Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Thimo Wilken

Bedeutung von automatisierten Akzeptanztests
innerhalb der Deployment Pipeline

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Thimo Wilken

Bedeutung von automatisierten Akzeptanztests innerhalb der Deployment Pipeline

Bachelorarbeit eingereicht im Rahmen Bachelorprüfung

im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Zhen Ru Dai
Zweitgutachter : Prof. Dr. Bettina Buth

Abgegeben am 23.03.2016

Thimo Wilken

Thema der Bachelorarbeit
Bedeutung von automatisierten Akzeptanztests innerhalb der Deployment Pipeline

Stichworte
Testautomation, Test-Editor, Akzeptanztests, Deployment, Deployment Pipeline, Benutzeroberflächentests, Continuous Delivery, Continuous Deployment, DevOps, Continuous Software Engineering

Kurzzusammenfassung
Ziel der Arbeit ist es die Akzeptanztests innerhalb der Deployment Pipeline zu beschreiben und exemplarisch anhand eines Kundenprojektes zu beleuchten. Dabei handelt es sich um ein Projekt der Firma akquinet für den Kunden WEAT, das sich mit der Abrechnung großer Mengen an EC-Karten Transaktionendaten beschäftigt. Für das Projekt wird mit Hilfe des Test-Editors beispielhaft ein repräsentativer Akzeptanztest automatisiert.

Thimo Wilken

Title of the paper
Importance of automated acceptance tests within the deployment pipeline

Keywords
Test automation, Test-Editor, acceptance testing, deployment, deployment pipeline, GUI tests, Continuous Delivery, Continuous Deployment, DevOps, Continuous Software Engineering

Abstract
The goal of this work is to describe acceptance tests within the deployment pipeline, generally speaking and using a specific real world example. The mentioned project is developed by akquinet for their client WEAT and deals with a high volume of ec card transaction data. For that project, with the help of the Test-Editor, a representative acceptance test will be automated.

Danksagung

Ich danke herzlich allen Mitarbeitern der akquinet AG für ihre bereitwillige und freundliche Unterstützung, insbesondere meinem Betreuer Lothar Lipinski.
Weiterhin möchte ich meinen HAW Betreuern für die Bachelorarbeit Prof. Dr. Zhen Ru Dai sowie Prof. Dr. Bettina Buth für ihre Hilfe danken.
Ein abschließendes Dankeschön gilt meinen Eltern für ihre Unterstützung sowie allen Helfern der Bachelorarbeit für ihr hilfreiches Feedback.

Inhaltsverzeichnis

1 Einleitung

In der sich wandelnden Softwarelandschaft, weg vom Wasserfallmodell und spärlich gesäten Softwareupdates, hin zu agileren Entwicklungsmethoden und häufigen Releases, wird ein schneller Entwicklungsprozess immer wichtiger.
Im Mittelpunkt dieses Entwicklungsprozesses steht die sogenannte Deployment Pipeline. Diese besteht abstrakt aus den einzelnen Schritten um die Software aus dem VCS (Version Control System) in die Hände des Nutzers zu bringen (Humble, et al., 2011).
Die Deployment Pipeline (Beispiel in Abbildung 1) ist kein immer gleicher feststehender Prozess, sondern eine auf die Entwicklung jeder Software angepasste Folge von Schritten, mit dem jedoch immer präsenten Ziel die Software zu veröffentlichen, dem Kunden zu Verfügung zu stellen oder anderweitig in einer Produktionsumgebung einzusetzen.

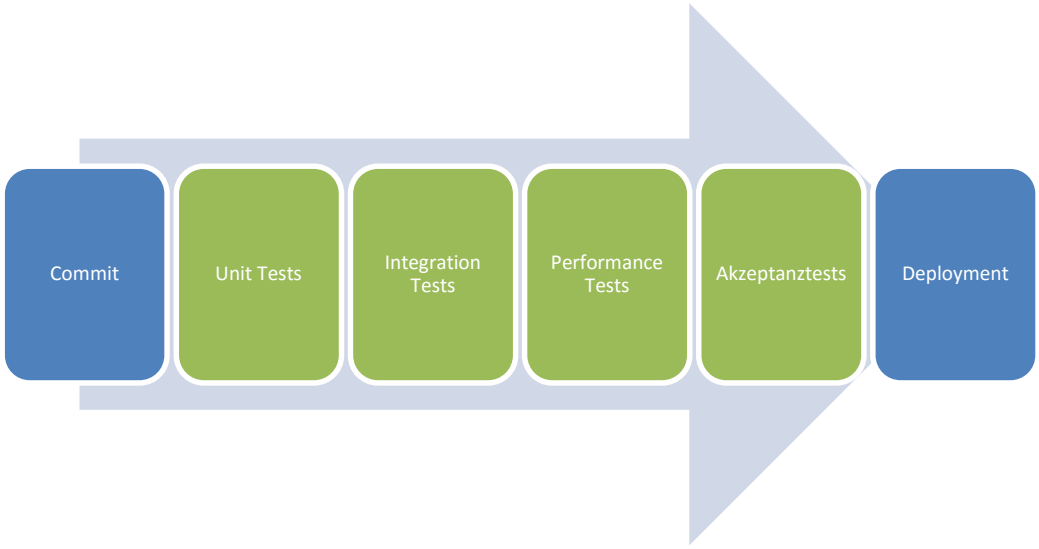

Abbildung 1 Beispiel einer Deployment Pipeline Der Pfeil stellt den zeitlichen Ablauf dar. Die einzelnen Blöcke repräsentieren Phasen. Die einzelnen Phasen sind zum jetzigen Zeitpunkt nicht relevant.

Ein wichtiger Bestandteil der Pipeline sind die Testphasen. Jede Phase stärkt die Zuversicht in die Fehlerfreiheit der Software (Humble, et al., 2011). Üblicherweise brauchen die späteren Phasen mehr Zeit, prüfen dafür aber in einer produktionsnäheren Umgebung oder einer höheren Abstraktionsschicht der Software.
Eine dieser Testphasen ist die Phase der Akzeptanztests. Ein solcher Test verifiziert, dass eine entwickelte Software auch tatsächlich eine vom Kunden formulierte Anforderung erfüllt (Wolff, 2015). Diese Tests befinden sich, im Gegensatz zu etwa Unit- oder Integrationstests, auf einer besonders hohen Abstraktionsebene. Es wird meist eine sehr produktionsnahe Umgebung oder Zusammenstellung von Komponenten getestet, was diese Art von Tests sehr anfällig und fragil, aber dafür sehr aussagekräftig im Hinblick auf das Funktionieren des gesamten Systems macht.
Die Vergleichsweise hohe Komplexität der Tests führt häufig dazu, dass diese in vielen Projekten nicht durchgeführt oder gar nicht erst spezifiziert werden.
Für den flexiblen Deployment Prozess der agilen Softwareentwicklung ist es unabdingbar, dass möglichst viele Schritte der Deployment Pipeline, inklusive der Akzeptanztests, automatisiert werden. Die Automatisierung von Tests macht nicht nur die Ausführung der Tests effizienter, sondern ermöglicht es auch die Tests häufiger durchzuführen. Somit steht benötigtes Feedback schneller zur Verfügung (Wolff, 2015).

1.1 Ziel der Arbeit

In dieser Arbeit wird das Konzept der Deployment Pipeline beleuchtet. Spezielles Augenmerk wird dabei auf die Akzeptanztests innerhalb der Pipeline gerichtet. Als Fallstudie dient im Anschluss das WEAT Entgelt-Abrechnungssystem (kurz WEAT EABR), ein Projekt, dass die akquinet AG (akq15) für den Kunden WEAT (WEA15) entwickelt. Die WEAT EABR Deployment Pipeline wird kurz beleuchtet, wieder mit speziellem Augenmerk auf die Phase der Akzeptanztests.
Der praktische Teil der Arbeit besteht darin exemplarisch einen repräsentativen Akzeptanztest des Projektes zu automatisieren. Dies geschieht mit Hilfe des Test-Editors, einem Programm das unter anderem zum Testen von Nutzeroberflächen verwendet werden kann. Der Test-Editor ist ein Open Source Projekt, das unter anderen von der akquinet AG und der SIGNAL IDUNA (SIG15) entwickelt wird.
Um exemplarisch zu zeigen wie die Akzeptanztestausführung in die Deployment Pipeline des WEAT EABR Projekts integriert werden kann, ist es ebenfalls Ziel den automatisierten Test aus einem lokalen Jenkins Server starten zu können. Jenkins (Jen15) ist ein Continuous Integration und Delivery Server, der es ermöglicht die einzelnen Phasen des Software Delivery Prozesses zu automatisieren.

1.2 Kapitelübersicht

Im Folgenden ein kurzer Überblick über die Inhalte der einzelnen Kapitel:

Deployment Pipeline
In diesem Kapitel wird auf das Konzept und den Nutzen der Deployment Pipeline eingegangen. Dazu wird der übliche Aufbau einer Deployment Pipeline mit ihren einzelnen Phasen beschrieben. Im Anschluss werden in aller Kürze Tools zur Implementierung einer Deployment Pipeline behandelt.

Akzeptanztests
Hier findet sich eine spezielle Betrachtung der Akzeptanztestphase innerhalb der Deployment Pipeline. Dazu werden Akzeptanztests im Allgemeinen beschrieben und nachfolgend wird der Blick auf verschiedene Arten und Implementierungsmöglichkeiten von Akzeptanztests gelegt.

Fallstudie
Dieses Kapitel enthält eine genauere Vorstellung des WEAT EABR Projekts. Dabei wird kurz die Deployment Pipeline die zur Entwicklung des Projekts verwendet wird betrachtet. Spezielles Augenmerk liegt dabei auf der Akzeptanztestphase. Abschließend wird der sogenannte Rechnungslauftest als repräsentatives Beispiel für einen zu implementierenden Akzeptanztest vorgestellt und der Test-Editor als verwendetes Tool betrachtet.

Implementierung
Hier angekommen wird der Prozess der Automatisierung des Rechnungslauftests mit Hilfe des Test-Editors beschrieben. Dazu wird anfangs eine Übersicht über die beteiligten Komponenten sowie deren Zusammenspiel gegeben. Ebenso werden Testschritte des Tests zu Gruppen ähnlicher Schritte zusammengefasst. Daraufhin wird kurz die Vorgehensweise bei der Implementierung des Tests erläutert und nachfolgend werden im Detail die Implementierungen der einzelnen Testschrittgruppen erläutert. Abschließend wird gezeigt wie der Test mit Hilfe von Jenkins gestartet und ausgewertet werden kann.

Related work
Losgelöst vom praktischen Anteil dieser Arbeit wird hier allgemein auf verwandte und weiterführende Konzepte der Deployment Pipeline eingegangen. Hierbei wird weit ausgeholt um einen besseren Überblick über die Position der Deployment Pipeline im Entwicklungsprozess zu bekommen und die Folgen einer Optimierung der Pipeline, sowie die dafür nötigen Änderungen anzureißen.

Fazit und Ausblick
Dieser Abschnitt enthält eine kurze Zusammenfassung der Arbeit sowie Bewertungen und einige direkt auf dem praktischen Teil aufsetzende weiterführende Ideen.

2 Deployment Pipeline

In der Einleitung schon kurz beschrieben, wird in diesem Kapitel die Deployment Pipeline genauer betrachtet. Das Ziel der Deployment Pipeline ist es den Prozess der Software Entwicklung auf sinnvolle Art und Weise zu strukturieren um dem Kunden auf möglichst verlässliche Art und Weise flexibel gut getestete Software ausliefern zu können. Dieses Ziel ist in Abbildung 2 dargestellt.
Eine der wichtigsten Eigenschaften bei der Softwareentwicklung ist schnelles Feedback. Checkt der Entwickler seinen Source Code in das VCS ein, möchte er schnell wissen ob seine Änderungen neue Fehler hervorgerufen haben. Gründliche Tests nehmen jedoch viel Zeit in Anspruch und der Entwickler möchte nicht stundenlang warten nur um dann zu erfahren, dass er einen sehr offensichtlichen Fehler gemacht hat. Die Deployment Pipeline ist eine Möglichkeit mit dieser Sachlage umzugehen, indem man Build und Tests in mehrere Phasen unterteilt und somit schneller Feedback erhält (Humble, et al., 2011).

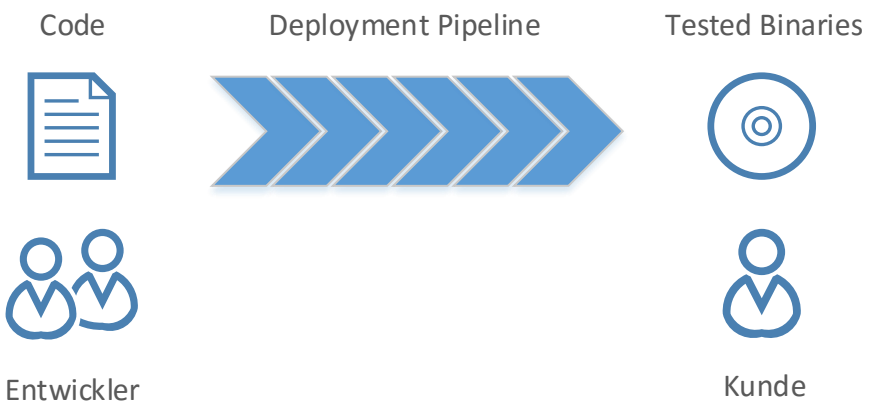

Abbildung 2 Das Ziel der Deployment Pipeline.

Frühere Testphasen finden die meisten grundlegenden Probleme und geben schnelle Rückmeldung. Spätere Phasen hingegen prüfen gründlicher oder auf einem höheren Abstraktionsniveau. Diese brauchen jedoch gängigerweise länger und geben damit erst viel später Rückmeldung an die Entwickler (Humble, et al., 2011). Häufig prüfen die späteren Phasen die Software auch in immer produktionsnäheren Umgebungen, mit immer mehr interagierenden Komponenten (z.B. Datenbanken, Netzwerken oder Dateisystemen). Zu sehen ist dieser für die Pipeline essenzielle Trade-Off in Abbildung 3.

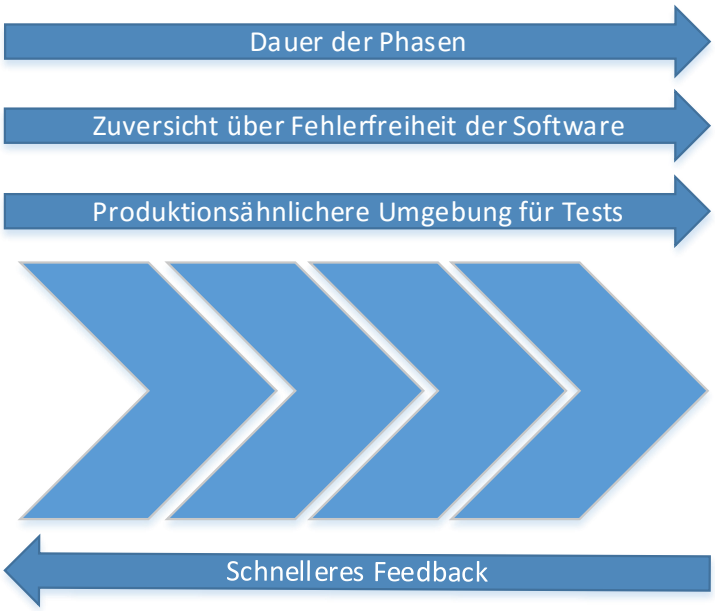

Abbildung 3 Vergleich von Eigenschaften späterer zu früheren Phasen. Der nicht beschriftete Mittelteil stellt abstrakt die Phasen einer Deployment Pipeline dar. Die Pfeile benennen Eigenschaften die bei späteren (nach rechts gerichteter Pfeil) oder bei früheren (nach links gerichteter Pfeil) Phasen tendenziell ausgeprägter sind. Die Abbildung ist an eine Darstellung aus (Humble, et al., 2011) angelehnt.

Die zeitliche Reihenfolge der Phasen sollte nach Möglichkeit nicht unbedingt nach Art der Tests (Performanz, Akzeptanz etc.) gestaltet werden, sondern nach der Wahrscheinlichkeit einen Fehler zu finden (Wolff, 2015). So sollte tendenziell erst ist in die Breite und dann in die Tiefe getestet werden. Erst schnelle Tests und dann langsame. Erst die Basisfunktionen und dann komplexe Szenarien (Humble, et al., 2011).
Der Kern der Deployment Pipeline ist so schnell wie möglich nützliches Feedback zu geben. Veranschaulicht ist das in Abbildung 4. Auf der vertikalen Achse ist die Zeit und auf der horizontalen Achse sind die Pipeline Phasen dargestellt. Dabei ist gezeigt wie der Entwickler oder Tester die Pipeline anstößt indem er den Code in das VCS eincheckt. Der Build der Applikation wird dann automatisch gestartet und im Anschluss werden einige Unit Tests durchgeführt. Tritt in dieser Phase ein Fehler auf, wird das dem Entwickler gemeldet. Beispielsweise in Form einer E-Mail oder einer Statusanzeige oder ähnlichem. Läuft die Phase erfolgreich ab, wird (in diesem Fall) automatisch die nächste Phase gestartet.
Die Phasen müssen nicht zwangsweise automatisch gestartet werden, siehe beispielsweise die Release Phase in der Abbildung 4. So kann es beispielsweise auch sein, dass die Ergebnisse der vorangegangenen Phase von einem Menschen begutachtet werden müssen um zu entscheiden ob die Phase erfolgreich war oder nicht (Humble, et al., 2011), bevor die nächste Phase gestartet wird. Ebenso muss auch die Durchführung nicht automatisiert ablaufen, wie hier die Phase der manuellen Tests veranschaulichen soll.

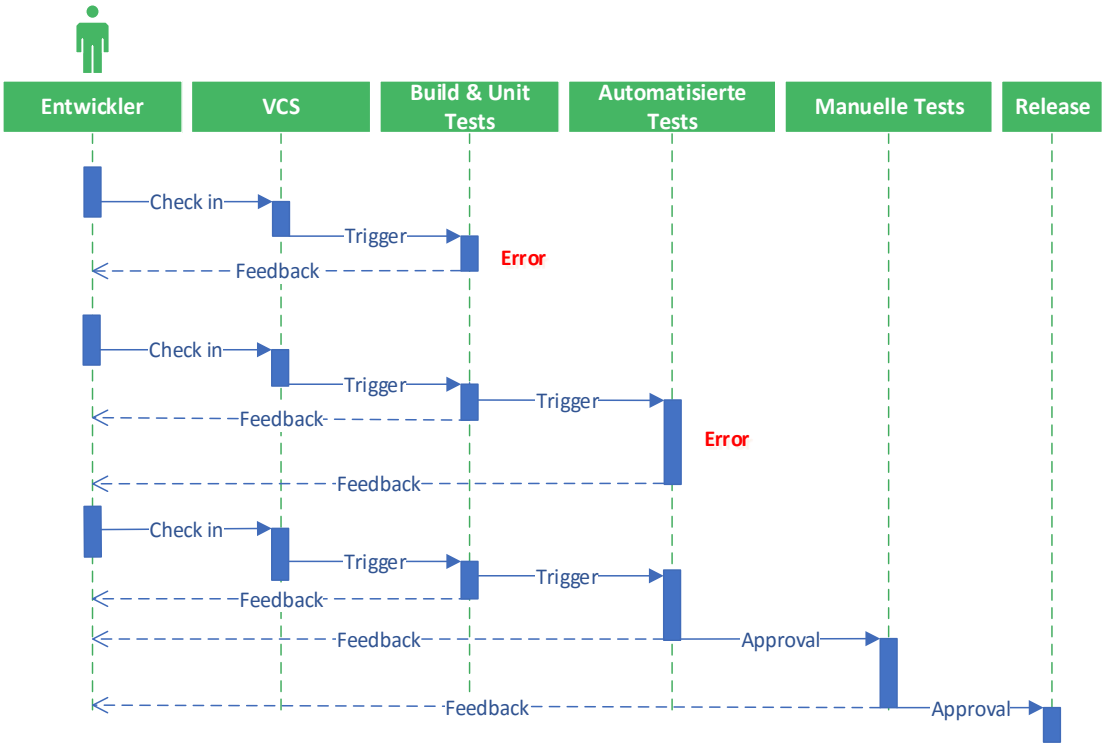

Abbildung 4 Sequenzdiagramm einer Beispiel Deployment Pipeline. Dargestellt ist wie die einzelnen Phasen getriggert werden und Feedback geben. Die Abbildung ist an eine Darstellung aus (Humble, et al., 2011) angelehnt.

Es muss sich bei den einzelnen Phasen nicht zwangsläufig um Testphasen handeln. Weiterhin sind, wie erwähnt, sowohl manuell als auch automatisiert ausgeführte Phasen möglich und gängig. Um schnell Feedback zu erhalten, ist es jedoch wünschenswert, dass so viele Phasen wie möglich automatisiert ablaufen. Dabei gibt es Ausnahmen, so ist etwa das explorative Testen, bei dem der Tester die Anwendung relativ frei erforscht um etwaige Fehler aufzuspüren, eine eher für den Menschen prädestinierte Aufgabe, die nur schwer zu automatisieren ist.
Die tatsächliche Implementierung einer Deployment Pipeline ist meist mit Hilfe eines Tools wie Jenkins oder GoCD realisiert. Solche Tools fallen unter Begriffe wie Continuous Integration Server, Release Automation Tool oder Build Automation Tool. Auf diese wird im Abschnitt 2.2 kurz eingegangen, nachdem im Folgenden der gängige Aufbau einer Deployment Pipeline beschrieben wird.

2.1 Üblicher Aufbau der Deployment Pipeline

Es folgt eine kurze Beschreibung des gängigen Aufbaus einer Deployment Pipeline (siehe Abbildung 5). Die hier vorgestellten Phasen und Gruppen von Phasen decken keineswegs alle Möglichkeiten einer Deployment Pipeline ab und sind sehr allgemein gehalten, es wird jedoch die Essenz der Deployment Pipeline beschrieben. Für ein konkretes Beispiel sei auf das Fallbeispiel in Abschnitt 4.1 verwiesen.

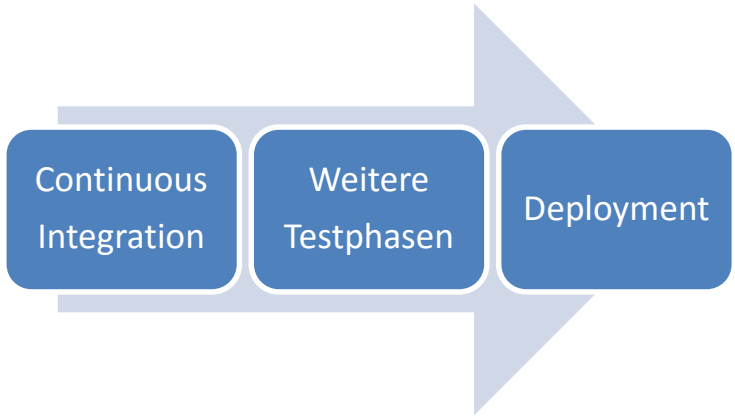

Abbildung 5 Abstrakter allgemeiner Aufbau einer Deployment Pipeline.

2.1.1 Continuous Integration

Continuous Integration beschreibt das fortlaufende Zusammenführen von Software Komponenten um Integrationsprobleme zu vermeiden (Fowler, 2006). Dies ist meist der erste Schritt der Pipeline. Häufig wird mehrmals täglich integriert. Diese Integration findet meist auf einem Continuous Integration (CI) Server statt, auf dem der Code aller Beteiligten integriert wird. Abbildung 6 zeigt die Rolle des CI Servers.
Alternativ ließe sich diese Phase in die drei Phasen Build, Unittests und Integrationstests unterteilen und auf ähnliche Art und Weise beschreiben. Eine solche Darstellung ist jedoch eine bereits zu konkrete Implementierung einer Deployment Pipeline, weshalb Continuous Integration als erste Phase beschrieben wird.

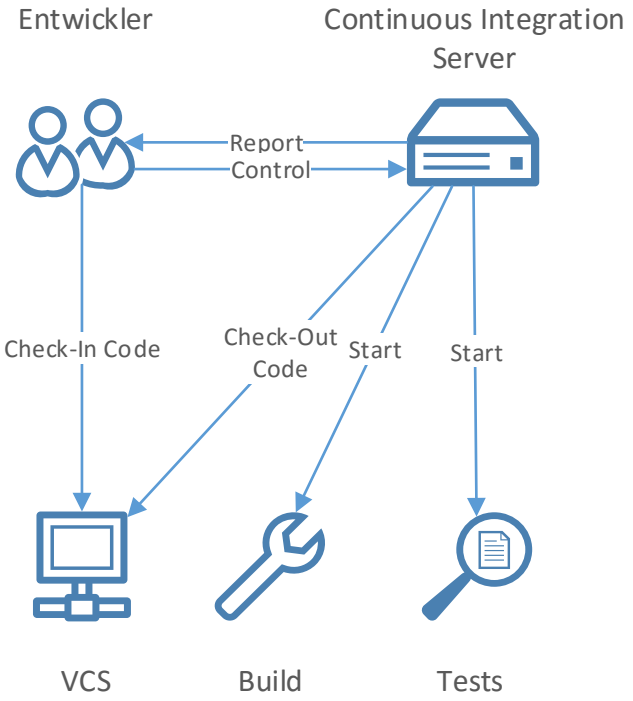

Abbildung 6 Rolle des Continuous Integration Servers. Der CI Server dient als zentrale Stelle zum Erstellen und Testen der Software aus dem VCS. Dabei kann er beliebig konfigurierte werden und gibt Feedback über Build und Tests an die Entwickler.

Stelle man sich ein Szenario, ähnlich dem in der Kapiteleinleitung beschriebenen, vor. Ein Entwickler ändert einen Teil des Source Codes, startet lokal den Build der Software, lässt lokal einige Tests laufen, integriert gegebenenfalls Änderungen anderer Entwickler aus dem zentralen VCS und checkt seine Änderungen bei Erfolg anschließend in das zentrale VCS ein. Nun will er möglichst schnell wissen, ob seinen Änderungen auf dem Server neue Fehler hervorrufen oder gegen anderweitige Projektrichtlinien (z.B. Coding Conventions) verstößt. Er möchte also, dass in kurzer Zeit die Grundfunktionalität geprüft wird und er Rückmeldung über das Ergebnis der Prüfung erhält.
Hierbei sei erwähnt, dass häufig die lokal von Entwickler ausgeführten Tests denen, nach dem Check-In, auf dem zentralen Server laufenden Tests entsprechen. Dies hat zum einen den Grund, dass falls die Tests lokal fehlschlagen, der fehlerhafte Code gar nicht erst in das zentrale VCS und damit an andere Entwickler gelangt oder den Entwicklungsprozess aufhält. Zum anderen bedeutet ein erfolgreicher Testdurchlauf auf der lokalen Entwicklermaschine nicht automatisch, dass der Code auf dem, häufig produktionsähnlicheren, Continuous Integration Server ebenso erfolgreich getestet wird.
Laufen die Tests lokal erfolgreich durch wird nach dem Check-In in das VCS der Code auf dem CI Server automatisch kompiliert, eine Suite von automatisierten Tests durchlaufen, der Code auf die Nutzung der richtigen Konventionen analysiert und gegebenenfalls die resultierenden Binaries für spätere Phasen gespeichert.

