MASTERARBEIT

EIN MODELL AUS VERKNÜPFTEN EINFLUSSFAKTOREN AUS DEN PHASEN DES SOFTWARE DEVELOPMENT LIFE CYCLE, WELCHE MIT TESTERGEBNISSEN DES AKZEPTANZTESTS IN VERBINDUNG STEHEN, ZUM BEURTEILEN DER QUALITÄT EINES SOFTWARE RELEASE

ausgeführt am

Studiengang

Informationstechnologien und Wirtschaftsinformatik

Von: Pawely Eskandar

Personenkennzeichen: 1610320016

Graz, am .....

Unterschrift

EHRENWÖRTLICHE ERKLÄRUNG

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die benutzten Quellen wörtlich zitiert sowie inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

.....

Unterschrift

DANKSAGUNG

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Masterarbeit unterstützt und motiviert haben.
Zuallererst möchte ich Gott für seinen Beistand und seine reichen Gnaden danken, denn er hat mich bis hierhin begleitet.
Ein großer Dank gebührt meinem Betreuer Herr Dr. Michael Amann-Langeder, der meine Masterarbeit betreut und begutachtet hat. Für die hilfreichen Anregungen und die konstruktive Kritik bei der Erstellung dieser Arbeit möchte ich mich herzlich bedanken.
Abschließend möchte ich mich bei meiner Ehefrau Mariam Eskandar und bei meinen Eltern und Geschwistern bedanken, für den starken emotionalen Rückhalt über die Dauer meines gesamten Studiums.

KURZFASSUNG

Die Softwarequalität ist ein wesentlicher Wettbewerbsfaktor in der Softwarebranche. Viele Softwareprojekte legen daher den Schwerpunkt ihrer Softwarequalität auf Abnahmetests und ignorieren andere wichtige Faktoren des gesamten Software-Lebenszyklus. Aus diesem Grund konzentriert sich diese Arbeit darauf, ob ausgewählte Einflussfaktoren aus dem gesamten Software-Lebenszyklus in Verbindung mit Testergebnissen von Abnahmetests eine detailliertere Bewertung der Qualität eines Software Release ermöglichen können. Hierzu wurden Einflussfaktoren untersucht und in ein Modell integriert. Darauf aufbauend wurde eine Bewertungsmatrix für die Implementierung entwickelt. In einem Experiment, in dem das Modell auf ein Beispielprojekt angewendet wurde, wurden 6 von 16 Features aus dem Software Release ausgenommen. Hätte sich dieses Experiment nur auf die Ergebnisse des Akzeptanztests gestützt, wären nur 4 von 16 Features aus dem Software Release ausgenommen worden. Diese Ergebnisse bestätigen, dass die ausgewählten Einflussfaktoren eine detailliertere Bewertung der Qualität des Software Release ermöglichen. Trotz dieser Schlussfolgerung würden die Erkenntnisse dieser Arbeit von weiteren Studien profitieren, bei denen mehr Zeit zur Verfügung steht und verschiedene Werkzeuge zur Überprüfung der Forschungsfrage verwendet werden.

ABSTRACT

Software quality is a key competitive factor in the software industry. Many software projects therefore lay their focus on quality in acceptance tests and ignore other important factors of the whole-software lifecycle. This paper focuses on whether selected influencing factors of the whole-software lifecycle in relation to the test results of acceptance tests can enable a more detailed evaluation of the quality of a software release. Influencing factors are explored and integrated in a model and based on this an evaluation matrix for the implementation is developed. In an experiment where the model is applied on an example project, 6 out of 16 features are excluded from the software release. Had this experiment relied on the results of the acceptance test, only 4 out of 16 features would have been excluded from the software release. These results verify that the selected influencing factors enable a more detailed evaluation of the quality of the software release. Despite this conclusion, the findings of this paper would benefit from further studies, where more time is available and different tools are used to verify the research question.

INHALTSVERZEICHNIS

1 EINLEITUNG

Die Sicherstellung der Qualität einer Software ist von einem Softwareprojekt nicht mehr wegzudenken. Gleichzeitig steigen die Ansprüche an die Qualität der Software seitens der Auftraggeber, welche wiederum durch die Endbenutzer und Endbenutzerinnen und den Markt getrieben werden, stetig an (Chatterjee, 2017; McConville, 2017). In einer digitalen Welt, in welcher sich die Time-To-Market immer weiter verkürzt und Releases immer häufiger werden, kann das Testing mit der Zeit zum Engpass werden. Die Zahl der unterschiedlichen Geräte, auf denen die Software reibungslos funktionieren muss, steigt und sorgt damit für eine höhere Komplexität in der Sicherstellung der Softwarequalität (Akula, 2017). Im Zeitalter der Globalisierung und der hohen Verbreitung von Social Media, können sich Firmen so gut wie keinen Fehler in ihrer Software erlauben, falls sie am Markt noch bestehen und eine ernst zu nehmende Konkurrenz darstellen wollen (Britton, 2017). Aus diesen Gründen ist es notwendig eine hohe Softwarequalität anzustreben und diese kontinuierlich zu gewährleisten. Um dies zu bewerkstelligen bedarf es eines Qualitätsprozesses durch den gesamten *Software Development Life Cycle (SDLC)* hindurch. Qualität kann nicht erst ganz am Ende des Prozesses geschaffen werden, indem kurz vor der Auslieferung der Software getestet wird. Qualität muss vom ersten Schritt des Lebenszyklus der Software begonnen und bis zum letzten Schritt weiterverfolgt werden. In agilen Softwareprojekten muss dieser Qualitätsprozess sogar kontinuierlich in jeder Iteration betrieben werden. (Minduca, 2014; Mark, 2017)

In der vorletzten Phase des SDLC, bevor die Software ausgeliefert wird, wird sie üblicherweise auf die Erfüllung aller Anforderungen des Auftraggebers geprüft. Dieser Test nennt sich *Akzeptanztest* und stellt die letzte Teststufe im Testprozess dar. Der Akzeptanztest wird in traditionellen Vorgehensmodellen, wie zum Beispiel beim Wasserfallmodell, manuell durchgeführt. In agilen Vorgehensmodellen hingegen, wie beispielsweise SCRUM, versucht man diesen Test so gut wie möglich zu automatisieren, um unter anderem Zeit und Aufwand und gleichzeitig Kosten zu ersparen. Unabhängig davon welche Methode gewählt wird, resultiert aus dem Akzeptanztest ein Testergebnis, welches in erster Linie eine Aussage darüber trifft, ob alle spezifizierten Anforderungen des Auftraggebers erfüllt wurden bzw. ihre Funktionalität gegeben ist oder nicht. Die Frage, die sich hier stellt ist, welche Aussage dieses Testergebnis über die Qualität des auszuliefernden Release machen kann? Ist der Release trotz eines positiven Testergebnisses bereit für ein Deployment? Diese Antwort kann das Testergebnis eines Akzeptanztests alleine nicht liefern. Andere Einflussfaktoren aus dem Requirements Engineering oder aus dem Release Management können ausschlaggebend sein für die Beantwortung dieser Frage. Ziel der vorliegenden Arbeit ist es daher diese Einflussfaktoren zu ermitteln, um die folgende Forschungsfrage zu beantworten:

> *Welche Einflussfaktoren aus den Phasen des Software Development Life Cycle ermöglichen in Verbindung mit Testergebnissen des Akzeptanztests eine genauere Beurteilung der Qualität eines Software Release?*

Dementsprechend geht die vorliegende Arbeit von folgender Hypothese aus:

> *Der Einsatz von ausgewählten Einflussfaktoren aus den Phasen des Software Development Life Cycle ermöglicht eine genauere Beurteilung der Qualität eines Software Release.*

1.1 Problemstellung

Die Wichtigkeit von Akzeptanztests in einem agilen Softwareentwicklungsprozess steht außer Frage und wird von Humble & Farley (2011) deutlich in ihrem Buch *Continuous Delivery* untermauert. Wird der Fokus jedoch lediglich auf Akzeptanztests gelegt, werden viele grundlegende Aspekte der Software außer Acht gelassen, welche vermeidbare Fehlerzustände im auszuliefernden Release hinterlassen können. Dies wird auch durch die Testpyramide von Cohn (2010) deutlich gemacht, welche besagt, dass der Fokus nicht zuerst auf Akzeptanztests gelegt werden soll, sondern auf den grundlegenden Teststufen darunter, welche einen größeren Bereich der Software abdecken.

Aus diesem falsch gelegtem Fokus resultiert das Problem, dass in agilen Softwareprojekten zu viel Vertrauen in den Testergebnissen von Akzeptanztests gelegt wird. Kritische Punkte aus anderen Phasen des SDLC werden bei der Analyse der Akzeptanztests außer Acht gelassen. Beispielsweise kann einem Software Release mit erfolgreichen Akzeptanztests, jedoch mit nicht ausreichender Code Coverage oder nicht erfolgreichen Unit Tests, keine hohe Qualität beigemessen werden.

Erfolgreiche Testergebnisse von Akzeptanztests können irreführend sein, da nicht erkannt wird, ob ein fehlerhaftes Feature eines Release, ein anderes fehlerfreies Feature aus dem Release direkt beeinflusst. In diesem Fall liegt das Problem schon früher im SDLC, nämlich während des Requirements Engineering. Mangelnde Implementierung von Traceability im Requirements Engineering Prozess durch alle Features hindurch, kann das gerade beschriebene Problem verursachen.

Durch solche Situationen lässt sich zusammengefasst sagen, dass eine Qualitätsaussage über ein Release mehr beinhaltet als nur das Testergebnis von Akzeptanztests. Über den gesamten SDLC hinweg existieren in den jeweiligen Phasen Einflussfaktoren, welche mit den Testergebnissen des Akzeptanztests zusammengeführt werden können, wodurch schlussendlich eine gehaltvollere Qualitätsaussage über ein Release getroffen werden kann. Die Aufgabe im Zuge dieser Masterarbeit ist es diese Einflussfaktoren zu ermitteln und sinnvoll miteinander innerhalb eines Modells zu integrieren, sodass bei Anwendung eine verbesserte Qualitätsaussage resultiert.

1.2 Ziele der Arbeit

Wie aus der Einleitung und der Problemstellung entnommen werden kann, ist das Hauptziel dieser Arbeit eine gehaltvollere Aussage über die Qualität eines Release treffen zu können. Dazu

soll über ausgewählte Einflussfaktoren, das Beurteilen der Qualitätsaussage von Testergebnissen aus Akzeptanztests möglich gemacht werden. Auf Basis der ermittelten Einflussfaktoren wird ein Modell zusammengestellt, welches in den Prozess eines Kundenprojektes integriert und angewendet wird. Aus diesen Punkten ergeben sich folgende Ziele für die vorliegende Meisterarbeit:

- **Ermittlung der Einflussfaktoren zur Beurteilung der Qualitätsaussage von Testergebnissen aus Akzeptanztests**

  Dazu werden Metriken aus den einzelnen Phasen des SDLC ermittelt, welche Einfluss auf die Qualität des Releases haben und mit der Teststufe Akzeptanztest in Verbindung gebracht werden können.

- **Erstellung eines Modells zur Integration der ermittelnden Einflussfaktoren**

  Ein Modell soll entwickelt werden, welches die ermittelten Einflussfaktoren sinnvoll miteinander verbindet und es ermöglicht diese in den Qualitätsprozess eines SDLC integrieren zu können.

- **Anwendung des Modells in einem Beispielprojekt**

  Das entwickelte Modell wird im Zuge des Praxisteils anhand eines Beispielprojektes validiert. Das Experiment soll einerseits die Anwendbarkeit des Modells demonstrieren, als auch die Hypothese verifizieren. Das Ergebnis der Anwendung des Modells führt letztendlich zu einer binären Entscheidung über die Auslieferung des Releases aus dem Beispielprojekt: Ja oder Nein. Die aus dem Experiment resultierende Entscheidung wird mit der Entscheidung verglichen, welche lediglich auf Basis des Akzeptanztests getroffen werden würde.

2 SOFTWAREQUALITÄT

Wie man dem Titel der vorliegenden Arbeit und der Forschungsfrage entnehmen kann, soll erforscht werden, wie eine allumfassende Qualitätsaussage über ein Software Release getroffen werden kann. Zunächst einmal soll der Begriff *Softwarequalität* definiert und beschrieben werden, wie diese gemessen werden kann, welche Metriken bereits existieren und wie Softwarequalität sichergestellt werden kann.

2.1 Definition von Softwarequalität

Das IEEE Std 610.12-19 (1990) definiert Qualität im Softwareentwicklungs-Kontext als den Grad, zu welchem ein System, eine Komponente oder ein Prozess eine spezifizierte Anforderung erfüllt. Des Weiteren definiert das IEEE Std 610.12-19 (1990) Qualität auch als den Grad, zu welchem ein System, eine Komponente oder ein Prozess die Wünsche oder Bedürfnisse des Auftraggebers oder der Endbenutzer und Endbenutzerinnen, erfüllt. Wie man aus diesen Definitionen entnehmen kann, besteht eine Verbindung zwischen Anforderungen, Qualität und Wünsche des Auftraggebers. Erst wenn die Anforderungen und Wünsche des Auftraggebers erfüllt sind, kann man von einer qualitativen Software sprachen.

Die Qualität einer Software ist jedoch ohne bestimmte Kriterien nur schwer zu definieren. Daher haben das ISO/IEC 9126-1 (2001) und das revidierte aktuellere ISO/IEC 25010 (2011) Qualitätskriterien ausgearbeitet, mit welchen man die Qualität einer Software konkretisieren kann. Bei diesen Qualitätskriterien handelt es sich um *Functional Suitability, Performance Efficiency, Compatibility, Usability, Reliability, Security, Maintainability* und *Portability*. Folgende Qualitätskriterien besitzen Sub-Qualitätskriterien, welche diese nochmals konkretisieren und die Eigenschaften ihres jeweiligen Qualitätskriteriums bilden, wie in Abbildung 1 zu sehen ist. Folgend wir jedes Qualitätskriterium zusammenfassend beschrieben.

