. *Building and Testing with Gradle*. Sebastopol: O’Reilly Media Inc, 2011. ISBN: 9781449304638. URL: <http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10763586> (besucht am 17.09.2021).
- [Bos21] Boston Consulting Group. *What Is the Growth Share Matrix?* 2021. URL: <https://www.bcg.com/de-de/about/our-history/growth-share-matrix> (besucht am 29.08.2021).
- [BR08] Armin Beer und Rudolf Ramler, Hrsg. *The Role of Experience in Software Testing Practice*. IEEE, 2008. ISBN: 978-0-7695-3276-9. DOI: [10.1109/SEAA.2008.28](https://doi.org/10.1109/SEAA.2008.28). (Besucht am 02.09.2021).
- [Bre+20] Michael Brenner u. a. *Praxisbuch ISO/IEC 27001*. München: Carl Hanser Verlag, 2020. ISBN: 978-3-446-46170-3.
- [CDM18] Andrei Contan, Catalin Dehelean und Liviu Miclea. „Test automation pyramid from theory to practice“. In: *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*. IEEE, 2018, S. 1–5. ISBN: 978-1-5386-2205-6. DOI: [10.1109/AQTR.2018.8402699](https://doi.org/10.1109/AQTR.2018.8402699). (Besucht am 27.08.2021).
- [CG14] Lisa Crispin und Janet Gregory. *Agile testing: A practical guide for testers and agile teams*. 1. ed., 10. print. A Mike Cohn signature book. Upper Saddle River: Addison-Wesley, 2014. ISBN: 9780321534460.

- [Cha14] Vinod Kumar Chauhan. „Smoke Testing“. In: *International Journal of Scientific and Research Publications* 4 (2014). Artikel 2. URL: [https://www.academia.edu/6733225/Smoke\\_Testing](https://www.academia.edu/6733225/Smoke_Testing) (besucht am 28.08.2021).
- [Cli12] Cliffdcw. *File:SDLC - Software Development Life Cycle.jpg*. Lizenz: <https://creativecommons.org/licenses/by-sa/3.0/deed.en>. 2012. URL: [https://en.wikipedia.org/wiki/File:SDLC\\_-\\_Software\\_Development\\_Life\\_Cycle.jpg](https://en.wikipedia.org/wiki/File:SDLC_-_Software_Development_Life_Cycle.jpg) (besucht am 08.09.2021).
- [codoJ] codeborne. *Documentation*. o.J. URL: <https://selenide.org/documentation.html> (besucht am 19.09.2021).
- [Coh10] Mike Cohn. *Succeeding with agile: Software development using Scrum*. The Addison-Wesley signature series A Mike Cohn signature book. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN: 9780321579362.
- [Cri11] Lisa Crispin. *Using the Agile Testing Quadrants*. 2011. URL: <https://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/> (besucht am 29.08.2021).
- [Cro20] Croncal. *File:The test automation pyramid.png*. Lizenz: <https://creativecommons.org/licenses/by-sa/4.0/deed.en>. 2020. URL: [https://commons.wikimedia.org/wiki/File:The\\_test\\_automation\\_pyramid.png](https://commons.wikimedia.org/wiki/File:The_test_automation_pyramid.png) (besucht am 27.08.2021).
- [Dav20] Adam L. Davis. „Building“. In: *Modern Programming Made Easy*. Hrsg. von Adam L. Davis. Berkeley, CA: Apress, 2020, S. 99–109. ISBN: 978-1-4842-5568-1. DOI: [10.1007/978-1-4842-5569-8\\_13](https://doi.org/10.1007/978-1-4842-5569-8_13). (Besucht am 25.09.2021).
- [Dee+20] N. Deepa u. a. „An analysis on Version Control Systems“. In: *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*. 2020, S. 1–9. DOI: [10.1109/ic-ETITE47903.2020.39](https://doi.org/10.1109/ic-ETITE47903.2020.39). (Besucht am 18.09.2021).
- [Dif+21] DiffPlug u. a. *diffplug/spotless: Spotless: Keep your code spotless*. 2021. URL: <https://github.com/diffplug/spotless> (besucht am 18.09.2021).
- [Dri20] Meike Drießen. *Zwickmühle Corona-Warn-App*. 2020. URL: <https://news.rub.de/presseinformationen/wissenschaft/2020-11-06-neues-projekt-zwickmuehle-corona-warn-app> (besucht am 04.08.2021).
- [Dvo09] Daniel L. Dvorak. *NASA Study on Flight Software Complexity: Final Report*. 2009. URL: [https://www.nasa.gov/pdf/418878main\\_FSWC\\_Final\\_Report.pdf](https://www.nasa.gov/pdf/418878main_FSWC_Final_Report.pdf) (besucht am 02.08.2021).
- [Ebe+16] Christof Ebert u. a. „DevOps“. In: *IEEE Software* 33.3 (2016), S. 94–100. ISSN: 0740-7459. DOI: [10.1109/MS.2016.68](https://doi.org/10.1109/MS.2016.68). (Besucht am 25.09.2021).