Die hier durchlaufenen Tests sind zum Großteil Unit- und Integrationstests. Unittests prüfen dabei isoliert die kleinsten Softwarebaustaue oder Komponenten, wie z.B. Klassen und ihre Methoden, auf ihre Richtigkeit (Spillner, et al., 2012). Integrationstests hingegen prüfen das Zusammenspiel der einzelnen Komponenten untereinander (Spillner, et al., 2012).
Andere Arten von Tests können jedoch ebenso ausgeführt werden. Es geht im Kern darum in dieser frühen Phase Tests auszuführen, die schnell ablaufen. Der Entwickler sollte nicht mehr als zehn Minuten auf seine Rückmeldung warten, ein guter Richtwert sind fünf Minuten (Humble, et al., 2011). Andere Quellen empfehlen zwei bis zehn Minuten, jedoch gibt es nicht ausreichend Untersuchungen dies zu begründen (Laukkanen, et al., 2015). Allerdings beeinflusst die Dauer wie häufig Entwickler integrieren. Eine kurze Dauer unterbricht nicht den Entwicklungsfluss und bedarf keiner separaten Berücksichtigung, während eine etwas längere Dauer eine Unterbrechung darstellt und Entwickler dazu verleitet nur in speziellen Zeitslots zu integrieren (Laukkanen, et al., 2015).
Ebenso können aber auch einzelne, längere Tests durchlaufen werden, wenn das Team es für sinnvoll erachtet. So kann beispielsweise ein Smoketest, der die Anwendung startet und einige Basisfunktionen testet frühzeitig Alarm schlagen, falls die Anwendung z.B. gar nicht erfolgreich gestartet werden kann (Humble, et al., 2011).
Diese Commit Test Suite, die nach dem Check-In ausgeführt wird, sollte kontinuierlich verbessert werden, mit dem Ziel schnell möglichst viele Fehler zu finden, sodass die Entwickler mit einem entsprechend kurzen Iterationszyklus arbeiten können, der jedoch viele wichtige Fehlerprüfungen abdeckt.
Der essenzielle Ablauf von Continuous Integration ist in Abbildung 7 dargestellt.

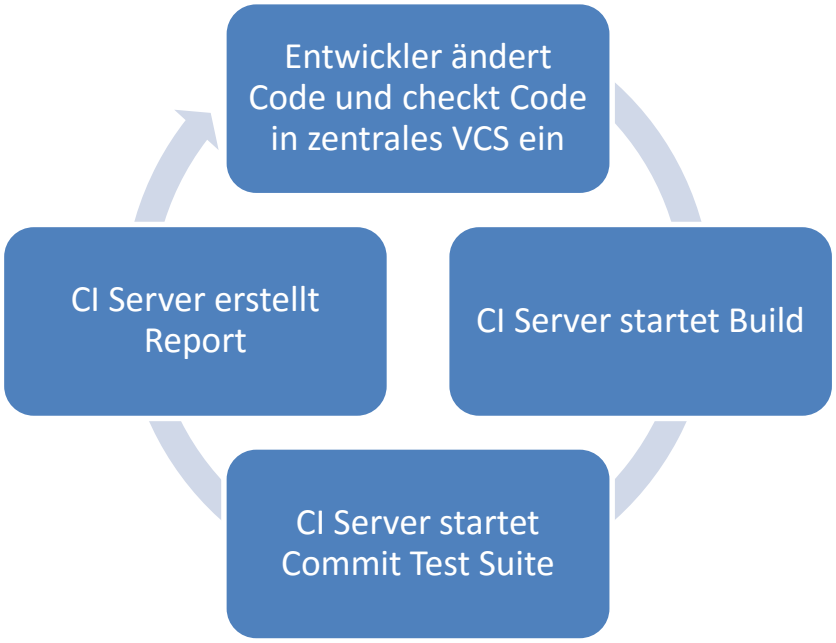

Abbildung 7 Grundlegender zyklischer Ablauf von Continuous Integration.

2.1.2 Weitere Testphasen

Hat ein Projekt viele oder langdauernde Tests, ist es nicht sinnvoll alle auf einen Schlag nach dem Check-In auszuführen, da der Entwickler viel zu lang auf eine Rückmeldung warten muss. Deshalb gliedert man bestimmte Gruppen von Tests in einzelne Phasen aus, welche nach der in Abschnitt 2.1.1 beschriebenen Commit Test Suite ausgeführt werden. Hierbei handelt es sich also um eine Erweiterung von Continuous Integration.
Die Tests der Commit Phase haben eine Sonderstellung, weil sie als erste schnelle Abwehr gegen neueingeführte Fehler dienen. Jetzt laufen anschließend alle Testphasen die mehr Zeit in Anspruch nehmen. Hierbei handelt es sich meist um verschiedene Arten von Akzeptanztests auf die in Kapitel 3 näher eingegangen wird.
Dabei wird es sich zum größten Teil um automatisierte Tests handeln, es sind jedoch auch manuelle Testphasen gängig. Diese werden jedoch meist nur durchgeführt nachdem alle automatisierten Tests erfolgreich durchgelaufen sind (Humble, et al., 2011).

2.1.3 Deployment

Hier angekommen, ist jetzt grob Folgendes über die Software bekannt (Humble, et al., 2011):

- Der Code lässt sich kompilieren.
- Der Code tut was die Entwickler denken, dass er tut, da er die Unit- und Integrationstests erfolgreich durchlaufen hat.
- Die Software tut was der Kunde erwartet, da die Akzeptanztests erfolgreich abgeschlossen wurden.

Die Software kann also guten Gewissens an den Endnutzer weitergegeben werden. Dieses bezeichnet man als Deployment oder Release. Dabei kann die Software beispielsweise Nutzern auf einer Webseite zum Download angeboten werden, oder aber etwa in die Produktionsumgebung des Kunden eingespeist werden.
Auch hier ist es sinnvoll diesen Schritt so weit wie möglich zu automatisieren.

2.2 Tools

Um das Erstellen und Verwalten einer Deployment Pipeline zu erleichtern existieren verschiedene Tools die meist unter Namen wie Continuous Integration Server, Release Management System, Continuous Delivery Server oder Continuous Deployment Server zu finden sind. Diese Tools bieten Möglichkeiten die Phasen einer Pipeline und deren Abhängigkeiten voneinander zu implementieren und den kompletten Ablauf zu visualisieren.

Hier werden beispielhaft zwei Tools in aller Kürze vorgestellt. Für eine genauere Beschreibung sei auf deren Webpräsenzen verwiesen. Weitere, hier nur erwähnte Tools sind TeamCity (Tea16), Bamboo (Bam16), Pulse (Pul16) und ElectricFlow (Ele16).

Jenkins
Jenkins ist ein Open Source Continuous Integration Server (Jen15), der aus dem Hudson Projekt hervorgegangen ist. Jenkins ist in erster Linie Continuous Integration Server. Durch seine weite Verbreitung existieren jedoch viele Plugins die auch das Verwalten einer kompletten Deployment Pipeline erleichtern.

Go
Go, um Verwechslungen mit der Programmiersprache Go (Gol15) zu vermeiden, meist GoCD (Go Continuous Delivery) genannt ist ein Continuous Delivery Server. Anders als beispielsweise Jenkins oder andere Continuous Integration Server, wurde GoCD explizit mit dem Gedanken der Deployment Pipeline entwickelt (GoC15). GoCD ist ebenfalls seit einiger Zeit Open Source und wird von ThoughtWorks (Tho15) entwickelt.

3 Akzeptanztests

Ein entscheidender Bestandteil der Deployment Pipeline sind die Akzeptanztests. Das Ziel der Akzeptanztests ist es, dass der Kunde die Software abnimmt, also “akzeptiert”. Es ist also eine explizit gewünschte Eigenschaft der Tests, dass sie auch für den Kunden verständlich verfasst werden (Wolff, 2015). Die Akzeptanztests beschreiben und testen die Anforderungen die der Kunde an die Software hat.
Auch andere Arten von Tests, wie beispielsweise Unittests tragen dazu bei, dass der Kunde die gewünschte Software bekommt. Jedoch sagen Unittests eher aus: „Die einzelnen Bausteine der Software tun das, was der Entwickler erwartet, was sie tun“. Akzeptanztests hingegen beschreiben explizit Abläufe und Kriterien die der Kunde fordert. Dies macht Akzeptanztests zu einer wichtigen Komponente der Deployment Pipeline.
Häufig später in der Pipeline angesiedelt verwenden Akzeptanztests meist ein relativ produktionsnahes Setting. Durch das Zusammenspiel von vielen komplexen Komponenten und dem expliziten Ziel die Anforderungen zu prüfen, werden Akzeptanztests eher als Black-Box-Tests betrachtet.
Dabei können die Tests sowohl manuell also auch automatisiert durchgeführt werden. In einigen Fällen mögen sie auch vereinzelt in der Commit Test Suite vorkommen um etwa zu gewährleisten, dass die Applikation sich überhaupt starten lässt (Smoke Test).
Im ganzen verifiziert eine Akzeptanztestsuite, dass die vorliegende Software dem Kunden den erwarteten Mehrwert bringt (Humble, et al., 2011).

3.1 Akzeptanztests vs. funktionale Tests

Akzeptanztests und funktionale Tests werden häufig gleichgesetzt, sind jedoch nicht dasselbe. Ein Akzeptanztest verifiziert die Akzeptanzkriterien einer Anforderung oder einer User Story. Diese können sowohl funktional aus auch nichtfunktional sein. Nichtfunktionale Akzeptanzkriterien sind beispielsweise: Capacity, Performance, Modifiability, Availability, Security, Usability etc. Ein funktionales Kriterium legt fest, was die Software explizit tun soll, z.B. soll die Software eine Liste aller Nutzer anzeigen können oder etwa die Möglichkeit bieten einen Nutzer aus dem System löschen zu können.
Die Haupteigenschaft eines Akzeptanztests ist hierbei, dass er bei erfolgreichem Durchlauf demonstriert, dass eine gewisse User Story oder eine spezielle Anforderung erfüllt ist (Humble, et al., 2011), funktional oder nichtfunktional.

3.2 Durchführungsarten von Akzeptanztests

Es existieren verschiedene Möglichkeiten die Akzeptanzkriterien einer Software als Tests zu implementieren. Im Folgenden werden diese kurz beschrieben. Ebenfalls werden generelle Vor- und Nachteile der drei Vorgehensweisen genannt. Dabei sind diese Vor- und Nachteil nicht absolut und immer gültig, sie sollten hier lediglich der Orientierung dienen. Alle drei Arten der Testimplementierung haben ihre Berechtigung und eine Kombination der Testarten ist sinnvoll.

3.2.1 Manuelles Testen

Das manuelle Testen beschreibt das Testen durch einen Menschen. Beispielsweise eine Person die sich durch eine Webanwendung klickt und vorher definierte Schritte ausführt oder aber die Prüfung weniger konkreter Eigenschaften der Anwendung wie etwa Usability oder das Look & Feel auf verschiedenen Plattformen.

Vorteile:

- Es besteht kein Initialer Aufwand für die Automatisierung von Tests, da überhaupt nichts automatisiert wird. Es kann sofort mit dem Testen begonnen werden.
- Auch vorher nicht bedachte Probleme können aufgedeckt werden. Speziell bei explorativem Testen. Probleme die den aktuellen Test nicht betreffen, können bei manuellen Tests zufällig entdeckt werden (z.B. Fehler in GUI Layout Proportionen).
- Eher schwer automatisierbare Kriterien wie Usability und Look & Feel einer Applikation können getestet werden.

Nachteile:

- Die eigentliche Testdurchführung nimmt viel Zeit in Anspruch. Die benötigte Zeit hängt stark vom durchführenden Tester ab.
- Im Vergleich zu automatisierten Tests ist das manuelle Testen, abgesehen von den ersten paar Durchläufen teurer, da sich die Automatisierungskosten nach einiger Zeit amortisieren (Humble, et al., 2011).
- Der durchführende Tester benötigt oft Know-how.
- Sich wiederholende Regressionstests sind meist sehr uninteressant in der Durchführung für den Tester. Ohne Automation müssen die, für diesen Fall oft überqualifizierten, Tester wiederholt monotone Arbeit durchführen.
- Die bei einem Testdurchlauf entstandenen Testergebnisse sind teilweise schwer reproduzierbar aufgrund von menschlichen Fehlern (Wolff, 2015).
- Die meisten Tests müssen vor jedem Release durchgeführt werden um die Software möglichst zuversichtlich veröffentlichen zu können. Auf Grund des teilweise hohen manuellen Testaufwands wird die Software in einigen Fällen weniger häufig

veröffentlicht (Humble, et al., 2011). Die manuellen Teste können also eine Veröffentlichungsbarriere bilden.
- Die Tests werden meist kurz vor dem Release durchgeführt, einer relativ hektischen Zeit in der das Team unter vergleichsweise hohem Druck steht. Als Folge davon wird häufig zu wenig Zeit eingeplant um die beim Testen gefundene Probleme noch rechtzeitig vorm Release zu beheben (Humble, et al., 2011).
- Werden kurz vor dem Release komplexe Probleme gefunden, besteht die Möglichkeit, dass durch die gemachten Änderungen zur Fehlerbehebung kurz vor dem Release weitere Regressionsprobleme entstehen.
- Manuelle Tests lassen sich schlecht skalieren. Wächst die Anzahl oder die Häufigkeit der Durchführung der Tests, wächst in gleichem Maße ebenso der Aufwand für die manuelle Durchführung.

3.2.2 Automatisiertes Testen über eine API

Eine weitere Möglichkeit Akzeptanzkriterien zu überprüfen ist das Testen über eine API (Application Programming Interface). Dabei ist zwischen der hier gemeinten High-level-Geschäftslogik-API und der Low-level-API, die für Unittests verwendet wird zu unterscheiden. Die hier beschriebene High-level-Geschäftslogik-API könnte beispielsweise auch von der GUI verwendet werden.
Es geht nicht darum extra für die Tests eine API bereitzustellen, dies wiederspräche dem Ziel das zu testen, was der Nutzer später auch (indirekt) wirklich verwendet. Zusätzlich geht es nicht darum Zusatzfunktionalitäten speziell für die Tests bereitzustellen. Es sollte im Gegenteil der letzte Ausweg sein, Funktionalität speziell für die Tests in die API aufzunehmen (Humble, et al., 2011).

Vorteile:
- Es wird nicht für jeden Testdurchlauf ein Tester zur Durchführung benötigt.
- Die Tests können deutlich schneller durchgeführt werden als manuelle Tests.
- Die Tests können schneller durchgeführt werden als Tests, die die Applikation über die GUI ansprechen.
- Änderungen am GUI Code führen nicht zu fehlerhaften Testdurchläufen, da die GUI nicht mitgetestet wird.

Nachteile:
- Eine entsprechend nutzbare vollständige, gut strukturierte und gut testbare API ist nicht immer verfügbar.
- Die GUI wird in keiner Weise getestet.
- Kriterien wie beispielsweise Usability können im Gegensatz zu manuellen Tests nicht vernünftig getestet werden.

3.2.3 Automatisiertes Testen über eine GUI

Als dritte Möglichkeit bleibt das automatisierte Ausführen der Tests über eine GUI. Hierbei wird mit Hilfe eines entsprechenden GUI Treibers, wie etwa Selenium (Sel15) für Weboberflächen, direkt die Interaktion mit der Oberfläche automatisiert.

Vorteile:

- Es wird nicht für jeden Testdurchlauf die Zeit eines Testers benötigt.
- Die Tests sind näher am realistischen Anwendungsfall als API Tests, da der Test noch näher an der Sicht des Nutzers operiert.
- Im Gegensatz zum Testen über eine API können auch GUI Layout Probleme oder Browserinkompatibilitäten getestet werden (Wolff, 2015).
- Die Tests laufen schneller durch als manuelle Tests.

Nachteile:

- GUI Tests sind langsam im Vergleich zu API Tests (Wolff, 2015).
- GUI Tests sind teilweise sehr fehleranfällig bei kleinen GUI Änderungen (Wolff, 2015). Dadurch kann es zu einem hohen Wartungsaufwand kommen.
- Die benötigte Testumgebung ist oft sehr komplex, da sehr produktionsnah (Wolff, 2015).
- Kriterien wie beispielsweise Usability können im Gegensatz zu manuellen Tests nur schwer vernünftig getestet werden.

3.2.4 Vergleich der Durchführungsarten

Automatisierte Akzeptanztests ersetzten nicht unbedingt das manuelle Testen. Es befreit den Tester eher von Routinetests der Grundfunktionalitäten (Wolff, 2015). Die Frage ist also nicht unbedingt ob das automatisierte Testen das manuelle Testen ablösen soll. Es geht eher darum wichtige Funktionalität regelmäßig automatisiert zu testen. Der initial höhere Aufwand für automatisierte Tests zahlt sich nach einiger Zeit aus. Kommen mehr Funktionen zur Software hinzu, müssen auch mehr Tests durchgeführt werden, zusätzlich müssen aber auch alle alten Tests durchgeführt werden um Regression zu vermeiden. Der manuelle Aufwand hierfür ist extrem hoch. Die Automation ermöglicht es einmal automatisierte Tests immer wieder auszuführen und ist somit, abgesehen vom initialen Aufwand, weniger kostenintensiv.
Das manuelle Testen hat weiterhin seinen Platz in der Deployment Pipeline. Sind sich wiederholende Regressionstests automatisiert, und der Tester somit davon befreit Routineaufgaben abzuarbeiten, kann er sich auf die Arbeiten konzentrieren die dann zusätzlich zu den automatisierten Tests einen hohen Mehrwert bringen (Humble, et al., 2011). Dazu zählen beispielsweise das explorative Testen, Usability Tests, die Testplanung, Produktvorführungen und das Warten der automatisierten Tests.

Die Frage ob über eine GUI oder eine API getestet werden soll, ist sehr projektspezifisch. Häufig steht eine entsprechende API nicht zur Verfügung, oder stellt nicht ausreichend Funktionalität bereit um damit vernünftig zu Testen. Die fehlende GUI verringert jedoch die Komplexität enorm und beschleunigt die Testausführung beträchtlich. Sollte es also nötig sein, sehr viele ähnliche Tests auszuführen oder schnelleres Feedback zu geben, so ist die API eine gute Wahl. Allgemein gilt, dass am ehesten eine Mischung aller Varianten im richtigen Verhältnis den größten Nutzen bringt.
Für den weiteren Verlauf der Arbeit, also das Fallbeispiel in Kapitel 4 und die exemplarische Implementierung eines Akzeptanztests in Kapitel 5 ist die Variante mit der Steuerung über die GUI vorgegeben.

4 Fallstudie

Die akquinet AG entwickelt fortlaufend für die Firma WEAT eine Software, genannt WEAT Entgelt-Abrechnungssystem, kurz WEAT EABR. Aufgabe der Software ist es die Monatsabrechnung von EC-Karten Transaktionen an Tankstellen zu verwalten. Die Software muss dabei mit einer sehr großen Menge an Transaktionen umgehen können.
Das System verfügt über ein Web-Interface (siehe Abbildung 8) und berechnet die EC-Entgelte nach den individuellen Vereinbarungen zwischen Händlern (Tankstellen) und Banken, beide Parteien werden über sogenannte Konzentratoren gebündelt.

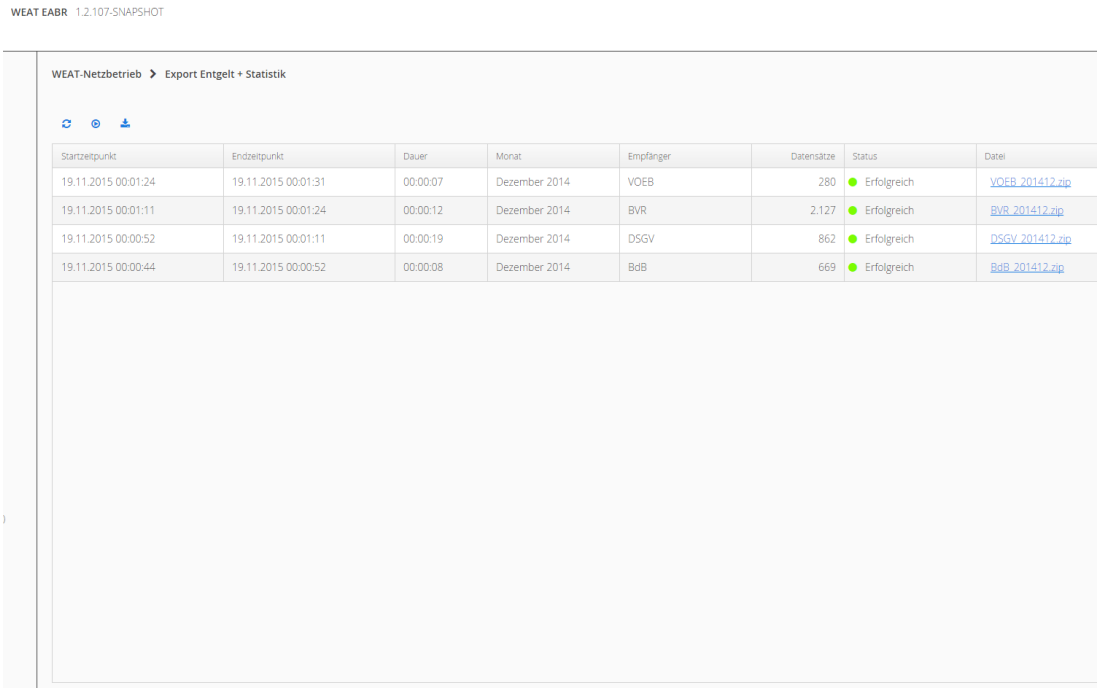

Abbildung 8 Die WEAT EABR Weboberfläche. Die linke Seite zeigt normalerweise Navigationspunkte um beispielsweise neue Elemente im System anzulegen, Dateien zu importieren, Dateien zu exportieren oder anderweitige Aktionen auszuführen, wurde jedoch aus Kundenschutzgründen hier weggeschnitten. Rechts ist der jeweilige Inhalt des Navigationspunktes zu sehen. Bei dieser Abbildung geht es lediglich darum ein Gefühl für die Oberfläche zu bekommen, der konkrete Inhalt ist nicht relevant.

Ein typischer Ablauf für eine Monatsabrechnung sieht im Groben wie folgt aus:

1. Daten der EC-Transaktionen ins System importieren.
2. Das System führt die Entgeltberechnung durch.
3. Das Ergebnis wird in Form von Berichten / Rechnungen dargestellt und exportiert.

Die akquinet AG, die hier als Dienstleister auftritt, ist ein Unternehmen, dessen Kerngeschäft als „Geschäftsprozessoptimierung durch integrierte IT-Lösungen“ beschrieben werden kann. Dabei wird sowohl Standardsoftware als auch individualisierte Software verwendet. Weiterhin betreibt das Unternehmen Rechenzentren und ist im Business Consulting Bereich aktiv (akq15).
Die WEAT Electronic Datenservice GmbH (WEA15), hier der Kunde, ist ein Unternehmen welches europaweit Tankstellenbetreiber beim Management und Controlling unterstützt. Nach eigenen Angaben bestehen folgende Geschäftsfelder:

- Abwicklung des bargeldlosen Zahlungsverkehrs.
- Erfassung und Verarbeitung der Daten.
- Bereitstellung von Datendiensten wie Preisinformationen.
- Service & Beratung in allen Angelegenheiten des Netzbetriebs.

4.1 Betrachtung der WEAT EABR Deployment Pipeline

Die von der akquinet AG zur Entwicklung des WEAT EABR Systems verwendete Deployment Pipeline ist in Abbildung 9 zu sehen. Dabei besteht die erste Phase, direkt nach dem Commit aus Build, Unit- und Integrationstests. Gefolgt von zwei weiteren manuell gestarteten, jedoch automatisierten ablaufenden Phasen, welche die Performance- und Loadtests (Lasttests) ausführen. Im Anschluss folgt das manuelle Ausführen der bestehenden funktionalen Akzeptanztests. Dabei werden über die GUI die entworfenen User Stories durchgespielt.

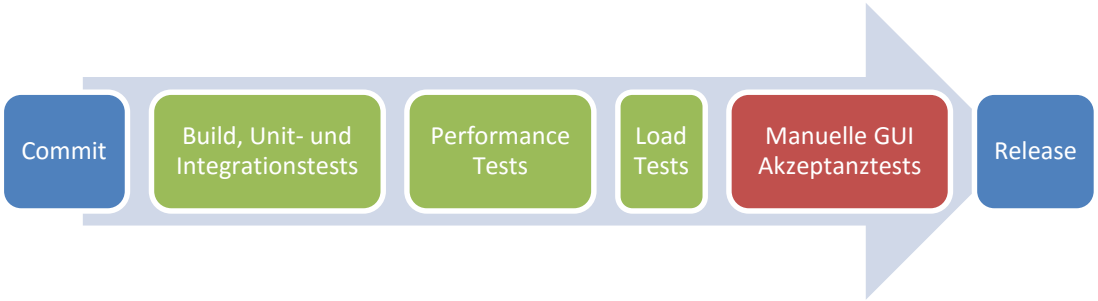

Abbildung 9 Deployment Pipeline des WEAT EABR Projekts. Die Größte der Phasen dient lediglich der besseren Lesbarkeit. Blau beschreibt Anfangs- und Endpunkt der Pipeline, Grün normale Phasen und Rot Phasen mit Verbesserungsbedarf.

Tabelle 1 zeigt welche der hier relevanten Phasen wie gestartet werden, wie lange die Durchführung in etwa dauert und wieviel Arbeitsstunden ca. investiert werden müssen.

Phase	Auslöser der Phase	Ungefähre Dauer in Std:Min	Etwa benötigte Arbeitszeit in Std:Min
Build, Unit- und Integrationstests	VCS commit	01:10	-
Performance Tests	Manuell	00:20	-
Loadtests	Manuell	06:00	-
Manuelle GUI Akzeptanztests	Manuell	08:00	08:00

Tabelle 1 Auslöser, Dauer und benötigte Arbeitsstunden der einzelnen Phasen der WEAT EABR Deployment Pipeline.

4.2 Problem Akzeptanztestphase

Wie in Tabelle 1 zu sehen ist dauern die manuellen GUI Akzeptanztests zwar nicht deutlich länger als etwa die Loadtests, jedoch benötigen sie eine Person zur Durchführung und stellen damit die größte Arbeitslast dar. Zusätzlich ist das Ausführen der Akzeptanztests eine sehr monotone Aufgabe, da es sich quasi um Regressionstests handelt, welche die Grundfunktionalitäten der Anwendung prüfen. Es werden immer die gleichen Schritte ausgeführt. Somit wird kostbare Arbeitszeit verschwendet und gleichzeitig ist der Ausführende wenig gefordert. Als weitere Folge kommt hinzu, dass durch die oben genannten Gründe, die Tests nicht so häufig ausgeführt werden wie es möglich wäre.
Ziel muss es also sein die vorliegenden Akzeptanztests zu automatisieren. Da die Implementierung aller Tests den Rahmen dieser Arbeit sprengen würde, wird ein einzelner repräsentativer Testfall automatisiert.