Das erste Qualitätskriterium *Functional Suitability* beschäftigt sich mit der Funktionalität der Software, welche sich aus den Anforderungen des Auftraggebers ergibt. Dazu wird die Software auf den Erfüllungsgrad der Anforderungen gemessen und gleichzeitig auf die Korrektheit ihrer Funktionalität. Wie durch die Definition des IEEE Std 610.12-19 (1990) herausgegangen ist, wird zusammengefasst die Qualität durch den Erfüllungsgrad der Anforderungen definiert. Hier sehen wir jedoch, dass dies nur ein Teil des gesamten Qualitätsmodells darstellt. Dennoch bekommt dieses Qualitätskriterium üblicherweise die meiste Aufmerksamkeit der Softwarequalitätssicherung, da dort auch die meisten Fehler entdeckt werden. Genau dann, wenn man im Systemtest oder Akzeptanztest vom Testen von funktionalen Anforderungen spricht, handelt es sich um das Qualitätskriterium *Functional Suitability.* (ISO/IEC 25010, 2011; Wagner, 2013)

Abbildung 1 - Qualitätsmodell nach ISO/IEC 25010, angelehnt an *ISO/IEC 25010 (2011)*

Spricht man jedoch von nicht-funktionalen Anforderungen, handelt es sich unter anderem um das Qualitätskriterium *Performance Efficiency*. Bei diesem Qualitätskriterium wird hauptsächlich die Effizienz der Leistung des Systems analysiert. Dies beinhaltet beispielsweise die Reaktionszeit der Software gegenüber der Interaktion einer Benutzerin oder eines Benutzers. Dabei wird die Ressourcennutzung der Software durch andere Hardware oder Software und das maximale Leistungsvermögen der Software während der Laufzeit gemessen. Oftmals sind konkrete Anforderungen seitens des Auftraggebers für die konkrete Leistung der Software gegeben. In einigen Fällen jedoch sind diese nicht spezifiziert und man orientiert sich an Erfahrungswerten oder Standards der Branche. (ISO/IEC 9126-1, 2001; Hoffmann, 2008; ISO/IEC 25010, 2011)

Als nächstes Qualitätskriterium laut dem ISO/IEC 25010 (2011) kommt *Compatibility* oder auch *Kompatibilität*. Dieses Qualitätskriterium betrachtet hauptsächlich die Kompatibilität der Software mit anderen Softwaresystemen. Ebenfalls wird geprüft, wie problemlos ein Softwaresystem arbeitet, während es Basisfunktionalitäten und Informationen mit anderen Softwaresystemen teilt. Mit diesem Qualitätskriterium kann neben der Software selbst auch die Hardware und ihre Kompatibilität zu anderen Hardwaresystemen betrachtet werden. Im Kontext der derzeitigen Archtitektur-Trends mit Micro-Services und Clouds gewinnt dieses Qualitätskriterium immer mehr an Bedeutung. (Wagner, 2013)

Das nächste Qualitätskriterium, *Usability* oder auch *Benutzbarkeit*, bezieht sich auf alle Faktoren, welche die Interaktion zwischen dem System und der Endbenutzerin oder dem Endbenutzer bestimmen und Einfluss auf ihre oder seine Zufriedenheit haben. Aus diesem Grund meint Wagner (2013), dass dieses Qualitätskriterium weitere andere Qualitätskriterien einschließt, wie *Functional Suitability* oder *Performance*, da diese einen unmittelbaren Einfluss auf die Interaktion der Endbenutzer und Endbenutzerinnen mit dem System haben. Usability sagt auch aus wie

verständlich, erlernbar und attraktiv ein System für Endbenutzer und Endbenutzerinnen ist. (ISO/IEC 9126-1, 2001; ISO/IEC 25010, 2011)

Reliability oder auch *Zuverlässigkeit* beschreibt die Stabilität des Systembetriebs. Hier wird beispielsweise die Verfügbarkeit des Systems betrachtet und geschaut wie stabil das System trotz Ausfälle ist. Gleichzeitig beinhaltet dieses Qualitätskriterium auch die Geschwindigkeit, in der ein ausgefallenes System und dessen Daten wiederhergestellt werden. Je nach Anwendungsfall der Software kann die Zuverlässigkeit eines Systems äußerst wichtig sein, wie beispielsweise im Medizinsektor. Dieses Qualitätskriterium fokussiert sich daher hauptsächlich auf die Hardware. Die Zuverlässigkeit der Software hingegen, wird durch den fehlerfreien Betrieb unter bestimmten Konditionen definiert. (ISO/IEC 25010, 2011)

Das folgende Qualitätskriterium *Security* war im ISO/IEC 9126-1 (2001) lediglich ein Sub-Qualitätskriterien von *Functional Suitability*, jedoch etablierte es sich im ISO/IEC 25010 (2011) zu einem eigenständigen Qualitätskriterium. Es beschreibt das Ausmaß, wie sehr eine Software vor Attacken von außen geschützt ist. Des Weiteren wird auch Sicherheit im Sinne der Autorisierung betrachtet in Bezug auf Zugriffen bestimmter Benutzer und Benutzerinnen. (Wagner, 2013)

Das Ausmaß wie wartbar, erweiterbar, modifizierbar oder testbar eine Software ist, wird vom Qualitätskriterium *Maintainability* oder *Wartbarkeit* beschrieben. Im Unterschied zu den vorherigen Qualitätskriterien, welche direkt oder indirekt die Interaktion mit Benutzern und Benutzerinnen miteinschließt, betrachtet *Maintainability* zusätzlich die Sicht der Entwicklungstätigkeit. (ISO/IEC 25010, 2011; Wagner, 2013)

Als letztes Qualitätskriterium verbleibt *Portability* oder auch *Übertragbarkeit*. Dieses Qualitätskriterium ähnelt zwar dem Qualitätskriterium *Maintainability*, jedoch legt es den Fokus auf das Übertragen, Migrieren, oder Ersetzen der Software. Es evaluiert im Grunde genommen den Grad der Stärke der Bindung einer Software an eine bestimmte Plattform. (Spillner & Linz, 2012; Wagner, 2013)

2.2 Messung von Softwarequalität

Softwarequalität muss für den Auftraggeber sichergestellt werden, um sein Vertrauen in die Qualität der Software zu gewährleisten. Diese Disziplin nennt sich *Software-Qualitätssicherung* und beinhaltet verschiedene Modelle und Techniken, um dies zu erreichen (Liggesmeyer, 2009). Um in weiterer Folge die Qualität einer Software garantieren zu können, muss diese gemessen werden. In der Literatur wird auch von der sogenannten *Softwaremessung* gesprochen (Dumke & Zuse, 1994). Wie schon anfangs bei der Definition erwähnt, ist der Begriff Softwarequalität ein abstrakter Begriff, der sich schwer messen lässt. Aufgrund dessen wurden Modelle und Normen entwickelt, wie die der ISO/IEC 25010 (2011), welche die Software in Qualitätskriterien aufspaltet, sodass eine Softwaremessung möglich wird.

Die Aufspaltung der Softwarequalität in unterschiedliche Qualitätskriterien ermöglicht es die Qualität der Software mit bestimmten Verfahren und Techniken zu testen. Dies führt dazu, dass

die getesteten Qualitätskriterien eine gesammelte Aussage zur allgemeinen Softwarequalität liefern können (Rentrop, 2017). Das im vorherigen Kapitel beschriebene Qualitätskriterium *Functional Suitability* wird in vielen Softwareprojekten als das wichtigste Kriterium für das Messen der Softwarequalität betrachtet, da für den Auftraggeber die Anforderungserfüllung das konkreteste Maß dafür ist (Krypczyk , 2014). Aus diesem Grund liegt auch der Schwerpunkt in der Software-Qualitätssicherung beim *Testing*. Das Testing ist eine Methode, um die Softwarequalität sicherzustellen, welche unterschiedliche Verfahren zur Überprüfung einsetzt. Dabei ist zu beachten, dass sich das Testing über den gesamten SDLC erstreckt und nicht nur der Überprüfung der Anforderungserfüllung am Ende des SDLC dient (Hoffmann, 2008). Die Disziplin der Software-Qualitätssicherung mit ihren verschiedenen Verfahren und Techniken wird in Kapitel 6 *Quality Assurance* genauer betrachtet.

2.3 Metriken zur Softwarequalität

Metriken helfen dabei den Status und den Fortschritt einer bestimmten Aktivität, bis hin zum Fortschritt eines gesamten Projektes zu einem bestimmten Zeitpunkt festzustellen. Zugleich verhelfen sie dazu diesen Status oder Fortschritt zu quantifizieren, um damit konkrete Aussagen treffen zu können. Allerdings sind Metriken nur dann effektiv, wenn sie auch dem gesamten Team deutlich ersichtlich sind und entsprechend präsentiert werden. Ein großer Vorteil von Metriken ist die Möglichkeit Messungen vergleichen zu können, entweder mit vergangenen Werten des eigenen Projekts oder mit anderen Werten vergleichbarer Projekte. Auch wird es durch Metriken möglich konkrete Zielwerte zu definieren, die auf ihre Erreichbarkeit geprüft werden können. Zusammenfassend kann gesagt werden, dass Metriken eine organisatorische Unterstützung für das Projektmanagement darstellen und zur Besserung der Qualität der Software verhelfen. (Ebert & Dumke, 1996; Rossberg, 2014)

Metriken im Kontext der Softwarequalität werden grundsätzlich in zwei Kategorien unterteilt: *Produktmetriken* und *Prozessmetriken*. Produktmetriken beziehen sich auf den Quellcode und dienen in erster Linie zur Unterstützung des Entwicklungsteams und der Qualitätsmanager. Prozessmetriken werden im Projektmanagement eingesetzt und ermittelt Kriterien wie beispielsweise Aufwand, Kosten oder Ähnliches (Ebert & Dumke, 1996). Die vorliegende Arbeit fokussiert sich lediglich auf Produktmetriken, von welchen eine direkte Verbindung zu Features aus Akzeptanztests hergestellt werden kann. Diese werden im Kontext dieser Arbeit wie folgt *Softwaremetriken* genannt.

Der IEEE Std. 1061 (1998) definiert Softwaremetriken folgendermaßen:

„*A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality.*“

Die wesentliche Aussage dieser Definition ist, dass Softwaremetriken den Erfüllungsgrad eines Software-Attributs in numerischen Werten darstellen und damit die Qualität dieses Attributs interpretierbar machen. Somit verhelfen Softwaremetriken dem Entwicklungsteam und dem Auftraggeber einen Überblick über den Status der Entwicklung und eine Aussage über die Qualität der Software zu erhalten. (Fenton & Bieman, 2015)

Zur Kategorie der Softwaremetriken existieren bereits eine große Anzahl an Metriken. Bekannte Softwaremetriken sind beispielsweise die *zyklomatische Komplexität* oder die *Code Coverage* (Ebert & Dumke, 1996). Diese beziehen sich meist, wie vorhin erwähnt, auf den Quellcode der Software. In der vorliegenden Arbeit sollen jedoch auch Metriken aus allen anderen Phasen des SDLC ermittelt werden, welche direkt mit Features aus Akzeptanztests verknüpft werden können, um eine genauere Aussage über die Qualität eines Software Release treffen zu können. Am Ende jedes Kapitels der Phasen des SDLC befinden sich die zugehörigen ausgewählten Metriken jeder Phase.

3 SOFTWARE DEVELOPMENT LIFE CYCLE

Beim *Software Development Life Cycle (SDLC)*, welcher in der Literatur auch *Software Live Cycle* oder auch *Software Development Cycle* genannt wird, handelt es sich um jenen Prozess, welcher die gesamte Entwicklung eines Softwareproduktes betreut, angefangen von der Ermittlung der Anforderungen bis hin zur Auslieferung des Softwareproduktes. Dieser Prozess besteht aus mehreren Phasen, welche zwar in der Literatur keine einheitlichen Bezeichnungen besitzen und dessen Anzahl je nach Detailgrad ebenfalls variieren können, jedoch ist man sich über die Kernphasen des Prozesses einig. (IEEE Std 610.12, 1990; Everett & McLeod, 2007; Langer, 2012)

Softwarequalität sollte nicht erst vor der Auslieferung der Software gewährleistet werden, sondern über den gesamten SDLC hindurch. Deshalb beschäftigt sich die vorliegende Arbeit damit Metriken aus allen Phasen des SDLC zu ermitteln, um eine genauere Beurteilung der Qualität eines Software Release zu ermöglichen.

Der SDLC Prozess besteht im Kern laut IEEE Std 610.12 1990 (1990) aus folgenden Phasen: „*requirements phase, design phase, implementation phase, test phase, and, installation and checkout phase*“. Diese Phasen spiegeln sich in der vorliegenden Arbeit in den folgenden Themenabschnitten wider: *Requirements Engineering, Implementation Phase, Quality Assurance & Release Management*. In den kommenden Kapiteln, in welchen diese Themenabschnitte behandelt werden, werden Einflussfaktoren aus jeder einzelnen dieser Phasen ermittelt.

Diese zuvor genannten Phasen werden in Form von unterschiedlichen Vorgehensmodellen durchlaufen. Es existieren Vorgehensmodelle wie beispielsweise das *Wasserfallmodell*, in denen die Phasen des SDLC sequentiell abgearbeitet werden, aber auch Vorgehensmodelle wie zum Beispiel das *Spiralmodell*, in denen die Phasen iterativ abgearbeitet werden. Agile Vorgehensmodelle wie beispielsweise *Scrum* sind wesentlich jünger als die zuvor genannten Beispiele und basieren zwar auf das Prinzip der iterativen Vorgehensmodelle, jedoch forcieren diese schnellere Auslieferung und frühes Feedback, kontinuierliche Verbesserung und höhere Flexibilität (Beck, et al., 2001; Hoffmann, 2008). Aus diesem Grund bezieht sich diese Arbeit und dessen Erkenntnisse ausschließlich auf agile Vorgehensmodelle.

3.1 Continuous Delivery

Innerhalb der Vorgehensmodelle existieren verschiedene Ansätze und Methoden, welche das Vorgehensmodell operativ unterstützen. In der Welt der agilen Vorgehensmodelle existiert ein weit verbreiteter Ansatz, der sich *Continuous Delivery* nennt. Continuous Delivery bezeichnet in der Softwareentwicklung eine Vorgehensweise, welche es erlaubt automatisiert in kurzen Zyklen funktionierende Software an den Auftraggeber auszuliefern. Diese Vorgehensweise zielt darauf ab den gesamten Prozess der Auslieferung einer Software zu optimieren und kontinuierlich zu machen (Humble & Farley, 2011; Chen, 2015; Wolff, 2016). Wie in der Einleitung und in der

Forschungsfrage bereits angedeutet, basiert die vorliegende Arbeit und in Folge dessen das entwickelte Modell auf diesen Ansatz.