- [Fie20] Lena Fiedler. *Sieben Fehler der Corona-Warn-App und was sie bedeuten*. 2020. URL: <https://www.zeit.de/digital/mobil/2020-07/covid-19-corona-warn-app-fehler-smartphone-android-ios/komplettansicht> (besucht am 04.08.2021).
- [Fil+21] Katarzyna Filus u. a. „Efficient Feature Selection for Static Analysis Vulnerability Prediction“. In: *Sensors (Basel, Switzerland)* 21.4 (2021). DOI: [10.3390/s21041133](https://doi.org/10.3390/s21041133). (Besucht am 18.09.2021).
- [FNT20] FNT GmbH. *Test Policy*. 2020. URL: [https://fntgrp.sharepoint.com/:w:/r/sites/DevelopmentOperations/\\_layouts/15/Doc.aspx?sourcedoc=%7B6566DCBA-C8B3-4C3D-BD5F-06CE86F6B2A8%7D&file=Test\\_Policy.docx&action=default&mobileredirect=true&DefaultItemOpen=1](https://fntgrp.sharepoint.com/:w:/r/sites/DevelopmentOperations/_layouts/15/Doc.aspx?sourcedoc=%7B6566DCBA-C8B3-4C3D-BD5F-06CE86F6B2A8%7D&file=Test_Policy.docx&action=default&mobileredirect=true&DefaultItemOpen=1) (besucht am 08.09.2021).
- [FNT21a] FNT Quality Assurance. *Best Practices and Testing Guidelines*. 2021. URL: <https://fntgrp.sharepoint.com/sites/DevelopmentOperations/SitePages/QA-Best-Practices.aspx?csf=1&web=1&e=WslPlahttp://> (besucht am 08.09.2021).
- [FNT21b] FNT Quality Assurance. *QA Know how: Approach / Tools / Technologies / Methodology / Communication*. 2021. URL: <https://fntgrp.sharepoint.com/sites/DevelopmentOperations/SitePages/QA-General.aspx> (besucht am 08.09.2021).
- [FNTtoJ] FNT GmbH. *Das Unternehmen: FNT Software*. o.J. URL: <https://www.fntsoftware.com/fnt/das-unternehmen> (besucht am 01.08.2021).
- [Fui19] Flaviu Fuior. „An overview of some tools for automated testing of software applications“. In: *Romanian Journal of Information Technology and Automatic Control*. Bd. 29. 2019, S. 97–106. URL: [https://rria.ici.ro/wp-content/uploads/2019/10/08-art.8-Fuior\\_Flaviu.pdf](https://rria.ici.ro/wp-content/uploads/2019/10/08-art.8-Fuior_Flaviu.pdf) (besucht am 23.09.2021).
- [Gar+20] Boni García u. a. „A Survey of the Selenium Ecosystem“. In: *Electronics* 9.7 (2020), S. 1067. DOI: [10.3390/electronics9071067](https://doi.org/10.3390/electronics9071067). (Besucht am 20.09.2021).
- [Gra21] Gradle Inc. *Releases*. 20.08.2021. URL: <https://gradle.org/releases/> (besucht am 24.09.2021).
- [GS17] Shekhar Gulati und Rahul Sharma. „Integrating Tools“. In: *Java Unit Testing with JUnit 5*. Hrsg. von Shekhar Gulati und Rahul Sharma. Berkeley, CA: Apress, 2017, S. 99–119. ISBN: 978-1-4842-3014-5. DOI: [10.1007/978-1-4842-3015-2\\_6](https://doi.org/10.1007/978-1-4842-3015-2_6). (Besucht am 18.09.2021).
- [GuroJa] Gurock. *Configuring the Jira Cloud Integration*. o.J. URL: <https://www.gurock.com/testrail/docs/integrate/tools/jira/cloud> (besucht am 27.09.2021).



- [GuroJb] Gurock. *TestRail API Reference Guide*. o.J. URL: <https://www.gurock.com/testrail/docs/api> (besucht am 19. 09. 2021).
- [Hal21] Johan Haleby. *REST Assured*. 21.05.2021. URL: <https://rest-assured.io/> (besucht am 19. 09. 2021).
- [Her+13] Israel Herraiz u. a. „The evolution of the laws of software evolution: A Discussion Based on a Systematic Literature Review“. In: *ACM Computing Surveys* 46.2 (2013), S. 1–28. ISSN: 0360-0300. DOI: [10.1145/2543581.2543595](https://doi.org/10.1145/2543581.2543595). (Besucht am 02. 08. 2021).
- [HH08] Reinhard Höhn und Stephan Höppner. *Das V-Modell XT*. eXamen.press. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. ISBN: 978-3-540-30249-0. DOI: [10.1007/978-3-540-30250-6](https://doi.org/10.1007/978-3-540-30250-6). URL: <https://link.springer.com/content/pdf/10.1007%5C%2F978-3-540-30250-6.pdf> (besucht am 09. 09. 2021).
- [HSv17] Les Hatton, Diomidis Spinellis und Michiel van Genuchten. „The long-term growth rate of evolving software: Empirical results and implications“. In: *Journal of Software: Evolution and Process* 29.5 (2017), e1847. ISSN: 20477473. DOI: [10.1002/smr.1847](https://doi.org/10.1002/smr.1847). (Besucht am 02. 08. 2021).
- [IEE93] IEEE. *IEEE Standard Classification for Software Anomalies*. Piscataway, NJ, USA, 1993. DOI: [10.1109/IEEESTD.1994.121429](https://doi.org/10.1109/IEEESTD.1994.121429). (Besucht am 10. 09. 2021).
- [ISO13a] ISO/IEC/IEEE. *Software and systems engineering Software testing Part 1: Concepts and definitions*. Piscataway, NJ, USA, 2013. DOI: [10.1109/IEEESTD.2013.6588537](https://doi.org/10.1109/IEEESTD.2013.6588537). (Besucht am 10. 09. 2021).
- [ISO13b] ISO/IEC/IEEE. *Software and systems engineering Software testing Part 2: Test processes*. Piscataway, NJ, USA, 2013. DOI: [10.1109/IEEESTD.2013.6588543](https://doi.org/10.1109/IEEESTD.2013.6588543). (Besucht am 23. 08. 2021).
- [ISO13c] ISO/IEC/IEEE. *Software and systems engineering Software testing Part 3: Test documentation*. Piscataway, NJ, USA, 2013. DOI: [10.1109/IEEESTD.2013.6588540](https://doi.org/10.1109/IEEESTD.2013.6588540). (Besucht am 10. 09. 2021).
- [ISO15] ISO/IEC/IEEE. *ISO/IEC/IEEE International Standard - Software and systems engineering–Software testing–Part 4: Test techniques*. Piscataway, NJ, USA, 2015. DOI: [10.1109/IEEESTD.2015.7346375](https://doi.org/10.1109/IEEESTD.2015.7346375). (Besucht am 10. 09. 2021).
- [ISO17] ISO/IEC/IEEE. *ISO/IEC/IEEE International Standard - Systems and software engineering–Measurement process*. Piscataway, NJ, USA, 2017. DOI: [10.1109/IEEESTD.2017.7907158](https://doi.org/10.1109/IEEESTD.2017.7907158). URL: <https://ieeexplore.ieee.org/document/7907158> (besucht am 10. 09. 2021).