4.3 Rechnungslauftest als repräsentatives Beispiel

Als repräsentatives Beispiel für einen der Akzeptanztests wird in Kapitel 5 der WEAT EABR Rechnungslauftest mit Hilfe des Test-Editors automatisiert. Der Test-Editor ist ein Tool, dass den Tester bei der Erstellung von Testfällen unterstützt und ist in Abschnitt 4.4 beschrieben.
Der Rechnungslauftest testet die fachliche Funktionalität der Entgeltberechnung sowie einiger daraus resultierender oder im Anschluss separat ausgeführter Dateiexporte. Dafür werden im Grunde wie am Anfang von Kapitel 4 beschrieben zuerst alle Transaktionen für den kompletten Monat importiert, dann die Entgeltberechnung gestartet und durchgeführt, und anschließend daraus resultierende Ergebnisse und Exporte überprüft. Eine genauere Beschreibung der dafür notwendigen Schritte ist in Kapitel 5 zu finden. Ein

allzu genaues Verständnis der WEAT EABR Anwendung ist für die Testimplementierung nicht notwendig, da es sich eher um einen Black-Box Test handelt der die Ergebnisse des Tests mit Referenzen abgleicht.
Der Rechnungslauftest ist gut als repräsentatives Beispiel geeignet, da er die Hauptfunktionalität der Anwendung testet und sehr umfangreich ist (etwa vier DIN A4 Seiten), somit sind sehr viele der vorkommenden Schritte auch in anderen Tests wiederzufinden. D.h. einmal implementiert können diese Schritte ohne allzu große Umstände auch für anderweitige Tests wiederverwendet werden.

4.4 Test-Editor

Der Test-Editor ist ein Open Source Projekt, das darauf abzielt die Erstellung automatisierter Akzeptanztests, speziell für Nichtentwickler, zu erleichtern. Entwickelt wird das Projekt größtenteils von der akquinet AG und der SIGNAL IDUNA.
Der Test-Editor bietet eine grafische Benutzeroberfläche zum Erstellen von Testfällen.

Unterstützte Benutzeroberflächen sind z.B.

- Webanwendungen
- Swing Fat-Clients
- SWT\RCP Fat-Clients
- Webservices (SOAP und Rest)
- Mainframe Anwendungen (Kostenpflichtiges Plugin erforderlich)

Das Programm ist jedoch keineswegs auf die Erstellung von Tests für Benutzeroberflächen beschränkt. Es kann beliebiger Code durch die einzelnen Testschritte aufgerufen und ausgeführt werden. Dieser aufgerufene Code wird hier als Fixture bezeichnet.
Weitere Infos zum Test-Editor gibt es auf der Test-Editor Webseite (Tes).

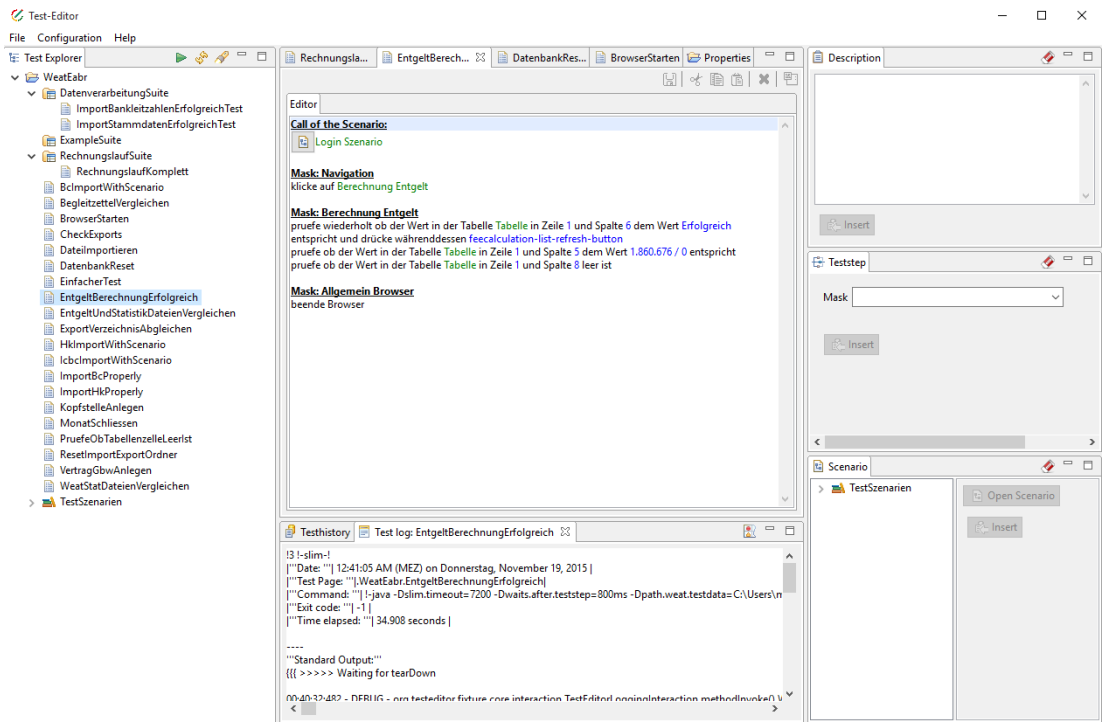

Abbildung 10 Benutzeroberfläche des Test-Editors. Die Lesbarkeit der Schrift ist hier nicht relevant, es geht lediglich um die einzelnen Sektionen der GUI.

Abbildung 10 zeigt den wesentlichen Teil der Benutzeroberfläche des Test-Editors. Die linke Seite dient als Browser für die erstellten Testfälle, Testsuiten und Testszenarien. Ein Testszenario bündelt einzelne Testschritte, die wiederverwendet werden sollen, zu einem Szenario, in etwa wie eine Funktion beim Programmieren. Ein Szenario kann dann als Testschritt aufgerufen werden. In der Mitte oben ist der eigentliche Editor zu sehen, in dem die einzelnen Testschritte dargestellt werden. Darunter können Testlogs und eine Historie der ausgeführten Tests betrachtet werden. Die rechte Seite bietet Möglichkeiten um einen einzelnen Testschritt zu erstellen. Im oberen Teil kann eine Beschreibung eingefügt werden, der mittlere Abschnitt dient dem Erstellen des eigentlichen Testschrittes und ganz unten kann ein Testszenario als Testschritt eingefügt werden.
Es erfolgt keine Evaluierung entsprechender Tools zur Automatisierung des Akzeptanztests, da für die vorliegende Arbeit der Test-Editor vorgegeben ist. Es wird die Version 1.8 des Test-Editors verwendet.

4.4.1 FitNesse

Der Test-Editor basiert auf FitNesse. FitNesse begann als Fork von Fit (jam15) und ist ein Webserver, Wiki und automatisches Test Tool (Fit15). Der Fokus liegt dabei auf der Automation von Akzeptanztests, welche im Wiki in einer speziellen Syntax eingepflegt

werden können um dann ausgeführt zu werden. Die genutzte Syntax ist auch für nicht Programmierer erlernbar.
FitNesse bietet also ein Wiki um Akzeptanztests zu dokumentieren welche dann durch sogenannte Fixtures ausgeführt werden. Diese Fixtures sind der Programmcode der die eigentlichen Testschritte ausführt. Zum Testen einer Webseite kann beispielsweise ein Akzeptanztest im FitNesse Wiki beschrieben werden, welcher dann z.B. von einer Fixture, welche die Selenium API nutzt, ausgeführt werden kann. Würde etwa, wie in diesem Fallbeispiel, Java verwendet um die Fixture zu erstellen, so wäre die Fixture ein Java Archive (JAR).
Der Test-Editor setzt auf FitNesse auf und bietet zusätzlich zur Funktionalität von FitNesse eine Benutzeroberfläche zur einfacheren Erstellung von Testfällen, sobald die Fixture einmal programmiert ist und mit der Oberfläche verbunden wurde.
FitNesse wird hier separat erwähnt, da das spätere Ausführen der Tests über Jenkins in Abschnitt 5.11 komplett ohne den Test-Editor stattfindet. Der Test-Editor dient also lediglich dazu das Entwerfen von FitNesse Tests benutzerfreundlicher zu gestalten.

5 Implementierung

Wie in Kapitel 4 dargestellt, ist der wohl größte Schwachpunkte der WEAT EABR Deployment Pipeline, die Phase der manuell ausgeführten Akzeptanztests. In diesem Kapitel wird die Automatisierung des repräsentativen Rechnungslauf Akzeptanztests im Detail beschrieben. Dazu wird zu Anfang eine Übersicht über die am Test beteiligten Komponenten und deren Zusammenhänge gegeben. Anschließend wird die Vorgehensweise erläutert um dann exemplarisch im Detail die Implementierung einzelner Testschritte zu beschreiben. Am Ende des Kapitels wird ein kurzer Vorher-Nachher-Vergleich der manuellen gegen die automatisierte Variante gezogen und gezeigt wie der Test in Jenkins integriert werden kann.

5.1 Übersicht

Um zu verstehen was zur Automatisierung des Tests unternommen wurde, ist es wichtig im Folgenden einige Zusammenhänge darzustellen. Als erster Schritt wird der grundlegende Aufbau des WEAT EABR Systems betrachtet, gefolgt vom Blick auf den Test-Editor, um im Anschluss darauf eingehen zu können an welchen Punkten der Test-Editor mit WEAT EABR interagiert.

5.1.1 WEAT EABR Komponenten

In Abbildung 11 sind die relevanten WEAT EABR Komponenten zu sehen. Dazu gehört der JBoss Server (JBo16), auf dem die eigentliche Applikation läuft, die dahinterliegende Datenbank, sowie ein Import- und Exportverzeichnis auf dem Dateisystem, mit deren Hilfe die Applikation Daten im- und exportiert.
Dabei steuert der Anwender, ein WEAT Mitarbeiter, die Applikation über die Weboberfläche. Der Webbrowser wird hier ebenfalls zum WEAT EABR System gezählt. Er fungiert als Client und bietet mit seiner Weboberfläche eine Schnittstelle zum Server. Für diese Arbeit wurde der Mozilla Firefox Browser (Moz16) in der Portable Edition Version 43.0.3 verwendet. Dieses Setup wird auf der Test-Editor Webseite empfohlen (Tes).
Die eigentliche Applikation läuft auf dem JBoss Server in Form eines WARs (Web application ARchive) (Wik16). Die Weboberfläche sowie die Client-Server-Kommunikation basiert auf Vaadin (Vaa15). Zur persistenten Speicherung der Daten wird eine PostgreSQL (Pos16) Datenbank eingesetzt. Bei den Im- und Exportverzeichnissen handelt es sich um beim Start des JBoss Servers als Parameter mitgegebene Pfade auf dem Dateisystem. Das

Importverzeichnis dient dazu Daten in das System zu importieren. Das Exportverzeichnis fungiert als Outputverzeichnis, welches Ergebnisse von Berechnungen enthält, wie etwa dem Rechnungslauf.

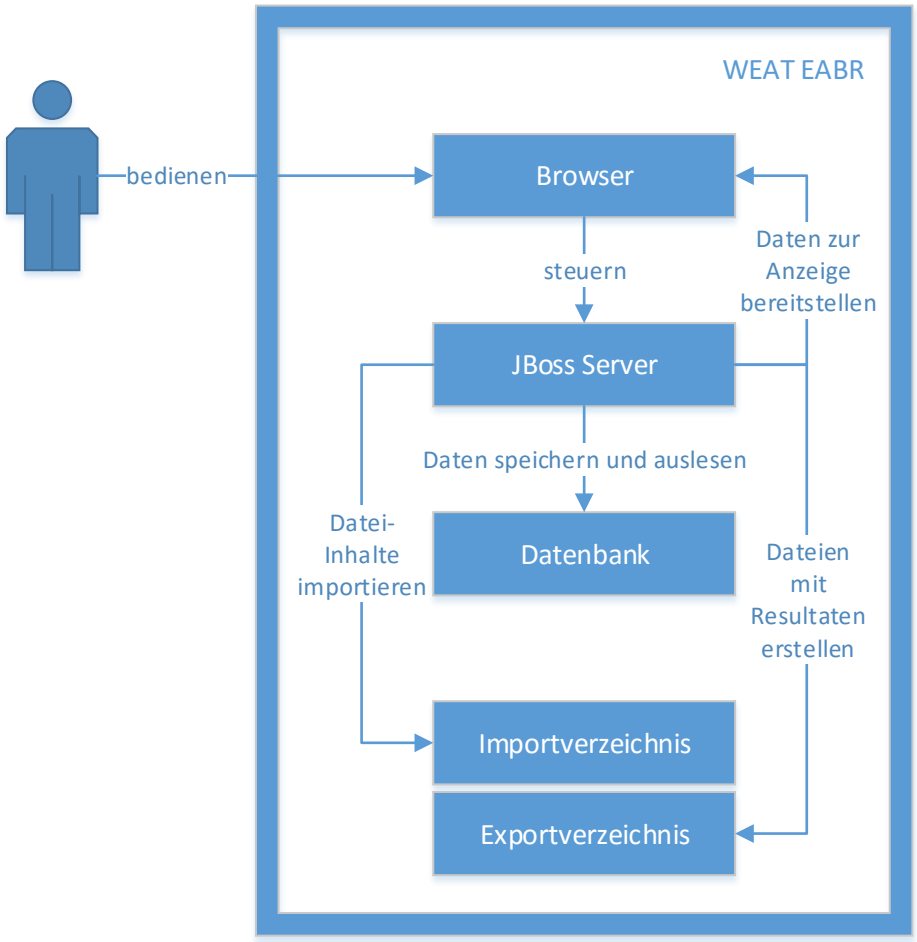

Abbildung 11 Für die vorliegende Arbeit wichtige Komponenten des WEAT EABR Systems.

5.1.2 Test-Editor Komponenten

Als nächstes wird der grobe Zusammenhang einiger dem Test-Editor zugehöriger Komponenten betrachtet. Abbildung 12 zeigt, dass der Nutzer den Test-Editor bedient um damit den FitNesse Rechnungslauftest zu erstellen und zu starten. Der, im Normalfall vom Test-Editor gestartete, FitNesse Server ruft bei Teststart nach und nach die den Testschritten entsprechenden Java Methoden in der Fixture auf. Hier werden jetzt die eigentlichen Testschritte ausgeführt die mit WEAT EABR interagieren. Mehr dazu in 5.1.3.

Der FitNesse Server kann auch ohne den Test-Editor gestartet werden und wird hier getrennt aufgeführt, siehe Abschnitt 5.11 für ein konkretes Beispiel.

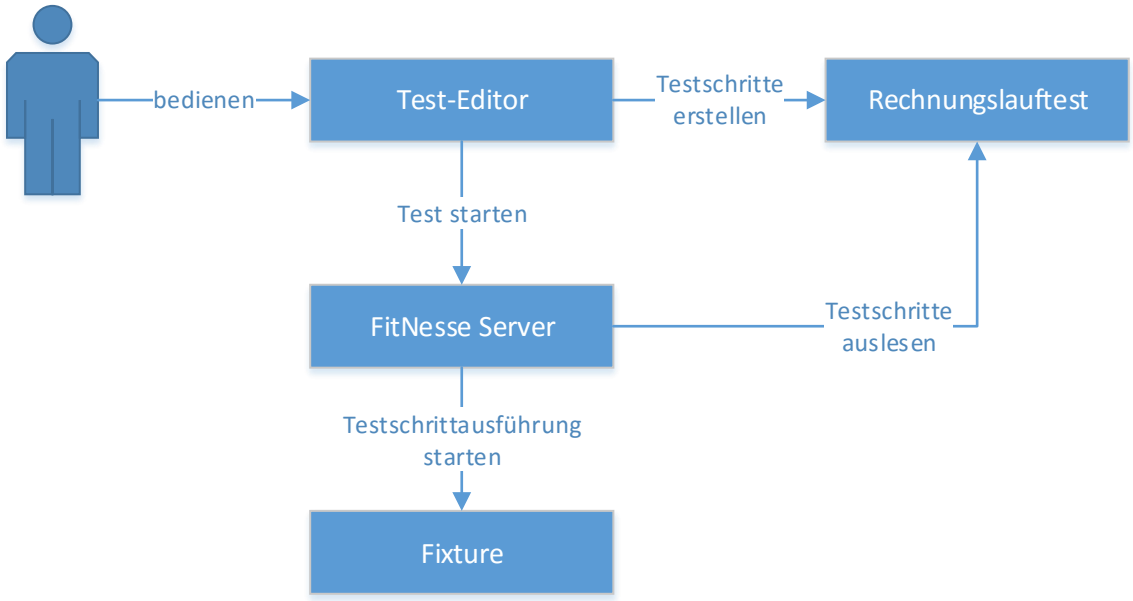

Abbildung 12 Zusammenhänge einzelner Komponenten beim Arbeiten mit dem Test-Editor.

5.1.3 Berührungspunkte zwischen Fixture und WEAT EABR

Die Java Fixture ist die zu erweiternde Komponente, wenn es darum geht neue Funktionalität in den Test-Editor zu bringen. Viele gängige Schritte kann der Test-Editor, mit Hilfe von vorhergehender Konfiguration, bereits ausführen. So ist etwa das Klicken von Links bereits implementiert. Um anderweitige neue Testschritte ausführen zu können, auch solche die keine GUI involvieren und WEAT EABR spezifisch sind, wird die *HtmlWebFixture* Klasse um neue Methoden erweitert.
In Abbildung 13 sind die Zusammenhänge zwischen der Fixture und WEAT EABR abgebildet. Die Fixture ist die Schnittstelle um mit WEAT EABR zu interagieren. Dabei wird der Browser mit Hilfe von Selenium gesteuert um zu navigieren und angezeigte Daten auszulesen. Mit dem JBoss Server wird nicht direkt interagiert. Die Datenbank wird mit Hilfe der Fixture geleert und initialisiert. In das Importverzeichnis werden Dateien kopiert, damit sie vom JBoss Server automatisch importiert werden. Aus dem Exportverzeichnis werden Dateien gelesen um die Ergebnisse des Rechnungslaufs abzugleichen. Beide Verzeichnisse werden zum entsprechenden Zeitpunkt gelöscht und neu aufgesetzt.
Als weiteres, nicht zu WEAT EABR gehöriges, aber speziell für die WEAT EABR Tests erstelltes, Verzeichnis ist das Testdatenverzeichnis zu sehen. Dieses enthält sowohl Dateien

mit Testdaten zum Importieren ins System, als auch Referenzdateien zum Abgleichen der Ergebnisse sowie weitere für den Test relevante Daten.

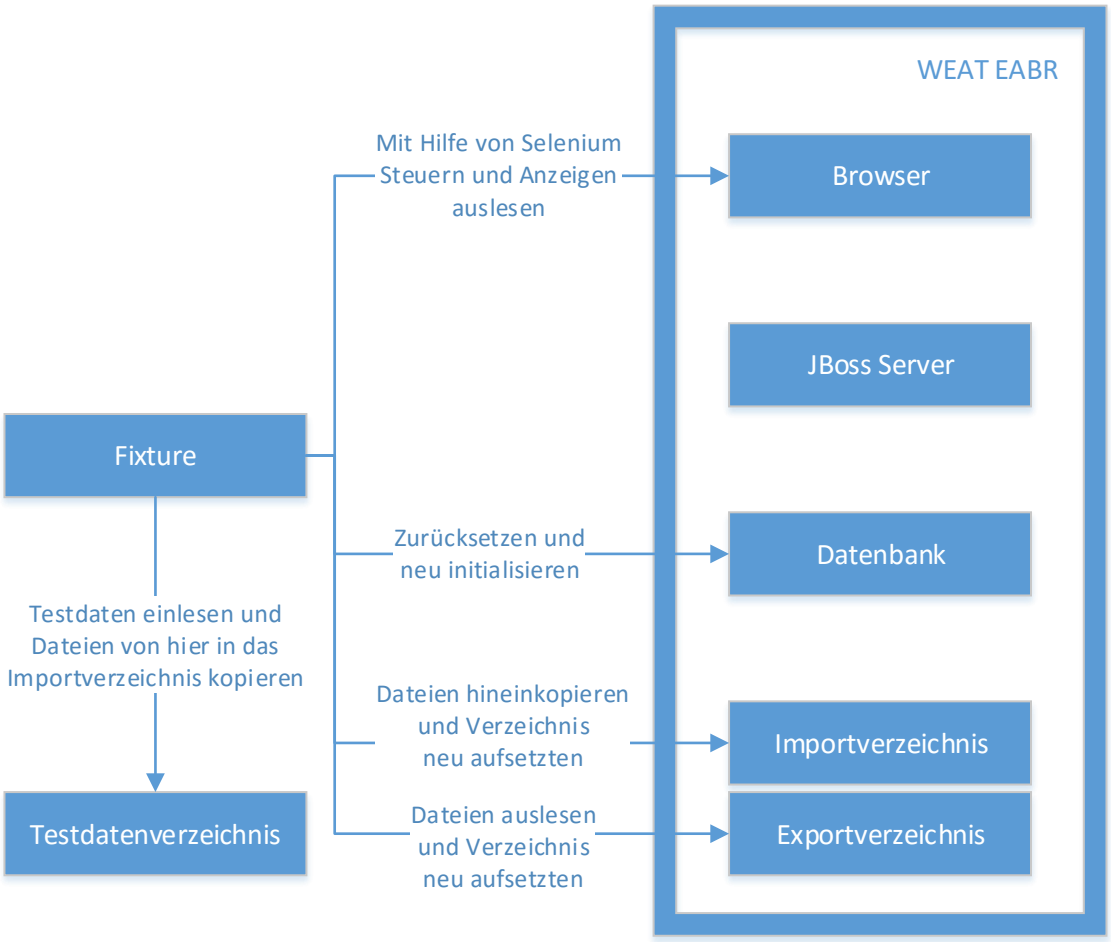

Abbildung 13 Berührungspunkte zwischen Java Fixture und WEAT EABR. Auf der rechten Seite sind die WEAT EABR Komponenten zu sehen. Auf der linken Seite ist die Fixture als Verbindungspunkt zum Test-Editor, sowie das extra für die WEAT EABR Tests angelegte Testdatenverzeichnis, zu sehen.

5.2 Aufbau der Fixture

Der Fixture Code, der zum Ausführen der Testschritte dient und mit WEAT EABR interagiert, ist von der allgemeinen Struktur her wenig komplex. Es wird die, vom Test-Editor vorgegebene *HtmlWebFixture* Klasse erweitert, die bereits einige Funktionalität enthält. Es werden also vom FitNesse Server die den Testschritten entsprechenden Methoden aus der *HtmlWebFixture* aufgerufen. So wird z.B. für das Prüfen einer Tabellenzelle der Benutzeroberfläche die Methode *checkTextIsPresentInTableCell* mit den entsprechenden

Parametern aufgerufen. Abbildung 14 zeigt zur Übersicht ein UML Klassendiagramm des neu hinzugefügten Fixture Codes.

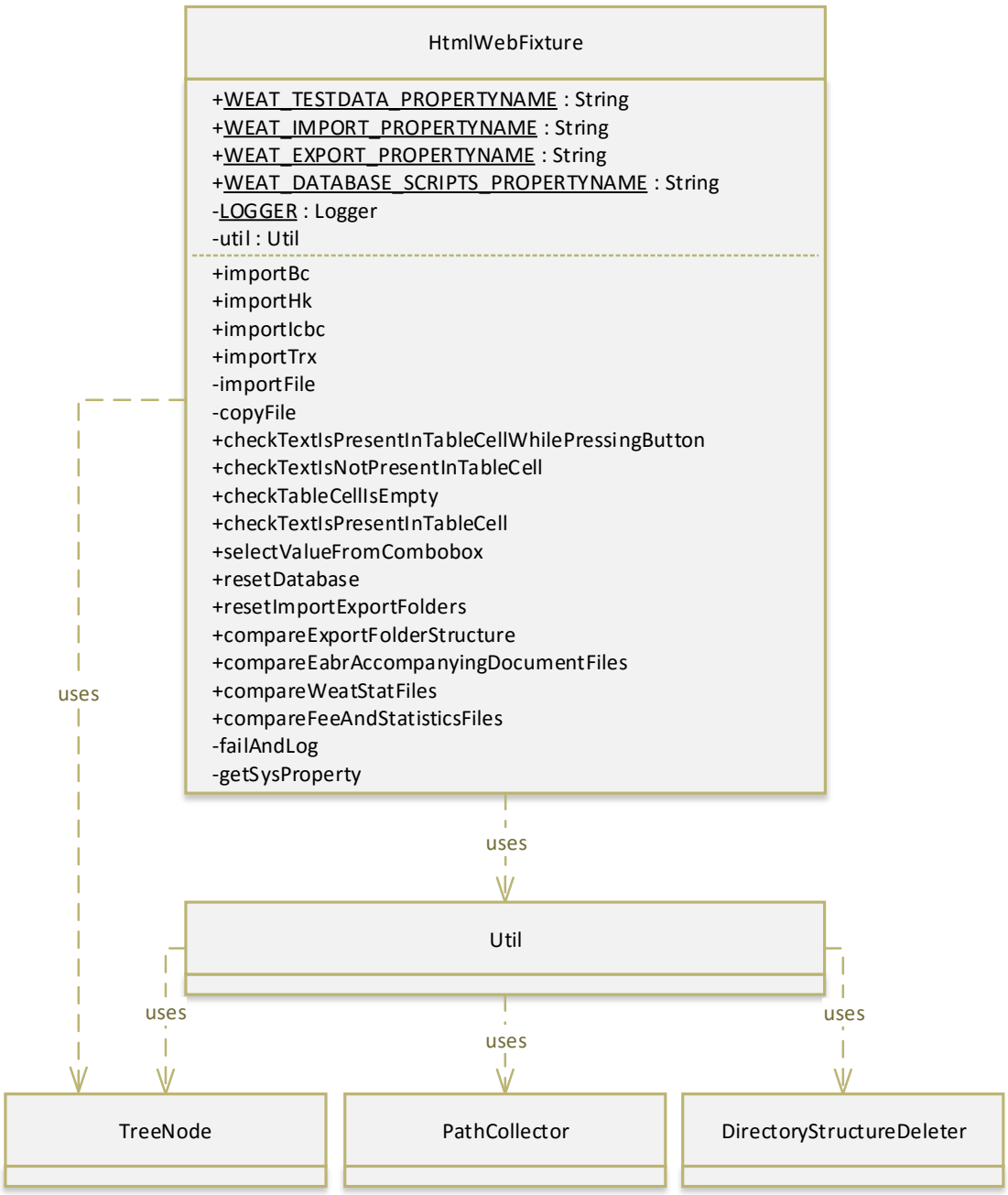

Abbildung 14 Vereinfachtes UML Klassendiagramm des neuen Fixture Codes.

Das Diagramm ist vereinfacht, so sind beispielsweise nur die neuen Methoden der *HtmlWebFixture* aufgeführt und nicht die bereits bestehenden oder Methoden anderer Klassen. Weiterhin sind der Übersicht halber keine Funktionsparameter angegeben.

Folgende Klassen sind enthalten:

- **HtmlWebFixture:** Einstiegspunkt in den Fixture Code.
- **Util:** Enthält Helfer Methoden.
- **TreeNode:** Datenstruktur zum Umgang mit Dateiverzeichnissen.
- **PathCollector:** Sammelt eine Liste aller Pfade in einem Verzeichnis.
- **DirectoryStructureDeleter:** Zum Löschen aller Inhalte eines Verzeichnisses.