3.1.1 Deployment Pipeline

Das Vorgehen von Continuous Delivery baut auf den Prozess der sogenannten *Deployment Pipeline* auf, welcher aus mehreren Phasen besteht, wie in Abbildung 2 zu sehen ist. Die Deployment Pipeline bildet den gesamten Prozess, vom ersten Einchecken von Codezeilen in das Versionsverwaltungssystem der Software bis hin zur Auslieferung der Software an den Auftraggeber, ab. Der Prozess besteht aus folgenden Abschnitten: *Commit, Automated Acceptance Testing, Automated Capacity Testing, Manual Testing* & *Release*. Nur wenn ein Abschnitt erfolgreich abgeschlossen wurde, kann es mit nächsten Abschnitt weitergehen. Jeder Abschnitt gibt bei Erfolg oder Misserfolg Feedback an den vorangegangenen Abschnitt ab.

Abbildung 2 - Deployment Pipeline (Humble & Farley, 2011)

In der ersten Phase *Commit* wird der eingecheckte Code überprüft. Diese Phase soll nach dem Continuous Delivery Vorgehen automatisiert getriggert werden, sobald die Entwicklerin oder der Entwickler den Code eingecheckt hat. Je nach Projekt werden dann Unit Tests ausgeführt, welche den Code überprüfen. Außerdem wird die Struktur des Codes geprüft und, ob dieser Code mit dem bestehenden Code in Konflikt steht. (Humble & Farley, 2011; Chen, 2015; Wolff, 2016)

Da ausschließlich Unit Tests zur Sicherstellung der Qualität der Software nicht genügen, werden im zweiten Abschnitt automatisierte Akzeptanztests durchgeführt. Im vorherigen Abschnitt war es wichtig den Code rein technisch auf seine Funktionalität zu überprüfen. Dieser Abschnitt hingegen fokussiert sich auf die Funktionalität aus Sicht des Auftraggebers. (Humble & Farley, 2011; Chen, 2015; Wolff, 2016)

Die zwei ersten Abschnitte der Deployment Pipeline sind in vielen Fällen ausreichend für einen zur Auslieferung reifen Release. Der dritte und der vierte Abschnitt beinhalten weitere mögliche Tests für die Software. Im dritten Abschnitt wird die Software automatisiert auf ihre Belastbarkeit und Sicherheit getestet und im vierten Abschnitt werden manuelle explorative Tests durchgeführt und die Usability der Software wird überprüft. (Humble & Farley, 2011; Chen, 2015; Wolff, 2016)

Der letzte Abschnitt behandelt die Auslieferung der Software an den Auftraggeber. Genauer gesagt geht es um das Ausrollen des Release auf das Produktivsystem des Auftraggebers. Mit diesem Abschnitt ist ein hohes Geschäftsrisiko verbunden. In Folge dessen bedeutet das, dass hier keine Fehler gemacht werden dürfen. Deshalb strebt auch hier das Continuous Delivery einen automatisierten Prozess an. (Humble & Farley, 2011; Chen, 2015; Wolff, 2016)

Ein Continuous Delivery Prozess basierend auf der Deployment Pipeline könnte wie in Abbildung 3 dargestellt aussehen:

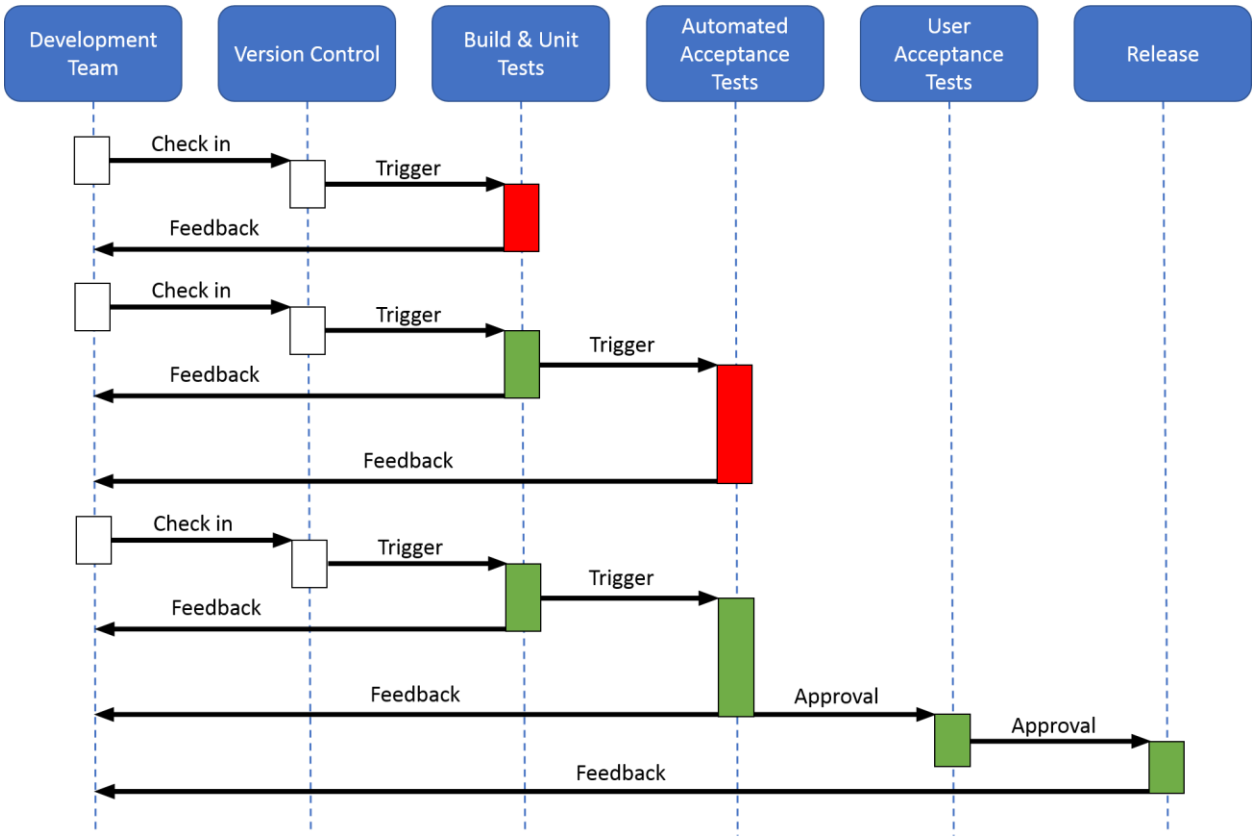


Abbildung 3 - Continuous Delivery im Deployment Pipeline Prozess (Humble & Farley, 2011)

3.1.2 Vorteile von Continuous Delivery

Der Einsatz von Continuous Delivery bringt je nach Situation des Projektes einige Vorteile mit sich. Prinzipiell erhält man durch dieses Vorgehen den Vorteil, dass der Release-Prozess optimiert wird, wie schon anfangs erwähnt. Genauer gesagt wird der Release-Prozess durch die Automatisierung der jeweiligen Prozessschritte beschleunigt und man erhält einen wiederholbaren und planbaren Prozess. Die Automatisierung allein hilft dabei das Risiko, welches durch manuelles Eingreifen entsteht, zu minimieren und schafft einen schnelleren Prozess, erhöht die Qualität und reduziert die Kosten, die durch manuelles Ausrollen der Software entsteht (Humble & Farley, 2011; Chen, 2015; Wolff, 2016). Humble & Farley (2011) meinen dazu: „*Das Leben ist viel zu kurz um unsere Wochenenden in Serverräumen mit dem Ausrollen von Applikationen zu verschwenden*."

Ein weiterer Vorteil ist das häufige und schnellere Feedback, welches durch mehrmalige Deployments entsteht. Fehler können zeitnah durch den Entwickler oder der Entwicklerin behoben werden und Feedback der Auftraggeberin oder des Auftraggebers kann zeitnah eingearbeitet werden, ohne dass sich der Entwickler oder die Entwicklerin lange in dieses Feature einarbeiten und eindenken muss. (Humble & Farley, 2011; Wolff, 2016)

Für die Auftraggeberin oder den Auftraggeber ergibt sich ein weiterer Vorteil dadurch, dass das Risiko für ein instabiles Release auf dem Produktivsystem minimiert wird. Durch die kürzeren Zyklen entstehen Releases in kürzeren Zeitabschnitten. Landet ein instabiles Release aus bestimmtem Gründen auf das Produktivsystem, kann immer noch das vorherige Release als Backup benutzt werden ohne dabei viele Funktionalitäten einzubüßen. (Wolff, 2016)

Des Weiteren hat die Auftraggeberin oder der Auftraggeber durch Continuous Delivery den Geschäftsvorteil, schneller auf Änderung im Markt reagieren zu können. Die Zeit, die gebraucht wird, um eine solche Änderung zu implementieren und auszuliefern wird durch Continuous Delivery verringert. (Chen, 2015; Wolff, 2016)

3.2 Continuous Integration

Continuous Integration kommt im ersten Schritt der Deployment Pipeline zum Einsatz und ist ein Teil des Continuous Delivery. Es bezeichnet ein Vorgehen in der Softwareentwicklung, welches vorsieht, dass die Codebasis jedes Mal integriert wird, sobald eine Entwicklerin oder ein Entwickler Code in das Versionsverwaltungssystem eincheckt. Sobald der neue Code eingecheckt ist, sieht Continuous Integration vor, dass das ganze System gebaut wird, um zu überprüfen, ob die Software kompiliert werden kann. Im Zuge dessen wird die Software durch ein automatisiertes Test-Set, welche als Integrationstests bezeichnet werden, geprüft. Continuous Integration verlangt deshalb auch, dass jede Entwicklerin und jeder Entwickler mindestens einmal pro Tag ihren Code einchecken, um damit einen Build zu triggern. Gleichzeitig soll jedoch dieser Prozess nicht dazu führen, dass das Ausführen des Builds und der Unit-Tests lokal in der Entwicklungsumgebung der Entwickler und der Entwicklerinnen vernachlässigt wird. Abbildung 4 veranschaulicht einen simplen Continuous Integration Prozess. (Fowler, Continuous Integration, 2006; Humble & Farley, 2011; Baumgartner, Klonk, Pichler, Seidl, & Tanczos, 2013)

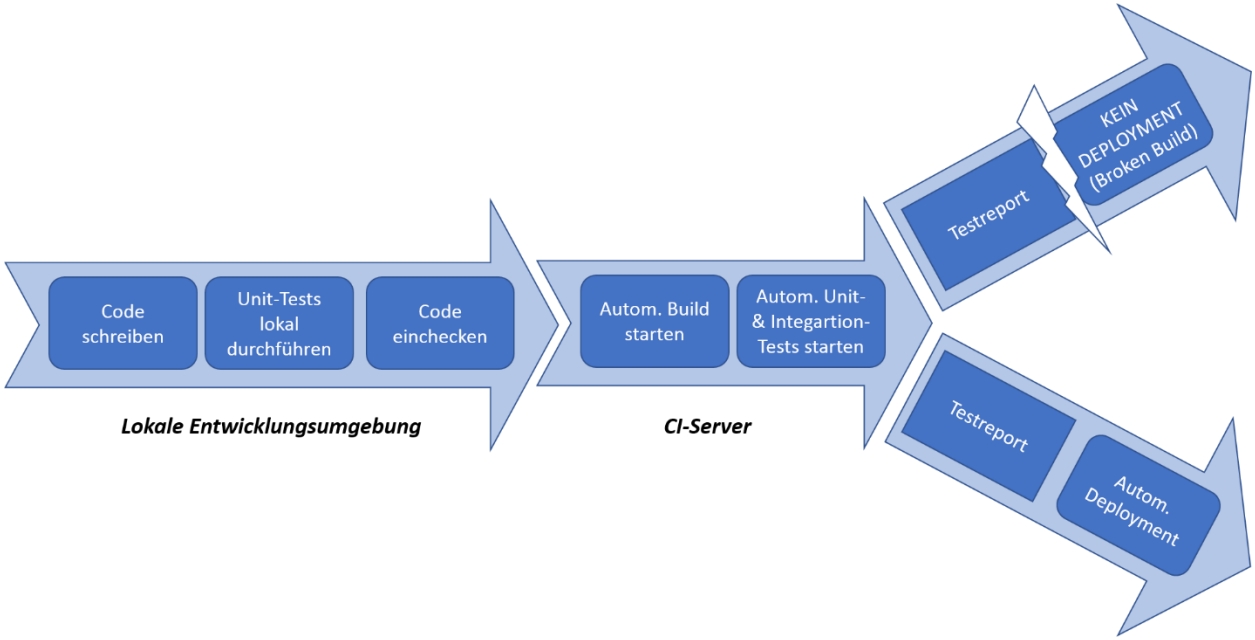


Abbildung 4 - Continuous Integration Prozess, angelehnt an *Baumgartner, Klonk, Pichler, Seidl, & Tanczos (2013)*

Dieser Prozess involviert einen weiteren Schritt worin das Entwicklungsteam in irgendeiner Form über einen fehlgeschlagenen Build oder Test benachrichtigt wird. Dies kann beispielsweise über den Email-Verkehr erfolgen oder auch über diverse Instant Messaging Dienste. Dadurch werden Entwickler und Entwicklerinnen dazu gebracht ein aufgetretenes Build-Problem oder den fehlgeschlagenen Test sofort zu lösen, um so das große Ziel von Continuous Integration zu erfüllen, ein ständig stabiles System zu gewährleisten. Desweitern erhält man durch diesen Prozess den Vorteil, dass Fehler im Code schneller und früher in der Pipeline aufgedeckt werden und somit Zeit und Geld für das Lösen der Fehler gespart wird. (Fowler, Continuous Integration, 2006; Humble & Farley, 2011; Baumgartner, Klonk, Pichler, Seidl, & Tanczos, 2013)