- [IST20] ISTQB®. *About ISTQB®: ISTQB® - Advancing the software testing profession*. 2020. URL: <https://www.istqb.org/about-us.html> (besucht am 07.09.2021).
- [Jam+16] Muhammad Abid Jamil u. a. *Software Testing Techniques: A Literature Review*. Piscataway, NJ: IEEE, 2016. ISBN: 978-1-5090-4521-1. URL: <http://ieeexplore.ieee.org/servlet/opac?punumber=7813518> (besucht am 27.08.2021).
- [JBo13] JBoss Community. *Chapter 45. Resteasy Client API: 45.2. Resteasy Proxy Framework*. 2013. URL: [https://docs.jboss.org/resteasy/docs/3.0-beta-3/userguide/html/RESEasy\\_Client\\_Framework.html#d4e2049](https://docs.jboss.org/resteasy/docs/3.0-beta-3/userguide/html/RESEasy_Client_Framework.html#d4e2049) (besucht am 13.09.2021).
- [KG04] Daryl Kulak und Eamonn Guiney. *Use Cases: Requirements in Context*. 2. Aufl. Boston: Addison-Wesley, 2004. ISBN: 9780133085150.
- [KG13] Harpreet Kaur und Gagan Gupta. „Comparative Study of Automated Testing Tools: Selenium, Quick Test Professional and Testcomplete“. In: *Int. Journal of Engineering Research and Applications*. Bd. 3. 2013, S. 1739–1743. URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.448.6743&rep=rep1&type=pdf> (besucht am 13.09.2021).
- [KK15] Claus Klammer und Albin Kern. „Writing unit tests: It’s now or never!“ In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2015, S. 1–4. ISBN: 978-1-4799-1885-0. DOI: [10.1109/ICSTW.2015.7107469](https://doi.org/10.1109/ICSTW.2015.7107469). (Besucht am 25.09.2021).
- [Kle19] Stephan Kleuker. *Qualitätssicherung durch Softwaretests: Vorgehensweisen und Werkzeuge zum Testen von Java-Programmen*. 2nd ed. 2019. Wiesbaden: Springer Fachmedien Wiesbaden und Imprint: Springer Vieweg, 2019. ISBN: 978-3-658-24886-4. DOI: [10.1007/978-3-658-24886-4](https://doi.org/10.1007/978-3-658-24886-4). (Besucht am 25.09.2021).
- [KM19] Sithembiso Khumalo und Martie Mearns. „SharePoint as enabler for collaboration and efficient project knowledge sharing“. In: *SA Journal of Information Management* 21.1 (2019). ISSN: 2078-1865. DOI: [10.4102/sajim.v21i1.1044](https://doi.org/10.4102/sajim.v21i1.1044). (Besucht am 24.09.2021).
- [LA17] Stefan Luber und Stephan Augsten. *Definition „Programmierschnittstelle“ Was ist eine API?* 2017. URL: <https://www.dev-insider.de/was-ist-eine-api-a-583923/> (besucht am 15.09.2021).
- [Las18] Brent Laster. *Jenkins 2: Up and Running: Evolve Your Deployment Pipeline for Next Generation Automation*. Sebastopol: O’Reilly Media, Inc., 2018. ISBN: 978-1-491-97959-4.



- 
- [Len+20] Valentina Lenarduzzi u. a. „Are SonarQube Rules Inducing Bugs?“ In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, S. 501–511. ISBN: 978-1-7281-5143-4. DOI: [10.1109/SANER48275.2020.9054821](https://doi.org/10.1109/SANER48275.2020.9054821). (Besucht am 18.09.2021).
  - [Len+21] Valentina Lenarduzzi u. a. *A Critical Comparison on Six Static Analysis Tools: Detection, Agreement, and Precision*. 2021. URL: <http://arxiv.org/pdf/2101.08832v1> (besucht am 18.09.2021).
  - [Li11] Hongjun Li. „RESTful Web service frameworks in Java“. In: *2011 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC)*. 2011, S. 1–4. DOI: [10.1109/ICSPCC.2011.6061739](https://doi.org/10.1109/ICSPCC.2011.6061739). (Besucht am 13.09.2021).
  - [Lig09] Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. 2. Aufl. Heidelberg: Spektrum Akademischer Verlag, 2009. ISBN: 978-3-8274-2056-5. DOI: [10.1007/978-3-8274-2203-3](https://doi.org/10.1007/978-3-8274-2203-3). (Besucht am 25.08.2021).
  - [Lio96] Jaques-Louis Lions. *Ariane 5 Flight 501 Failure: Report by the Inquiry Board*. Paris, 1996. URL: <https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf> (besucht am 01.09.2021).
  - [Lou11] Panos Louridas. „Test Management“. In: *IEEE Software* 28.5 (2011), S. 86–91. ISSN: 0740-7459. DOI: [10.1109/MS.2011.111](https://doi.org/10.1109/MS.2011.111). (Besucht am 15.09.2021).
  - [LS18] Richard Lackes und Markus Siepermann. *Datenintegration*. 2018. URL: <https://wirtschaftslexikon.gabler.de/definition/datenintegration-31223/version-254785> (besucht am 25.08.2021).
  - [Mad+21] Matej Madeja u. a. „Empirical Study of Test Case and Test Framework Presence in Public Projects on GitHub“. In: *Applied Sciences* 11.16 (2021), S. 7250. DOI: [10.3390/app11167250](https://doi.org/10.3390/app11167250). (Besucht am 20.09.2021).
  - [Mae12] Kazuaki Maeda. „Performance evaluation of object serialization libraries in XML, JSON and binary formats“. In: *2012 Second International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*. 2012, S. 177–182. DOI: [10.1109/DICTAP.2012.6215346](https://doi.org/10.1109/DICTAP.2012.6215346). (Besucht am 13.09.2021).
  - [McC15] David McCandless. *Codebases: Millions of lines of code*. 2015. URL: <https://informationisbeautiful.net/visualizations/million-lines-of-code/> (besucht am 02.08.2021).
  - [McC96] Steve McConnell. „Daily Build and Smoke Test“. In: *IEEE Softw.* 13 (1996), S. 143–144. URL: <https://stevemcconnell.com/wp-content/uploads/2017/08/DailyBuildandSmokeTest.pdf> (besucht am 28.08.2021).
-