Wie genannt ist die *HtmlWebFixture* der Einstiegspunkt für die Ausführung aller Testschritte. Wird ein Testschritt ausgeführt wird im Endeffekt immer eine Methode aus der *HtmlWebFixture* aufgerufen, sei es bereits vorhandener oder neu hinzugefügter Code.
Die Klasse enthält vier WEAT spezifische Konstanten um System Properties (Sys16) abzurufen zu können. Diese Konstanten enthalten die Bezeichner der System Properties. Die System Properties wurden vorher in der Test-Editor Konfiguration gesetzt. Die System Properties enthalten Pfade auf dem Dateisystem. Diese werden aus der Fixture heraus abgerufen und somit können die vorher konfigurierten Dateipfade in dem Fixture Code der einzelnen Testschritte verwendet werden.
Dabei beschreibt *TESTDATA* den Pfad zum Verzeichnis mit allen benötigten Testdaten. *IMPORT* und *EXPORT* zeigen auf die Im- und Exportverzeichnisse des WEAT EABR JBoss Servers. *DATABASE_SCRIPTS* verweist auf das Verzeichnis, dass die zur Initialisierung der Datenbank benötigten SQL Skripte enthält.
Der *LOGGER* dient, wie der Name vermuten lässt, dem Logging.
Im unteren Bereich der *HtmlWebFixture* sind die Methoden der Klasse gezeigt. Zur besseren Übersicht sind, wie erwähnt, nur die Namen, ohne Parameter und Rückgabewert, angegeben. Alle, mit einem Plus markierten, *public* Methoden sind direkte Einstiegspunkte für Testschritte, werden also direkt vom FitNesse Server aufgerufen und reichen einen *boolean*, der den Erfolg der Durchführung beschreibt, zurück. Die, mit einem Minus markierten, *private* Methoden dienen lediglich als kleine Helfer.
Für etwas größere Helfer, z.B. zum Erstellen und Löschen von Verzeichnisstrukturen wird die *Util* Helfer Klasse verwendet. Diese verwendet wiederum die Klassen *TreeNode*, *PathCollector* und *DirectoryStructureDeleter*. Diese drei Klassen sowie die *Util* Klasse sind hier und im Diagramm nicht weiter ausgeführt, da sie als Helfer für die *HtmlWebFixture* dienen und die Beschreibung der genauen Implementierung für den Verlauf der Arbeit nicht weiter interessant ist.
Abgesehen von den gezeigten Klassen, wurden, um die Funktionalität der Testschritte gegen Regression abzusichern, einige Regressionstests geschrieben. Dabei handelt es sich größtenteils um Unit- und Integrationstests. Letztere greifen auf das Dateisystem zu, um beispielsweise das Erstellen von Ordnerstrukturen oder Vergleichen von Dateien zu testen.

5.3 Signatur einer Fixture Methode

Die einzelnen Fixture Methoden folgen dem Aufbau in Listing 1. Die Methoden geben alle einen *boolean* zurück, der besagt ob der Testschritt erfolgreich ausgeführt werden konnte oder nicht. Zusätzlich werfen, wo sinnvoll, einige Methoden eine sogenannte *StopTestException*. Diese ist bereits vom Test-Editor Fixture Code vorgegeben und signalisiert, dass der komplette Test gestoppt werden soll.

```
public boolean methodName() throws StopTestException
```

Listing 1 Allgemeine Signatur einer Fixture Methode.

Als Kriterium zum Werfen einer *StopTestException*, und damit dem Abbruch des gesamten Tests wurde folgendes Kriterium gewählt: Wenn ein Testschritt als Grundlage für weitere Testschritte dient, d.h. beispielsweise das Zurücksetzen der Datenbank oder die Eingabe von Daten in eine Maske, also in allen Fällen in denen nicht nur eine separate alleinstehende Prüfung vorgenommen wird, wie etwa dem Vergleich zweier Dateien, wird im Fehlerfall eine *StopTestException* geworfen, da sonst nicht für einen vernünftigen Testdurchlauf garantiert werden kann.
Der Testablauf und im speziellen die Fehler können in der Log Datei des Tests eingesehen werden.

5.4 Test-Editor Konfigurationsdateien

Um mit Hilfe des Test-Editors einen Test auszuführen sind vorab einige Konfigurationen nötig. In Tabelle 2 sind die Dateien, die für die Testschrittimplementierungen angepasst werden müssen, sowie deren Zugehörigkeit (Test-Editor oder FitNesse) und jeweilige Funktion, aufgeführt. Diese Dateien sind quasi das Bindemittel zwischen der leserlichen Test-Editor GUI und dem eben gezeigten Code in der Fixture, der dann wiederum die WEAT EABR Komponenten anspricht. Der Einfachheit und Vollständigkeit halber wird hier die bereits beschriebene Java Fixture ebenfalls aufgeführt.

Dateiname	Zugehörigkeit	Funktion
ElementList.conf	Test-Editor	Definition von Konstantennamen für WEAT EABR GUI Elemente. HTML IDs und XPath Elemente, die referenziert werden sollen, bekommen leserliche Namen.
AllActionGroups.xml	Test-Editor	Deklaration von Menustrukturen (ActionGroups) zur Erstellung von Testschritten in der Test-Editor GUI, sowie das Verbinden von Konstanten aus der ElementList.conf mit einem *TechnicalBindingType* aus der *TechnicalBindingTypeCollection.xml*.
TechnicalBindingTypeCollection.xml	Test-Editor	Festlegen der Struktur eines *TechnicalBindingTypes*. Ein TechnicalBindingType ist in etwa gleichzusetzten mit einer Blaupause für einen Testschritt.
content.txt (SzenarioLibrary)	FitNesse	Verbinden von *TechnicalBindingTypes* mit Methoden aus der *Fixture.jar.*
content.txt (Projekt)	FitNesse	Allgemeine Projektkonfigurationen, sowie das Anlegen von Konstanten, die in der Fixture verwendet werden sollen (Mit Hilfe von *System Properties*).
Fixture.jar	FitNesse	Java Archiv mit bereits vorgegebenen sowie neu programmierten Methoden die im Test-Editor verwendet werden sollen.

Tabelle 2 Funktion der einzelnen Konfigurationsdateien des Test-Editors.

Zum besseren Übersicht sind in Abbildung 15 noch einmal die Konfigurationsdateien und deren grobe Zusammenhänge gezeigt. Dabei enthält die Abbildung nicht die *content.txt (Projekt)*, da diese nur indirekt mit der Fixture interagiert und eher allgemeine Konfigurationen enthält.

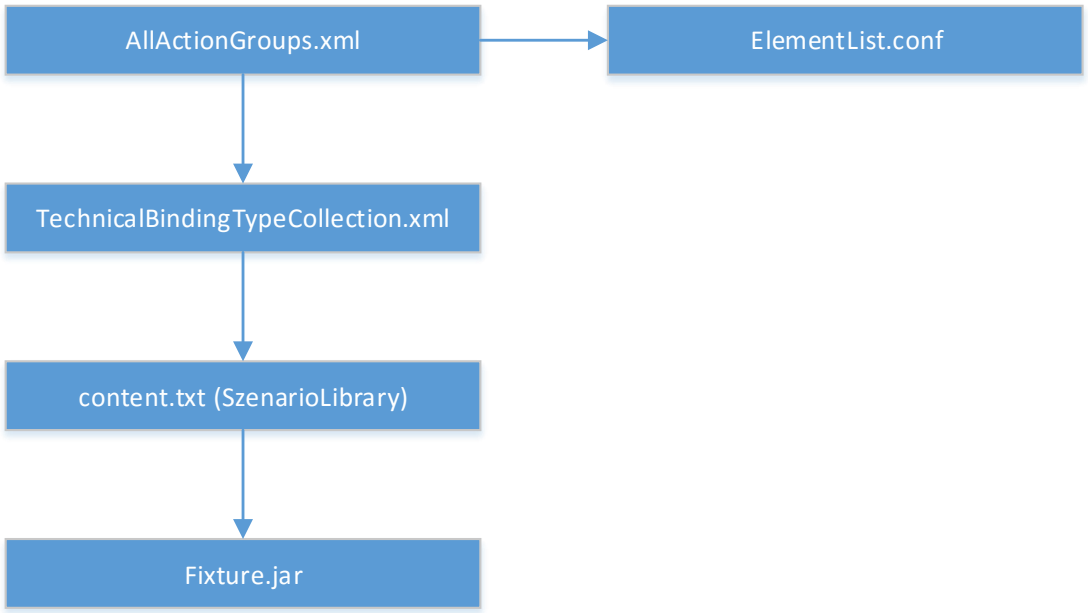

Abbildung 15 Zusammenhänge der Konfigurationsdateien. Die Pfeile beschreiben eine Referenz.

Ein etwas genaueres Verständnis der Dateien ergibt sich beim Betrachten der weiter unten aufgeführten Beispiele zur Implementierung eines konkreten Schritts, hier wird lediglich eine Übersicht gegeben. Beispiele sind in Abschnitt 5.8.5. und 5.9.4 zu finden.

5.5 Vorgehensweise bei der Testschrittimplementierung

Um die einzelnen nötigen Testschritte des Rechnungslaufs zu implementieren wurde in etwa wie in Abbildung 16 dargestellt vorgegangen. Das Aktivitätsdiagramm zeigt grundlegend zwei Pfade, deren Wahl davon abhängt ob es für den benötigten Schritt bereits eine passende Fixturemethode gibt oder diese noch erstellt werden muss.
Existiert diese bereits, muss lediglich in der *AllActionGroups.xml* die entsprechende Funktionalität in der Test-Editor Oberfläche verfügbar gemacht werden. Falls es sich um einen Testschritt, der die GUI involviert, handelt, müssen zusätzlich die entsprechenden GUI Elemente aus der *ElementList.conf* referenziert werden. In dieser Abbildung wird davon ausgegangen, dass alle benötigten GUI Elemente bereits vorab in die *ElementList.conf* eingepflegt wurden.

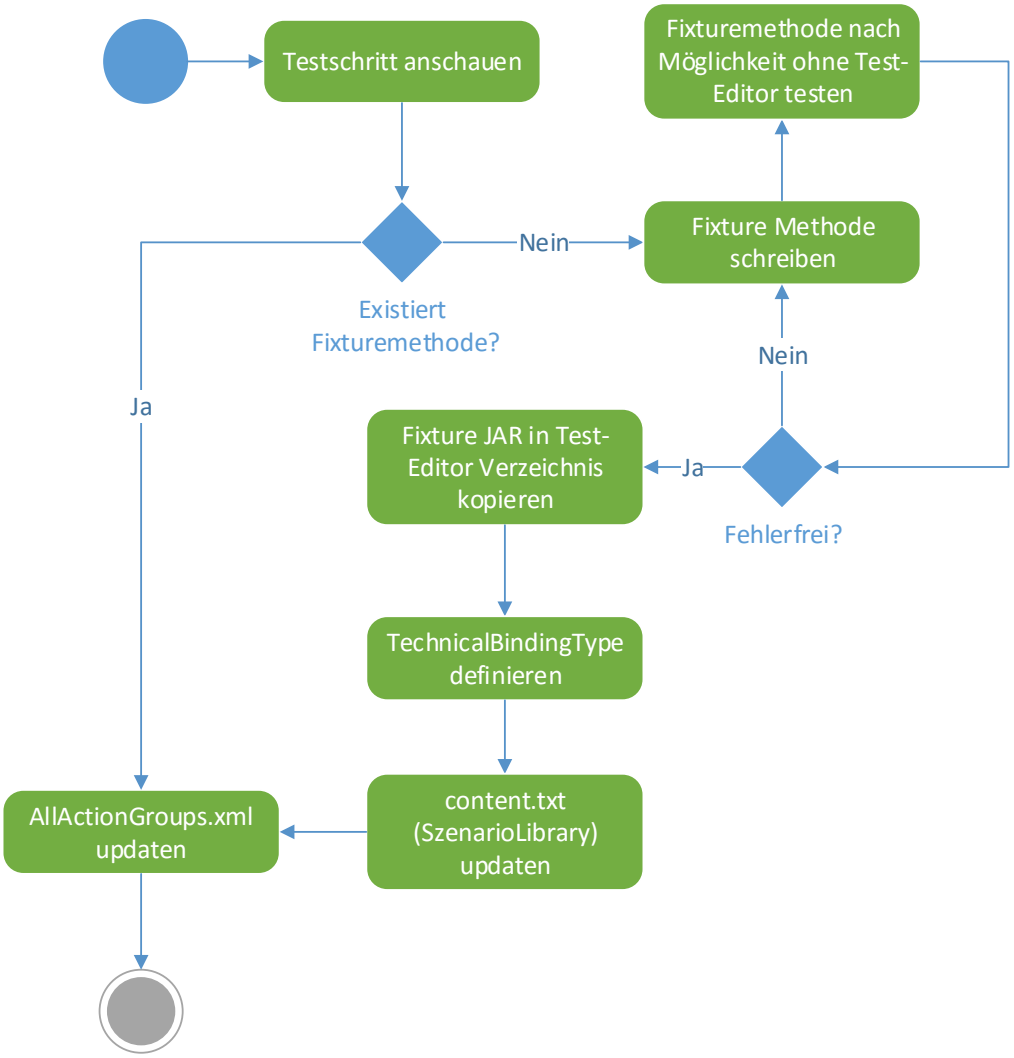

Abbildung 16 Vorgehensweise beim Implementieren der einzelnen Testschritte.

Ist die Methode noch nicht vorhanden, so muss sie erstellt, getestet und anschließend in ein bestimmtes Test-Editor Verzeichnis kopiert werden. Damit wird die alte Fixture durch eine neue, erweiterte, ersetzt. Jetzt müssen noch einige Konfigurationen am Test-Editor vorgenommen werden. Für Details hierzu sei ebenfalls auf die Bespiele in den Abschnitten 5.8.5 und 5.9.4 verwiesen.

5.6 Nötige Schritte

Um den Rechnungslauftest zu automatisieren ist eine Vielzahl von kleinen Schritten erforderlich. Um nicht zu viel preiszugeben werden hier die einzelnen Schritte des Rechnungslaufs nicht genannt, zusätzliche ist der genaue Ablauf des Tests nicht von Belang. Die Schritte lassen sich grob kategorisieren. Zum einen sind Interaktionen mit der GUI nötig, dazu zählen sowohl Eingaben, als auch Prüfungen von Anzeigen. Zum anderen gibt es Schritte die nicht direkt mit der GUI interagieren, beispielsweise Dateiimporte und Vergleiche von Dateien.

5.6.1 Grober Ablauf des Tests

Aus der Vogelperspektive gesehen, stellt sich der Test wie folgt dar:

- **Herstellen eines bekannten Systemzustandes**. Dieser Schritt ist nicht spezifisch für den Rechnungslauftest, sondern beschreibt das Zurücksetzen des Systems auf einen definierten Zustand. Dazu gehört beispielsweise das Leeren der Datenbank.
- **Vorbedingungen des Rechnungslauftests erfüllen**. Vorbereiten des Systems für einen Rechnungslauf für einen kompletten Monat. Unter anderem das Erstellen von Verträgen oder Importieren von Transaktionsdaten.
- **Start und Ausführung des Rechnungslaufs**. Dies geschieht automatisch, sobald Transaktionen importiert werden und bedarf somit keiner Aktion.
- **Prüfen der Ergebnisse**. Im Anschluss wird geprüft ob der Rechnungslauf fehlerfrei durchgelaufen ist und ob die entstandenen exportierten Dateien und erstellten Verzeichnisse den Erwartungen entsprechen.

5.6.2 Konkret benötigte Schritte

Der in 5.6.1 beschriebenen Ablauf enthält eine große Menge einzelner Testschritte. Dabei gibt es sich wiederholende oder zumindest ähnliche Schritte mit gleicher Abfolge. Im Folgenden werden diese Typen von Schritten, zur besseren Übersicht in drei Gruppen gegliedert, aufgelistet:

Reset (Abschnitt 5.7)

- Datenbankinhalt löschen und mit Skripten initialisieren
- WEAT EABRS Import- und Exportverzeichnisse löschen und benötigte Verzeichnisstrukturen neu erstellen

Interaktionen mit der GUI (Abschnitt 5.8)

- Buttons und Links klicken
- Textfelder ausfüllen
- Elemente aus Dropdown-Menu auswählen
- Tabelleneinträge prüfen
- Prüfen ob ein bestimmter Text vorhanden ist
- Browser starten
- Browser stoppen

Dateiopererationen (Abschnitt 5.9)

- Dateiinhalte importieren
- Exportverzeichnisstruktur mit Referenz vergleichen
- Dateiinhalte vergleichen

Einige dieser Schritte sind bereits ohne großen Aufwand mit Hilfe des Test-Editors möglich, andere Erfordern speziellen Fixture Code. In den drei Abschnitten im Anschluss werden die hier gezeigten Gruppen von Schritten beschrieben.

5.7 Reset

Um zu gewährleisten, dass jeder Testdurchlauf mit einem Systemzustand arbeitet, der immer gleich ist und den Erwartungen entspricht, müssen bestimmte WEAT EABR Komponenten vor jedem Test auf einen definierten Zustand gesetzt werden. Dazu zählen in diesem Fall die Datenbank und die Import- und Exportverzeichnisse.

5.7.1 Datenbank Reset

Als erster Schritt des Rechnungslauftests wird die Datenbank mit Hilfe von Flyway (fly15), einer Library zur Datenbankmigration, geleert und im Anschluss mit Initialisierungsskripten auf einen definierten Zustand gebracht. Die verwendeten Skripte entsprechen denen, die für die Initialisierung einer für die Produktion vorgesehenen Datenbank ausgeführt werden. Somit wird auf einem möglichst produktionsnahen Zustand getestet und es müssen keine neuen Skripte erstellt und gepflegt werden.
Das verwendete Flyway JAR muss ebenfalls dem Test-Editor zugänglich gemacht werden. In diesem Fall wurde es in das Test-Editor lib-Verzeichnis kopiert.

5.7.2 Import- und Exportverzeichnis Reset

WEAT EABR verwendet sowohl ein Import- als auch ein Exportverzeichnis als Dateischnittstelle mit dem Server.

Das Importverzeichnis enthält verschiedene Ordner für verschiedenartige Dateien, beispielsweise Bankleitzahlen oder Transaktionen. Wird eine Datei in diese Ordner kopiert, wird sie automatisch vom WEAT EABR Server importiert und in die Datenbank eingepflegt. Die Metadaten eines solchen Imports können dann über die Weboberfläche betrachtet werden. Dazu gehört beispielsweise die Dauer des Imports, ob der Import erfolgreich war und die Anzahl der importierten Elemente. Ein Beispiel für die Anzeige der Metadaten nach einem Bankleitzahlimport ist in Abbildung 17 zu sehen.

Abbildung 17 Tabelle mit Metadaten zu einem Import einer Datei mit Bankleitzahlen.

Um evtl. auftretenden Komplikationen durch Dateirückstände voriger Testdurchläufe aus dem Weg zu gehen, werden sowohl Import- als auch Exportverzeichnis komplett gelöscht und nach den im WEAT EABR Wiki definierten Vorgaben die Ordnerstrukturen neu erstellt. Damit die Dateistruktur flexibel geändert werden kann, ohne dass die Java Fixture geändert und neu kompiliert werden muss, wurde ein Programmteil entwickelt der aus einer mit Tabs eingerückten Textdateien die Ordnerstrukturen einliest. Es existiert sowohl eine Datei mit der Import-, als auch eine mit der Exportverzeichnisstruktur. Ein kleiner Ausschnitt einer solchen Datei, mit geänderten Namen ist in Listing 2 zu sehen.

```
folder-a
    folder-b
    folder-c
        folder-d
    folder-e
folder-f
folder-g
```

Listing 2 Beispielordnerstruktur.

Nach der Ausführung dieser beiden Testschritte, dem neu aufsetzten der Datenbank und dem Neuerstellen der Import- und Exportverzeichnisse, kann der eigentliche Rechnungslauftest gestartet werden.

5.8 Interaktionen mit der GUI

Viele der benötigten Schritte können ohne Änderungen am Fixture Code, nur durch entsprechende Konfiguration des Test-Editors verwendet werden. Dazu gehören:

- Buttons und Links klicken
- Textfelder ausfüllen
- Prüfen ob ein bestimmter Text vorhanden ist
- Browser starten
- Browser stoppen

Neuen Fixture Code benötigen die folgenden Schritte:

- Element aus Dropdown-Menu auswählen (siehe Abschnitt 5.8.3)
- Tabelleneinträge prüfen (siehe Abschnitt 5.8.4)

In dem zu schreibenden Fixture Code verwenden wir, genau wie der Rest des Test-Editors Selenium (Sel15). Selenium stellt eine API zur Steuerung von Web Browsern bereit, um Browserinteraktionen automatisieren zu können. Es wird größtenteils zur Testautomatisierung verwendet und kann die meisten modernen Browser fernsteuern.
Bevor jedoch Elemente aus den beiden oben gezeigten Kategorien angesprochen werden können, müssen für alle verwendeten GUI Elemente, für beide Kategorien von Schritten, im WEAT EABR GUI Code HTML IDs gesetzt werden und diese dann in die *ElementList.conf* eingepflegt werden.

5.8.1 Setzten von HTML IDs

Um mit dem Test-Editor über die *ElementList.conf* die WEAT EABR GUI Elemente ansprechen zu können werden im WEAT EABR Source Code, also nicht im Fixture Code, für alle benötigten Elemente HTML IDs gesetzt. Das sind größtenteils Buttons, Textfelder, Tabellen sowie Dropdownmenüs.
Das dazu verwendete Schema wird im neu hinzugefügten WEAT EABR Wikieintrag in Abbildung 18 beschrieben. Dabei ist das Hauptziel die IDs so zu setzten, dass sie in der ganzen Applikation einzigartig sind.

Die IDs der Komponenten sind in Englisch.

Sie werden nach folgendem Schema vergeben:
<section>-<subsection>-*<name>-<component type>

Die subsection kann null oder mehrmals vorkommen.

Beispiel: Das IBAN TextField in der Detailansicht unter dem Navigationspunkt Kopfstellen (headends) hat die ID:
headend-details-iban-textfield

Beispiel 2: Der im header befindliche logout button hat die ID:

header-logout-button

Abbildung 18 Neuer Eintrag aus dem WEAT EABR Wiki, der beschreibt wie die HTML IDs gesetzt werden.

Listing 3 zeigt ein Beispiel wie die HTML IDs im Source Code gesetzt wurden. Die gezeigte Methode wird bei der Initialisierung der Komponente aufgerufen. Die *DetailsViewRootComponent* dient als Superklasse für mehrere Detailansichten zum Anlegen neuer Elemente über die WEAT EABR GUI. Die weiter unten auftauchende Abbildung 23 zeigt ein Beispiel für solch eine Ansicht.

```
public void setHtmlIds(String... sections) {
    StringBuilder builder = new StringBuilder();
    String delimiter = "-";

    for (String section : sections) {
        builder.append(section);
        builder.append(delimiter);
    }
    String prefix = builder.toString();

    this.buttonBack.setId(prefix + "back-button");
    this.buttonEdit.setId(prefix + "edit-button");
    this.buttonCancel.setId(prefix + "cancel-button");
    this.buttonSave.setId(prefix + "save-button");
}
```

Listing 3 Ausschnitt aus DetailsViewRootComponent.java Datei aus dem WEAT EABR Projekt. Zeigt wie hier die HTML IDs gesetzt werden.

5.8.2 Warum kein XPath?

Es ist auch möglich Elemente mit dem Test-Editor über deren XPath anzusprechen. XPath beschreibt einen Pfad in XML Dokumenten um einzelne Elemente oder aber Gruppen von Elementen zu adressieren (W3S15). Der Vorteil dieser Variante ist es, dass nicht extra HTML IDs gesetzt werden müssen.
Diese Möglichkeit wurde jedoch verworfen, da der Test-Editor zum einen stabiler mit HTML IDs arbeitet und zum anderen das Ansprechen der GUI Elemente etwas

änderungsresistenter ist. Ändert das WEAT EABR Team beispielsweise die GUI und verschiebt etwa einen Button in ein anderes div-Element, oder ändert sich der Elementaufbau in einer neueren Version des GUI Frameworks, hier Vaadin, so sind die Elemente in den meisten Fällen weiterhin über HTML IDs adressierbar. Mit XPath ist dies nicht der Fall.
Ein weiterer Faktor ist die bessere Lesbarkeit und geringere Komplexität von vernünftig gesetzten HTML IDs gegenüber XPath Elementen. Ebenso können, falls benötigt, konkrete Informationen über z.B. den einzutragenden Datentyp eines Textfeldes in der zugehörigen HTML ID gespeichert werden (siehe Abschnitt 7.3.2).

5.8.3 Element aus Dropdown-Menu auswählen

Es ist zwar potenziell möglich mit dem Test-Editor Elemente aus Dropdown-Menüs auswählen, jedoch wurde die WEAT EABR GUI mit Vaadin erstellt. Vaadin Dropdown-Menüs, genannt *ComboBoxen*, verwenden kein *<select>* Tag zum Erstellen solcher Menus. Hier bestehen diese aus einer Kombination von * * und *<td>* Elementen. Somit funktioniert die bestehende Methode zur Auswahl von Elementen nicht.
Es muss also eine neue Fixture Methode geschrieben werden, welche als Parameter die HTML ID der *ComboBox*, sowie den Text des auszuwählenden Eintrags bekommt. Diese wählt dann, wenn vorhanden, den entsprechenden Eintrag aus.
Für ein besseres Verständnis des nötigen Prozesses zur Erstellung und Integration neuer Fixture Funktionalität sei auf die Beispiele in Abschnitt 5.8.5 und 5.9.4 verwiesen.

5.8.4 Tabelleneinträge prüfen

Ein Großteil der WEAT EABR Weboberfläche besteht aus Tabellen (siehe Abbildung 19). Tabellen zur Anzeige importierter Dateien, laufender Prozesse wie z.B. dem Rechnungslauf und zur Auflistung eingepflegter Elemente, beispielsweise von Verträgen.

Abbildung 19 Tabelle zur Anzeige von importierten Transaktionsdateien. Hier sind drei erfolgreich abgeschlossene separate Importe angezeigt.

Um zu prüfen ob die Dateien korrekt importiert wurden, die Prozesse erfolgreich abgeschlossen wurden etc. muss es also möglich sein den Inhalt von einzelnen Zellen zu überprüfen, d.h. mit einem vorher festgelegten *String* vergleichen zu können.
Die zu diesem Zweck geschriebene Fixture Methode hat die in Listing 4 gezeigte Signatur. Sie bekommt also aus dem Test-Editor den erwarteten Text, die Zeilen- und Spaltennummer, sowie die zuvor in Abschnitt 5.8.1 gesetzte HTML ID der Tabelle. Dann wird der Text der Tabellenzelle mit dem mitgegebenen Text verglichen.
Als Besonderheit müssen die, z.B. in Abbildung 19 gezeigten, runden farbigen Statusindikatoren, falls vorhanden, ignoriert werden.

```
public boolean checkTextIsPresentInTableCell(String text, String row, String col,
                                             String tableId)
```

Listing 4 Signatur der Fixture Methode zum Vergleichen von Tabellenzellenwerten mit einem gegebenen Text.

Es gibt jedoch einen Fall der wesentlich häufiger auftritt als der oben beschriebene. In Abbildung 20 ist ein noch nicht abgeschlossener Importprozess zu sehen. Oben links über der Tabelle ist der Refresh Button zu sehen. Dieser aktualisiert den Status des jeweils laufenden Prozesses. Der Status zeigt an, dass der Prozess noch nicht beendet ist und die Laufzeit gibt an, dass er bereits 37 Sekunden läuft.

Abbildung 20 Tabelle die einen laufenden Importprozess anzeigt.