Der wesentliche Teil im Continuous Integration Prozess ist der automatische Build-Prozess. Wie schon vorhin erwähnt muss dieser Build-Prozess dann angestartet werden, sobald jemand aus dem Entwicklungsteam seinen neuen Code eincheckt. Der nächste Schritt im Build-Prozess ist es nun ein Set aus automatisierten Tests auszuführen und diesen neuen Code zu überprüfen. Es werden dabei hauptsächlich Unit-Tests sowie Integrationstests ausgeführt. In manchen Teams wird auch zusätzlich ein *Nightly Build* eingeplant, welcher, wie der Name schon sagt, einmal täglich während der Nacht ausgeführt wird. Die Unterscheidung zwischen den beiden Build-Prozessen liegt im Umfang der dabei ausgeführten Tests. Der Build startet nach jedem Einchecken von neuem Code nur die wichtigsten Tests zur Überprüfung an, indem der Nightly Build das komplette Test-Set ausführt. Um diesen Prozess zu bewerkstelligen, gibt es diverse kommerzielle, als auch ebenbürtige Open-Source Tools. (Fowler, Continuous Integration, 2006; Rossberg, 2014; Baumgartner, Klonk, Pichler, Seidl, & Tanczos, 2013)

3.3 Continuous Testing

Ein wesentlicher Aspekt des Continuous Delivery, welcher noch in jungen Schuhen steckt, ist *Continuous Testing* und stellt einen weiteren Teil im Zyklus des Continuous Delivery dar. Ziel von Continuous Testing ist es die Feedbackschleifen zu verkürzen und gleichzeitig auch ein kontinuierliches Feedback anzustreben. Um dieses Ziel zu erreichen wird versucht den höchsten Grad an Testautomatisierung zu etablieren, wobei manuelles Testen nicht gänzlich ausgeschlossen wird. Continuous Testing erstreckt sich über alle Teststufen, angefangen bei den Komponententests bis hin zu den Akzeptanztests. Es wird bei jedem dieser Teststufen versucht diesen bestmöglich zu automatisieren. Des Weiteren beschränkt sich Continuous Testing nicht nur auf funktionale Tests, sondern auch auf nicht-funktionale Tests, wie Performance Tests oder Security Tests. (Rady & Coffin, 2011; Ariola & Dunlop, 2014; Auerbach, 2015; Pryce, 2017)

Je fortgeschrittener und umfangreicher ein Projekt ist, desto höher ist und wird das Geschäftsrisiko gegenüber dem Auftraggeber. Continuous Testing versucht durch das rasche und kontinuierliche Feedback auf das potenzielle Geschäftsrisiko aufmerksam zu machen und dieses zu minimieren. Fehler, welche während des Prozesses des Continuous Testings gefunden werden, werden kategorisiert und priorisiert, damit sofort das Geschäftsrisiko jedes einzelnen Fehlers ersichtlich wird. Der Prozess versucht dadurch eine Aussage zu treffen, ob der neueste Release qualitätsgemäß und risikofrei an den Auftraggeber ausgeliefert werden kann oder nicht.

Gleichzeitig wird dadurch der gesamte Prozess des Continuous Delivery kürzer, da Fehler schneller gefunden werden. (Ariola & Dunlop, 2014; Philipp-Edmonds, 2014; Pryce, 2017)

3.3.1 Testautomatisierung

Im Kontext des Softwaretestens ist das automatisierte Testen von Software nichts anderes als die Verwendung einer Testsoftware, welche manuelle Test-Tätigkeiten des Testers oder der Testerin übernimmt. Testautomatisierung kann in allen Teststufen angewendet werden, aber hat jedoch auch ihre Grenzen. In vielen Fällen können nämlich nicht alle Tätigkeiten des Testers oder der Testerin automatisiert werden. Oftmals entscheidet der Tester oder die Testerin intuitiv etwas Bestimmtes zu testen und hat dabei sein oder ihre eigenen Ideen und ein individuelles Vorgehen. Eine Testsoftware ist somit nur ein unterstützendes Werkzeug für den Tester oder die Testerin und ersetzt diese Rolle nicht. In jedem Projekt muss abgewogen werden in welchem Bereich und inwieweit automatisierte Tests sinnvoll sind, wobei Testautomatisierung in agilen Vorgehensmodellen nahezu unausweichlich ist, im Unterschied zu traditionellen Vorgehensmodellen. (Dustin, Rashka, & Paul, 2001; Baumgartner, Klonk, Pichler, Seidl, & Tanczos, 2013; Seidl, Bucsics, Gwihs, & Baumgartner, 2015)

Testautomatisierung bietet die Möglichkeit sich einen Überblick über den derzeitigen Zustand der Software zu verschaffen und diesen in Folge dessen unter Kontrolle zu haben. Diese Möglichkeit ergibt sich dadurch, dass ein bestimmtes automatisiertes Test-Set mit verschiedenen Testfällen, welches einen festgelegten Output hat, wiederholte Male ausgeführt werden kann. Somit können Testdurchläufe miteinander verglichen werden und bilden dadurch eine Metrik, welche benutzt werden kann, um beispielsweise die Qualität eines neuen Release beurteilen zu können. Beim manuellen Test hingegen können die Testschritte von Durchlauf zu Durchlauf variieren und sind daher weniger zuverlässig und schwer messbar. (Dustin, Rashka, & Paul, 2001; Seidl, Bucsics, Gwihs, & Baumgartner, 2015)

3.3.2 Vorteile der Testautomatisierung

In der Literatur ist man sich einig, dass Testautomatisierung im Kontext des Softwaretestens Zeit- und Kosteneinsparungen mit sich bringt und in Folge dessen die Softwarequalität steigert. Gründe sind beispielsweise komplexe Anforderungen, welche eine große Anzahl an Testfällen benötigen. Diese zu testen bringt einen großen Testaufwand mit sich, welcher Zeit und Geld aufbraucht. Des Weiteren werden Tests nicht nur einmal durchgeführt, sondern bei jedem neuen Stück Code, sogenannte „Regressionstests“, oder bei jedem Release, sogenannte „Smoke-Tests“. Regressionstests sind dazu da, um bei jeder Veränderung der Software zu überprüfen, ob alle anderen Funktionalitäten der Software immer noch einwandfrei funktionieren. Smoke-Tests überprüfen die wesentlichen Funktionalitäten der Software nachdem ein Release bereit ist für die Auslieferung an den Auftraggeber und sind eine kürzere Variante des Regressionstests. Beide Arten von Tests jedes Mal manuell durchzuführen, besonders Regressionstest, erfordert viel Zeit und Geduld. Der Tester oder die Testerin werden auch fehleranfällig und unkonzentriert, je länger so ein Test dauert. Daher eignen sich besonders für

diese zwei Fälle automatisierte Tests einzuführen, um auch hier Zeit und Geld einzusparen und die Qualität der Software hochzuhalten. (Dustin, Rashka, & Paul, 2001; Hoffmann, 2008; Crispin & Gregory, 2009; Baumgartner, Klonk, Pichler, Seidl, & Tanczos, 2013; Malhotra & Chug, 2013; Saleem, Qadri, Hassan, Bashir, & Ghafoor, 2014; Seidl, Bucsics, Gwihs, & Baumgartner, 2015)

Dasselbe Prinzip gilt natürlich auch für die Teststufen am Anfang des V-Modells. Sowohl Komponententests als auch Integrationstests würden ohne Automatisierung sehr viel Aufwand benötigen. Mit jeder Veränderung der Software und zusammenführen der neu entwickelten Codezeilen werden diese beiden Tests automatisiert angeworfen und liefern ein zuverlässiges Ergebnis und somit wird die Qualität und Stabilität des Codes gewährleistet. (Dustin, Rashka, & Paul, 2001; Seidl, Bucsics, Gwihs, & Baumgartner, 2015)

Wie bereits im Abschnitt 3.1.1 erwähnt, werden mit Hilfe von Testautomatisierung Wiederholbarkeit und Vergleichbarkeit ermöglicht und somit können Metriken zur Analyse und Optimierung geschaffen werden. Dadurch wird es möglich kritische Fragen zum Release oder zum aktuellen Status der Code Qualität zu beantworten. (Dustin, Rashka, & Paul, 2001; Seidl, Bucsics, Gwihs, & Baumgartner, 2015)

Testautomatisierung wirkt ebenso der Monotonie mancher bestimmter manuellen Tests entgegen, da ein Tester oder eine Testerin mit der Zeit sehr fehleranfällig wird, je öfter diese Testschritte durchgeführt werden müssen, wobei durch Automatisierung dieser Tests die Fehleranfälligkeit in diesem Fall auf null sinkt. Durch diese Optimierung wird, wie auch schon weiter oben erwähnt, dem Tester oder der Testerin mehr Zeit für andere Aufgaben zur Sicherung der Qualität der Software verschafft. (Dustin, Rashka, & Paul, 2001; Seidl, Bucsics, Gwihs, & Baumgartner, 2015)

Ein weiterer Vorteil ist die vereinfachte Reproduzierbarkeit eines Fehlers, da es oftmals schwer ist nach dem Entdecken eines Fehlers beim manuellen Testen die Testschritte dafür nachzustellen. Bei einem automatisierten Test hingegen ist es wesentlich einfacher, da man den Test nochmals ausführen oder die Schritte im Testfall betrachten kann. (Dustin, Rashka, & Paul, 2001; Seidl, Bucsics, Gwihs, & Baumgartner, 2015)

Außerdem machen es automatisierte Tests einfacher die Funktionalität eines Systems zu verstehen. Dies ist vor allem dann hilfreich, wenn jemand neu zum Projekt dazustößt und der Wissensträger zurzeit nicht verfügbar ist, um denjenigen mit der Software vertraut zu machen. Durch ein Abspielen oder Betrachten der Testfälle können auf einfache Art und Weise die Funktionalitäten der Software erlernt werden. (Dustin, Rashka, & Paul, 2001; Seidl, Bucsics, Gwihs, & Baumgartner, 2015)

Zuletzt soll die Wiederholbarkeit der Tests nochmals erwähnt werden, wobei diesmal der Fokus auf den Zeitpunkt der Ausführung liegt. Dieser Zeitpunkt kann nämlich bei automatisierten Tests beliebig gewählt werden im Vergleich zu manuellen Tests. Beispielsweise ist es üblich die Tests nachts auszuführen und in weiterer Folge die Testergebnisse am Morgen zu analysieren. Dadurch kann die Zeit des Testers oder der Testerin effizient genutzt werden. (Dustin, Rashka, & Paul, 2001; Seidl, Bucsics, Gwihs, & Baumgartner, 2015)

3.3.3 Bewertung von Testergebnissen

Die Bewertung und Beurteilung von Testergebnissen ist das Gegenüberstellen des tatsächlichen Testergebnisses zu dem erwarteten Testergebnis. Auf Basis dieser Gegenüberstellung werden Entscheidungen abgeleitet, um darauf basierend Maßnahmen einzuleiten. Je nachdem wie das Testergebnis aussieht, können Fehler festgehalten werden, welche in Folge dessen behoben werden müssen. Die Bewertung kann auch ausschlaggebend dafür sein, dass der Release nicht an den Auftraggeber ausgeliefert werden kann. Es könnte auch so weit gehen, dass Meilensteine, wie zum Beispiel das Go-Live, dadurch verschoben werden müssen. Die Bewertung könnte jedoch auch lediglich darauf hinweisen, dass die Testfälle unzureichend sind und weitere hinzukommen oder die bestehenden verbessert werden müssen. Somit wird ersichtlich, dass die Bewertung der Testergebnisse eine wesentliche Maßnahme für das gesamte Projekt darstellt. (Schmitz, Bons, & Megen, 1983; Denert, 1992; Dustin, Rashka, & Paul, 2001; Hoffmann, 2008; Liggesmeyer, 2009; Spillner & Linz, 2012; Winter, Ekssir-Monfared, Sneed, Seidl, & Borner, 2012; Baumgartner, Klonk, Pichler, Seidl, & Tanczos, 2013)

Um überhaupt eine Bewertung von Testergebnissen vornehmen zu können, werden Qualitätsrichtlinien und klare Klassifikationen von Schweregrad und Priorität von Fehlern benötigt. Die Priorität würde aussagen wie dringend dieser Fehler aus der Sicht des Auftraggebers behoben werden muss und der Schweregrad würde bestimmen wie schwerwiegend sich der Fehler auf die Funktionalität der Software auswirkt. Beides muss klar definiert sein, um eine klare Bewertung durchführen zu können und auf Basis dessen Entscheidungen treffen und Maßnahmen ableiten zu können. Dabei kann eine Tabelle zur Übersicht sehr hilfreich sein, worin die Testergebnisse, der vermutliche Grund des Fehlers, die Maßnahmen zur Fehlerbehebung und der Lösungsvorschlag abgebildet sind. (Dustin, Rashka, & Paul, 2001)

Die erwartenden Testergebnisse, mit denen das tatsächliche Testergebnis verglichen wird, basieren auf den Spezifikationen der Anforderungen oder den Akzeptanzkriterien der User Stories des Auftraggebers. Die Spezifikationen jedes Mal heranzuziehen beim Bewerten der Testergebnisse kann sehr umfangreich und mühsam werden, je größer und komplexer ein Projekt wird. Eine Lösung dafür sind Testverfahren, aus den diversifizierenden Testtechniken. Diversifizierenden Testtechniken verfolgen den Ansatz Testergebnisse miteinander zu vergleichen. Dazu zählen hauptsächlich *Back-to-Back-Tests* und *Mutationen-Tests*. Laut Liggesmeyer (2009) können unter Umständen auch Regressionstests in diese Kategorie eingeordnet werden. Beispielsweise wird das Testergebnis des Regressionstests eines bestimmten Software-Builds mit dem des vorangegangenen Software-Builds verglichen. Durch die Verwendung dieses Verfahrens kann die Bewertung der Testergebnisse automatisiert werden. (Liggesmeyer, Sneed, & Spillner, 1992; Liggesmeyer, 2009)

Dustin, Rashka, & Paul (2001) warnen allerdings davor automatisierten Testergebnissen blind zu vertrauen, selbst wenn diese darauf hinweisen, dass alles in Ordnung ist. Hinter einem positiven Resultat könnte immer noch ein Fehler in der Software versteckt sein. Deshalb ist es wichtig von Zeit zu Zeit stichprobenartig alle Testfälle zu überprüfen, um festzustellen, ob diese auch korrekt durchgeführt werden.