- [Mih17] Alexandra Mihalache. „Project Management Tools for Agile Teams“. In: *Informatica Economica* 21.4/2017 (2017), S. 85–93. ISSN: 14531305. DOI: [10.12948/issn14531305/21.4.2017.07](https://doi.org/10.12948/issn14531305/21.4.2017.07). URL: <https://pdfs.semanticscholar.org/aaba/28b1f1070fe759b59764c7879321581ce485.pdf> (besucht am 23. 09. 2021).
- [MR03] Daniel D. McCracken und Edwin D. Reilly. „Backus-Naur Form (BNF)“. In: *Encyclopedia of Computer Science*. GBR: John Wiley and Sons Ltd, 2003, S. 129–131. ISBN: 0470864125. URL: <https://dl.acm.org/doi/abs/10.5555/1074100.1074155> (besucht am 27. 09. 2021).
- [MWI17] Ahmad Mustafa, Wan M.N. Wan-Kadir und Noraini Ibrahim. „Comparative Evaluation of the State-of-art Requirements-based Test Case Generation Approaches“. In: *International Journal on Advanced Science, Engineering and Information Technology*. Bd. 7. 2017, S. 1567–1573. URL: <https://core.ac.uk/reader/325990310> (besucht am 27. 09. 2021).
- [Nik20] Zornitsa Nikolova. „Testing Strategies in an Agile Context“. In: *The Future of Software Quality Assurance*. Hrsg. von Stephan Goericke. Cham: Springer International Publishing, 2020, S. 111–121. ISBN: 978-3-030-29508-0. DOI: [10.1007/978-3-030-29509-7\\_9](https://doi.org/10.1007/978-3-030-29509-7_9). (Besucht am 15. 09. 2021).
- [OF16] Anton Okolnychyi und Konrad Fögen. „A Study of Tools for Behavior-Driven Development“. In: *Full-scale Software Engineering / Current Trends in Release Engineering Seminar 2015/16*. 2016. URL: <https://www2.swc.rwth-aachen.de/docs/teaching/seminar2016/FsSE%20CTRelEng%202016.pdf#page=11> (besucht am 20. 09. 2021).
- [ÖM19] Deniz Özkan und Alok Mishra. „Agile Project Management Tools: A Brief Comprative View“. In: *Cybernetics and Information Technologies* 19.4 (2019), S. 17–25. DOI: [10.2478/cait-2019-0033](https://doi.org/10.2478/cait-2019-0033). (Besucht am 18. 09. 2021).
- [OO19] Bohdan Oliinyk und Vasyl Oleksiuk. „Automation in software testing, can we automate anything we want?“. In: *CEUR Workshop Proceedings of the 2nd Student Workshop on Computer Science and Software Engineering, CS and SE @ SW 2019*. Bd. 2. 2019, S. 224–234. URL: <http://ceur-ws.org/Vol-2546/paper16.pdf> (besucht am 27. 08. 2021).
- [Pat17] Nikhil Pathania. *Learning continuous integration with Jenkins: A beginner’s guide to implementing continuous integration and continuous delivery using Jenkins 2*. Second edition. Birmingham, UK: Packt Publishing, 2017. ISBN: 9781788475198. URL: <http://proquest.tech.safaribooksonline.de/9781788479356> (besucht am 28. 09. 2021).

- [PDE17] Igor Pyrko, Viktor Dörfler und Colin Eden. „Thinking together: What makes Communities of Practice work?“ In: *Human relations; studies towards the integration of the social sciences* 70.4 (2017), S. 389–409. ISSN: 0018-7267. DOI: [10.1177/0018726716661040](https://doi.org/10.1177/0018726716661040). (Besucht am 14.09.2021).
- [Pez+16] Felipe Pezoa u. a. „Foundations of JSON Schema“. In: *Proceedings of the 25th International Conference on World Wide Web*. Hrsg. von Jacqueline Bourdeau u. a. Republic und Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2016, S. 263–273. ISBN: 9781450341431. DOI: [10.1145/2872427.2883029](https://doi.org/10.1145/2872427.2883029). (Besucht am 29.09.2021).
- [Plu20] Werner Pluta. *Boeing entdeckt zwei weitere Softwarefehler*. 2020. URL: <https://www.golem.de/news/boeing-737-max-boeing-entdeckt-zwei-weitere-softwarefehler-2004-147785.html> (besucht am 14.09.2021).
- [Pra+19] Gede Artha Azriadi Prana u. a. „Categorizing the Content of GitHub README Files“. In: *Empirical Software Engineering* 24.3 (2019), S. 1296–1327. ISSN: 1382-3256. DOI: [10.1007/s10664-018-9660-3](https://doi.org/10.1007/s10664-018-9660-3). (Besucht am 29.09.2021).
- [Red21] RedHat. *RESTEasy*. 8.07.2021. URL: <https://resteasy.github.io/> (besucht am 19.09.2021).
- [RSS17] Paruchuri Ramya, Vemuri Sindhura und P. Vidya Sagar. „Testing using selenium web driver“. In: *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. IEEE, 2017, S. 1–7. ISBN: 978-1-5090-3239-6. DOI: [10.1109/ICECCT.2017.8117878](https://doi.org/10.1109/ICECCT.2017.8117878). (Besucht am 25.09.2021).
- [SAZ17] Mojtaba Shahin, Muhammad Ali Babar und Liming Zhu. „Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices“. In: *IEEE Access* 5 (2017), S. 3909–3943. DOI: [10.1109/ACCESS.2017.2685629](https://doi.org/10.1109/ACCESS.2017.2685629). (Besucht am 15.09.2021).
- [SBC12] Abhijit A. Sawant, Pranit H. Bari und P. M. Chawan. „Software Testing Techniques and Strategies“. In: *International Journal on Engineering Research and Applications*. Bd. 2. 2012, S. 980–986. URL: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.416.7931&rep=rep1&type=pdf> (besucht am 27.09.2021).
- [SBS07] Harry M. Sneed, Manfred Baumgartner und Richard Seidl. *Der Systemtest: Anforderungsbasiertes Testen von Software-Systemen*. München: Hanser, 2007. ISBN: 3-446-40793-6. URL: <https://www.hanser-elibrary.com/doi/10.3139/9783446428614> (besucht am 25.09.2021).

- [Sey06] S. Seyfert. *File:V-Modell.png*. Lizenzen: <https://creativecommons.org/licenses/by-sa/3.0/deed.en> und <https://www.gnu.org/licenses/fdl-1.3.html>. 2006. URL: <https://commons.wikimedia.org/wiki/File:V-Modell.png> (besucht am 09.09.2021).
- [SG15] Peter Stoyanov Sabev und Katalina Grigorova. „Transforming Automated Software Functional Tests for Performance and Load Testing“. In: *International Journal of Scientific Engineering and Applied Science (IJSEAS)*. Bd. 1. 2015, S. 447–453. URL: [https://www.researchgate.net/publication/278328926\\_Transforming\\_Automated\\_Software\\_Functional\\_Tests\\_for\\_Performance\\_and\\_Load\\_Testing](https://www.researchgate.net/publication/278328926_Transforming_Automated_Software_Functional_Tests_for_Performance_and_Load_Testing) (besucht am 23.09.2021).
- [SG17] Peter Stoyanov Sabev und Katalina Grigorova. „A Comparative Study of GUI Automated Tools for Software Testing“. In: *SOFTENG 2017, The Third International Conference on Advances and Trends in Software Engineering*. Bd. 3. 2017. URL: [https://www.researchgate.net/publication/319465398\\_A\\_Comparative\\_Study\\_of\\_GUI\\_Automated\\_Tools\\_for\\_Software\\_Testing](https://www.researchgate.net/publication/319465398_A_Comparative_Study_of_GUI_Automated_Tools_for_Software_Testing) (besucht am 19.09.2021).
- [Sil21] Veronika Silberg. *Folgenschwerer Bug: kann die Corona-Warn-App so leicht ausgetrickst werden?* 2021. URL: <https://www.merkur.de/welt/corona-warn-app-bug-sicherheit-fehler-impfschutz-daten-datum-zr-90811318.html> (besucht am 04.08.2021).
- [SIM11] Muhammad Shahid, Suhaimi Ibrahim und Mohd Naz’ri Mahrin. „A Study on Test Coverage in Software Testing“. In: *2011 International Conference on Telecommunication Technology and Applications*. Bd. 5. 2011, S. 207–215. URL: [https://www.researchgate.net/profile/Dr-Muhammad-Shahid/publication/228913406\\_A\\_Study\\_on\\_Test\\_Coverage\\_in\\_Software\\_Testing/links/00b7d52859b2459da0000000/A\\_Study\\_on\\_Test-Coverage-in-Software-Testing.pdf](https://www.researchgate.net/profile/Dr-Muhammad-Shahid/publication/228913406_A_Study_on_Test_Coverage_in_Software_Testing/links/00b7d52859b2459da0000000/A_Study_on_Test-Coverage-in-Software-Testing.pdf) (besucht am 27.09.2021).
- [Sin20] Jigar Singh. *DevOps impact on Software Testing Life Cycle*. 2020. URL: [https://repository.stcloudstate.edu/cgi/viewcontent.cgi?article=1045&context=csit\\_etds](https://repository.stcloudstate.edu/cgi/viewcontent.cgi?article=1045&context=csit_etds) (besucht am 23.09.2021).
- [SL19] Andreas Spillner und Tilo Linz. *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB®-Standard*. 6., überarbeitete und aktualisierte Auflage. Heidelberg und Ann Arbor: dpunkt.verlag und ProQuest eBook Central, 2019. ISBN: 978-3-86490-583-4.
- [SMG13] Marian Stoica, Marinela Mircea und Bogdan Ghilic-Micu. „Software Development: Agile vs. Traditional“. In: *Informatica Economica* 17.4/2013 (2013),