Das Ziel ist es auf das Ende des Prozesses zu warten und dann und dann zu prüfen ob er erfolgreich abgeschlossen wurde. Der Status wird allerdings ausschließlich durch das Klicken des Refresh Buttons aktualisiert. Einige Prozesse können mehr als 10 Minuten laufen. Wenn der Refresh Button gedrückt wird gibt es also keine Garantie, dass der Prozess danach als abgeschlossen angezeigt wird.
Weiterhin bietet der Test-Editor keine Funktionalität um direkt Schleifen in den Testablauf zu integrieren um beispielsweise 10 Sekunden lang zu warten, dann den Refresh Button zu drücken, anschließend den Status aus der Tabelle zu prüfen und den Ablauf solange zu wiederholen bis der Prozess beendet ist.
Eine weitere Möglichkeit wäre pauschal etwa 15 Minuten zu warten bevor der Refresh Button gedrückt wird und dann die Tabelle zu prüfen. Der Test-Editor bietet die Möglichkeit einen Testschritt einzufügen der für einen bestimmten Zeitraum wartet. Diese Variante hat jedoch zwei gravierende Nachteile. Zum wird sehr viel Zeit verschwendet, wenn der Prozess eigentlich weniger Zeit benötigt. Zum anderen gibt es keine Sicherheit, dass nach Ablauf der

Zeit der Prozess tatsächlich beendet ist. Unvorhergesehene Systemlast kann die Laufzeit deutlich strecken.
Um dieses Problem zu lösen muss das Warten in die Fixture Methode verlagert werden. In der entworfenen Methode wird der Refresh Button gedrückt, kurz gewartet (z.B. 5 Sekunden) und dann beispielsweise geprüft ob der Status den Wert *Erfolgreich* anzeigt. Auch bei dieser Variante wird durch das letzte Warteinterval Zeit verschwendet, jedoch deutlich weniger.
Als weiterer Nachteil dieser Variante ergibt sich, dass nun sehr viel Zeit in der Fixture Methode verbracht wird, es muss also für einen, sehr großzügigen, Timeout gesorgt werden, der den Methodenaufruf nach einer zu langer Zeit beendet, damit der Testschritt in einem unvorhergesehenen Fehlerfall nicht unendlich viel Zeit beansprucht. Für diesen Fehlerfall wird also weiterhin Zeit verschwendet.
Zusätzlich zu den Parametern der in Listing 4 gezeigten Methode muss für diese Implementierung ebenfalls die HTML ID des Refresh Buttons übergeben werden, um darauf zugreifen zu können. Hierbei stößt man auf eine weitere Limitation des Test-Editors. Zwar ist es möglich beliebige Parameter an Fixture Methoden zu übergeben, jedoch kann ein Testschritt nicht zwei Elemente aus der *ElementList.conf* Konfigurationsdatei referenzieren. In Abbildung 21 ist zu sehen dass der Name der Tabelle, hier *Tabelle*, einen leserlichen Namen hat und als Dropdown-Menu dargestellt wird, obwohl auch hiermit indirekt eine HTML ID referenziert wird. Für den zu drückenden Button bleibt nur die Möglichkeit über ein Textfeld direkt die HTML ID einzugeben. Dies ist, wenn auch unschön, ein Workaround um trotzdem auf zwei HTML IDs in einem Testschritt zugreifen zu können.

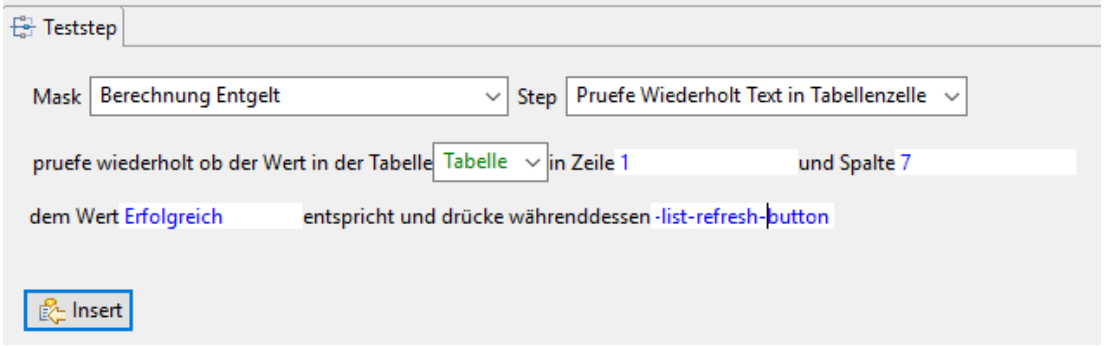

Abbildung 21 Test-Editor GUI Ausschnitt, der das Erstellen eines Testschrittes zur wiederholten Prüfung eines Tabelleneintrages zeigt. Das hier gezeigte Bild wurde bearbeitet um, zur besseren Lesbarkeit, die Anzeige des Schrittes auf mehrere Zeilen aufzuteilen. Zusätzlich ist durch die Darstellungsweise des Test-Editors in dem letzten Textfeld nicht die Komplette HTML ID sichtbar.

Nun ist es möglich Testschritte einzufügen, die wiederholt den Refresh Button einer Tabelle drücken und zwischendurch eine Tabellenzelle mit einem vorgegebenem Wert vergleichen und dies solange tun bis der erwartete Text erscheint oder aber der Timeout abgelaufen ist.

5.8.5 Beispiel ELV-Konzentrator anlegen

Im Folgenden wird ein einfach gehaltenes, aber repräsentatives Beispiel zum Erstellen eines Testschrittes gezeigt. Das Beispiel soll dabei helfen einen Einblick in die Details der Arbeit zu bekommen. Dies dient nicht nur dem Nachvollziehen, sondern ist auch wichtig für das spätere Verstehen von Kritik und Optimierungsvorschlägen in den Abschnitten 7.2, 7.3.1 und 7.3.3.
Ein ELV-Konzentrator beschreibt einen Anbieter des Elektronischen Lastschriftverfahrens. Abbildung 22 zeigt den entsprechenden Wiki-Ausschnitt aus den Vorbedingungen des Rechnungslauftests, mit der Anweisung einen neuen ELV-Konzentrator anzulegen.

- Folgende ELV-Konzentratoren wurden über die UI im System angelegt
    - Routingkennzeichen *20* und Name *InterCard*

Abbildung 22 Ausschnitt aus dem Rechnungslauftest. Der gezeigte Eintrag beschreibt das Anlegen eines ELV-Konzentrators in den Vorbedingungen für den Test.

Um einen solchen im WEAT EABR System neu einzupflegen muss der Nutzer zur entsprechenden Maske navigieren, Name und Routingkennzeichen eintragen und auf „Speichern“ klicken. Das Routingkennzeichen dient der Identifikation des entsprechenden ELV-Konzentrators einer Transaktion. Ein Verständnis der Domäne ist aber für die hier gezeigten Abläufe nicht weiter von Belang.

Abbildung 23 zeigt die Maske. Ebenso zu sehen ist hier die gesetzte HTML ID des Name Feldes (auf Englisch): *#eddsconcentrator-details-name-textfield*. Die ID beschreibt also, dass das Element sich in der ELV-Konzentrator Sektion, in der Detailansicht befindet und es sich um das Textfeld namens Name handelt.

Abbildung 23 GUI-Ausschnitt der die Maske zum Anlegen eines neuen ELV-Konzentrators zeigt. Zusätzlich wird mit Hilfe der Chrome DevTools die HTML ID des Name Textfeldes gezeigt.

Um nun mit dem Test-Editor auf das Textfeld zugreifen zu können müssen im entsprechenden WEAT EABR Test-Editor Projekt einige Konfigurationen vorgenommen werden. Als erster Schritt werden in der Datei *ElementList.conf* einige Konstanten zur

Verwendung in anderen Konfigurationsdateien definiert. Diese Datei verbindet Elemente einer GUI, in diesem Fall einer Webseite, mit leserlichen Namen, die in der *AllActionGroups.xml* verwendet werden. Listing 5 zeigt den Ausschnitt aus der *ElementList.conf*, der das Anlegen eines ELV-Konzentrators betrifft.

```
navigation-edds_concentrators-button    = navigation-edds_concentrators-button
eddsconcentrators-list-create-button    = eddsconcentrators-list-create-button
eddsconcentrator-details-back-button    = eddsconcentrator-details-back-button
eddsconcentrator-details-edit-button    = eddsconcentrator-details-edit-button
eddsconcentrator-details-cancel-button  = eddsconcentrator-details-cancel-button
eddsconcentrator-details-save-button    = eddsconcentrator-details-save-button
eddsconcentrator-details-name-textfield = eddsconcentrator-details-name-textfield
eddsconcentrator-details-code-textfield = eddsconcentrator-details-code-textfield
```

Listing 5 ElementList.conf Einträge für ELV-Konzentratoren

Zu sehen sind zur Linken die Namen der Konstanten die anlegt wurden und zur Rechten die HTML IDs die zugewiesen werden. In diesem Fall sind die Namen exakt gleich, da bereits sinnvolle Namen für die HTML IDs verwendet werden. Es ist aber z.B. auch vorstellbar, dass sich auf der rechten Seite XPath Elemente oder unleserliche HTML IDs befinden, die dann einen lesbaren Namen bekommen.
Diese Konstanten können dann, und wirklich auch nur, wenn sie in der *ElementList.conf* definiert sind, in der Datei *AllActionGroups.xml* referenziert werden. In dieser Datei wird angegeben unter welcher Bezeichnung und in welcher Kategorie die oben definierten Konstanten in der GUI angezeigt werden und welche Operationen auf ihnen ausgeführt werden können.
Klarer wird das Ganze bei der Betrachtung des für den Moment wichtigen Auszugs aus der *AllActionGroups.xml* in Listing 6 und dem entsprechenden Test-Editor GUI Ausschnitt zum Anlegen eines neuen Testschrittes in Abbildung 24.

```
<!--
#########################################################################
# ELV-Konzentratoren - Details view
######################################################################### -->
<ActionGroup name="ELV-Konzentrator anlegen">
    <action technicalBindingType="Klicke_Element">
        <actionName locator="eddsconcentrator-details-back-button">
            Zurück
        </actionName>
    </action>
    <action technicalBindingType="Klicke_Element">
        <actionName locator="eddsconcentrator-details-edit-button">
            Editieren
        </actionName>
    </action>
    <action technicalBindingType="Klicke_Element">
        <actionName locator="eddsconcentrator-details-cancel-button">
            Abbrechen
        </actionName>
    </action>
    <action technicalBindingType="Klicke_Element">
        <actionName locator="eddsconcentrator-details-save-button">
            Speichern
        </actionName>
    </action>
    <action technicalBindingType="Eingabe_Wert">
        <actionName locator="eddsconcentrator-details-name-textfield">
            Name
        </actionName>
    </action>
    <action technicalBindingType="Eingabe_Wert">
        <actionName locator="eddsconcentrator-details-code-textfield">
            Routingkennzeichen
        </actionName>
    </action>
</ActionGroup>
```

Listing 6 Ausschnitt aus *AllActionGroups.xml*. Gezeigt werden die Einträge die das Erstellen eines neuen *ELV*-Konzentrators betreffen.

Dabei korrespondiert die *ActionGroup* mit der *Mask*. Der *actionName* referenziert mit dem *locator* die in der *ElementList.conf* aus Listing 5 angelegten Konstanten und damit indirekt die GUI Elemente. Die *action* gibt mit Hilfe des *technicalBindingType* an welche Optionen in dem *Step* Dropdownmenü zur Verfügung stehen.

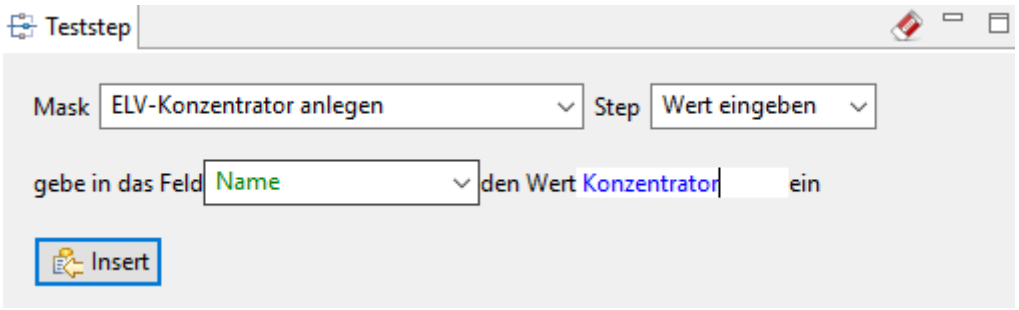

Abbildung 24 Teil der Test-Editor GUI. Zeigt das Erstellen eines neuen Testschrittes, der in das ELV-Konzentrator Feld *Name* den Wert *Konzentrator* eingibt.

Die beiden *technicalBindingTypes Klicke_Element* und *Eingabe_Wert* werden mit dem Test-Editor ausgeliefert und können somit in der *AllActionGroups.xml* ohne Weiteres verwendet werden. Definiert sind diese in der Konfigurationsdatei *TechnicalBindingTypeCollection.xml*. Der relevante Ausschnitt ist in Listing 7 zu sehen.

```
<TechnicalBindingType id="Klicke_Element" name="Element klicken">
    <actionPart position="1" type="TEXT" value="klicke auf"/>
    <actionPart position="2" type="ACTION_NAME"/>
</TechnicalBindingType>
<TechnicalBindingType id="Eingabe_Wert" name="Wert eingeben">
    <actionPart position="1" type="TEXT" value="gebe in das Feld"/>
    <actionPart position="2" type="ACTION_NAME"/>
    <actionPart position="3" type="TEXT" value="den Wert"/>
    <actionPart position="4" type="ARGUMENT"/>
    <actionPart position="5" type="TEXT" value="ein"/>
</TechnicalBindingType>
```

Listing 7 Ausschnitt aus TechnicalBindingTypeCollection.xml der für das Anlegen eines ELV-Konzentrators relevant ist.

Mit Hilfe der *id* wird aus der *AllActionGroups.xml* ein *TechnicalBindingType* referenziert. Das Attribut *name* zeigt an wie in der Test-Editor GUI in Abbildung 24 der *TechnicalBindingType* angezeigt wird. Hier wird der *TechnicalBindingType Eingabe_Wert* als *Wert eingeben* angezeigt.
Die *actionPart* Elemente legen fest wie die untere Zeile in der GUI aussieht. Die *position* ordnet die Elemente von links nach rechts an. Der *type* hat folgende Optionen:

- ***TEXT*** Elemente dienen der besseren Lesbarkeit der Testschritte und an ihrer Stelle wird lediglich der in *value* enthaltene Text angezeigt. Mit Hilfe dieser Elemente kann ein verständlicher Satz formuliert werden.
- ***ACTION_NAME*** referenziert den Namen der *action*, die den *TechnicalBindingType* verwendet. In diesem Fall ist das *Name*. Das Dropdownmenü bietet jedoch auch die Auswahl *Routingkennzeichen*, da sich diese *action* in derselben *ActionGroup* befindet und ebenfalls den *TechnicalBindingType Eingabe_Wert* besitzt.

- ***ARGUMENT*** bietet die Möglichkeit ein Argument zu übergeben. In der Test-Editor GUI wird diese wie in Abbildung 24 zu sehen, als Textfeld angezeigt. In diesem Fall handelt es sich um den Text *Konzentrator*, der in das Textfeld *Name* eingegeben werden soll.

Wird der Testschritt mit dem *Insert* Button unten links in Abbildung 24 bestätigt, so wird der Testschritt dem Testablauf hinzugefügt und im Editor in der Mitte der Test-Editor GUI angezeigt. Abbildung 25 zeigt den entsprechenden Ausschnitt.

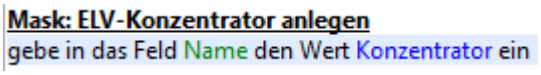

Abbildung 25 Ausschnitt aus der Test-Editor GUI. Zeigt einen Schritt um in das Feld *Name* den Wert *Konzentrator* einzugeben.

In diesem Beispiel ist weder gezeigt wie auf die Seite mit der Maske zum Anlegen eines ELV-Konzentrators navigiert wird, noch wie das Routingkennzeichen eingetragen oder der Speichern Button aus Abbildung 23 gedrückt wird. Es veranschaulicht jedoch den Prozess der vollzogen werden muss, um einen neuen Testschritt anzulegen, der einen bereits existierenden *TechnicalBindingType* verwendet.
Dazu sei erwähnt, dass diese im Beispiel erstellte Möglichkeit einen ELV-Konzentrator anzulegen, nun in vielen verschiedenen Tests genutzt werden könnte, ohne nochmal die Konfigurationsdateien zu ändern. Dies ist Teil des Grundgedankens hinter dem Test-Editor. Ein Entwickler oder eine anderweitig technisch versierte Person, pflegt beispielsweise eine Webseite in die Test-Editor Konfiguration einmalig ein und der Tester kann sich die Tests dann „zusammenklicken“ ohne sich mit der Konfiguration auseinandersetzen zu müssen.

5.9 Dateioperationen

Neben dem Herstellen eines definierten Systemzustandes in Abschnitt 5.7 und den Interaktionen mit der GUI in Abschnitt 5.8 wurde als dritte Kategorie die Dateioperationen gruppiert. Dazu werden Dateiimport, der Vergleich von Verzeichnisstrukturen sowie der Vergleich von Dateiinhalten gezählt. Diese werden in den folgenden drei Abschnitten erläutert.
Dabei werden alle Daten die für den Test in das System eingespeist werden und alle Referenzdateien zum Abgleich der Ergebnisse aus dem speziell angelegten Testdatenverzeichnis geladen. Hierbei handelt es sich, wie weiter oben erwähnt, um einen auf dem Dateisystem befindlichen Ordner auf den aus der Fixture heraus zugegriffen werden kann.

5.9.1 Dateiimporte

Abgesehen von den Daten die über die Weboberfläche in die Datenbank eingepflegt werden, gibt es andere Daten die über Dateiimporte ins System gelangen. Diese Daten sind für den Rechnungslauftest bereits vorgegeben und können aus dem internen WEAT EABR Wiki heruntergeladen werden.
Importiert werden können die Dateien entweder über einen GUI Dialog der Weboberfläche oder aber durch das Kopieren der entsprechenden Dateien in das Importverzeichnis des Servers. Für den hier implementierten Test wird ausschließlich die Variante mit dem Importverzeichnis verwendet. Das hat den Grund, dass der Importdialog der GUI sehr plattformspezifisch ist und mit Hilfe des Test-Editors nur sehr schwer zu testen ist.
Der Dateiimport wird automatisch gestartet, sobald die Datei in das Importverzeichnis kopiert worden ist. Mit Hilfe der in Abschnitt 5.8.4 gezeigten Methode zum wiederholten prüfen eines Tabelleneintrags wird gewartet bis der Import abgelaufen ist und anschließend geprüft ob die in der Tabelle angezeigten Metadaten den erwarteten Daten entsprechen. So wird z.B. die Anzahl der Datensätze oder der Dateiname mit Referenzen abgeglichen.
Der Rechnungslauf verlangt das Importieren verschiedenartiger Dateien, wie beispielsweise Transaktionen oder Bankleitzahlen und weiteren, diese variieren für die Testzwecke jedoch lediglich durch das Zielimportverzeichnis sowie durch die entsprechende Webseite der WEAT EABR GUI mit der richtigen Tabelle für die Art des Imports.

5.9.2 Abgleich der Exportverzeichnisstruktur

Nachdem im Rechnungslauftest ein definierter Zustand herstellt wurde, alle nötigen Daten über die GUI Masken eingepflegt worden sind und alle Importe erfolgreich beendet wurden, startet automatisch die Entgeltberechnung für den Testmonat. Nach erfolgreichem Abschluss dieser müssen die Ergebnisse der Berechnung kontrolliert werden. Diese liegen in Form von Dateien im Exportverzeichnis vor. Dabei erläutert Abschnitt 5.9.3 den Vergleich der einzelnen Dateien. Zuvor wird jedoch die komplette Verzeichnisstruktur des Exportverzeichnisses mit einer Referenz abgeglichen.
Unter anderem zu diesem Zweck wurde eine Baumdatenstruktur, genannt *TreeNode* entwickelt, welche ein Dateiverzeichnis als Baum abbildet. In eine solche Datenstruktur wird der Inhalt einer mit Tabs eingerückten Textdatei, welche die erwartete Exportverzeichnisstruktur darstellt, eingelesen. Diese Struktur wird nun mit dem aktuellen Exportverzeichnis des Tests verglichen. Dabei werden lediglich die relativen Pfade der Dateien und Verzeichnisse abgeglichen, nicht jedoch die Dateiinhalte.

5.9.3 Dateivergleiche

Da nach dem im vorherigen Abschnitt beschriebenen Vergleich des Exportverzeichnisses mit einer Referenz nun davon ausgegangen werden kann, dass alle erwarteten Dateien

existieren, werden jetzt die Inhalte der einzelnen Dateien ebenfalls mit Referenzen abgeglichen. Dabei handelt es sich um 17 Dateien und drei Typen von Dateien, jedoch steht für alle eine Referenzdatei zum Vergleich zur Verfügung.
Der Vergleich geschieht Zeilenweise, da es im Fehlerfall, wenn eine Zeile aus der Referenz nicht genauso in der zu testenden Datei auftaucht, genügt den Tester auf die entsprechende Fehlerhafte Zeile aufmerksam zu machen.
Da einige Dateien beispielsweise das Datum ihrer Erstellung anzeigen, muss es möglich sein, bestimmte Teile vom Vergleich im Fixture Code auszuschließen, da die Referenzdateien zum Vergleich in jedem Fall ein älteres Datum enthalten.

5.9.4 Beispiel Dateivergleich

Dieses Beispiel hat zwei Ziele: Zum einen soll es zeigen, dass die Funktionalität des Test-Editors über die im Beispiel aus Abschnitt 5.8.5 gezeigten *Klicke_Element* und *Eingabe_Wert* hinaus erweitert werden kann. Zum anderen wird klar, dass der Test-Editor beliebige Operationen ausführen kann und nicht auf die Ansteuerung einer GUI beschränkt ist. Beide Sachverhalte sind für die Implementierung der Dateivergleiche und des Rechnungslauftests unabdingbar.
Es wird ein weiterer Schritt aus dem Rechnungslauftest betrachtet der gegen Ende des Tests eine beim Rechnungslauf entstandene und exportierte Datei mit einer Referenzdatei vergleicht und erwartet, dass die beiden Dateien identisch sind. Der entsprechende Auszug aus dem WEAT EABR Wiki ist in Abbildung 26 zu sehen. Auf der linken Seite ist der relative Pfad zur entstandenen Datei, relativ zum WEAT EABR Export Verzeichnis, zu sehen. Auf der rechten Seite kann im Wiki die Referenzdatei für den Vergleich heruntergeladen werden.

Prüfe, ob die folgenden beiden Dateien identischen Inhalt haben

Datei EABR	Referenzdatei
weat_stat/export/STAT1412	expected_STAT1412

Abbildung 26 Ausschnitt aus dem WEAT EABR Wiki, der einen Testschritt zum Vergleich zweier Dateien im Rechnungslauftest beschreibt.

Der genaue Inhalt der Dateien ist für das Testen nicht von Bedeutung. Es handelt sich um eine Übersicht über alle Transaktionen je Station und Art und enthält beispielsweise Informationen wie Betrag, Währung oder Tankstellennummer. Die Datei ist etwa 10000 Zeilen lang.
Da lediglich auf das Dateisystem zugegriffen wird, nicht aber auf die GUI, brauchen keine neuen Einträge in die *ElementList.conf* einpflegt werden und es kann stattdessen gleich mit der *AllActionGroups.xml* in Listing 8 begonnen werden.

```
<!--
########################################################################
# Vergleiche WEAT STAT Dateien
######################################################################## -->
<ActionGroup name="WEAT STAT Dateien vergleichen">
    <action technicalBindingType="WEAT_STAT_Dateien_Vergleichen">
        <actionName locator="">WEAT STAT Dateien vergleichen</actionName>
    </action>
</ActionGroup>
```

Listing 8 WEAT STAT Dateivergleich betreffender Eintrag in der AllActionGroups.xml Konfigurationsdatei.

Da kein Element aus der *ElementList.conf* referenziert wird kann der *locator* leer bleiben. In diesem Fall heißt die *ActionGroup* genau wie die *action* und bietet lediglich diese eine Auswahloption an.
In Abbildung 27 ist der zugehörige GUI Ausschnitt zum Erstellen eines Testschrittes gezeigt. Die in der unteren Zeile eingetragenen Werte entsprechen den Werten aus dem Wiki Eintrag in Abbildung 26. Die teils abgeschnittenen Werte sind lediglich ein Darstellungsproblem des Test-Editors, sie sind vollständig eingetragen.

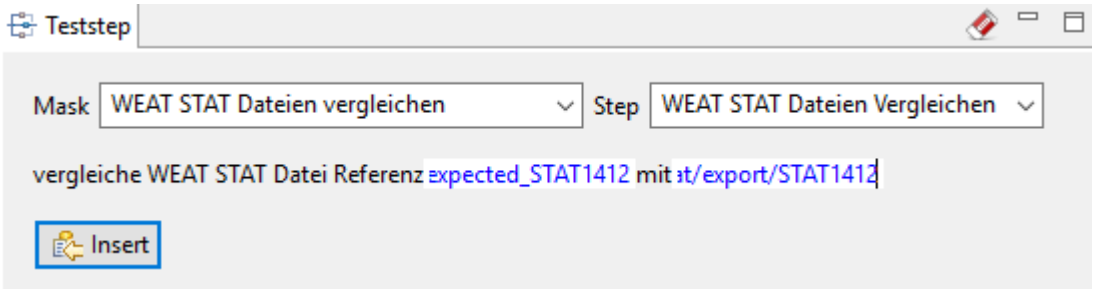

Abbildung 27 Ausschnitt aus der Test-Editor GUI, der das Erstellen eines Testschrittes zum Vergleich von WEAT STAT Dateien zeigt.

Die *ActionGroup* korrespondiert wieder mit der *Mask* und der *Step* mit dem *actionName*. Was noch fehlt ist ein Eintrag in der *TechnicalBindingTypeCollection.xml* der die untere Zeile in Abbildung 27 festlegt. Im Gegensatz zum vorangegangenen Beispiel zum Anlegen eines ELV-Konzentrators in Abschnitt 5.8.5, bei dem auf die mitgelieferten und somit bereits existierenden *TechnicalBindingTypes Klicke_Element* und *Eingabe_Wert* zugegriffen wurde, existiert das *WEAT_STAT_Dateien_Vergleichen* Binding noch nicht. Also muss der Eintrag neu erstellt werden (siehe Listing 9).

```
<TechnicalBindingType id="WEAT_STAT_Dateien_Vergleichen"
        name="WEAT STAT Dateien Vergleichen">
    <actionPart position="1" type="TEXT"
            value="vergleiche WEAT STAT Datei Referenz"/>
    <actionPart position="2" type="ARGUMENT"/>
    <actionPart position="3" type="TEXT" value="mit"/>
    <actionPart position="4" type="ARGUMENT"/>
</TechnicalBindingType>
```

Listing 9 Neu erstellter TechnicalBindingType zum Vergleich von WEAT STAT Dateien.