4 REQUIREMENTS ENGINEERING

Das *Requirements Engineering* ist ein systematischer Ansatz zum Verwalten und Organisieren von Anforderungen mit dem Ziel ein Projekt erfolgreich zu leiten. Das Requirements Engineering bildet den allerersten Schritt im SDLC. Zu den Aufgaben dieses Ansatzes zählen hauptsächlich das Identifizieren, Analysieren, Dokumentieren, Validieren, Kommunizieren und Nachverfolgen von Anforderungen (Fuchs, Fuchs, & Hauri, 2002; Berenbach, Paulish, Kazmeier, & Rudorfer, 2009; Hull, Jackson, & Dick, 2011). Klaus Pohl & Chris Rupp (2015) fokussieren und reduzieren diese Aufgaben auf die vier wesentlichsten: Ermitteln, Dokumentieren, Prüfen und Verwalten von Anforderungen. Im folgenden Kapitel werden alle im Kontext dieser Arbeit relevanten Disziplinen dieses Ansatzes behandelt. Dazu gehört die Disziplin der Priorisierung von Anforderungen und auch deren Nachverfolgung. Des Weiteren wird auch auf Anforderungen aus Sicht der Qualitätssicherung eingegangen.

Anforderungen werden am Anfang des Projektes gänzlich erhoben. Dies ist bei traditionellen Vorgehensmodellen oft der Fall. Bei agilen Vorgehensmodellen, wie zum Beispiel *Scrum* oder *Kanban,* müssen nicht alle Anforderungen am Projektanfang zur Gänze ermitteln werden. Das Ermitteln der Anforderungen in agilen Vorgehensmodellen gestaltet sich kontinuierlich über den gesamten Projektverlauf hindurch. (Chemuturi, 2013; Pohl & Rupp, 2015)

Anforderungen definieren sich laut IEEE Std 610.12-1990 (1990) und der Übersetzung von Klaus Pohl & Chris Rupp (2015) wie folgt:

„*Eine Anforderung ist:*

(1) Eine Bedingung oder Fähigkeit, die von einem Benutzer (Person oder System) zur Lösung eines Problems oder zur Erreichung eines Ziels benötigt wird.

(2) Eine Bedingung oder Fähigkeit, die ein System oder Teilsystem erfüllen oder besitzen muss, um einen Vertrag, eine Norm, eine Spezifikation oder andere, formell vorgegebene Dokumente zu erfüllen.

(3) Eine dokumentierte Repräsentation einer Bedingung oder Eigenschaft gemäß (1) oder (2).“

Anforderungen sind also wesentlich für ein System und die Basis jedes Projektes. Sie beschreiben dessen Verhalten und Charakteristiken und tragen dazu bei dem Auftraggeber einen Wert zu liefern. Anforderungen sind auch die Basis für alle Schritte des SDLC, angefangen beim Entwickeln der Anforderungen bis hin zum Testen dieser. Damit diese für alle Personen, welche im Projekt involviert sind, verständlich sind, werden die Anforderungen in natürlicher Sprache formuliert und für alle zugänglich gemacht. Für das Entwickeln der Anforderungen, werden diese noch genauer in Form von Akzeptanzkriterien verfasst, sodass die Anforderung für Entwickler und Entwicklerinnen leichter verständlich und präziser formuliert sind. (Young, 2004; Hull, Jackson, & Dick, 2011)

Anforderungen und deren Akzeptanzkriterien stellen die Basis für den Akzeptanztest dar, wenn der Auftraggeber das System auf dessen Vollständigkeit und Erfüllung überprüft. Deshalb ist es schon von vornherein wichtig diese präzise zu formulieren, sodass diese einerseits ohne

Missverständnisse getestet werden können und andererseits keinen Interpretationsspielraum zulassen. (Hull, Jackson, & Dick, 2011)

4.1 Arten von Anforderungen

Anforderungen kann man in viele verschiedene Arten unterteilen, wie es beispielsweise Ralph Young (2004) in seinem Buch „*The Requirements Engineering Handbook*“ macht, jedoch unterteilen die meisten Autoren Anforderungen in zwei wesentliche Arten: Funktionale Anforderungen und Nicht-funktionale Anforderungen/Qualitätsanforderungen. (Hull, Jackson, & Dick, 2011; Chemuturi, 2013; Pohl & Rupp, 2015)

Bei den funktionalen Anforderungen handelt es sich um jene Anforderungen, welche die fachliche Spezifikation des Systems beschreiben. Durch diese Anforderungen beschreibt der Auftraggeber wie das System zu funktionieren hat und welches Ergebnis erwartet wird. Diese Art von Anforderung stellt auch die Mehrheit der Anforderungen in einem Projekt dar. Das sind auch jene Anforderungen, welche durch den gesamten SDLC wandern bis sie dem Auftraggeber ausgeliefert werden. Wie bereits vorhin erwähnt, enthalten diese Anforderungen auch meist Akzeptanzkriterien, welche diese Anforderung noch genauer beschreiben und definieren. Diese werden dann während des Entwickelns herangezogen, für die manuellen oder automatisierten Tests verwendet und vom Auftraggeber zum verifizieren des Systems benutzt. (Young, 2004; Hull, Jackson, & Dick, 2011; Pohl & Rupp, 2015)

Nicht-funktionale Anforderungen oder auch Qualitätsanforderungen beziehen sich mehr auf das allgemeine Verhalten des Systems als Ganzes und behandeln keine bestimmte Funktionalität, sondern verschiedene Qualitätsmerkmale. Beispiele für solche Qualitätsmerkmale wären Sicherheit, Performanz, Verfügbarkeit, Skalierbarkeit, Benutzerfreundlichkeit oder Portabilität. Nicht-funktionale Anforderungen fließen meist nicht in den Software-Lebenszyklus ein, wie es die funktionalen Anforderungen tun, sondern sind unabhängig davon. Dennoch sind Anforderungen dieser Art ebenso wichtig wie die funktionalen Anforderungen und werden genauso vom Auftraggeber verifiziert und abgenommen. (Chemuturi, 2013; Pohl & Rupp, 2015)

Das Requirements Engineering hat viele untergeordnete Aufgaben, wie bereits am Anfang des Kapitels beschrieben, jedoch fokussiert sich diese Arbeit lediglich auf jene Aufgaben, welche mögliche Einflussfaktoren auf das automatisierte Beurteilen von automatisierten Akzeptanztests besitzen.

4.2 Priorisierung von Anforderungen

Das Priorisieren von Anforderungen im Requirements Engineering ist erforderlich, um die Umsetzung dieser besser planen zu können. Insbesondere wenn die Zeit für die Umsetzung des Projekts begrenzt ist. Diese Planung fließt schlussendlich in die Release Planung ein und entscheidet, welche Anforderungen in welcher Reihenfolge und zu welchem Zeitpunkt ausgeliefert werden. Somit wird sichergestellt, dass jene Anforderungen mit dem höchsten

Business Value zuerst umgesetzt werden. Aber auch andere Gründe neben der verfügbaren Zeit können ausschlaggebend sein für die Priorisierung, wie zum Beispiel das verfügbare Budget, das Risiko der einzelnen Anforderungen oder die Relevanz aus Sicht des Auftraggebers. Diese Priorisierung der Anforderungen ist zwar grundsätzlich Aufgabe des Requirements Engineer oder des Projektleiters, wird aber in den meisten Fällen vom Auftraggeber getroffen. Diese wiederum versuchen die Anforderungen aus Sicht der Endbenutzer zu priorisieren. (Berenbach, Paulish, Kazmeier, & Rudorfer, 2009; Chemuturi, 2013; Wiegers & Beatty, 2013; Pohl & Rupp, 2015)

Zum Priorisieren von Anforderungen gibt es einige Ansatzweisen und Techniken. Zwar gleichen sich die meisten von ihnen, jedoch betrachtet jede dieser Techniken Anforderungen aus einem anderen Blickwinkel. Die Bekanntesten von ihnen werden etwas näher vorgestellt.

4.2.1 Ein-Kriteriums-Klassifikation

Diese Art der Priorisierung ist eine sehr leichte in ihrer Anwendung und betrachtet lediglich einen Gesichtspunkt in Bezug auf Anforderungen, nämlich die geschäftliche Relevanz der Anforderung. Dabei werden Anforderungen auf drei Klassen aufgeteilt: *Mandatory*, *Optional* und *Nice-to-have*. Alle Anforderungen in der Klasse *Mandatory* haben die höchste Priorität und müssen unbedingt umgesetzt werden und werden dann üblicherweise auch als erstes umgesetzt. Anforderungen, welche der Klasse Optional zugeordnet sind müssen nicht zwingend umgesetzt werden, da sie auch keinen zwingenden oder maßgeblichen geschäftlichen Vorteil bieten. *Nice-to-have* Anforderungen hingegen haben die niedrigste Priorität und müssen gar nicht in der Umsetzung berücksichtigt werden. Wenn es die Ressourcen, die Zeit und das Budget, erlauben können diese Anforderungen selbstverständlich berücksichtigt werden. (Pohl & Rupp, 2015)

4.2.2 MoSCoW

Die Priorisierungstechnik *MoSCoW* teilt Anforderungen in vier Kategorien ein. Diese vier Kategorien lauten: *Must, Should, Could* und *Won't* und bilden somit das Akronym *MoSCoW*. Diese Technik ähnelt stark der Einteilung der Ein-Kriteriums-Klassifikation, jedoch beinhaltet diese Technik eine zusätzliche Kategorie. Wie bei der Ein-Kriteriums-Klassifikation müssen Anforderungen der Kategorie *Must* zwingend umgesetzt werden. Anforderungen aus der Kategorie *Should* hingegen sollten auch umgesetzt werden, jedoch nicht mit dieser hohen Dringlichkeit wie Anforderungen aus der *Must* Kategorie. Diese Anforderungen können zu einem späteren Zeitpunkt umgesetzt werden, jedoch können sie in Bezug auf die Endauslieferung des Systems nicht ausgelassen werden. Jene Anforderungen aus der *Could* Kategorie ähneln der *Optional* Klasse aus der Ein-Kriteriums-Klassifikation. Auch hier ist die Umsetzung dieser Anforderung nicht zwingend notwendig für das Projekt. Zuletzt bleibt noch die *Won't* Kategorie, in welche deren Anforderungen zwar aus der aktuellen Planung ausgeschlossen werden, jedoch können diese Anforderungen für die Zukunft dennoch berücksichtigt werden. (Brennan, 2009; Watkins, 2009; Wiegers & Beatty, 2013)

4.2.3 Three-level scale

Eine weitere Technik nennt sich *Three-level scale* und basiert auf einer Matrix mit zwei Dimensionen, Relevanz und Dringlichkeit. Diese zwei Dimensionen werden miteinander in Beziehung gesetzt, um die Priorität der Anforderungen ermitteln zu können. Daraus entstehen vier Quadranten: *High Priority, Medium Priority, Low Priority* und *Don't do these*, wie in Abbildung 5 zu sehen ist. Aus diesen Quadranten resultieren schlussendlich drei Kategorien, zu denen Anforderungen zugeordnet werden, nämlich *High Priority, Medium Priority* und *Low Priority*. Die vierte Kategorie *Don't do these* wird dabei außer Acht gelassen, da die Anforderungen dieser Kategorie nicht umgesetzt werden. (Wiegers & Beatty, 2013)

Im Unterschied zur *Ein-Kriteriums-Klassifikation* und der *MoSCoW* Technik wird hier zusätzlich die Dimension der Dringlichkeit und nicht nur die geschäftliche Relevanz berücksichtigt. Somit kann die Priorität einer Anforderung noch genauer ermittelt und mit anderen unterschieden werden.

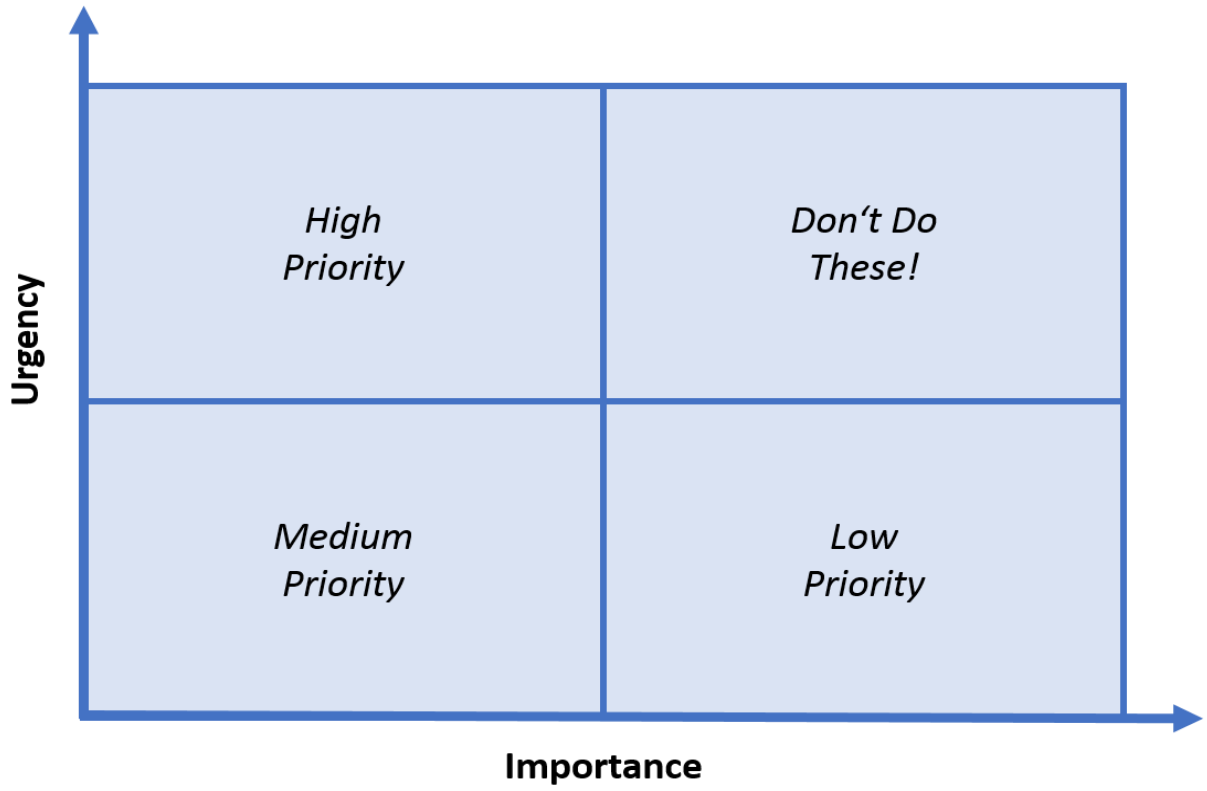


Abbildung 5 - Three-level scale (Wiegers & Beatty, 2013)