- S. 64–76. ISSN: 14531305. DOI: [10.12948/issn14531305/17.4.2013.06](https://doi.org/10.12948/issn14531305/17.4.2013.06). (Besucht am 15.09.2021).
- [Sou+19] Manel Souibgui u. a. „Data quality in ETL process: A preliminary study“. In: *Procedia Computer Science* 159 (2019), S. 676–687. ISSN: 18770509. DOI: [10.1016/j.procs.2019.09.223](https://doi.org/10.1016/j.procs.2019.09.223). (Besucht am 25.08.2021).
- [Spi17] Diomidis Spinellis. „State-of-the-Art Software Testing“. In: *IEEE Software* 34.5 (2017), S. 4–6. ISSN: 0740-7459. DOI: [10.1109/MS.2017.3571564](https://doi.org/10.1109/MS.2017.3571564). (Besucht am 27.08.2021).
- [SS20] Ken Schwaber und Jeff Sutherland. *The Scrum Guide: Der gültige Leitfaden für Scrum: Die Spielregeln*. Lizenz: <http://creativecommons.org/licenses/by-sa/4.0/>. 2020. URL: [https://www.itsmgroup.com/fileadmin/user\\_upload/pdfs/2020-Scrum-Guide-German.pdf](https://www.itsmgroup.com/fileadmin/user_upload/pdfs/2020-Scrum-Guide-German.pdf) (besucht am 15.09.2021).
- [SVP18] Jayasudha Subburaj, Soundarya C. Veni und Shanthi Palaniappan. „Optimization of Software Reuse in Agile Software Development (OSRAD)“. In: *International Journal of Pure and Applied Mathematics*. Bd. 119. 2018, S. 1959–1967. URL: <https://acadpubl.eu/hub/2018-119-12/articles/7/1738.pdf> (besucht am 28.09.2021).
- [SW06] Jochen Seemann und Jürgen Wolff von Gudenberg. *Software-Entwurf mit UML 2: Objektorientierte Modellierung mit Beispielen in Java*. 2. Aufl. Xpert.press. Berlin: Springer, 2006. ISBN: 3-540-30949-7. URL: [http://deposit.dnb.de/cgi-bin/dokserv?id=2728459&prov=M&dok\\_var=1&dok\\_ext=htm](http://deposit.dnb.de/cgi-bin/dokserv?id=2728459&prov=M&dok_var=1&dok_ext=htm) (besucht am 22.07.2021).
- [Tsa+01] W. T. Tsai u. a. „End-to-end integration testing design“. In: *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*. IEEE, 2001, S. 166–171. ISBN: 0-7695-1372-7. DOI: [10.1109/CMPSAC.2001.960613](https://doi.org/10.1109/CMPSAC.2001.960613). (Besucht am 25.09.2021).
- [TT21] John T. Taylor und Wayne T. Taylor. „Software Architecture“. In: *Patterns in the Machine: A Software Engineering Guide to Embedded Development*. Berkeley, CA: Apress, 2021, S. 63–82. ISBN: 978-1-4842-6440-9. DOI: [10.1007/978-1-4842-6440-9\\_5](https://doi.org/10.1007/978-1-4842-6440-9_5). (Besucht am 20.09.2021).
- [Ulm21] Ulmer, Kathrin. *Schätzungen für die manuelle Durchführungszeit von ausgewählten Testfällen*. Interview durch Email-Befragung. Das Interview ist im Anhang aufgeführt. Ellwangen, 14.09.2021.
- [Urb21] Monika Urban. „„Die Hoffnung, informiert zu sein“. Effekte der Corona-Warn-App“. In: *Prävention und Gesundheitsförderung* (2021). ISSN: 1861-6755. DOI: [10.1007/s11553-021-00854-9](https://doi.org/10.1007/s11553-021-00854-9). (Besucht am 25.09.2021).