In diesem Fall wurde zweimal der *type ARGUMENT* (die in der GUI einzugebenden Pfade zu den Dateien) und keine *action* verwendet, da nicht auf ein GUI Element zugegriffen wird.
Allerdings ist der Eintrag nicht ausreichend um einen Testschritt auszuführen. Dafür muss die relevante Fixture des Test-Editors erweitert werden. In diesem Zusammenhang beschreibt die Fixture, wie erwähnt, den Java Code der angesprochen wird um die Testschritte auszuführen. In diesem Fall ist das die *HtmlWebFixture*, da diese die bereits genutzten Funktionen *Klicke_Element* und *Eingabe_Wert* enthält. Die Fixture wird um die in Listing 10 gezeigte Methode plus hier nicht aufgeführte benötigte Hilfsfunktionen erweitert.

```
public boolean compareWeatStatFiles (String referenceFile,
        String fileUnderTest) throws StopTestException {
    Path testDataPath = Paths.get(getSysProperty(WEAT_TESTDATA_PROPERTYNAME));
    Path reference = testDataPath.resolve(Paths.get(referenceFile));
    Path exportPath = Paths.get(getSysProperty(WEAT_EXPORT_POPERTYNAME));
    Path underTest = exportPath.resolve(Paths.get(fileUnderTest));

    return util.compareFiles(reference, underTest);
}
```

Listing 10 Einfache Java Methode aus der Fixture, welche zwei WEAT STAT Dateien vergleicht.

Die Fixturemethoden empfangen als Parameter *Strings* aus dem Test-Editor. In diesem Fall sind das *referenceFile=expected_STAT1412* und *fileUnderTest=weat_stat/export/STAT1412*.
Am Anfang der Methode werden die benötigten Pfade zusammengesetzt. Dabei ist *getSysProperty* eine Helfermethode die über *System.getProperty* eine Systemvariable abfragt. *WEAT_TESTDATA_PROPERTYNAME* ist eine Konstante die den Namen der *Property* enthält, welche den Pfad zum Verzeichnis mit den Testdaten enthält. Dieses Testdatenverzeichnis enthält etwa die Referenzdatei *expected_STAT1412*. Existiert die *Property* wird mit Hilfe von *Paths.get* ein entsprechendes *Path* Objekt erstellt und in *testDataPath* gespeichert.
Im nächsten Schritt wird über die *resolve* Methode der komplette Pfad zur Referenzdatei zusammengesetzt. Liegen die Testdaten beispielsweise in */path/to/testdata*, so enthält der Path *reference* nun */path/to/testdata/expected_STAT1412*.

In den folgenden beiden Zeilen passiert das gleiche für die zu prüfende Datei, nur dass der WEAT EABR Exportpfad als Basis dient. Der *underTest Path* enthält also beispielsweise */path/to/weateabr/serverdata/export/weat_stat/export/STAT1412*.
In der letzten Zeile werden nun mit der auch in anderen Testschritten verwendeten Helfermethode *compareFiles* die Inhalte beider Dateien verglichen und das Ergebnis zurückgeliefert.
Ist die Java Fixture kompiliert und in das entsprechende Test-Editor Verzeichnis kopiert worden, fehlt noch ein weiterer Schritt zum Verbinden der Fixture Methode mit dem Test-Editor. Listing 11 zeigt den benötigten Eintrag in die *content.txt* Datei aus dem Order *ScenarioLibrary*.

```
'''HtmlWebFixture - WEAT STAT Dateien Vergleichen(WEAT_STAT_Dateien_Vergleichen)'''
!|scenario|vergleiche WEAT STAT Datei Referenz|reference|mit|undertest|
|compareWeatStatFiles;|@reference|@undertest|
```

Listing 11 Eintrag aus der content.txt Datei der ScenarioLibrary. Der Eintrag betrifft das Vergleichen von WEAT STAT Dateien.

Hierbei handelt es sich eine Datei die direkt zum vom Test-Editor genutzten FitNesse gehört. *HtmlWebFixture* bezeichnet den Namen der erweiterten Fixture Klasse. Es folgt der Name des *TechnicalBindingType* aus Listing 9, gefolgt von der in Klammern gesetzten entsprechenden *id* des *TechnicalBindingType*.
Zeile zwei entspricht den *actionParts*, lediglich mit dem Wort *scenario* vorangestellt, mit | getrennt und die *ARGUMENT*s besitzen beliebige Namen.
Zeile drei beginnt mit dem Namen der erstellten Java Fixture Methode, gefolgt von den beiden, Zeile zwei referenzierenden, Argumenten. Diese entsprechen den Parametern der Java Methode in der Fixture.
Sind keine Fehler gemacht worden kann nun der in Abbildung 28 gezeigte Testschritt ausgeführt werden.

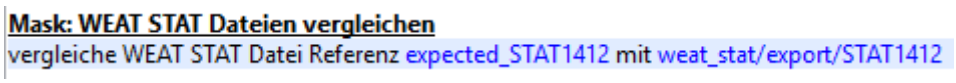

Abbildung 28 Auszug aus der Test-Editor GUI. Zeigt den erstellten Testschritt zum Vergleich zweier WEAT STAT Dateien.

Bei fehlerfreier Durchführung zeigt der Test-Editor das Ergebnis in Abbildung 29. Dabei sei klargestellt, dass für diesen Testschritt allein kein Browser gestartet werden würde, da nicht mit der GUI interagiert wird.

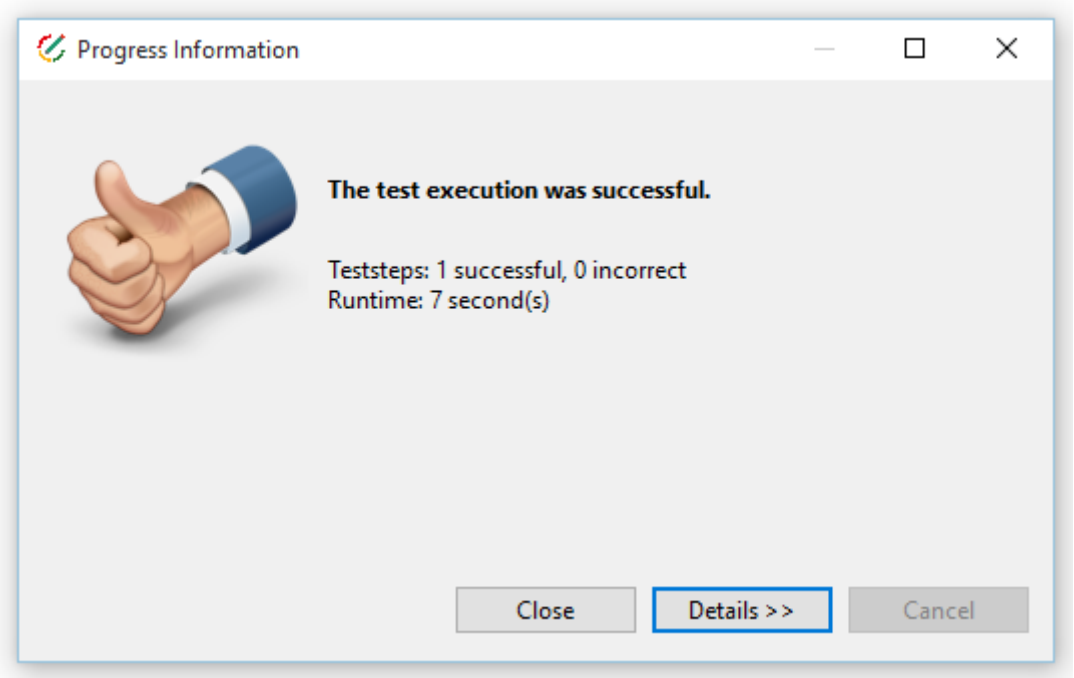

Abbildung 29 Test-Editor Ergebnisanzeige eines erfolgreich durchgelaufenen Tests.

5.10 Szenarien für wiederverwendete Schrittfolgen

Bestimmte Abfolgen von mehreren Testschritten kommen immer wieder vor, somit ist es sinnvoll diese eine Art Funktion auszulagern. Im Fall des Test-Editors sind das Szenarien. Ein Szenario enthält eine Abfolge von Testschritten. Dieses Szenario selbst kann dann wiederum als Testschritt in einen Test eingefügt werden.
Für den Rechnungslauftest wurden mehrere verschiedene Szenarien für beispielsweise den Login in das System, das Ausfüllen von Masken (Verträge etc) und das Importieren von Dateien angelegt.

5.11 Beispielhafte Integration in die Pipeline

Nach der Automatisierung des Rechnungslauf Akzeptanztests ist es sinnvoll über die Integration des Tests in die WEAT EABR Deployment Pipeline nachzudenken. Diese Deployment Pipeline ist mit Hilfe von Jenkins realisiert. Jenkins (Jen15) ist ein Continuous Integration und Delivery Server, der es ermöglicht die einzelnen Phasen des Software Delivery Prozesses zu automatisieren.
Hier wird angerissen wie der Test aus einem Jenkins CI Server gestartet werden kann. Dazu wurde ein lokaler Jenkins Server auf dem Entwicklungsrechner für die vorliegende Arbeit

installiert. Zusätzlich wurde das Jenkins FitNesse Plugin (Jen16) installiert, dass den Prozess des Ausführens von FitNesse Tests etwas vereinfacht und nach Durchlauf der Tests einen Report zu Anzeige in Jenkins generiert.
Der Test-Editor spielt in diesem Zusammenhang keine Rolle mehr und diente lediglich als Unterstützung zur Testfallerstellung. Der erstellte Rechnungslauftest ist ein FitNesse Test und kann somit komplett ohne den Test-Editor ausgeführt werden. Dies ist der Grund für die getrennte Darstellung zwischen Test-Editor und FitNesse im bisherigen Verlauf der Arbeit.
Abbildung 30 zeigt das Zusammenspiel von Jenkins, FitNesse und WEAT EABR. Zur besseren Übersicht sind die Interaktionen (über die Java Fixture) zwischen dem FitNesse Server und WEAT EABR aus Abbildung 13 nicht erneut dargestellt, aber trotzdem alle WEAT EABR Komponenten gezeigt.

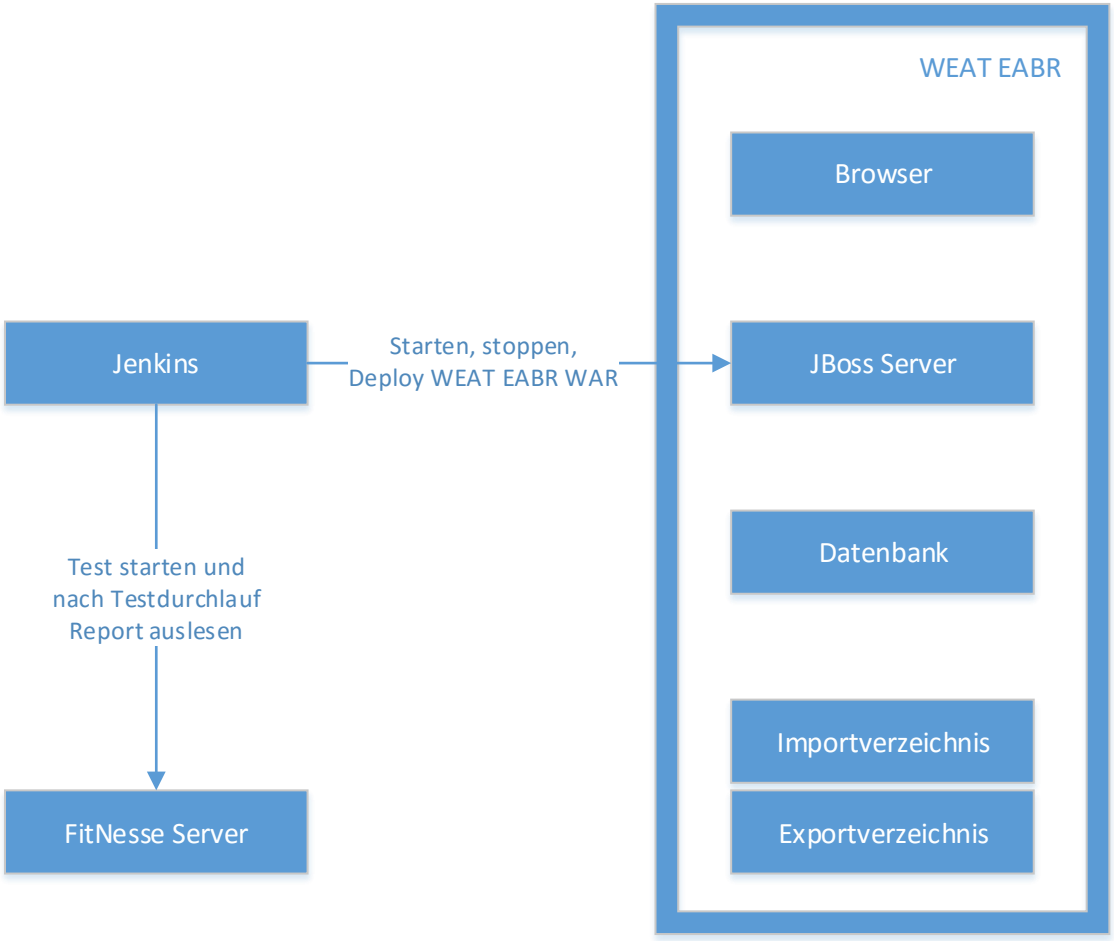

Abbildung 30 Zusammenhang zwischen Jenkins und WEAT EABR sowie Jenkins und dem FitNesse Server.

Der zeitliche Ablauf der Testdurchführung durch Jenkins ist wie folgt:

1. JBoss Server starten
2. WEAT EABR WAR deployen
3. Rechnungslauftest auf dem FitNesse Server starten
4. Report auslesen
5. JBoss Server herunterfahren

Für die gezeigte Abfolge wurde ein Jenkins Job erstellt. Die Schritte 1,2 und 5 sind mit Hilfe von Skripten (für Linux und Windows), die aus dem Jenkins Job aufgerufen werden realisiert. Listing 12 zeigt beispielhaft das Herunterfahren des JBoss Servers aus einer Batch Datei. Dabei wird der Server über das JBoss Command Line Interface (CLI) angesprochen. Diese Batchdatei wird für Schritt 5 aus dem Jenkins Job heraus aufgerufen, zu sehen in Abbildung 31.

```
cd D:\weat\jboss-eap-6.3\bin

./jboss-cli.bat --connect command=:shutdown
```

Listing 12 Batch Datei zum Herunterfahren des JBoss Servers per Skript.

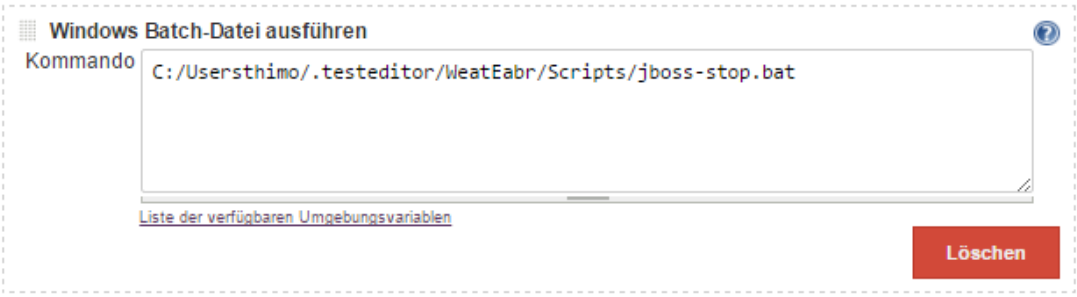

Abbildung 31 Der Build Schritt aus dem Jenkins Job zum Aufrufen des Skripts, dass den JBoss Server herunterfährt.

Schritte 3 und 4 sind mit Hilfe des Jenkins FitNesse Plugins implementiert.
Der benötigte FitNesse Server wird einmalig manuell vorher gestartet und ist ein unveränderlicher Bestandteil. Im Gegensatz dazu soll das WEAT EABR WAR sowie der JBoss Server später in der WEAT EABR Deployment Pipeline immer der neuesten Version aus dem VCS entsprechen. D.h. der JBoss Server muss aus Jenkins gestartet werden und das WEAT EABR WAR auch von hier auf den JBoss Server deployed werden.

5.11.1 Report des Testdurchlaufs

Mit Hilfe des Jenkins FitNesse Plugins wird für die ausgeführten Tests ein Report angezeigt. Abbildung 32 zeigt die Übersicht in der alle durchgelaufenen Tests angezeigt werden. In diesem Fall ist nur ein Test mit dem Namen *BrowserStarten* gelaufen, der etwa 15 Sekunden benötigt hat und erfolgreich abgeschlossen wurde.

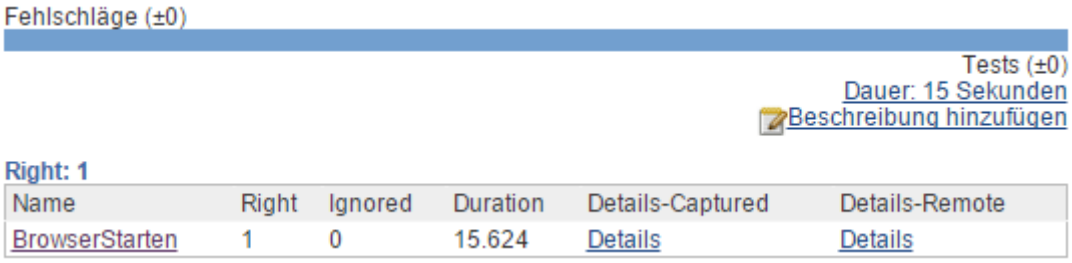

Abbildung 32 In Jenkins angezeigter Report nach dem Durchlaufen eines Tests der nur den Browser startet.

Unter *Details-Captured* können die Details des Testdurchlaufs mit den einzelnen Schritten eingesehen werden. In Abbildung 33 sind die Details des *BrowserStarten* Testdurchlaufs zu sehen. Im unteren Teil werden die einzelnen Testschritte angezeigt. Um das Beispiel kurz zu halten wurde hier nur der Browser gestartet. Für den Fall, dass ein Testschritt nicht erfolgreich durchgeführt wurde, wäre er hier entsprechend rot eingefärbt.

Abbildung 33 Detailansicht für den *BrowserStarten* Test aus dem Jenkins FitNesse Plugin Test Report.

5.12 Zeitlicher Vorher-Nachher-Vergleich

Für einen zeitlichen Vergleich zwischen der manuellen und der automatisierten Durchführung des Rechnungslauftests wurde der Test einmal manuell mit ungefähren Zeitmessungen und einmal automatisiert mit Zeitmessungen durchgeführt. Ein manueller Durchlauf des kompletten Tests dauert ca. 1 Stunde. Der automatisierte Durchlauf etwa 23 Minuten.
Der Test-Editor wartet nach jedem Testschritt (z.B. Button klicken, Textfeld ausfüllen, Datenbank zurücksetzten etc.) eine gewisse Zeit. Für diesen automatisierten Durchlauf wurde ein Intervall von 0,8 Sekunden gewählt. D.h. bei einigen Schritten wird ein Großteil der Zeit, wenn beispielsweise viele Textfelder ausgefüllt werden, mit Warten verbracht. Die Zeiten können in dieser Hinsicht also noch optimiert leicht werden.
Die manuelle Zeit ist lediglich ein grober Richtwert, da jeder Tester unterschiedlich schnell und genau arbeitet. Der hier gezeigte Vergleich ist also nur als Tendenz zu werten. Sehr rechenintensive Schritte wie das Importieren der vielen Transaktionen oder das Berechnen des Entgeltes sind sowohl manuell als auch automatisiert sehr zeitaufwendig, da die Berechnung oder der eigentliche Import der Daten den größten Teil der Zeit in Anspruch nimmt und nicht das Navigieren auf der GUI.
Andere Schritte hingegen, wie etwa das genaue Prüfen von Verzeichnisstrukturen und Dateiinhalten zeigen einen deutlichen Zeitvorteil für die automatisierte Variante. Auch für das manuelle Vergleichen von Dateiinhalten muss ein Diff-Tool verwendet werden, da die Dateien teilweise etwa 10000 Zeilen enthalten.
Die automatisierte Variante benötigt also etwa die Hälfte der Zeit. Selbst mit einer kürzeren Wartezeit der Schritte ist der Test nicht unter 15 Minuten zu bringen. Der Löwenanteil bleibt durch die Importe und Berechnungen bestehen. Hier hilft voraussichtlich, falls tatsächlich gewünscht, nur eine Minimierung der Testdatenmenge um die Importprozesse und in der Folge auch die Entgeltberechnung zu beschleunigen.

6 Related Work

Um für den Leser dieser Arbeit einen roten Faden zu wahren und nicht vom eigentlichen Kern abzulenken, wurden einige weiterführende Deployment Pipeline Konzepte und Hintergründe ausgelassen. In diesem Kapitel wird dies nachgeholt und der Kontext der Deployment Pipeline sowie weiterführende Konzepte erläutert.
Das Kapitel ist völlig losgelöst vom praktischen Teil dieser Arbeit und dient in erster Linie dazu einen Kontext zu vermitteln, einen Trend in der Softwareentwicklung zu beschreiben und Inspiration für neue Arbeiten zu liefern.

6.1 Continuous Delivery und Continuous Deployment

Der Begriff der Deployment Pipeline ist eng verbandelt mit dem Continuous Delivery Konzept. Continuous Delivery beschreibt eine Vorgehensweise Software in kurzen Zyklen so zu entwickeln, sodass es jederzeit möglich ist sie verlässlich in eine Produktionsumgebung zu deployen (Humble, et al., 2011). Der Kern hierbei ist Automation aller möglichen Vorgänge und das Arbeiten nach dem Motto: „If it's hard, do it often". Somit wird der Fokus auf die Verbesserung der komplizierten, und damit meist fehleranfälligen und langsamen, Vorgänge gelenkt.
Continuous Delivery wird manchmal mit Continuous Deployment verwechselt. Letzteres bedeutet, dass jede Änderung an der Software die erfolgreich durch die automatisierte Pipeline läuft, auch automatisch deployed wird. Continuous Delivery beschreibt dagegen nur, dass es möglich ist zu jedem Zeitpunkt mehrmals am Tag zu deployen, aber bewusst nicht automatisch jede Änderung deployed wird. Um Continuous Deployment umsetzten zu können ist es nötig Continuous Delivery zu implementieren. Im Folgenden geht es um beide Konzept, der Schwerpunkt liegt jedoch auf Continuous Deployment, da das Konzept etwas weiterführend ist und Continuous Delivery beinhaltet. Der Unterschied beider ist nochmal in Abbildung 34 dargestellt.
Im weiteren Verlauf dieses Abschnitts, wird auf die Vorteile, die Herausforderungen und die mit Continuous Delivery/Deployment im Zusammenhang stehenden Praktiken eingegangen um ein besseres Verständnis der Konzepte zu vermitteln.

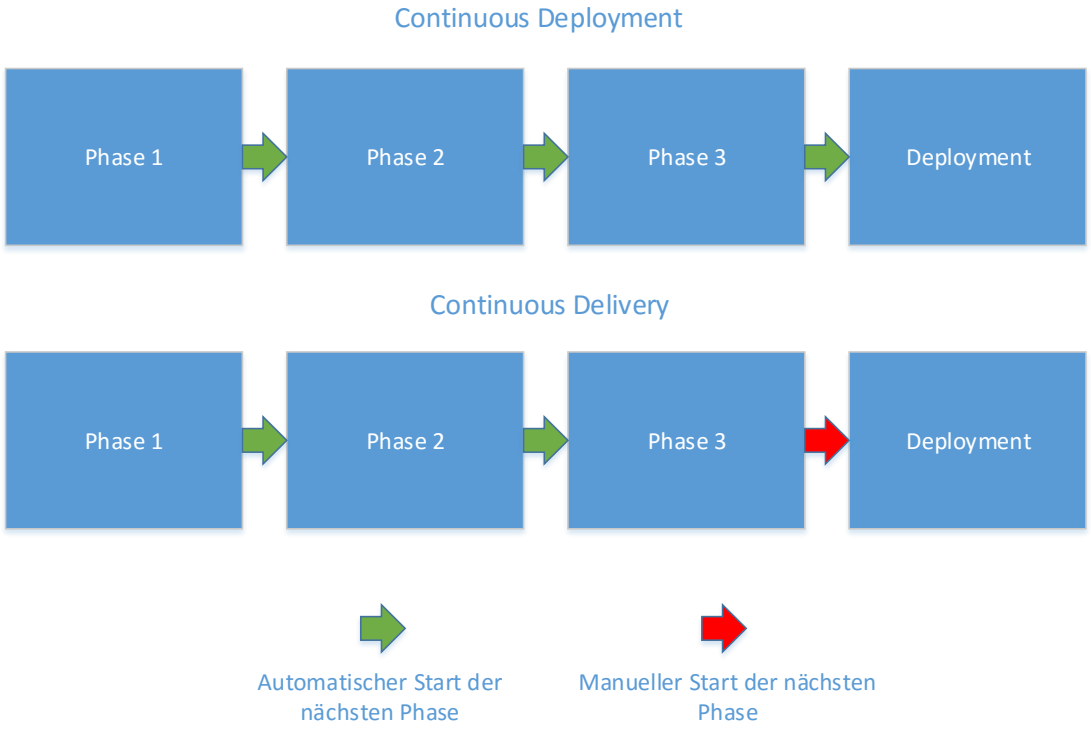

Abbildung 34 Unterschied zwischen Continuous Deployment und Continuous Delivery.

6.1.1 Vorteile

Um die Adoption von Continuous Delivery oder Continuous Deployment zu rechtfertigen, bedarf es dem Wissen um die Vorteile. Dabei basieren viele der hier aufgelisteten Vorteile auf empirischen Umfragen oder Erfahrungsberichten und bieten lediglich einen Anhaltspunkt.

Beschleunigte time to market
Die Verwendung einer größtenteils automatisierten Deployment Pipeline führt, wenn gewünscht, zu deutlich häufigeren Releases. Durch das häufige releasen wird der Releaseprozess immer weiter optimiert und gefestigt (Chen, 2015) und dadurch von Mitarbeitern oft als weniger verschwenderisch wahrgenommen (Leppänen, et al., 2015).

Entwicklung des richtigen Produkts durch schnelles Feedback
Häufige Releases führen zu schnellem Kundenfeedback (Leppänen, et al., 2015). Dadurch wird seltener Zeit in Features investiert, die beim Kunden keinen Anklang finden. Dies steht im Gegensatz zu seltenen Releases, etwa alle paar Monate bis Jahre. Hierbei kann es leicht zu einer Entwicklung in die falsche Richtung kommen (Chen, 2015).

Verbesserte Produktivität und Effizienz (Leppänen, et al., 2015)
Entwickler und Tester verbringen kaum Zeit mehr damit ihre Testumgebungen aufzusetzen und zu warten, da dies automatisiert geschieht. Ebenso wird durch das häufige automatisierte releasen Zeit für die Problembehebung bei komplexen Releaseprozessen eingespart (Chen, 2015).

Verlässliche Releases
Das Risiko für Fehler beim Release wird deutlich gesenkt. Die Abläufe sind durch das häufige Durchführen besser getestet und viele Probleme bereits behoben. Häufiges releasen bedeutet ebenso kleine Releases mit weniger Änderungen (Commits). Zusätzlich sollte es möglich sein per Knopfdruck ein Rollback auszuführen und somit in der Produktionsumgebung im dringenden Fehlerfall eine vorige Version wiederherstellen zu können (Chen, 2015).

Verbesserte Produktqualität (Leppänen, et al., 2015)
Jede Änderung am Source Code muss die komplette Pipeline erfolgreich durchlaufen und wird somit gründlich getestet. Hierdurch gibt es beim Kunden deutlich weniger Bugs. Gefundene Bugs werden durch die häufigen Releases deutlich schneller für den Kunden behoben (Chen, 2015).