4.2.4 Kano-Modell

Mit der Nutzung des Kano-Modells zur Priorisierung von Anforderungen wird ein neuer Gesichtspunkt betrachtet. Das zu implementierende System wird aus dem Gesichtspunkt des Marktes und dessen Entwicklung betrachtet und folglich aus dem Blickwinkel der Endbenutzer

und ihrer Zufriedenheit. Wie in Abbildung 6 zu sehen ist, teilt das Kano-Modell diese Sicht in drei verschiedene Kategorien ein: *Basismerkmale*, *Leistungsmerkmale* und *Begeisterungsmerkmale*. Eine Anforderung, welche als Basismerkmal klassifiziert wird, ist eine Mindestanforderung für das System, damit es auf dem Markt bestehen kann. Die Abwesenheit eines solchen Merkmals löst schwere Unzufriedenheit bei der Endbenutzerin oder beim Endbenutzer aus. Anforderungen aus der Kategorie der Leistungsmerkmale wirken sich positiv auf das System am Markt aus und erhöhen in Folge dessen die Zufriedenheit der Endbenutzer und Endbenutzerinnen. Anforderungen aus der Kategorie der Begeisterungsmerkmale schaffen einen Wettbewerbsvorteil am Markt und erhöhen die Kundenzufriedenheit der Endbenutzer und Endbenutzerinnen exponentiell, da sie diese Funktionalitäten nicht erwartet hätten. (Kano, Seraku, Takahashi, & Tsuji, 1984; Brennan, 2009; Pohl & Rupp, 2015)

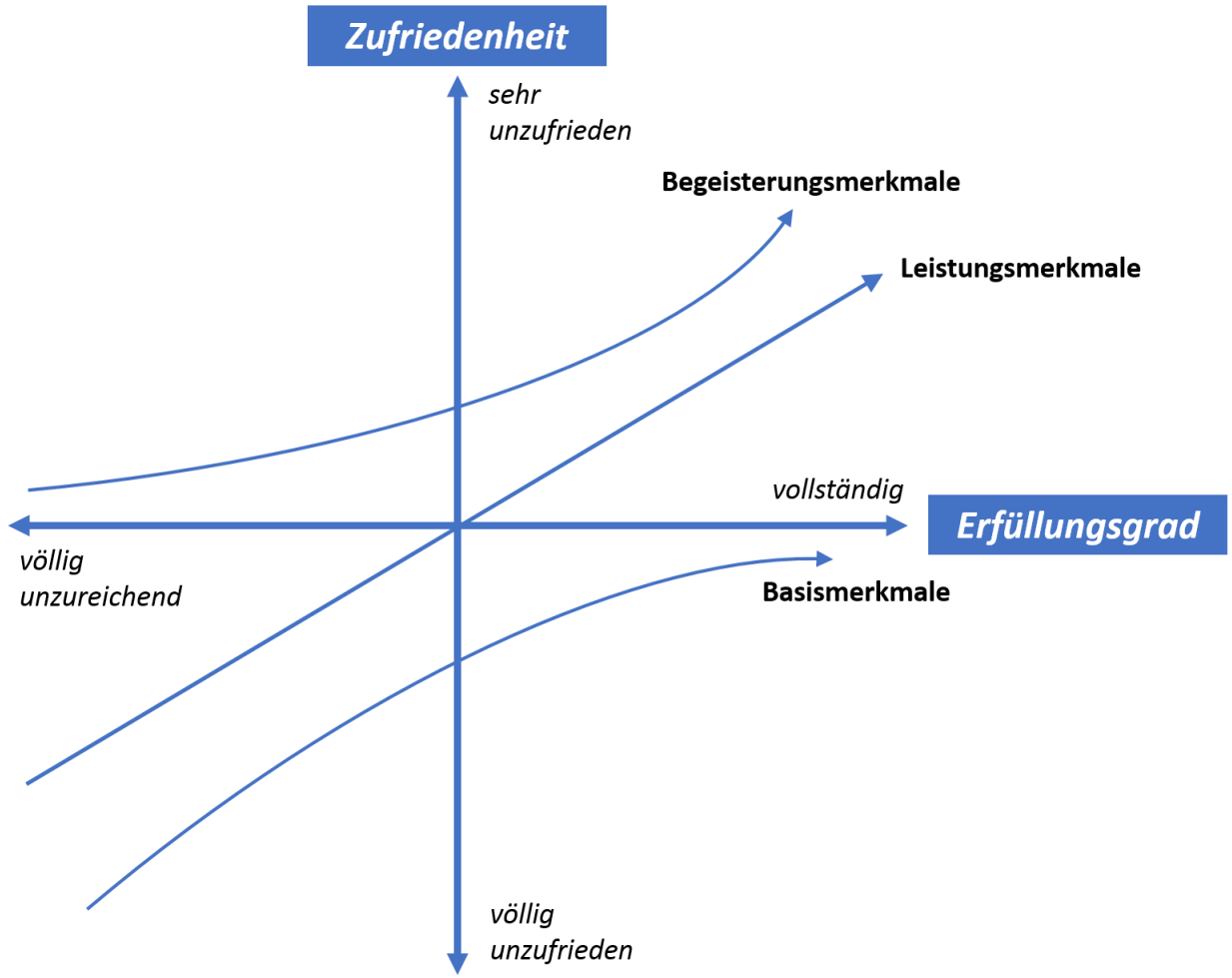


Abbildung 6 - Kano Modell, angelehnt an *Kano, Seraku, Takahashi, & Tsuji (1984)*

4.3 Verfolgbarkeit von Anforderungen

Die Sicherstellung der Verfolgbarkeit von Anforderungen ist essenziell in einem Softwareprojekt und angesichts des Kontextes dieser Arbeit ebenso für das Testen von Anforderungen. Unter

anderem wird dieser Aspekt als Qualitätskriterium für Anforderungen gesehen (Pohl & Rupp, 2015). Gotel & Finkelstein (1994) definieren die Verfolgbarkeit von Anforderungen als „*das Beschreiben und Folgen des Lebens einer Anforderung, sowohl vorwärts als auch rückwärts*“. Der IEEE Std 610.12-1990 (1990) definiert diesen Aspekt als den Grad der Zuordenbarkeit von Design und Anforderung einer Software. Es handelt es sich hier um die Verknüpfung von Anforderungen zu bestimmten Spezifikationen, Testfällen oder anderen Anforderungen. (Chemuturi, 2013; Rossberg, 2014)

Die Verfolgbarkeit von Anforderungen teilt sich grundsätzlich in zwei Arten auf: *Pre-Requirements-Specification(RS)-Traceability* und *Post-RS-Traceability*. Unter Pre-RS-Traceability versteht man die Verfolgbarkeit der Anforderung bis zu ihrem Ursprung der Spezifikation, aus welcher die Anforderung entstanden ist. Post-RS-Traceability hingegen bezeichnet die nachgelagerte Verfolgung von Artefakten, die mit einer bestimmten Anforderung in Verbindung stehen, wie zum Beispiel die Testfälle einer Anforderung (Gotel & Finkelstein, 1994). Neben dieser Hauptunterscheidung in zwei Arten gibt es noch eine dritte Art: die *Verfolgbarkeit zwischen den Anforderungen*. Diese definiert die Abhängigkeit von Anforderungen, welche sich ergänzen oder eine Ableitung oder Änderung einer anderen Anforderung darstellen. (Pohl & Rupp, 2015)

Grundsätzlich erlangt man durch die Verfolgbarkeit von Anforderungen eine gewisse Übersicht im Projekt und über dessen Fortschritt (Hull, Jackson, & Dick, 2011; Rossberg, 2014), jedoch gibt es eine Reihe weitere Vorteile dieses Aspekts. Es erleichtert beispielsweise die Zuordnung von Entwicklungsaufwänden zu einer bestimmten Anforderung und erleichtert somit die Verrechnung und weitere Aufwandsschätzungen im Verlauf des Projekts. Durch die Verfolgbarkeit von Anforderungen ist die Feststellung einer Auswirkung auf eine Anforderung, durch eine Änderung einer verknüpften Anforderung, mit wenig Aufwand möglich und hilft frühzeitig Probleme festzustellen. (Pohl & Rupp, 2015)

Ein weiterer Vorteil dieses Aspekts ist die Unterstützung beim Definieren des Rahmens (Scope) des Softwareprojektes. Meist bleibt der definierte Rahmen zu Anfang des Projektes nicht derselbe wie zum Ende des Projektes. Oftmals werden Anforderungen hinzugenommen, herausgenommen oder gar verändert. Die Verfolgbarkeit von Anforderungen hilft dabei über den gesamten Projektverlauf hinweg die Übersicht über den Scope zu bewahren und Änderungen festzustellen. (Chemuturi, 2013; Rossberg, 2014)

Die Verfolgbarkeit von Anforderungen hilft dabei Testabdeckung der Anforderungen festzustellen und diese dadurch zu erhöhen. Des Weiteren wird durch die vorhin erwähnte Analyse von Auswirkungen auf Anforderungen durch Anforderungsänderungen dabei geholfen, die damit verknüpften Testfälle dementsprechend anzupassen. (Berenbach, Paulish, Kazmeier, & Rudorfer, 2009; Chemuturi, 2013)

4.4 Testen von Anforderungen

Requirements Engineering und Akzeptanztests stehen unmittelbar miteinander in Beziehung. Anforderungen, die im Zuge des Requirements Engineerings erstellt wurden, werden in der

vierten Teststufe auf die Korrektheit ihrer Implementierung überprüft. Aus diesem Grund muss eine Anforderung so geschrieben sein, dass sie auch prüfbar ist und durch einen Testfall abgedeckt werden kann. Diese Eigenschaft wird auch als Qualitätskriterium einer Anforderung angesehen. (Berenbach, Paulish, Kazmeier, & Rudorfer, 2009; ISO/IEC & IEEE, 2011; Chemuturi, 2013; Pohl & Rupp, 2015)

Aus dem vorherigen Abschnitt kann abgeleitet werden, dass eine Anforderung, welche die Prüfbarkeit in ihrer Formulierung berücksichtigt, mit weniger Aufwand validiert werden kann. Wird nun dabei ein bestimmtes Schema oder Model benutzt, können Akzeptanztest leichter automatisiert werden. Eine Technik, welche auf dieses Vorgehen zurückgreift, nennt sich *Behavior-Driven-Development* (BDD). Diese versucht mithilfe eines bestimmten textuellen Schemas Anforderungen so zu formulieren, dass diese von einem Automatisierungs-Tool sofort verstanden wird und den automatisierten Testfall generiert. Somit wird die Generierung von automatisierten Testfällen erleichtert und es bedarf auch keinen Programmierer oder Programmiererin, um die Testfälle zu generieren. (Berenbach, Paulish, Kazmeier, & Rudorfer, 2009; Gärtner, 2012).

4.5 Metriken aus dem Requirements Engineering

Im Bereich des Requirements Engineerings können Metriken dazu verwendet werden, um eine Aussage über die Qualität oder Stabilität einer Anforderung (und in weiterer Folge einer Funktionalität) zu treffen (Chemuturi, 2013). Es handelt sich dabei weniger um Metriken, welche eine Aussage über das Requirements Engineering selbst treffen, sondern mehr um Metriken, welche zum Schluss dabei helfen eine Aussage über die Qualität der Features in einem Software Release treffen zu können. Folgende Metriken können manuell ermittelt werden oder auch mithilfe von Ticket-Verwaltungstools.

4.5.1 Volatility of Requirements

Die *Volatilität von Anforderungen* oder auch *Änderungsrate von Anforderungen* misst wie oft eine Anforderung aktualisiert oder bearbeitet wurde. Diese Änderungsrate kann ein Hinweis darauf sein, dass ein höheres Risiko bei jener Anforderung zu Grunde liegt. Daraus kann abgeleitet werden, dass beim Testen ein stärkerer Fokus auf die Funktionalität, welche aus jener Anforderung entstanden ist, gelegt werden soll. Auch kann diese Metrik als Entscheidungshilfe genutzt werden, ob eine Funktionalität mit einem Release ausgeliefert werden soll oder nicht. (Berenbach, Paulish, Kazmeier, & Rudorfer, 2009; Pohl & Rupp, 2015)

4.5.2 Requirements Test Case Coverage

Diese Metrik hilft dabei festzustellen, ob alle Anforderungen zumindest einen Test Case beinhalten. Der Hintergrund ist jener, dass, auch wenn eine Funktionalität erfolgreich getestet wird, dennoch auf spezielle Konditionen und Varianten innerhalb der Funktionalität vergessen

wird. Es soll hier somit sichergestellt werden, dass schon im Vorhinein, bevor eine Anforderung implementiert wird, Test Cases für Anforderungen überlegt werden. (Craig & Jaskiel, 2002)

4.5.3 Requirements Stability Metric

Mit folgender Metrik soll die Stabilität aller Anforderungen gemessen werden. Dabei wird die Anzahl der *Change Requests* herangezogen und durch die Gesamtanzahl aller Anforderungen dividiert. Ein Change Request bezeichnet eine vom Auftraggeber geäußerte Aufforderung zur Änderung einer bestehenden, korrekt implementierten Funktionalität. Die daraus resultierende Rate kann als Qualitätskriterium genutzt werden und auch dazu verwendet werden, um eine Aussage über den Release zu treffen, sofern eine bestimmte festgelegte Grenze überschritten wurde. (Chemuturi, 2013)

5 IMPLEMENTATION PHASE

Die zweite Phase des SDLC ist die *Implementation Phase.* Diese Phase bezieht sich auf die Commit-Phase des Continuous Delivery Prozesses, welche den ersten Schritt der Deployment Pipeline, wie bereits in Kapitel 3.1.1 beschrieben wurde, bezeichnet. Diese Phase dient im Continuous Delivery Prozess hauptsächlich dazu sicherzustellen, dass kein ungeprüfter Code in das Repository eingecheckt wird. Dazu sind einige Maßnahmen nötig, welche dabei helfen den Code zu überprüfen, wie beispielsweise Code-Kompilierung oder Unit Tests. Diese Maßnahmen sind zwar alle notwendig und hilfreich, doch sie prüfen nur den funktionalen Aspekt des Quellcodes, indem er ausgeführt wird. Nicht funktionale Aspekte wie Effizienz oder Wartbarkeit benötigen andere Maßnahmen zur Überprüfung. Diese Maßnahmen werden unter dem Titel statische Code-Analyse zusammengefasst und im folgenden Unterkapitel näher beschrieben. (Humble & Farley, 2011)