- [Vel+18] John Velandia u. a. „JAX-RS Implementations: A Performance Comparison“. In: *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)* 10.1-8 (2018), S. 139–144. URL: <https://jtec.utem.edu.my/jtec/article/view/3750> (besucht am 24.09.2021).
- [vH12] Michiel van Genuchten und Les Hatton. „Compound Annual Growth Rate for Software“. In: *IEEE Software* 29.4 (2012), S. 19–21. ISSN: 0740-7459. DOI: [10.1109/MS.2012.79](https://doi.org/10.1109/MS.2012.79). URL: <https://ieeexplore.ieee.org/document/6265076> (besucht am 02.08.2021).
- [Win+13] Mario Winter u. a. *Der Integrationstest: Von Entwurf und Architektur zur Komponenten- und Systemintegration*. München: Hanser, 2013. ISBN: 9783446429512. DOI: [10.3139/9783446429512](https://doi.org/10.3139/9783446429512). URL: <http://www.hanser-elibrary.com/isbn/9783446425644> (besucht am 05.09.2021).
- [Wit18] Frank Witte. *Metriken für das Testreporting*. Wiesbaden: Springer Fachmedien Wiesbaden, 2018. ISBN: 978-3-658-19844-2. DOI: [10.1007/978-3-658-19845-9](https://doi.org/10.1007/978-3-658-19845-9). (Besucht am 09.09.2021).
- [Wit19] Frank Witte. *Testmanagement und Softwaretest*. Wiesbaden: Springer Fachmedien Wiesbaden, 2019. ISBN: 978-3-658-25086-7. DOI: [10.1007/978-3-658-25087-4](https://doi.org/10.1007/978-3-658-25087-4). (Besucht am 25.08.2021).
- [Wit20] Frank Witte. *Strategie, Planung und Organisation von Testprozessen*. Wiesbaden: Springer Fachmedien Wiesbaden, 2020. ISBN: 978-3-658-31227-5. DOI: [10.1007/978-3-658-31228-2](https://doi.org/10.1007/978-3-658-31228-2). (Besucht am 09.09.2021).
- [YYN11] Hiroshi Yamada, Takeshi Yada und Hiroto Nomura. „Developing network configuration management database system and its application—data federation for network management“. In: *Telecommunication Systems* (2011). ISSN: 1018-4864. DOI: [10.1007/s11235-011-9607-0](https://doi.org/10.1007/s11235-011-9607-0). (Besucht am 25.09.2021).
- [ZM17] Ahmed Zerouali und Tom Mens. „Analyzing the evolution of testing library usage in open source Java projects“. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, S. 417–421. ISBN: 978-1-5090-5501-2. DOI: [10.1109/SANER.2017.7884645](https://doi.org/10.1109/SANER.2017.7884645). (Besucht am 23.09.2021).

# Anhang

## Anhangsverzeichnis

A - Das Anwendungsfalldiagramm

B - Experteninterview

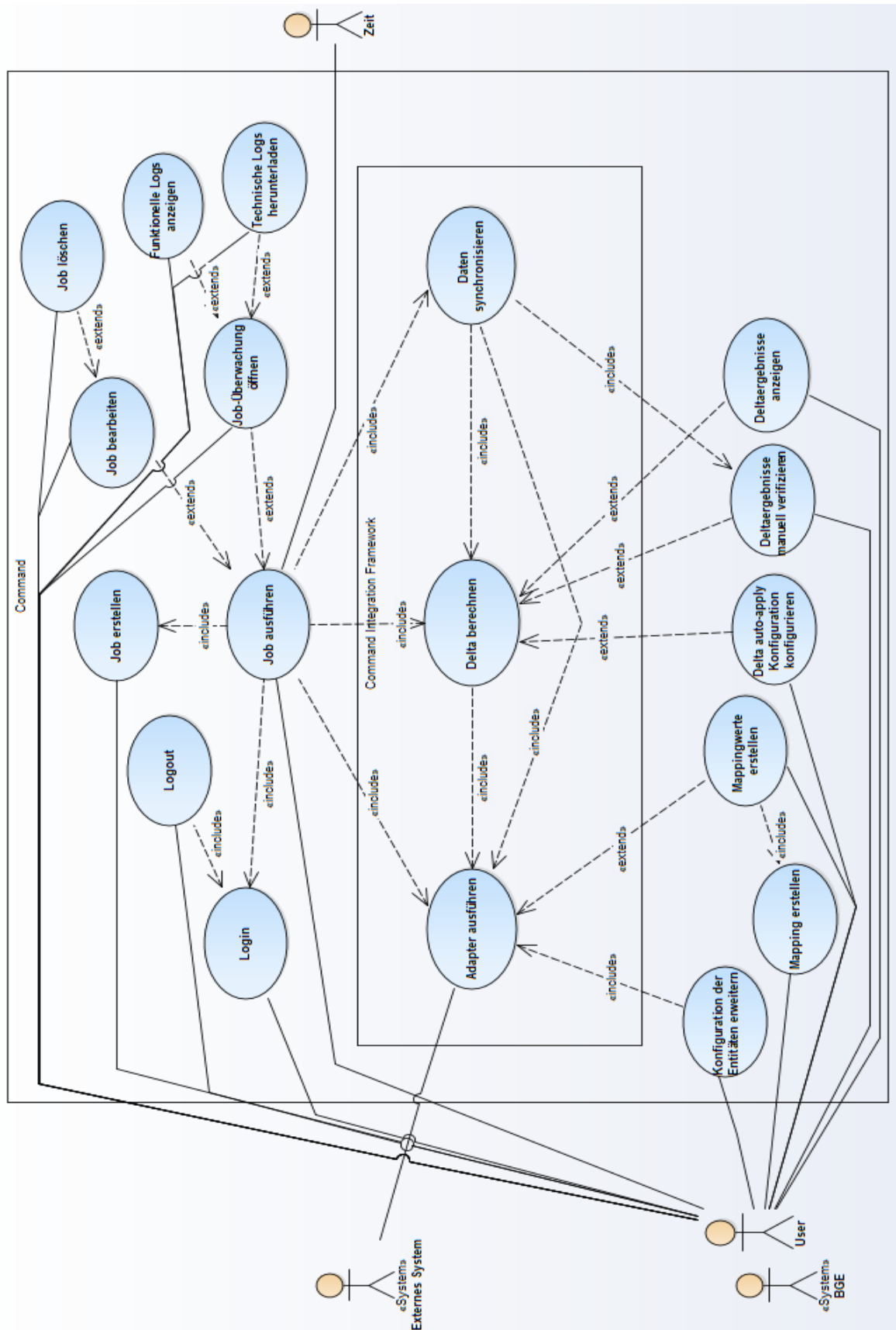
C - Die ausformulierten, im Fließtext nicht ausgeführten Anwendungsfälle

D - Ein Auszug aus der erstellten Traceability Matrix

E - Ein von JUnit erstellter [HTML](#)-Testbericht



## A. Das Anwendungsfalldiagramm





## B. Experteninterview

Interview mit einem Test Engineer, der das CIF bereits seit über drei Jahren regelmäßig manuell testet. Das Interview hat am 14.09.2021 in Form einer Email-Befragung stattgefunden. [Ulm21] Der Test Engineer möchte anonym bleiben.

**Interviewer:** Hier die Testfälle, für die eine Schätzung des Aufwands für manuelle Tests benötigt wird: Erstellen eines neuen Jobs mit den 13 Additional Settings, die in Testfall C19356883 aufgelistet sind.

**Test Engineer:** 0,5-0,75 Stunden.

**Interviewer:** Erstellen eines Campus, eines darin liegenden Building, eines darin liegenden Floor und eines darin liegenden Room.

**Test Engineer:** Über die Oberfläche: 0,25 Stunden. Mit dem CIF Job, wenn Importdateien nicht vorhanden sind und erstellt werden müssen und die Delta auto-apply Konfiguration nicht vorhanden ist und erstellt werden muss: 0,75–1 Stunde.