Verbesserte Work-Life-Balance
Die großen aufwendigen, stressigen und fehleranfälligen Releases sind Vergangenheit (Chen, 2015) (Neely, et al., 2013).

Einfacheres Einarbeiten
Die vielen automatischen Tests, die Möglichkeit schnell Fixes zu deployen und notfalls ein Rollback durchzuführen hilft dabei neuen Mitarbeitern die Angst davor zu nehmen ihren Code zu veröffentlichen (Neely, et al., 2013).

Experimentierfreudigeres Arbeiten
Die Sicherheit von Rollbacks ermöglicht es den Entwicklern zu experimentieren und eine weniger konservative Herangehensweise an den Tag zu legen (Neely, et al., 2013). Ebenso trägt das schnelle Kundenfeedback zu experimentierfreudigerem Arbeiten bei (Leppänen, et al., 2015). Ein Beispiel hierfür sind A/B Tests. Hierbei werden verschiedenen Nutzergruppen z.B. verschiedene Versionen eines Features bereitgestellt und im Anschluss ausgewertet welche Version dem Nutzer einen größeren Mehrwert bringt oder welche leichter zu verwenden ist.

Erhöhte Kundenzufriedenheit
Die vorangegangenen Punkte erhöhen deutlich die Kundenzufriedenheit (Chen, 2015) und ermöglicht durch die häufigeren Releases sowie das schnellere Kundenfeedback einen besseren, zeitigeren Kundenservice (Leppänen, et al., 2015).

6.1.2 Praktiken

Nachdem die wahrgenommenen Vorteile von Continuous Delivery/Deployment begutachtet wurden, werden hier verwendete Praktiken zur Umsetzung derer erläutert. In (Rahman, et al., 2015) fassen die Autoren Praktiken im Zusammenhang mit Continuous Deployment zusammen. Dafür wurden 19 Unternehmen untersucht (Facebook, Google Flickr, GitHub u.a.) um gemeinsame Vorgehensweisen aufzudecken. Einige der aufgeführten Punkte erscheinen offensichtlich, andere hingegen sind eher neu. Ebenfalls werden die Praktiken nicht exklusiv im Zusammenhang mit Continuous Deployment verwendet. Nicht alle aufgelisteten Praktiken werden von allen untersuchten Unternehmen verwendet. Weiterhin sind nicht alle Praktiken für alle Arten von Software geeignet. Eine speziell zugeschnittene Software für einige wenige Nutzer in einem Unternehmen in einem stark regulierten Markt bietet andere Herausforderungen als eine Webapp für Millionen von Nutzern. Nichtsdestotrotz können viele der folgenden Praktiken von Nutzen sein um ein Unternehmen näher an Continuous Delivery oder Continuous Deployment zu bringen:

Automated Deployment:
Die Software wird dem Endnutzer automatisch zur Verfügung stellen. Das Release geschieht also nicht durch manuelle Abläufe.

Automated Testing:
Das automatische Durchführen von Tests oder anderen Testaspekte wie etwa Testgeneration oder Testdatengeneration. Ein interessanter Punkt hierbei ist, dass die meisten der untersuchten Unternehmen kein separates Testteam für Continuous Deployment haben.

Code Review:
Die Voraussetzung, dass Entwickler Codeänderungen den Kommentaren anderer aussetzen und einer expliziten Genehmigung bedürfen, bevor die Änderung übernommen wird. Dies erhöht die Qualität, findet Defekte, findet Alternativvorschläge und dient ebenfalls dazu gute Programmierpraktiken im Unternehmen zu verbreiten.

Dark Launching:
Das deployen von Änderungen am Code, welche vor dem Endnutzer versteckt sind. Die Motivation hinter dieser Praxis ist es Feedback über die Qualität und Performance von Änderungen aus der Produktivumgebung zu bekommen, z.B. mit Hilfe von Dummydaten, ohne den Endnutzer zu beeinflussen. Ebenso genannt in (Neely, et al., 2013).

End-User Communication:
Das Nutzen von Kommunikation mit dem Endnutzer um Feedback und Anforderungen an die Software sammeln. Dies kann über Telefonate, Foren oder andere

Kommunikationskanäle geschehen. So ist es für bestimmte Produkte etwa sinnvoll Feedback aus sozialen Netzwerken zu sammeln.

Feature Flag:
Wird auch als *Feature Toggle* oder *Feature Flipper* bezeichnet. Dies beschreibt die Praxis Features und Änderungen mit Hilfe konditioneller Logik an- und ausschaltbar zu machen. Ist die Bedingung erfüllt wird der entsprechende Zweig ausgeführt. So kann etwa in der Produktion ein fehlerhaftes Feature einfach ausgeschaltet werden oder durch eine andere bestehende Featureimplementierung ersetzt werden. Zusätzlich können unfertige Features deployed werden (z.b. durch Dark Launching) und auf einfache Art und Weise an- und ausgeschaltet werden. Ebenso erwähnt in (Neely, et al., 2013)**.**

Intercommunication:
Ausgeprägte Kommunikation zwischen Teammitgliedern. Dieser Punkt spielt bei örtlich verteilten Teams eine größere Rolle und kann etwa mit Hilfe von Chats und einer Codereviewplattform, wie z.B. Gerrit (Ger16), verbessert werden.

Monitoring:
Das Sammeln und strukturierte Auswerten von Deployment- und Nutzungsdaten. Dies hilft beim Aufstöbern von Fehlerquellen sowie beim schnellen Auswerten der Featureadoption oder anderen Nutzungsmetriken. Ebenso genannt in (Neely, et al., 2013)**.** Tools fürs Monitoring sind beispielsweise: Splunk (Spl16), Ganglia (Gan16) oder Nagios (Nag16).

Repository Use:
Die Nutzung eines Version Control Systems, dass alle nötigen Software Artefakte enthält.

Shepherding Changes:
Beschreibt die Denkweise, dass jeder Entwickler für seine Änderungen verantwortlich ist. Und zwar durch den kompletten Deployment Prozess hindurch. Dies fördert die persönliche Verantwortung die übernommen wird, so wie das Mitwirken an der Deployment Pipeline und auftretenden Fehlern, ohne ein spezielles Qualitätssicherungsteam. Eine zusätzliche spezielle Implementierung dieser Praxis könnte z.B. sein, dass beim Release alle Entwickler die eine Änderung vorgenommen haben anwesend sein müssen (Feitelson, et al., 2013). Ebenso genannt in (Neely, et al., 2013)**.**

Staging:
Beschreibt das interne Verwenden (Dog Fooding) oder das graduelle Veröffentlichen (Gradual Rollout) von Änderungen an der Software bevor sie allen Endnutzern zur Verfügung gestellt wird. Zum Beispiel verwendet Facebook neue Versionen erst intern. Wenn keine Fehler auftreten wird die Änderung an 1% der Nutzer weitergegeben und wenn keine Schwierigkeiten auftreten erhalten alle Nutzer die neue Version (Feitelson, et al., 2013).

Abgesehen von den soeben aufgeführten Punkten, hat die Umsetzung von Continuous Delivery/Deployment auch direkte Auswirkungen auf die Softwarearchitektur. In *Towards Architecting for Continuous Delivery* (Lianping, 2015) versucht der Autor Architecturally Significant Requirements (ASRs), also für Continuous Delivery wichtige Softwarearchitektureigenschaften, zu erfasssen:

Deployability:
Häufige Releases erfordern ein zero-downtime Release, da eine häufige Serviceunterbrechung für den Nutzer nicht tolerierbar ist. Ebenso muss dafür gesorgt werden, dass es nicht wochenlang dauert bis z.B. eine Testumgebung aufgesetzt ist.

Security:
Die Applikation wird durch die häufigen Releases auch häufiger gestartet und spezielles Augenmerk sollte der Sicherheit der sonst meist vernachlässigten Start-Up-Phase zukommen.

Loggability:
Beim deployen einer neuen Version wird oft die neue Version auf neuen Serverinstanzen aufgesetzt und nach Prüfung der Funktionsfähigkeit werden die alten Serverinstanzen heruntergefahren. Bei dieser Art zu deployen kann es, ohne weitere Vorkehrungen, zum Verlust aller Logs auf den alten Instanzen kommen. Log Aggregation und Log Server gewinnen also an Bedeutung. Dabei müssen die Logs ausreichend Informationen zum Debuggen bieten, aber keine unnötigen Informationen enthalten, da die Speicherung und Aggregation mit Kosten verbunden sind.

Modifiability:
Bei der Verwendung von Continuous Delivery sind die meisten User Stories eher klein, um schnell released werden zu können und schnell Kundenfeedback zu bekommen. Die Architektur der Applikation sollte es so einfach wie möglich machen Veränderungen durchzuführen.
So können beispielsweise auch Dark Deployments und Feature Toggles mit eingeplant werden.

Monitorability:
Nach Möglichkeit sollte die Applikation ein spezielles Monitoringinterface anbieten. Zum einen um nach dem Release zu prüfen ob alles funktioniert, speziell bei Applikationen die über mehrere Server verteilt sind. Zum anderen um z.B. schnell nach dem Release Userfeedback und Nutzungsmetriken zu neuen Featurereleases zu bekommen.

Testability:
Ein großer Teil der Deployment Pipeline besteht aus Testphasen. Es ist also sinnvoll das Testen der Applikation so einfach wie möglich zu gestalten und dies bei der Architektur zu

bedenken. So könnte besonders Augenmerk auf die Testbarkeit der API und des GUI Frameworks gelegt werden.

Der Autor argumentiert nicht, dass die oben genannten ASRs für konventionelle Software nicht auch von Wichtigkeit sind, sondern dass bei Implementierung von Continuous Delivery die genannten Punkte an Bedeutung gewinnen um effektiv zu sein.
Weiterhin wird erwähnt, dass zum jetzigen Zeitpunkt nicht genügend erprobte Patterns und Taktiken für Continuous Delivery Software gibt. Ebenso mangelt es seiner Meinung nach an Strategien um bestehende Applikationen effektiv zu migrieren.

Als generelle Vorgehensweise zur Umsetzung aller in diesem Abschnitt beschriebenen Praktiken um den Releaseprozess zu verbessern, scheint eine inkrementelle Verbesserung besser zu funktionieren als zu versuchen alles auf einmal umzustellen (Neely, et al., 2013). Kann eine Änderung nicht released werden oder gibt es anderweitige Probleme empfiehlt es sich ein kurzes Meeting abzuhalten um Ursachen und verbessernde Schritte zu besprechen und Problempunkte so schnell wie möglich zu beheben (Neely, et al., 2013). Weiterhin wird die frühzeitige Parallelisierung von Tests empfohlen um möglichst schnell releasen zu können (Neely, et al., 2013).

6.1.3 Herausforderungen

Nachdem ein Blick auf mögliche Vorteile und verwendete Praktiken geworfen wurde, werden hier oft empfundene Schwierigkeiten beim Implementieren der beiden Konzepte erläutert:

Behinderung durch Altlasten:
Behinderungen durch traditionelle langsame Prozesse in der Organisation (Chen, 2015).

Domainspezifische Restriktionen:
Einigen Unternehmen, z.B. in der Telekommunikation, ist es nicht gestattet ständig zu releasen. Ähnliche Restriktionen gelten für einige Embedded Systeme in der Medizin, zur Sicherheit der Patienten. Software durch Kontrolle von Fabrikanlagen erfordert häufig das Anhalten der Anlage. Ein solcher Stopp ist kostenintensiv und muss vorher eingeplant werden, wodurch häufige Releases unattraktiv werden. Wieder andere Unternehmen haben keine volle Kontrolle über die Distribution ihrer Software und müssen sich an die Richtlinien und Releasezyklen ihres Zwischenhändlers anpassen. Ebenso wird beim Releasen über einen App-Store in einigen Fällen ein länglicher Reviewprozess des Store Betreibers vorausgesetzt (Leppänen, et al., 2015).

Kein klarer Adoptionspfad:
Fehlende wissenschaftliche Untersuchungen zur Einführung von Continuous Delivery in eine Organisation, sowie damit im Zusammenhang stehende organisatorische Probleme (Chen, 2015).

Legacy Code:
Alte Codelasten, speziell solche mit wenig automatisierten Tests und solche, die nicht mit automatisiertem Testen im Hinterkopf erstellt wurden, erfordern speziellen Aufwand (Leppänen, et al., 2015).

Manuelles und nichtfunktionales Testen:
Häufig kann auf das manuelle Testen vor dem Release nicht leicht verzichtet werden. Teils ist der Aufwand zur kompletten Automatisierung sehr hoch oder aber es wird auf exploratives Testen zum Aufstöbern zusätzlicher Defekte bestanden.
Nichtfunktionale Tests wie Performancetests sind oft sehr ressourcenintensiv und damit bei sehr häufigen Releases nicht immer praktikabel. Ähnliche Schwierigkeiten bieten sich bei Security Tests, komplexen Netzwerksetups und sehr vielen unterschiedlichen Nutzerumgebungen (Leppänen, et al., 2015).

Softwareunterstützung:
Fehlende einfache, anpassbare und standardisierte Softwareunterstützung von Continuous Delivery/Deployment, die einen Hersteller Lock-In vermeidet (Chen, 2015).

Mangelndes Vertrauen in Tests:
Nicht-deterministische Tests untergraben das Vertrauen in Testautomation (Neely, et al., 2013).

Präferenzen des Kunden:
Nicht alle Kunden empfinden häufige Releases als Bereicherung, einige präferieren seltenere Releases (Leppänen, et al., 2015).

Projektgröße:
Projekte mit sehr vielen oder langlaufenden Tests, langandauernden Builds oder einer großen Codebase sehen sich besonderer Herausforderungen ausgesetzt ihre Prozesse zu optimieren (Leppänen, et al., 2015).

Unterschiede zwischen Entwicklungs-, Test- und Produktionsumgebungen:
Die genannten Umgebungen und deren Konfigurationen in Einklang zu halten erweist sich teilweise als Schwierigkeit oder zumindest als zusätzlicher initialer Aufwand um die Umgebungen anzugleichen (Leppänen, et al., 2015).

Reibung in der Organisation und zwischen Teams:
Die Vorgesetzten müssen mitziehen um wirklich etwas zu ändern (Leppänen, et al., 2015). Zusätzlich sind häufig die vorhandenen Organisationsstrukturen nicht sehr empfänglich oder fördernd für Veränderung und neue Ideen. Reibung zwischen Teams bietet Schwierigkeitenpotential (Chen, 2015). Ein kürzerer Release Cycle bedeutet beispielsweise für Sales und Marketing, dass sie nicht mehr so weit vorausplanen können und nicht genau wissen wann welche Features released werden. Sie brauchen also mehr Transparenz um besser planen zu können und müssen gegebenenfalls ihre Abläufe anpassen (Neely, et al., 2013).
Das fehlende manuelle Testen vor dem Release führt teilweise zu Ängsten in der QA-Abteilung überflüssig zu werden. Dabei verlagert sich die Arbeit nur in andere Gebiete wie die Test Planung (Neely, et al., 2013).

Die Möglichkeit die Häufigkeit von Releases zu erhöhen, bedeutet nicht gleich, dass dies für jedes Unternehmen die richtige Entscheidung ist. So kann es, wie erwähnt, zum Beispiel sein, dass häufige Releases für den Kunden als Störungen empfunden werden (Leppänen, et al., 2015). Allerdings können auch Unternehmen die nicht zwangsweise eine schnelleren Releasezyklus anstreben von einem kontrollierten Releaseprozess profitieren (Leppänen, et al., 2015). Jedoch sind dabei, wie in diesem Abschnitt gezeigt, einige Schwierigkeiten zu überwinden.

6.2 DevOps

Continuous Delivery/Deployment ist eng verwandt mit dem DevOps (Development and Operations) Konzept. Continuous Deployment zielt darauf ab Änderungen an der Software mit Hilfe von automatisierten Builds, Tests und Deployments dem Endnutzer schnell zur Verfügung zu stellen (Humble, et al., 2011). Auf der anderen Seite hat sich DevOps als Methodik entwickelt, welche den kompletten Produktzyklus von Marketing, Planung, Personal, Vertrieb, Entwicklung (Development) bis hin zu der Systemadministration (Operations) betrifft (Rahman, et al., 2015). Allerdings kommen die Autoren von *DevOps: A Definition and Perceived Adoption Impediments* (Smeds, et al., 2015) nach einer Literaturrecherche zu dem Schluss, dass zum jetzigen Zeitpunkt noch keine einheitliche Definition von DevOps gegeben werden kann. Einige Autoren sehen DevOps eher als eine Berufsbezeichnung, andere als eine Firmenkultur und wieder andere als Ansammlung von Methoden oder eine beliebige Mischung aus den drei genannten Aspekten.
Die Autoren fassen zusammen und beschreiben DevOps als Mischung aus *Capabilities*, *Cultural Enablers* und *Technological Enablers*:

Capabilities:

- Continuous Planning
- Collaborative and Continuous Deployment
- Continuous Integration and testing
- Continuous release and deployment
- Continuous infrastructure monitoring and optimization
- Continuous user behaviour monitoring and feedback
- Service failure recovery without delay

Cultural Enablers:

- Shared goals, definition of success, incentives
- Shared ways of working, responsibility, collective ownership
- Shared values, respect, trust
- Constant, effortless communication
- Continuous experimentation and learning

Technological Enablers:

- Build automation
- Test automation
- Deployment automation
- Monitoring automation
- Recovery automation
- Infrastructure automation
- Configuration management for code and infrastructure

Ohne alle aufgeführen Begriffe zu erläutern stechen für die drei Kategorien die Eigenschaften *Continuous* (Capabilities), *Shared* (Cultural Enablers) und *Automation* (Technological Enablers) heraus. Es handelt sich also grob um einen Satz kontinuierlicher Praktiken, unterstütz von Automation, teamübergreifender gemeinsamer Werte und Kommunikation, hauptsächlich mit dem Ziel den Releaseprozess so reibungslos wie möglich zu gestalten. Vergleicht man diese Sichtweise auf DevOps mit Continuous Delivery und Continuous Deployment, unterscheiden sich diese besonders in der Betonung der *Cultural Enablers*.
Was anfangs noch auf die Development und Operations Teams beschränkt war, ist nun meist weiter gefasst und inkludiert weitere Teams, mit dem Ziel Grenzen abzubauen und Reibung zu minimieren. Nach Meinung von (Dyck, et al., 2015) ist, obwohl DevOps eine Abkürzung für Development and Operations ist, die Gültigkeit der Definition nicht auf diese beschränkt, und weitere Akronyme wie *DevSecOps* und *DevNetOps* sind nicht nötig sind.
In *Towards Definitions for Release Engineering and DevOps* (Dyck, et al., 2015) wird unterschieden zwischen DevOps und Release Engineering, wobei Release Engineering definiert ist als “software engineering discipline concerned with the development, implementation, and improvement of processes to deploy”, also nicht allzuweit entfernt

von Continuous Delivery. Während hier DevOps als “organizational approach that stresses empathy and cross-functional collaboration within and between teams – especially development and IT operations – in software development organizations, in order to operate resilient systems and accelerate delivery of changes“ definiert ist, also eher den Aspekt der teamübergreifenden Zusammenarbeit in den Vordergrund stellt.
Weiterhin scheint dabei ein gemeinsames Verständnis wichtiger geteilter Performancemetriken und deren automatische Messung und Analyse von Bedeutung zu sein (Gottesheim, 2015). Ziel hierbei ist es Mutmaßungen und Fingerzeigen zu minimieren und auftretende Probleme schnell zu erkennen.
Die Autoren von (Lehtonen, et al., 2013) beschreiben wie Metadaten, hier Taskmanagementdaten, dazu verwendet werden können um Visualisierungen zu erstellen. Diese Visualisierungen dienen dann der einfacheren Kommunikation über den Projektstatus. Diese bieten einen übersichtlicheren Blick auf die Daten. Zusätzlich wird hierdurch die Kommunikation vereinfacht und sonst untergangene Probleme im Softwareentwicklungsprozess werden teilweise sichtbar. Ebenso bestünde die Möglichkeit einige Probleme gleich automatisch zu erkennen und, z.B. für das Management, zu visualisieren.
Um das für DevOps wichtige Verwalten von Infrastruktur, zu vereinfachen bietet es sich an Infrastruktur ebenso in einem VCS zu verwalten wie Code. In *„Co-evolution of Infrastructure and Source Code – An Empirical Study”* (Jiang, et al., 2015) haben die Autoren 265 Repositories des Open Source Projekts OpenStack (Ope16) untersucht und die Änderungsrate sowie andere Eigenschaften der Dateien im Vergleich zu Test- und Builddateien sowie Source Code Dateien untersucht.
Infrastructure-as-code (IaC) beschreibt dabei die Praxis die benötigten Test- und/oder Deploymentumgebungen einer Software zu spezifizieren und zu automatisieren. Beispielsweise das Aufsetzen einer Virtual Machine (VM) und allen benötigten Libraries. Der Suffix *as-Code* beschreibt, dass die Spezifikation der Infrastruktur in Source Code ähnlichen Dateien entwickelt wird und somit in einem VCS verwaltet werden kann. Beispiele für eine solche Sprache sind etwa Puppet (Pup16) oder Chef (Che16). Beide sind für das Verwalten von Deployments auf Servern, VMs oder Cloud Umgebungen geeignet.
Ohne den eigentlichen Mehrwert dieser Art von Infrastrukturverwaltung zu vermindern, stellen die Autoren fest, dass die Infrastruktur Dateien nur einen kleinen Teil der gesamten Dateien ausmachen, relativ groß sind, sich häufig ändern und eng an Testdateien gekoppelt sind.

In diesem Abschnitt ist bereits leicht zu erkennen, dass häufig unklare Definitionen bestehen und sich Konzepte teilweise stark mit anderen überschneiden. Ziel von Continuous Delivery, Continuous Deployment und DevOps ist es gleichermaßen den Softwareentwicklungsprozess zu optimieren, nur der Fokus variiert. Im folgenden Abschnitt wird dieses Bestreben noch etwas ausgeweitet.

6.3 Continuous Software Engineering

Die verkürzten Releasecycles führen dazu, dass auch in anderen Bereichen außer der Entwicklung Umstellungen vorgenommen werden müssen. Continuous Software Engineering versucht die Lücken zu schließen um das *Continuous* Konzept auf andere Teile der Organisation auszuweiten um eine ganzheitliche Verbesserung zu erlangen.
Genauer bezeichnet Continuous Software Engineering die Fähigkeit einer Organisation in sehr kurzen Zyklen, typischerweise Stunden, Tage oder wenige Wochen, Software zu entwickeln, zu releasen und daraus zu lernen. Dies beinhaltet das Bestimmen neuer Funktionalitäten und deren Priorisierung, das Weiterentwickeln und Überarbeiten der Architektur, das Implementieren und Validieren der Funktionalität, sowie die Weitergabe an den Endnutzer und das Sammeln von Feedback um den nächsten Zyklus zu gestalten (Tichy, et al., 2015).
Die Befähigung dazu erfordert signifikante Änderungen in der Softwareentwicklung, wie die Parallelisierung von Abläufen, funktionsübergreifende Teams um schnelle Entscheidungen treffen zu können, sowie reibungslose Kommunikation zwischen Teams. Weiterhin erforderlich ist die Umsetzung und Weiterentwicklung der Verfahren, wie etwa Continuous Integration und Continuous Deployment, das Sammeln von Nutzungsdaten nach dem Deployment und Unterstützung für live Experimente um Alternativen auszutesten, z.B. durch A/B Tests (Tichy, et al., 2015).
In *Continuous Software Engineering and Beyond: Trends and Challenges* (Fitzgerald, et al., 2014) argumentieren die Autoren, dass speziell die Verbindung zwischen Business Strategie und Development ausgebaut werden muss und schlagen dafür die Bezeichnung *BizDev* vor. BizDev wird hier als fehlendes Glied in der Kette zu Continuous Software Engineering bezeichnet. BizDev enthält in diesem Zusammenhang hauptsächlich die Idee des *Continuous Planning*. Continuous Planning bezeichnet die Sicht auf Pläne als dynamische, ergebnisoffene Artefakte, welche sich in Reaktion auf das Geschäftsumfeld ändern und eine stärkere Zusammenarbeit zwischen Planung und Exekution forcieren.
Weiterhin wird argumentiert, dass Konzepte wie Continuous Delivery starke Ähnlichkeiten mit Konzepten aus *Lean Thinking* haben. Ein wesentlicher Kerngedanke des Lean Thinking ist das permanente Streben nach Perfektion, mit dem Ziel jegliche Verschwendung im Unternehmen zu minimieren (Womack, 2003). Geprägt wurde der Begriff im Zusammenhang mit der Automobilindustrie in *„Triumph of the lean production system“* (Krafcik, 1988).
Genauere Ähnlichkeiten bestehen zu dem *Flow* Konzept aus Lean Thinking. Anstatt eine Abfolge diskreter Aktivitäten, durchgeführt von klar abgegrenzten Teams, ist das Ziel eine kontinuierliche Bewegung (Fitzgerald, et al., 2014).
Brian Fitzgerald identifiziert in *Continuous Software Engineering and Beyond: Trends and Challenges* (Fitzgerald, et al., 2014) folgende *Continuous ** Aktivitäten:

- Continuous Planning
- Continuous Integration

- Continuous Deployment
- Continuous Delivery
- Continuous Verification
- Continuous Testing
- Continuous Compliance
- Continuous Security
- Continuous Use
- Continuous Trust
- Continuous Run-Time Monitoring
- Continuous Improvement
- Continuous Innovation

Um den Umfang der Arbeit nicht zu sprengen wird hier nicht genauer auf die einzelnen Punkte eingegangen, dem interessierten Leser sei die entsprechende Publikation von (Fitzgerald, et al., 2014) ans Herz gelegt.
Abbildung 35 aus derselben Publikation bietet abschließend einen Überblick über all diese Aktivitäten, zusammen mit den entsprechenden Teambereichen und Bezeichnern für deren Zusammenarbeit (BizDev und DevOps).

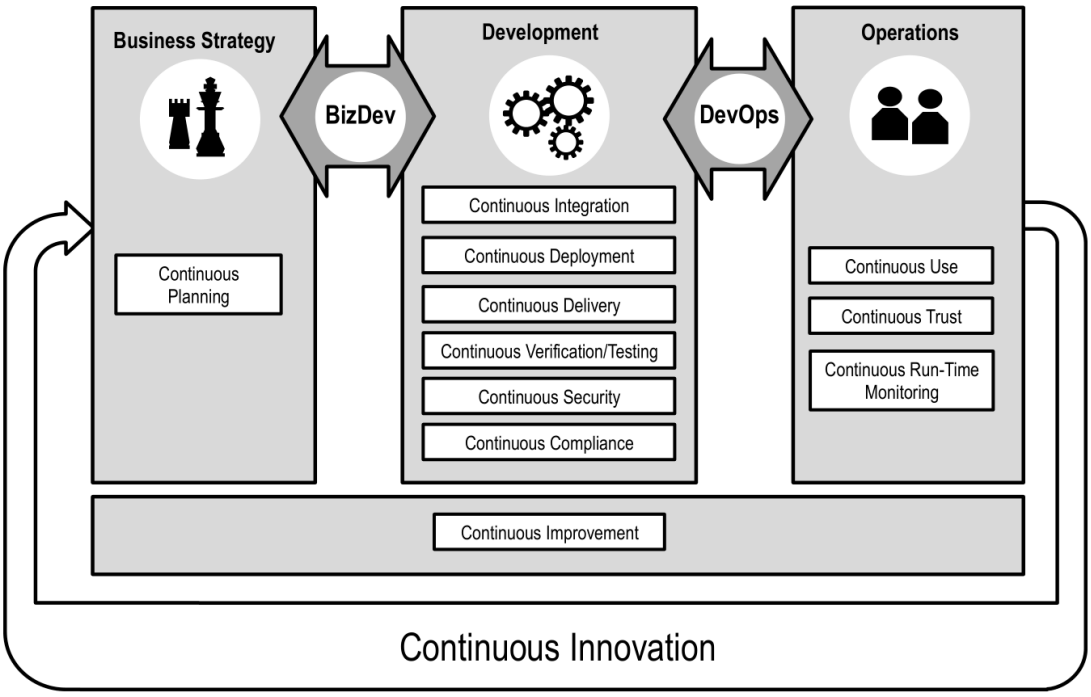

Abbildung 35 Zusammenfassung aller Continuous * Tätigkeiten die zusammen Continuous Software Engineering beschreiben. Das Bild wurde entnommen aus (Fitzgerald, et al., 2014).