5.1 Statische Code-Analyse

Die *statische Code-Analyse* ist eine Prüfmethode, welche den Quellcode analysiert, ohne dabei den Code auszuführen. Für diese Prüfmethode gibt es eine Reihe von automatisierten Werkzeugen, welche den Quellcode auf Basis einer Softwaremetrik analysieren. Die statische Code-Analyse kann jedoch auch manuell durchgeführt werden in Form eines Reviews beispielsweise. Ziel ist es frühzeitig Fehler, Probleme, oder die Nichteinhaltung von Standards und Konventionen zu erkennen, bevor diese erst später gefunden werden und deshalb größeren Aufwand für das Lösen mit sich bringen. Es wird zwar nicht möglich sein alle Fehler durch die statische Code-Analyse aufzudecken, jedoch werden dadurch einige Fehler schon frühzeitig abgefangen. (Hoffmann, 2008; Spillner & Linz, 2012)

5.2 Softwaremetriken

Softwaremetriken oder *Softwaremaße* geben eine quantitative Auskunft über den aktuellen Status des Quellcodes aus verschiedensten Sichten und können mit Hilfe von automatisierten Tools errechnet und visualisiert werden. Die quantitative Aussage der Softwaremetriken hilft in weiterer Folge dabei die zuvor beschriebenen Qualitätskriterien, wie *Functional Suitability*, *Compatibility* oder *Maintainability* zu beurteilen. (Hoffmann, 2008; Liggesmeyer, 2009)

5.2.1 Code Coverage

Eine dieser Metriken ist die *Testabdeckung* (*Code Coverage*). Sie gibt Auskunft darüber wieviel Quellcode durch Unit Tests abgedeckt wird. Für diese Metrik existieren Tools, welche in den Build Prozess eingebaut werden und folglich die Testabdeckung ermitteln können. Die Testabdeckung des Quellcodes minimiert das Risiko fehlerhaften Code auszuliefern. Dies bedeutet je höher die Code Coverage ist, desto geringer ist das Risiko für fehlerhaften Code. Diese Metrik sagt jedoch

nichts über die Qualität des Codes aus, sondern gibt nur ein gewisses Maß an Sicherheit für den Auftraggeber und das Entwicklungsteam über den Quellcode. Wie hoch die Code Coverage für den Quellcode sein soll, entscheidet entweder das Entwicklungsteam für sich selbst oder der Auftraggeber legt dies vertraglich fest. (Crispin & Gregory, 2009; Rossberg, 2014)

5.2.2 Code Reliability

Die *Code Reliability* ist eine weitere Metrik der statischen Code-Analyse. Wie bereits in Kapitel 2.1 beschrieben, handelt es sich bei der Zuverlässigkeit der Software um ein Qualitätskriterium von Software. Dieses betrachtet die Stabilität der Software im laufenden Betrieb (ISO/IEC 25010, 2011). Code Reliability betrachtet hingegen ausschließlich die Zuverlässigkeit des Quellcodes der Software bei seiner Ausführung. Diese Metrik wird durch die fehlerfreie Ausführung der Software definiert. Im Zuge dieser Metrik werden Fehler (Bugs) im Quellcode identifiziert. In diesem Kontext bezeichnen Fehler Fälle, in denen ein Quellcode-Abschnitt ein anderes Verhalten aufweist, als jenes, welches von diesem erwartet wurde (Kaur & Bahl, 2014). Für die Ermittlung dieser Metrik existieren einige Tools zur statischen Code-Analyse, welche solche Fehler automatisiert ermitteln können. (Schilling & Alam, 2006)

5.2.3 Code Maintainability

Die folgende Metrik betrachtet die Wartbarkeit und Erweiterbarkeit des Quellcodes und ist vom Qualitätskriterium *Maintainability* abgeleitet, welches in Kapitel 2.1 beschrieben wurde (ISO/IEC 25010, 2011). Die Metrik *Code Maintainability* versucht demnach Quellcode-Abschnitte zu ermitteln, welche schwer zum Warten sind. Dabei wird der Quellcode auf einige Faktoren abgeprüft, wie beispielsweise duplizierter Code, zu lange Methoden, zu große Klassen und andere ähnliche Faktoren. Quellcode-Abschnitte, welche gegen einen oder mehrere dieser Faktoren verstoßen, werden als *Code Smells* klassifiziert. *Code Smells* kann wortwörtlich als *stinkender Code* übersetzt werden. Dies soll metaphorisch darauf hindeuten, dass der schlecht strukturierte (*stinkende*) Quellcode-Abschnitt überarbeitet werden soll, um diesen *üblen Geruch* loszuwerden (Fowler, Beck, Brant, Opdyke, & Roberts, 1999). Ebenso existieren einige automatisierte Tools zur statischen Code-Analyse für die Ermittlung dieser Metrik. (Schilling & Alam, 2006)

5.2.4 Code Security

Die Gewährleistung der Sicherheit im Quellcode wird durch die Metrik *Code Security* festgestellt. Wie auch schon in Kapitel 2.1 beim übergeordneten Qualitätskriterium *Security* erwähnt, soll sichergestellt werden, dass keine sicherheitstechnischen Schwachstellen im Quellcode existieren. Falls im Quellcode eine potenzielle Schwachstelle entdeckt wird, wird der jeweilige Quellcode-Abschnitt als *Vulnerability* (Sicherheitslücke) ausgewiesen. Eine Sicherheitslücke besteht dann, wenn ein Quellcode-Abschnitt beispielsweise nicht gegen Cross Site Scripting, SQL Injection oder ähnliche Attacken geschützt ist (Fenton & Bieman, 2015). Genauso wie *Code*

Reliability und *Code Maintainability* kann auch *Code Security* durch Tools zur statischen Code-Analyse automatisiert ermittelt werden. (Schilling & Alam, 2006)

5.2.5 Cyclomatic Complexity

Eine weitere aussagekräftige Softwaremetrik ist die *zyklomatische Komplexität* (*Cyclomatic Complexity*) oder *McCabe-Metrik*. Aus dieser Metrik resultiert eine sogenannte zyklomatische Zahl, welche vereinfacht ausgedrückt darüber Auskunft gibt, wie Komplex die Struktur eines gewissen Codeabschnittes ist. Die McCabe-Metrik errechnet diese Zahl aus den Verbindungen, welche eine Komponente zu anderen Komponenten hat und kann dadurch ihre Komplexität bestimmen. Je höher die zyklomatische Zahl, desto komplexer ist der jeweilige Codeabschnitt. McCabe (1976) hält eine Obergrenze von 10 für akzeptabel, alles über 10 sollte überarbeitet werden. Aus dieser Metrik kann zusätzlich über den benötigten Testaufwand eines Codeabschnitts oder Features Rückschluss gezogen werden. Die zyklomatische Komplexität gibt schlussendlich eine Aussage über die Qualität und Beschaffenheit des Quellcodes aus und es sollte angestrebt werden die zyklomatische Zahl niedrig zu halten. (Hoffmann, 2008; Liggesmeyer, 2009)

5.2.6 Coupling

Ausschlaggebend für den Zustand des Quellcodes kann auch die Stärke der Verbindung oder Abhängigkeit zwischen Modulen oder Klassen sein. Die Metrik, welche diesen Umstand beschreibt, nennt sich *Kopplung* (*Coupling*). Die Kopplung zwischen Klassen kann entweder lose/schwach oder stark sein, wobei immer eine lose Kopplung angestrebt werden soll. Identifiziert man jedoch eine starke Kopplung zwischen Klassen ist die ein Hinweis darauf, dass diese Klassen und mit ihnen verbundenen Features einen erhöhten Testaufwand benötigen. (Myers, 1975; Gousset, Hinshelwood, Randell, Keller, & Woodward, 2014; Rossberg, 2014)

6 QUALITY ASSURANCE

Die *Quality Assurance (QA)* oder auch *Qualitätssicherung* ist die dritte Phase des SDLC und behandelt die zweite, dritte und vierte Phase der Deployment Pipeline des Continuous Delivery Prozesses. Diese wären *Automated Acceptance Testing, Automated Capacity Testing* und *Manual Testing*. Unter Quality Assurance versteht man alle Aktivitäten und Techniken, welche sowohl dabei helfen zu gewährleisten, dass alle Anforderungen des Auftraggebers korrekt umgesetzt worden sind und andererseits jene Aktivitäten und Techniken, welche dabei helfen Fehler oder Defekte in der Software festzustellen. Anders gesagt stellt die QA sicher, dass die gesamte Software oder ein Software Release möglichst fehlerfrei an den Auftraggeber ausgeliefert wird und für den produktiven Einsatz geeignet ist. Dabei ist es wichtig zu beachten, dass es so gut wie unmöglich ist eine Software komplett fehlerfrei zu halten. Jedoch hilft die QA dabei erstens diese Rate gering zu halten und zweitens die Fehler nur auf jene zu reduzieren, welche einen sehr geringen Schadensausmaß in der Software verursachen (Liggesmeyer, 2009). Wichtig ist es auch zu beachten, dass die Qualitätssicherung nicht nur in der Verantwortung einer Person oder einer bestimmten Gruppe liegt, sondern Teil des gesamten Entwicklungsteams ist. (Lewis, 2004)

Die Aktivitäten und Techniken der QA laufen darauf hinaus, die Erfüllung der Qualitätskriterien, wie beispielsweise *Functional Suitability, Performance Efficiency,* oder *Usability,* zu überprüfen und sicherzustellen. Zwei wichtige Aktivitäten der QA sind das *Defect Management* oder auch *Defektmanagement* und das *Testing*, welches alle Aktivitäten zum Testen der Software beinhaltet. (Lewis, 2004)

6.1 Defect Management

Ein *Defect* oder *Defekt* bezeichnet einen Fehlerzustand des Systems, welcher eine Fehlerwirkung verursacht, welche dann bei der Ausführung der Software sichtbar wird. Dieser Fehlerzustand ist für gewöhnlich ein Verhalten oder eine Funktionalität, welche nicht dem vom Auftraggeber spezifizierten Verhalten entspricht. Defects werden meist von den Testern des Entwicklungsteams entdeckt und festgehalten, aber auch vom Testteam des Auftraggebers. (Muller & Friedenberg, 2011; Spillner & Linz, 2012)

Beim *Defect Management* handelt es sich um jene Aktivitäten, welche sich um das Verwalten der festgehaltenen Defects kümmern. Zum Defect Management gehört zu allererst das strukturierte Festhalten oder Aufzeichnen eines Defects. Dieser muss einen aussagekräftigen Titel und eine detaillierte Beschreibung haben, mit welcher jener Defect analysiert werden kann. Des Weiteren müssen alle zu diesem Defect in Beziehung stehenden Dokumente und Spezifikationen mit diesem verlinkt werden. Diese wichtigen Punkte und gegebenenfalls weitere zusätzliche Punkte müssen für jede Person klar sein, welche einen Defect im Ticket-Verwaltungssystem anlegt. (Spillner & Linz, 2012)

Als weitere Aufgabe des Defect Managements zählt die Analyse des Defects. Ein festgehaltener Defect bedeutet nicht automatisch, dass ein Fehlerzustand existiert. Nach Analyse und Abgleich mit der Spezifikation kann sich herausstellen, dass der Ersteller des Defects falsch gelegen ist. Stellt sich jedoch heraus, dass es sich doch um einen Fehlerzustand handelt, muss zusätzlich analysiert werden, welche weiteren Auswirkungen dieser Defect mit sich bringt, als nur jene, welche im Bericht festgehalten sind. (Tian, 2005; Spillner & Linz, 2012)

Zum Defect Management gehört auch das Verfolgen eines Defects. Es muss zu jedem Zeitpunkt klar sein, wie der Status des Defects ist und welchen Verlauf dieser Defect, seitdem er festgehalten wurde, genommen hat. (Tian, 2005)

Abgesehen von den zuvor genannten Punkten Titel und Beschreibung, welche ein Defect-Ticket beinhalten sollte, gibt es zusätzlich drei wesentliche Punkte, welche dazu beitragen ein erfolgreiches Defect Management durchführen zu können. Die drei Punkte sind die *Klassifizierung*, der *Schweregrad* (*Severity*) und die *Priorität* eines Defects (Tian, 2005).

6.1.1 Klassifizierung

Unter der *Klassifizierung* eines Defects versteht man die Auswirkung eines Defects auf eine bestimmte Kategorie. Diese Kategorien können bestimmte Prozesse im System darstellen oder auch Qualitätsattribute, welche im Kapitel *Softwarequalität* beschrieben sind. Dies bedeutet konkret ausgedrückt, dass ein Defect beispielsweise eine Auswirkung auf die Kategorie *Security* haben kann. Auf Basis dieser Information können Defects beispielsweise gefiltert werden und die Einplanung von Defects wird dadurch unterstützt. (Tian, 2005; Muller & Friedenberg, 2011)

6.1.2 Schweregrad

Der nächste essenzielle Punkt ist der *Schweregrad* eines Defects. Dieser beschreibt wie *schwer* der konkrete Fehler das System belastet und in welchem Ausmaß das System nicht mehr benutzbar ist. Dadurch wird ebenfalls, wie durch die Klassifizierung, die Planung und auch die Priorisierung von Defects unterstützt. Ein einfaches Modell für den Schweregrad von Defects wäre beispielsweise: *Blocker*, *Critical, Major* & *Minor* (Lewis, 2004; Muller & Friedenberg, 2011). Diese vier Kategorien könnten folgendes bedeuten:

- **Blocker**: Blockiert essenzielle Prozesse zur Benutzung der Software
- **Critical**: Verursacht Datenverlust; verursacht Systemabsturz, etc.
- **Major**: Beeinträchtigt Funktionalitäten; Hauptfunktionalität des Systems ist jedoch benutzbar
- **Minor**: Beeinträchtigt Funktionalitäten nur zum Teil; Workaround ist möglich

6.1.3 Priorität

Als dritter Punkt verbleibt die Priorität eines Defects. Bei diesem Punkt handelt es sich nicht um die Priorisierung für die Einplanung von Defects, sondern um die Dringlichkeit der Behebung des Defects aus der Sicht des Auftraggebers. Jedoch wird die Priorität eines Defects gemeinsam mit seinem Schweregrad für die Priorisierung herangezogen und unterstützen damit das Defect Management. Gleich wie beim Schweregrad kann auch hier eine einfache Kategorisierung erfolgen mit Hilfe von *High, Medium* & *Low*. (Lewis, 2004; Muller & Friedenberg, 2011)

6.2 Testing

Die zweite Aktivität der Qualitätssicherung ist das *Testing* und ist gleichzeitig auch die berühmteste und meist ausgeübteste Aktivität der QA (Tian, 2005; Hoffmann, 2008). Vereinfacht gesagt versucht man im Testing die Software auszuführen, um folgend ihr Verhalten zu analysieren. Entweder schlägt die Ausführung fehl und die Fehlerursache muss behoben werden, oder es läuft alles wie erwartet und die erfolgreiche Ausführung stärkt das Vertrauen in die Software oder das jeweilige Software Release. Zusammengefasst ist das Ziel des Testings einerseits das Finden und Beheben von Fehlern und andererseits das Stärken des Vertrauens in die Software (Tian, 2005; IEEE Std 610.12, 1990). Schlussendlich wird mithilfe des Testings überprüft, ob die Anforderungen des Auftraggebers erfüllt worden sind, wodurch ihm demonstriert wird, dass die Software wie gewünscht funktioniert (Wagner, 2013).