**Interviewer:** Überprüfen, ob der Job „CIF\_AUTOMATED\_TEST\_JOB“ existiert.

**Test Engineer:** 5 Minuten.

**Interviewer:** Das Erstellen von Testdaten einer Device-Klasse, zum Beispiel Chassis, in den NMS-Tabellen (also das Einlesen der Excel-Dateien).

**Test Engineer:** Herunterladen der Datei, ausfüllen und Import: 20 Minuten.

**Interviewer:** Das Erstellen von (entsprechenden) Testdaten in Command, um zum Beispiel eine Delta-Berechnung überprüfen zu können.

**Test Engineer:** Nur mit dem Chassis, nur im Ist-Zustand: 25 Minuten.

**Interviewer:** Das Ausführen eines Jobs für eine Delta-Berechnung, nach dem die Testdaten im Delta-Report als Approved markiert werden und auf Apply gedrückt wurde. Zusätzlich soll anschließend geprüft werden, ob die Daten korrekt in Command gespeichert wurden.

**Test Engineer:** Ich gehe davon aus, dass es sich immer noch um das eine Chassis handelt. Mit dem Job machst du nur eine Delta Berechnung aber keine Synchronisation. Über deine Delta auto-apply Konfiguration hast du den Datensatz automatisch approved und klickst nur noch auf Apply. Im Anschluss Auswertung der Synchronisation. Insgesamt: 15-25 Minuten.

## **C. Die ausformulierten, im Fließtext nicht ausgeführten Anwendungsfälle**

Anwendungsfall	Ausloggen
ID	UC001.1
Beschreibung	Der Nutzer loggt sich aus Command aus.
Akteur	Benutzer
Auslösendes Ereignis	Der Benutzer befindet sich in der Oberfläche und möchte sich ausloggen.
Vorbedingungen	Command ist erreichbar und der Login war erfolgreich.
Schritte Basisszenario	1. Der Benutzer kehrt, falls noch nicht geschehen, zur Startseite zurück. 2. Der Benutzer loggt sich dort aus.
Fehlerfälle	Die Sitzung ist abgelaufen und der Benutzer kann sich nicht mehr selbst ausloggen.
Regeln	-
Automatisiert testen?	Ja, aber nicht den Fehlerfall.
Basis des CIF?	Ja.

Name Anwendungsfall	Job bearbeiten
ID	UC003
Beschreibung	Die Grundeinstellungen des Jobs werden bearbeitet.
Akteure	Benutzer, BGE
Auslösendes Ereignis	Der Akteur will eine Grundeinstellung am Job ändern.
Vorbedingungen	Command ist erreichbar und der Login war erfolgreich; das entsprechende Datenlexikon für das Source System existiert; der Job wurde erfolgreich mit dem Namen „CIF_AUTOMATED_TEST_JOB“ erstellt
Schritte Basisszenario	<ol style="list-style-type: none"> <li>1. Der Akteur öffnet die Einstellungen des Jobs und initiiert die Bearbeitung.</li> <li>2. Der Akteur ändert den Inhalt der acht benötigten Konfigurationsfelder.</li> <li>3. Der Akteur speichert seine Eingaben. Der Job ist modifiziert.</li> </ol>
Schritte 1. Alternativszenario	<ol style="list-style-type: none"> <li>3. Der Akteur ändert den Inhalt eines der nicht benötigten Konfigurationsfelder.</li> <li>4. Der Akteur speichert seine Eingaben. Der Job ist modifiziert.</li> </ol>
Schritte 2. Alternativszenario	<ol style="list-style-type: none"> <li>3. Der Akteur ändert den Inhalt eines der Additional Settings.</li> <li>4. Der Akteur speichert seine Eingaben. Die Additional Settings sind modifiziert.</li> </ol>
Schritte 3. Alternativszenario	<ol style="list-style-type: none"> <li>3. Der Akteur ändert die eingegebene Zeit in den Zeiteinstellungen.</li> <li>4. Der Akteur speichert seine Eingaben. Die Zeiteinstellung ist modifiziert.</li> </ol>
Schritte 4. Alternativszenario	<ol style="list-style-type: none"> <li>3. Der Akteur deaktiviert die Zeiteinstellungen.</li> <li>4. Der Akteur speichert seine Eingaben. Die Zeiteinstellung ist deaktiviert.</li> </ol>
Schritte 5. Alternativszenario	<ol style="list-style-type: none"> <li>3. Der Akteur dupliziert den Job.</li> <li>4. Der Akteur speichert die Eingaben. Der Job ist dupliziert.</li> </ol>
Schritte 6. Alternativszenario	<ol style="list-style-type: none"> <li>3. Der Akteur exportiert die Daten des Jobs in ein Excel-Dokument.</li> </ol>
Fehlerfälle	<ul style="list-style-type: none"> <li>• Geänderte Daten werden nicht entsprechend gespeichert.</li> <li>• Beim Duplizieren werden nicht alle wichtigen Daten mitgenommen.</li> </ul>

	<ul style="list-style-type: none"> <li>• Der Export funktioniert nicht.</li> </ul>
Regeln	Siehe UC002
Automatisiert testen?	Ja.

Name Anwendungsfall	Job löschen
ID	UC004
Beschreibung	Die Einstellungen der automatischen zeitlichen Ausführung des Jobs werden bearbeitet.
Akteure	Benutzer, BGE
Auslösendes Ereignis	Der Akteur will eine Einstellung in der Zeitkontrolle ändern.
Vorbedingungen	Command ist erreichbar und der Login war erfolgreich; das entsprechende Datenlexikon für das Source System existiert; der Job wurde erfolgreich mit dem Namen „CIF_AUTOMATED_TEST_JOB“ und einer Zeiteinstellung erstellt.
Schritte Basisszenario	1. Der Akteur löscht den Job.
Schritte 1. Alternativszenario	1. Der Akteur löscht ein oder mehrere Additional Settings.
Schritte 3. Alternativszenario	1. Der Akteur löscht die Zeiteinstellungen. 2. Der Akteur speichert seine Eingaben. Die Zeiteinstellung ist gelöscht.
Fehlerfälle	Ein Objekt kann nicht gelöscht werden.
Regeln	Siehe UC002
Automatisiert testen?	Ja

Name Anwendungsfall	Job-Überwachung öffnen
ID	UC005
Beschreibung	Die Job-Überwachung wird geöffnet und die Werte darin betrachtet.
Akteure	Ein Benutzer, die BGE.
Auslösendes Ereignis	Der Akteur öffnet den Job.
Vorbedingungen	Command ist erreichbar und der Login war erfolgreich, ein Job wurde mit Additional Settings erstellt.
Schritte Basisszenario	1. Der Nutzer öffnet die Job-Überwachung im Job-Menü. 2. Hier finden sich alle Einträge zu früheren Job-Durchläufen.
Fehlerfälle	Ein Job-Durchlauf wird nicht angezeigt.
Regeln	Es wird für jeden Job-Durchlauf angegeben, wann er lief, wie lang er lief und ob er erfolgreich war.
Automatisiert testen?	Ja.