6.4 Zusammenfassung und weitere Ideen

Frei nach dem Agile Motto: “Our highest priority is to satisfy the customer through early and continuous delivery of valuable software” (agi16) beschäftigen sich die in diesem Kapitel aufgeführten Konzepte mit der Verbesserung der Deployment Pipeline und der diese umgebenden Vorgänge in der Organisation.
Die hier genannten Konzepte sind keineswegs das Ende der Fahnenstange, bieten jedoch einen groben Überblick über den Trend in der Weiterentwicklung der Softwareentwicklung. Viele Publikationen geben Ideen um kleinere Teile dieses Prozesses zu verbessern. So beschäftigt sich (Liikkanen, et al., 2014) beispielsweise speziell mit dem Aspekt in enger Zusammenarbeit mit dem Endnutzer zu entwickeln.
(Gruhn, et al., 2015) schlägt eine, vielleicht umstrittene, andere Herangehensweise an die Zusammenarbeit zwischen Business und Development vor, indem das Business Department selber mit Hilfe einer DSL in einer Sandbox z.B. Business Logik mitentwickelt und somit durch das direkte Mitwirken am Code stärker investiert ist und die Verbindung zur Entwicklungsabteilung gestärkt wird.
In ihrer Publikation *„Personalised Continuous Software Engineering“* (Papatheocharous, et al., 2014) beschreiben die Autoren, mit Hilfe einer Durchgeführten Untersuchung, wie eine individuelle Zusammensetzung von Teams nach Persönlichkeiten und anderen menschlichen Faktoren, frei nach dem Agile Motto „individuals and interactions over processes and tools“ (agi16), Potential für die Verbesserung des Continuous Software Engineering Prozesses bieten könnte.
Weitere interessante Punkte sind die Parallelisierung von Abläufen, im Kleinen und im Großen, die Virtualisierung von Umgebungen, z.B. mit Hilfe von VirtualBox (Vir15), Docker (Doc15) oder Vagrant (Vag15), die teilweise automatische Generierung von Deployment Pipelines, die Verwendung von Developer Pipelines (auf Entwickler Maschinen) und parallele spekulative Pipelines zur Beschleunigung der eigentlichen Pipeline Durchläufe.

7 Fazit und Ausblick

Um die Arbeit abzuschließen wird hier kurz rekapituliert was erarbeitet wurde. Ebenso wird darauf eingegangen ob und inwiefern die Implementierung des Rechnungslauftests die Implementierung weiterer WEAT EABR Tests vereinfacht. Nachfolgend findet sich eine kurze Bewertung zum Arbeiten mit dem Test-Editor und es werden weiterführende Ideen beschrieben.

7.1 Was wurde erarbeitet?

Nach einleitender allgemeiner Betrachtung des Konzeptes der Deployment Pipeline (Kapitel 2) wurde die Deployment Pipeline des WEAT EABR Projekts kurz begutachtet und festgestellt, dass die Phase der Akzeptanztests komplett manuell durchgeführt wird und von Automatisierung profitieren kann (Kapitel 4). Im Hinblick darauf wurden vorher Akzeptanztests allgemein beleuchtet (Kapitel 3). Im Anschluss an die Analyse wurde ein repräsentativer Testfall, der Rechnungslauftest, aus der WEAT EABR Akzeptanztestsuite mit Hilfe des Test-Editors automatisiert und dessen Implementierung beschrieben (Kapitel 5). Um auf die Integration in die, auf einem Jenkins Server laufende, WEAT EABR Deployment Pipeline hinzuführen, wurde exemplarisch auf dem Arbeitsrechner für diese Arbeit der Test von einem lokalen Jenkins Server aus gestartet. In Kapitel 6 findet sich eine Beschreibung des Kontexts der Deployment Pipeline und damit im Zusammenhang stehender Publikationen.
Im weiteren Verlauf dieses Kapitels wird auf noch offene Aspekte eingegangen und es werden Ideen für eine eventuelle Nachfolgearbeit gegeben.

7.1.1 Übertragbarkeit auf andere Tests

Neben dem Rechnungslauftest bestehen für das WEAT EABR Projekt noch weitere Testfälle. Dabei sind die Tests in vier Kategorien unterteilt:

1. Datenverarbeitung
2. Eingaben UI
3. Rechnungslauf
4. Vertragskonstellationen

Kategorie 1 enthält Tests zum Im- und Export von Dateien und zu Berechnungen. Kategorie 2 besteht aus Tests die das Eingeben von Daten über die GUI testen. Kategorie 3 ist der hier implementierte Rechnungslauftest. Kategorie 4 testet ob für verschiedene Vertragskonstellationen bei der Entgeltberechnung die richtigen Ergebnisse geliefert werden.
Einige der Tests, besonders aus der Kategorie 1 sind lediglich ein Teil des Rechnungslauftests und sind nun leicht zu implementieren oder gegebenenfalls schon durch ein Test-Editor Szenario ausgelagert. Weiterhin ist ein Großteil der GUI über die *ElementList.conf* in den Test-Editor eingepflegt worden. Zum Einpflegen von neuen Elementen in WEAT EABR durch das Ausfüllen von GUI Masken, wie das Anlegen von Verträgen, bestehen bereits einige Szenarien für gängige Aktionen. Viele der häufig vorkommenden Testschritte wie: Login, Navigation und das Arbeiten mit Tabellen sind leicht im Test-Editor „zusammenklickbar“ ohne Fixture Code schreiben zu müssen oder Konfigurationsdateien des Test-Editors ändern zu müssen.
Für einige andere Tests muss mehr Aufwand betrieben werden und neuer Fixture Code geschrieben werden, wie beispielsweise für das Überprüfen ob eine Datei herunterladbar ist. Für wieder andere muss kein neuer Fixture Code geschrieben werden, jedoch müssen die Konfigurationsdateien des Test-Editors verändert werden.

7.2 Arbeiten mit dem Test-Editor

Im Hinblick darauf, dass evtl. andere diese Arbeit fortsetzten, anderweitig mit dem Test-Editor arbeiten oder aber den Test-Editor selber weiterentwickeln, werden hier kurz einige gewonnene Erkenntnisse über das Arbeiten mit dem Programm, in der Version 1.8, präsentiert.
Abgesehen von einigen Unannehmlichkeiten beim Arbeiten mit der Test-Editor GUI erfüllt er sein explizites Ziel Tests leicht „zusammenklicken“ zu können. Dies gilt allerdings nur für den Fall, dass für den zu erstellenden Test weder neuer Fixture Code eingebettet werden muss, noch Änderungen an der Konfiguration vorgenommen werden müssen. Beides musste für diese Arbeit getan werden.
Wird eine Änderung vorgenommen, die dies erfordert und es kommt zu einem Fehler, ist oft nur schwer ersichtlich wo der Fehler liegt, besonders wenn es um Änderungen an den Konfigurationsdateien geht. Dies führt dazu, dass die Struktur der Fixture sehr selten iterativ verbessert wird, da es unglaublich schnell zu Fehlern in der Konfiguration kommt, die meist keine hilfreiche Fehlermeldung erzeugen. Es ist oft nicht ersichtlich in welcher der Konfigurationsdateien ein Fehler vorliegt. Große Änderungen, die mehrere Aspekte auf einmal ändern sind damit fast ausgeschlossen. Erschwerend kommt hinzu, dass für das Integrieren neuer Funktionalität in die Fixture meist vier Konfigurationsdateien geändert werden müssen.
Der Test-Editor bietet Teilweise Hilfestellung beim Erstellen der initialen Konfiguration, jedoch wurde der größte Teil manuell erstellt und alle Änderungen wurden ebenfalls manuell ausgeführt. Für das vorliegende Projekt enthält die *ElementList.conf* 265 Zeilen, die

AllActionGroups.xml 737 Zeilen, die *TechnicalBindingTypeCollection.xml* 287 Zeilen und die *SzenarioLibrary (content.txt)* 213 Zeilen.
Abgesehen von diesem, relativ schwerwiegenden, Punkt ist der Test-Editor ein sehr hilfreiches Tool zur Testautomatisierung.

7.3 Ideen für weiterführende Arbeiten

Die hier kurz angerissenen Ideen sollen einem Interessierten Anhaltspunkte für eine weiterführende oder aber komplett neue Arbeit geben.

7.3.1 Test-Editor Debugging und Konfiguration verbessern

Zum Zeitpunkt des Schreibens der vorliegenden Arbeit ist es nicht klar in welche Richtung sich der Test-Editor entwickelt und ob sich damit die Kritikpunkte aus Abschnitt 7.2 erübrigen. Sollte dem nicht so sein, könnte eine Arbeit mit dem Ziel dies zu verbessern großen Mehrwert bringen. Als Zwischenlösung würde die Generierung aller Konfigurationsdateien Sinn machen.
In der Findungsphase dieser Arbeit wurde bereits ein Teil der Konfiguration testweise generiert.

7.3.2 Scannen von Webseiten

Für ein Szenario wie das hier vorliegende wird die komplette Webseite manuell (maximal mit Toolunterstützung) in die Test-Editor Konfiguration eingepflegt. Eine Alternative Herangehensweise wäre es die Webseite zu scannen und automatisch in die Test-Editor Konfiguration einzupflegen oder aber in andere Tools einzubetten oder sogar teilweise automatisch möglichst nützliche Tests für die Webseite zu generieren.
Ohne spezielles Wissen über den Aufbau der GUI ist es jedoch nicht möglich zu erkennen ob beispielsweise ein Button ein Button und ein Dropdown-Menu ein Dropdown-Menu ist. Diese sind keineswegs immer mit Hilfe der dafür vorgesehenen *button* und *select* Tags implementiert, sondern etwa mit *div* Elementen und sind somit nicht ohne Weiteres zu erkennen. Ein Beispiel hierfür ist das für die WEAT EABR GUI verwendete Vaadin. Das heißt selbst wenn die Webseite explizit mit dem Ziel entwickelt wird, sie später automatisch zu scannen, ist es wahrscheinlich nötig GUI Elemente mit Hilfe von HTML Klassen oder IDs genauer zu beschreiben.
In dieser Arbeit wurden die GUI Elemente mit Hilfe von HTML IDs referenziert. Um einzigartige Namen zu haben sind Elemente durch *section-subsection*-name-componentType* beschrieben. Hierbei könnte für eine automatische Testgenerierung bereits der Typ des Elements (Button, Textfeld etc.) herausgelesen werden. Weiterführend könnte z.B. für ein Textfeld der Dateityp und Wertegrenzen mit angegeben werden, so dass beim Testen klar ist, dass es sich hierbei z.B. um ein Integer Feld handelt. Hieraus könnten bereits

Testfälle generiert werden die mit falschen Werten prüfen. Allerdings ist ohne weiteres Wissen nicht klar wie die Webseite im Fehlerfall reagiert oder wo sich überhaupt eine Art Submit-Button befindet. Eine rein generische Generierung von Testfällen erweist sich also als sehr schwierig.

7.3.3 Automatische Generierung von Test-Editor Testfällen aus spezifizierten Testfällen im Wiki

Eine andere Herangehensweise könnte das automatische Erstellen von gegebenen Testfällen aus einer formalen Testfallbeschreibung sein. Sind beispielsweise die Testfälle im Wiki in einer bekannten formalen Sprache beschrieben, so könnten die Testfälle direkt generiert werden. Voraussetzung hierfür wäre jedoch, dass der benötigte Fixture Code bereits existiert und so strukturiert ist, dass er von den generierten Tests verwendet werden kann. Bestehen die Tests aus vielen immer unterschiedlichen Schritten, macht dies wenig Sinn. Existieren jedoch viele Testfälle, die aus immer ähnlichen Schritten aufgebaut sind, könnte dies einen Mehrwert bringen.
Eine weitere Variante hiervon wäre es Testfälle aus User Logs zu generieren. Ruft der Nutzer etwa einen Fehler hervor und berichtet diesen, so könnte nach Behebung der Ursache der Testfall schnell aus den aufgezeichneten Aktivitäten des Nutzers generiert werden. Hierbei ist es jedoch aller Voraussicht nach unabdingbar vorher zu prüfen welches Verhalten genau zum Fehler führte, wo der Test beginnen soll, was die Vorbedingungen sind und welche Schritte überflüssig sind. Die Generierung eines Testfalls aus einem formal spezifiziertem Logfile könnte jedoch trotzdem eine interessante Option sein.

8 Literaturverzeichnis

2nd International Workshop on Rapid Continuous [Konferenz] / Verf. Tichy Matthias [et al.] // 37th IEEE International Conference on Software Engineering. - [s.l.] : ACM/IEEE, 2015.

agilemanifesto [Online] // agilemanifesto. - 18. 01 2016. - http://www.agilemanifesto.org/.

akquinet AG [Online] // akquinet AG. - 26. 11 2015. - http://www.akquinet.de/.

Bamboo [Online]. - 16. 03 2016. - https://www.atlassian.com/software/bamboo.

Basiswissen Softwaretest [Buch] / Verf. Spillner Andreas und Linz Tilo. - [s.l.] : dpunkt.verlag, 2012.

BizDevOps: Because DevOps is Not the End of the Story [Artikel] / Verf. Gruhn Volker und Schäfer Clemens // Intelligent Software Methodologies, Tools and Techniques. - 2015.

Build Waiting Time in Continuous Integration – An Initial Interdisciplinary Literature Review [Konferenz] / Verf. Laukkanen Eero und Mäntylä Mika V. // 2nd International Workshop on Rapid Continuous Software Engineering. - [s.l.] : IEEE/ACM, 2015.

Challenges, Benefits and Best Practices of Performance Focused DevOps [Artikel] / Verf. Gottesheim Wolfgang. - [s.l.] : ACM, 2015.

Chef [Online]. - 01. 03 2016. - https://www.chef.io/.

Co-evolution of Infrastructure and Source Code - An Empirical Study [Konferenz] / Verf. Jiang Yujuan und Adams Bram // 12th Working Conference on Mining Software Repositories. - Montreal, Canada : IEEE, 2015.

Continuous Delivery - Der Pragmatische Einstieg [Buch] / Verf. Wolff Eberhard. - [s.l.] : dpunkt.verlag, 2015.

Continuous Delivery - Easy: Just Change Everything (Well, Maybe It Is Not That Easy) [Artikel] / Verf. Neely Steve und Stolt Steve. - [s.l.] : IEEE, 2013.

Continuous Delivery - Reliable Software Releases Through Build, Test, And Deployment Automation [Book] / auth. Humble Jez and Farley David. - [s.l.] : Pearson Education, 2011.

Continuous Delivery: Huge Benefits but Challenges Too [Artikel] / Verf. Chen Lianping. - 2015.

Continuous Integration [Artikel] / Verf. Fowler Martin. - 2006.

Continuous Software Engineering and Beyond: Trends and Challenges [Artikel] / Verf. Fitzgerald Brian und Stol Klaas-Jan // Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering. - [s.l.] : ACM, 2014.

Development and Deployment at Facebook [Artikel] / Verf. Feitelson Dror G., Frachtenberg Eitan und Beck Kent L.. - [s.l.] : IEEE, 2013.

DevOps: A Definition and Perceived Impediments [Artikel] / Verf. Smeds Jens, Nybom Kristian und Porres Ivan // Agile Processes, in Software Engineering, and Extreme Programming. - 2015.

Docker [Online] // Docker. - 08. 12 2015. - https://www.docker.com/.
ElectricFlow [Online]. - 16. 03 2016. - http://electric-cloud.com/products/electricflow/.
FitNesse [Online] // FitNesse. - 26. 11 2015. - http://www.fitnesse.org/.
flywaydb [Online] // flywaydb. - 29. 12 2015. - http://flywaydb.org/.
Ganglia [Online]. - 01. 03 2016. - http://ganglia.info/.
Gerrit [Online]. - 03. 02 2016. - https://www.gerritcodereview.com/.
Go CD [Online] // Go CD. - 07. 12 2015. - https://www.go.cd/.
Golang [Online] // Golang. - 07. 12 2015. - https://golang.org/.
jamesshore - The Problems With Acceptance Testing [Online] // jamesshore - The Problems With Acceptance Testing. - 01. 12 2015. - http://www.jamesshore.com/Blog/The-Problems-With-Acceptance-Testing.html.
JBoss [Online] // JBoss. - 07. 01 2016. - http://www.jboss.org/.
Jenkins [Online] // Jenkins. - 26. 11 2015. - http://jenkins-ci.org/.
Jenkins FitNesse Plugin [Online] // Jenkins FitNesse Plugin. - 10. 01 2016. - https://wiki.jenkins-ci.org/display/JENKINS/Fitnesse+Plugin.
Lean Thinking: Banish Waste And Create Wealth In Your Corporation [Buch] / Verf. Womack James P.. - 2003.
Lean UX - The Next Generation of User-Centered Agile Development? [Artikel] / Verf. Liikkanen Lassi A. [et al.]. - [s.l.] : ACM, 2014.
Mozilla Firefox [Online]. - 16. 03 2016. - https://www.mozilla.org/firefox/.
Nagios [Online]. - 01. 03 2016. - https://www.nagios.org/.
OpenStack [Online]. - 01. 03 2016. - https://www.openstack.org/.
Personalised Continuous Software Engineering [Artikel] / Verf. Papatheocharous Efi [et al.]. - [s.l.] : ACM, 2014.
PostgreSQL [Online] // PostgreSQL. - 07. 01 2016. - http://www.postgresql.org/.
Pulse [Online]. - 16. 03 2016. - http://zutubi.com/products/pulse/.
Puppetlabs [Online]. - 01. 03 2016. - https://puppetlabs.com/.
Selenium [Online] // Selenium. - 27. 11 2015. - http://www.seleniumhq.org/.
SIGNAL IDUNA [Online] // SIGNAL IDUNA. - 26. 11 2015. - https://www.signal-iduna.de.
Splunk [Online]. - 01. 03 2016. - http://www.splunk.com/.
Synthesizing Continuous Deployment Practices Used in Software Development [Konferenz] / Verf. Rahman Akond Ashfaque Ur [et al.] // Agile Conference. - [s.l.] : IEEE, 2015.
System Properties [Online] // System Properties. - 12. 01 2016. - https://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html.
TeamCity [Online]. - 16. 03 2016. - https://www.jetbrains.com/teamcity/.
Test-Editor [Online]. - 03. 08 2015. - http://testeditor.org/.
The Highways and Country Roads to Continuous Deployment [Artikel] / Verf. Leppänen Marko [et al.]. - [s.l.] : IEEE, 2015.
ThoughtWorks [Online] // ThoughtWorks. - 07. 12 2015. - https://www.thoughtworks.com/.

Towards Architecting for Continuous Delivery [Konferenz] / Verf. Lianping Chen // 12th Working IEEE/IFIP Conference on Software Architecture. - 2015.
Towards Definitions for Release Engineering and DevOps [Konferenz] / Verf. Dyck Andrej, Penners Ralf und Lichter Horst // 3rd International Workshop on Release Engineering. - [s.l.] : IEEE/ACM, 2015.
Triumph of the lean production system [Journal] / Verf. Krafcik J.. - 1988.
Vaadin [Online] // Vaadin. - 30. 11 2015. - https://vaadin.com.
Vagrant [Online] // Vagrant. - 08. 12 2015. - https://www.vagrantup.com/.
VirtualBox [Online] // VirtualBox. - 08. 12 2015. - https://www.virtualbox.org/.
Visualizations as a Basis for Agile Software Process Improvement [Konferenz] / Verf. Lehtonen Timo [et al.] // 20th Asia-Pacific Software Engineering Conference. - 2013.
W3Schools - Xpath Intro [Online] // W3Schools - Xpath Intro. - 09. 12 2015. - http://www.w3schools.com/xsl/xpath_intro.asp.
WEAT [Online] // WEAT. - 26. 11 2015. - http://www.weat.de/.
Wikipedia WAR [Online] // Wikipedia WAR. - 07. 01 2016. - https://en.wikipedia.org/wiki/WAR_(file_format).

9 Abbildungsverzeichnis

10 Listings

11 Tabellenverzeichnis

12 Abkürzungsverzeichnis

AG	Aktiengesellschaft
API	Application Programming Interface
CI	Continuous Integration
CD	Continuous Delivery
DSL	Domain Specific Language
EC	Electronic Cash
GUI	Graphical User Interface
UI	User Interface
VCS	Version Control System
WEAT EABR	WEAT Entgelt-Abrechnungssystem

13 Glossar

Agile Softwareentwicklung
Eine Zusammenstellung von Softwareentwicklungsmethoden, welche gute Zusammenarbeit und einen flexiblen Entwicklungsprozess fördern soll. Dabei fungiert agile als Überbegriff für konkretere Methoden wie Scrum oder Extreme Programming. Für mehr Informationen zur agilen Software Entwicklung siehe (agi16).

Akzeptanztestsuite
Eine Ansammlung oder Zusammenstellung von Akzeptanztests, die meist einen bestimmten Satz an Funktionalität verifizieren soll.

Anforderung
Eine von der Software zu erfüllende Leistung oder Eigenschaft.

Application Programming Interface
Ein Application Programming Interface, kurz API, beschreibt eine Software Komponente rein als Schnittstelle, also als Inputs, Outputs und Datentypen. Dies dient dem Zweck die, oft von anderen genutzte, Schnittstelle änderungsresistent zu halten ohne jedoch die Implementierung zu beschränken.

Black-Box-Test
Ein Black-Box-Test bezeichnet eine Methode des Softwaretests, bei der die Tests ohne Kenntnisse über Implementierung des zu testenden Systems entwickelt werden. Der Test prüft also nur das Verhalten nach außen.

Build
Eine Ansammlung von Schritten, wie etwa dem Kompilieren und Analysieren des Source Codes. Häufig gilt jeder Build als neue Version.

Coding Conventions
Ein Satz von Richtlinien und Empfehlungen für eine spezielle Programmiersprache, welche eine bestimmten Programmierstil vorgeben. Beispielsweise eine Festlegung über die Einrückung des Codes, die Art der Kommentare oder die Namensgebung von Variablen und Konstanten. Dies dient dazu die Zusammenarbeit durch einheitliche Standards zu erleichtern.

Commit
Hinzufügen der letzten Änderungen am Source Code zum Version Control System.

Deployment
Das Übertragen von Software Artefakten in eine Produktivumgebung des Kunden oder das anderweitige Bereitstellen der Software für den Endnutzer.

Domain Specific Language
Eine formale Sprache die speziell für eine bestimmte Domäne entworfen wurde.

Dummydaten
Eine Bezeichnung für Pseudodaten, die üblicherweise zu Testzwecken verwendet werden.

Fork
Das Abspalten einer Software mit separater Entwicklung weg vom eigentlichen Ursprung durch ein anderes Team.

Graphical User Interface
Ein Graphical User Interface, kurz GUI, beschreibt eine grafische Benutzeroberfläche mit deren Hilfe der Nutzer mit der Software interagieren kann. Dabei kann es sich beispielsweise um eine Webseite handeln, die als Schnittstelle für den Nutzer dient um mit dem dahinterliegenden Server interagieren zu können.

Lock-In
Lock-In oder Vendor Lock-In beschreibt die Abhängigkeit, welche bei der Verwendung bestimmter Produkte entsteht. Diese erschwert das Wechseln auf ein gleichwertiges Produkt eines anderen Herstellers.

Look & Feel
Beschreibt das Aussehen sowie die Benutzbarkeit einer GUI.

Maske
Ausschnitt einer GUI, der dem Einpflegen bestimmter Daten in das System dient.

Open Source
Open Source Software (OSS) bezeichnet eine Software, deren Source Code für die Öffentlichkeit verfügbar gemacht wurde, meist im Zusammenhang mit einer Lizenz, die festlegt wie der Source Code verwendet werden darf.

Regression
Regression bezeichnet einen Rückschritt oder eine Rückentwicklung. Im Zusammenhang mit Software ist oft eine Verschlechterung der Qualität gemeint.

Regressionstest
Ein Regressionstest hat die Intention eine Software gegen Regression abzusichern. Dabei sollen Regressionstests sowohl das Wiederauftreten alter Fehler, wie auch das Auftreten neuer Fehler durch vorgenommene Änderungen, vermeiden.

Release
Ein Release ist die Veröffentlichung oder das zur Verfügung stellen der Software für den Nutzer.

Sandbox
Eine Umgebung zur Ausführung eines Programms, welche starken Restriktionen unterliegt, üblicherweise um die Sicherheit des umgebenden Systems nicht zu gefährden.

Selenium
Ein Framework zur Automation von Webbrowsern.

SIGNAL IDUNA
Die SIGNAL IDUNA Gruppe (SIG15) ist ein wirtschaftlicher Zusammenschluss von Einzelunternehmen in Deutschland. Die Obergesellschaften sind:

- SIGNAL Krankenversicherung a. G., Dortmund.
- IDUNA Vereinigte Lebensversicherung aG für Handwerk, Handel und Gewerbe, Hamburg.
- SIGNAL Unfallversicherung a. G., Dortmund.
- Deutscher Ring Krankenversicherungsverein a. G., Hamburg.

Unit Tests
Ein schnell auszuführender low-level Test, der meist eine einzige Annahme über die Funktionalität eines Codeabschnitts (z.B einer Methode) überprüft.

Usability
Beschreibt, ganz allgemein, wie leicht ein von Menschen erstelltes Objekt zu erlernen und zu benutzen ist. In Bezug auf Software ist beispielsweise gemeint wie erlernbar und benutzbar der Umgang mit der Software(-oberfläche) ist.

User Story
Ein in der Agilen Software Entwicklung genutztes Tool um eine Anforderung aus Sicht des Endnutzers zu dokumentieren. Z.B. soll es dem Nutzer möglich sein sich einloggen zu können. Die zugehörige User Story beschreibt den Nutzer, sein Ziel und die Schritte die er vornehmen muss.

Vaadin
Vaadin ist ein serverseitiges Java Framework zum Erstellen von Single Page Webapplications (Vaa15). Vaadins besitzt unter anderem folgende Eigenschaften:

- Die Komplette Kommunikation zwischen Browser und Server ist automatisiert.
- Erweiterbar durch Plugins.
- Läuft auf allen gängigen Browsern sowie auf Mobilgeräten.
- Ist Open Source.

Version Control System
Ein System zum Aufzeichnen von Änderungen an einer Datei oder einem Satz von Dateien über die Zeit. Somit kann auf die komplette Historie zugegriffen werden. Diese dient der Dokumentation der Änderungen und stellt die Möglichkeit bereit auf ältere Versionen der Dateien zuzugreifen.

XPath
Definiert eine Syntax um auf Elemente in XML Dateien zu verweisen (W3S15).

14 Anhänge

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den _____ _____