Name Anwendungsfall	Funktionelle Logs anzeigen
ID	UC005.1
Beschreibung	Die funktionellen Logs eines Jobs werden eingesehen.
Akteure	Benutzer, BGE
Auslösendes Ereignis	Der Akteur öffnet den Job und möchte die Logs ansehen.
Vorbedingungen	Command ist erreichbar und der Login war erfolgreich; der Job wurde erstellt und wurde bereits mindestens einmal ausgeführt. Die Job-Überwachung zeigt alle Durchläufe korrekt an.
Schritte Basisszenario	<ol style="list-style-type: none"><li>1. Der Akteur öffnet die Ansicht der funktionellen Logs eines Jobdurchlaufs.</li><li>2. In der Ansicht werden Informationen und unter Umständen Warnungen oder Fehlermeldungen zu den einzelnen Schritten des Durchlaufs dargestellt.</li></ol>
Fehlerfälle	Die Logs wurden nicht korrekt eingetragen.
Regeln	Es gibt Info-Logs, Warning-Logs und Error-Logs. Die Info-Logs „Schnittstelle gestartet“ und „Schnittstelle beendet“ müssen immer vorhanden sein.
Automatisiert testen?	Ja

Name Anwendungsfall	Technische Logs herunterladen
ID	UC005.2
Beschreibung	Die technischen Logs eines Jobs werden heruntergeladen.
Akteure	Benutzer, BGE
Auslösendes Ereignis	Der Akteur öffnet den Job und möchte die Logs ansehen.
Vorbedingungen	Command ist erreichbar und der Login war erfolgreich; der Job wurde erstellt und wurde bereits mindestens einmal ausgeführt. Die Job-Überwachung zeigt alle Durchläufe korrekt an.
Schritte Basisszenario	1. Der Akteur lädt die technischen Logs über das Kontextmenü herunter.
Fehlerfälle	Logs werden nicht richtig angezeigt oder nicht alle Aktionen werden geloggt.
Regeln	-
Automatisiert testen?	Ja

Name Anwendungsfall	Deltaergebnisse manuell verifizieren
ID	UC010
Beschreibung	Einträge in der Delta-Tabelle werden verifiziert und für die Synchronisierung markiert.
Akteure	Benutzer, BGE
Auslösendes Ereignis	Die Delta-Tabelle mit den Ergebnissen der Berechnung wird betrachtet.
Vorbedingungen	Command ist erreichbar und der Login war erfolgreich; der Job wurde erstellt. Es befinden sich entsprechend Daten in den Delta-Tabellen.
Schritte Basisszenario	1. Die Daten werden als <i>Approved</i> markiert. 2. Die Markierung wird gespeichert.
Fehlerfälle	Die Markierung wird für einen Eintrag nicht oder für alle Einträge übernommen.
Regeln	-
Automatisiert testen?	Ja.



Name Anwendungsfall	Delta Auto-Apply-Konfiguration konfigurieren
ID	UC011
Beschreibung	Einträge in der Delta-Tabelle werden automatisch für die Synchronisierung markiert.
Akteure	Benutzer, BGE
Auslösendes Ereignis	Eine Auto-Apply-Konfiguration wird benötigt.
Vorbedingungen	Command ist erreichbar und der Login war erfolgreich; der Job wurde erstellt. Es befinden sich entsprechend Daten in den Delta-Tabellen.
Schritte Basisszenario	<ol style="list-style-type: none"> <li>1. Eine Auto-Apply-Konfiguration wird erstellt. Dabei wird angegeben, welche Daten genau jederzeit als <i>Approved</i> markiert werden sollen. Dies wird durch das Angeben von Sucheinschränkungen auf die Delta-Tabelle umgesetzt. So wird angegeben, welche Delta-Tabelle genutzt werden soll, und welche Attribute dieser Tabelle auf welche Werte untersucht werden sollen. Dadurch ergibt sich eine Menge an Daten.</li> <li>2. Die Daten werden als <i>Approved</i> markiert.</li> <li>3. Die Markierung wird gespeichert.</li> </ol>
Fehlerfälle	<ul style="list-style-type: none"> <li>• Die Markierung wird nicht übernommen oder für alle übernommen.</li> <li>• Die Filterfunktion funktioniert nicht ordnungsgemäß.</li> </ul>
Regeln	-
Automatisiert testen?	Ja.

## D. Ein Auszug aus der erstellten Traceability Matrix

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1																				
170	C19420491	Verify Job is not created when not all mandatory fields are filled																		
171	C19356883	Verify that job additional settings are added to job																		
172	C19420492	Verify Job is not created when it already exists																		
173																				
174	C19132175	Verify that job mandatory settings can be modified																		
175	C19423964	Verify that job not mandatory settings can be modified																		
176	C19574480	Verify that job is deleted successfully																		
177	C19420493	Verify that job additional settings can be modified																		
178	C19423965	Verify that job additional settings can be deleted																		
179	C19146851	Verify that Job can be opened																		
180	C19146855	Verify that job can be duplicated																		
181	C19146860	Verify that job can be exported to excel file																		
182	C19146864	Verify that job time control can be set																		
183	C19146859	Tools: Jobs: Release Job																		
184	C19423966	Verify that time control settings can be modified																		
185	C19721581	Verify that time control settings can be deactivated																		
186	C19423967	Verify that time control can be deleted																		
187	C19146866	Verify that technical log can be downloaded for job run																		
188	C19146881	Jobs: Monitoring: Functional Log: Errors																		
189	C19146882	Jobs: Monitoring: Functional Log: Warnings																		
190	C19423968	Verify that Infos will be logged in functional log																		
191	C18943134	Job is shown in monitoring																		
192																				
193	C18871730	Chassis: Create Type Mapping																		
194	C18943135	Chassis: Create Mapping Values for Type Mapping																		
195	C19146871	Mapping: Context menu: Delete																		
196	C19146874	Mapping: Mapping item: Values: Modify																		
197	C19146875	Mapping: Mapping item: Values: Remove																		
198	C19146867	Mapping: Search																		
199	C19146869	Mapping: Context menu: Open																		
200	C19146870	Mapping: Context menu: Duplicate																		
201	C19146872	Mapping: Context menu: Export																		
202																				
203	C19146793	Delta Report: Delta: Search																		

E. Ein von JUnit erstellter **HTML**-Testbericht