Das Testing oder auch in weiterer Folge die Aktivitäten der QA werden nicht zu einem einzigen bestimmten Zeitpunkt durchgeführt, sondern sind ein Prozess, welcher sich durch den gesamten Continuous Delivery Prozess durchzieht (Craig & Jaskiel, 2002). Aus diesem Grund besitzen Tests je nach Prozessschritt verschiedene Merkmale, welche in 3 verschiedenen Klassen eingeteilt werden: *Prüfebene*, *Prüfkriterium* & *Prüfmethodik*. Diese Klassifizierung lehnt sich an das Buch *Software-Qualität* von Hoffmann Dirk (2008) an und wird in den folgenden Unterkapiteln beschrieben.

6.2.1 Prüfebene

Die Prüfebene beschreibt vier Ebenen, welche in jedem Softwarelebenszyklus durchlaufen werden. In jeder dieser Ebenen werden verschiedene Arten von Tests durchgeführt (Hoffmann, 2008). Diese Ebenen beschreiben die chronologische Abfolge der Tests und werden beim Betrachten des V-Modells in Abbildung 7 verständlicher. Das V-Modell beschreibt ein Vorgehen, worin jeder Entwicklungstätigkeit eine Validierungstätigkeit zugeordnet ist. Wie der Name schon darauf hindeutet, ist das Modell wie der Buchstabe V aufgebaut, wobei der linke Ast die Entwicklungsstufen und der rechte Ast die korrespondierenden Test- und Integrationsstufen beschreibt. Das V-Modell wird Top-Down durchlaufen, angefangen mit dem linken Ast. (Thaller, 1994; Lewis, 2004; Hoffmann, 2008; Spillner & Linz, 2012)

Die erste Entwicklungsstufe bildet die *Anforderungsanalyse*, worin alle Wünsche, Spezifikationen und Anforderungen des Auftraggebers erhoben werden. Danach folgt der *funktionale Systementwurf*, worin die Anforderungen und Spezifikationen in konkrete Abläufe und Strukturen abgebildet werden. In Folge dessen werden diese Abläufe und Strukturen in der dritten Entwicklungsstufe für die technische Realisierung in eine Systemarchitektur überführt. Diese Stufe nennt sich *technischer Systementwurf*. Als letzte Entwicklungsstufe vor der konkreten Implementierung folgt die *Komponenten-Spezifikation*. Hierbei wird jedem Teilsystem in der Systemarchitektur seine konkrete Aufgabe und Funktion zugeteilt. Wie bereits erwähnt erfolgt als abschließende Tätigkeit die konkrete *Programmierung* der Spezifikationen. In den folgenden Unterabschnitten werden die jeweiligen Teststufen der Reihe nach näher erläutert. (Thaller, 1994; Lewis, 2004; Hoffmann, 2008; Spillner & Linz, 2012)

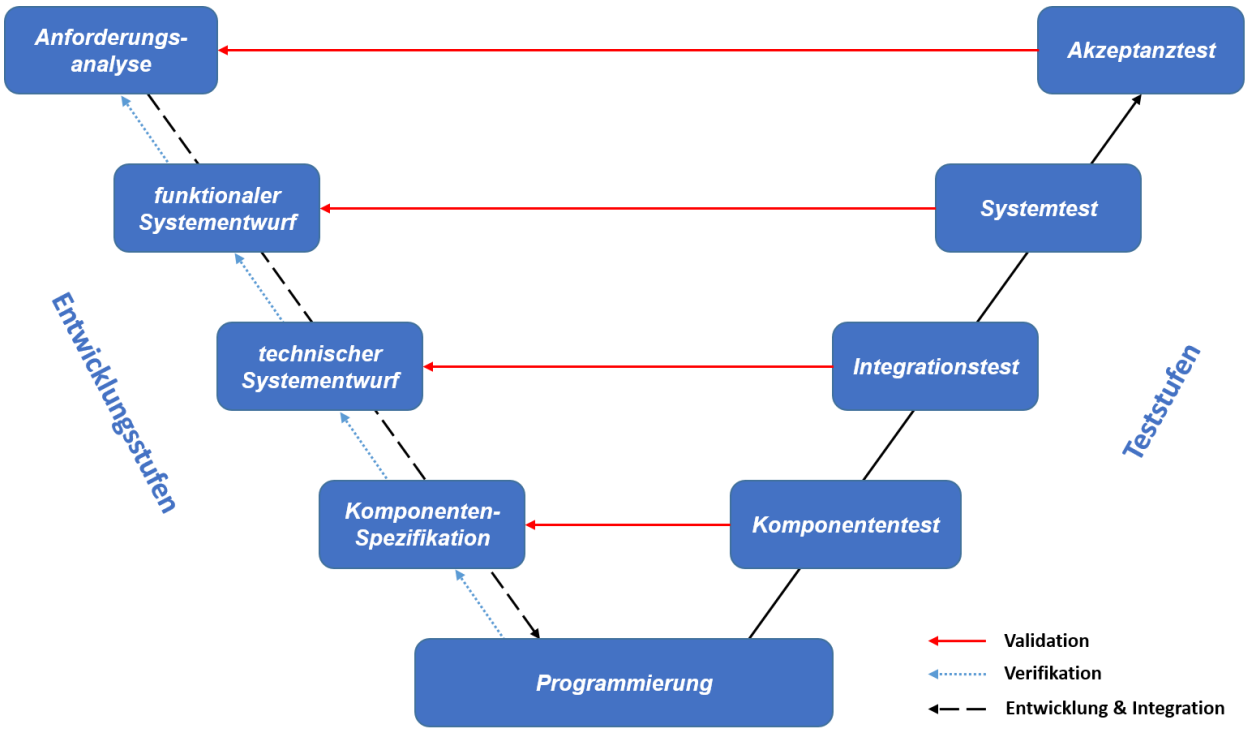


Abbildung 7 - V-Modell (Spillner & Linz, 2012)

6.2.1.1 Komponententest

Der *Komponententest*, auch *Modultest* oder *Unit Test* genannt, bildet die erste Stufe der Teststufen des V-Modells und erfolgt unmittelbar nach der Programmierung. Üblicherweise werden Komponententests aber während der Programmierung vom Programmierer oder von der Programmiererin selbst geschrieben. Es werden dabei eigenständig funktionierende Programmbausteine/Programmeinheiten oder auch Komponenten getestet. Abhängig von einer Programmiersprache können diese auch als Module, Units oder auch Klassen bezeichnet werden. Es werden alle Programmierbausteine isoliert von anderen Komponenten geprüft und jegliche benötigten Schnittstellen werden über Platzhalter (Mocks) abgebildet. Daher wird der

Komponententest der Kategorie des *White-Box-Testing* zugeordnet, da er direkt den Quellcode und seine Struktur validiert. Anders als beim *Black-Box-Testing*, welches die Software von außen betrachtet und ihre Funktion auf Basis der Anforderungen validiert (Hamill, 2004; Hoffmann, 2008; Liggesmeyer, Software-Qualität, 2009; Baumgartner, Klonk, Pichler, Seidl, & Tanczos, 2013).

Ziele des Komponententests sind einerseits den Code auf Basis der Komponentenspezifikation zu prüfen, ob die jeweiligen Programmbausteine die Ergebnisse liefern, welche gefordert waren und andererseits zu prüfen, wie robust jeder einzelne Programmbaustein ist, indem der spezifische Programmbaustein durch Grenzfälle oder negativ Beispiele getestet wird. Ein weiteres Ziel des Komponententests wäre die Effizienz des Programmbausteins, im Sinne von Speicherverbrauch oder Reaktionszeit (Spillner & Linz, 2012).

Komponententests können laut Hamill (2004) nach dem „*test twice, code once*" Schema erstellt werden, welches aus 3 Schritten besteht:

1. Schreibe einen Test für den neuen Code und sieh zu, wie er fehlschlägt
2. Schreibe auf einfachste Art und Weise einen neuen Code
3. Überarbeite (refactor) den Code nachdem der Test erfolgreich durchgelaufen ist

6.2.1.2 Integrationstest

Die nächste Teststufe, welche den technischen Systementwurf validiert, nennt sich *Integrationstest*. Hierbei werden die zuvor im Komponententest getesteten Programmbausteine herangezogen, zusammengesetzt und auf korrekte Zusammenarbeit geprüft (Spillner & Linz, 2012; Winter, Ekssir-Monfared, Sneed, Seidl, & Borner, 2012). In der Literatur findet man auch Unterteilungen für den Integrationstest, nämlich in *Komponentenintegrationstest* und *Systemintegrationstest*. Ersteres bezieht sich auf das Zusammenwirken von einzelnen Programmbausteinen, Methoden oder Klassen und die zweite Unterteilung auf die Integration von externen Systemen oder Subsystemen (Spillner & Linz, 2012; Winter, Ekssir-Monfared, Sneed, Seidl, & Borner, 2012; Baumgartner, Klonk, Pichler, Seidl, & Tanczos, 2013).

Der Integrationstest stellt also sicher, dass zwei Komponenten korrekt und erfolgreich miteinander kommunizieren können. Es wird hierbei darauf geachtet, ob die Kommunikation zwischen den zwei Schnittstellen überhaupt aufgebaut werden kann, die richtigen Daten ausgetauscht werden, das richtige Datenformat verwendet wird, oder auch ob die Daten zum richtigen Zeitpunkt versendet werden (Spillner & Linz, 2012; Winter, Ekssir-Monfared, Sneed, Seidl, & Borner, 2012).

Die Art und Weise wie Komponenten in einem System integrieret werden ist unterschiedlich. Einerseits kann die Strategie verfolgt werden alle Komponenten gleichzeitig in das System zu integrieren und erst dann den Integrationstest durchzuführen. Diese Strategie nennt sich „*Big-Bang-Integration*". Andererseits kann auch eine inkrementelle Strategie verfolgt werden, wobei diese noch in „*Bottom-Up-Integration*", „*Top-Down-Integration*", „*Outside-In-Integration*" und „*Inside-Out-Integration*" unterteilt werden kann (Spillner & Linz, 2012; Winter, Ekssir-Monfared, Sneed, Seidl, & Borner, 2012). Diese Strategien werden in der folgenden Arbeit jedoch

nicht näher behandelt, da sie nicht ausschlaggebend sind für die Untersuchung der Forschungsfrage.

6.2.1.3 Systemtest

Aufbauend auf den Komponenten- und Integrationstest folgt der *Systemtest*. Im Unterschied zu den beiden vorherigen Tests prüft der Systemtest nun das Zusammenspiel der Programmbausteine und Komponenten als Ganzes und versucht diese nicht aus programmiertechnische Sicht zu validieren, sondern aus Sicht des Auftraggebers. Anders formuliert wird das Gesamtsystem sowohl auf Basis der funktionalen Kundenanforderungen, als auch der nicht funktionalen validiert und zählt somit im Unterschied zum Komponenten- und Integrationstest zur Kategorie des *Black-Box-Testings*. Ein Systemtest für funktionale Anforderungen könnte beispielsweise das Anlegen eines neuen Benutzers in einem System sein und für nicht funktionale Anforderung beispielsweise das Messen der Reaktionszeit des Systems. Ziel ist es das Gesamtsystem so zu überprüfen, als wäre es im Echtbetrieb, indem dazu Testdaten benutzt werden, welche das Live-System simulieren. Es wird ebenfalls eine realitätsnahe Anzahl an Systemzugriffen zur Leistungsüberprüfung herangezogen, um ein tatsächliches Szenario im laufenden Betrieb simulieren zu können. (Schmitz, Bons, & Megen, 1983; Hoffmann, 2008; Liggesmeyer, 2009; Watkins, 2009; Spillner & Linz, 2012)

Ein wesentliches Hindernis beim Systemtest sind in vielen Fällen die unzureichend formulierten Kundenanforderungen. Dies erschwert das Validieren des Systems erheblich, da der testenden Person nicht ersichtlich ist, wie sich das System im konkreten Fall zu verhalten hat. Daher ist ein genaues *Requirements Engineering* unerlässlich für die Gewährleistung der Qualität des Produkts. (Hoffmann, 2008; Spillner & Linz, 2012)

Ineffiziente Testumgebungen, mangelnde Zeit, nicht ausreichend spezifizierte Kundenanforderungen, unzureichend qualifizierte Personen für Softwareautomatisierung, das Zusammenspiel verschiedener externer Systeme und andere Probleme zwingen Projekte zum manuellen Testen. Dennoch sollten automatisierte Systemtest angestrebt werden, jedoch nur in einem sinnvollen Ausmaß, der die verfügbare Zeit und das Budget nicht vollkommen ausreizt. Auch können nicht alle Testfälle aufgrund ihrer Komplexität automatisiert getestet werden. In Projekten mit agilen Vorgehensmodellen werden automatisierte Systemtests als *Regressionstest* vor jedem Release benutzt. (Hoffmann, 2008; Baumgartner, Klonk, Pichler, Seidl, & Tanczos, 2013; Linz, 2013)

6.2.1.4 Akzeptanztest

Der *Akzeptanztest*, welcher auch *Abnahmetest* oder *User Acceptance Test (UAT)* genannt wird verfolgt im Grunde genommen das gleiche Ziel wie der Systemtest. Der Akzeptanztest unterscheidet sich vom Systemtest jedoch darin, dass dieser nun vom Auftraggeber selbst und auf seiner eigenen Testumgebung durchgeführt wird und in seiner Verantwortung liegt. Mithilfe des Abnahmetestes prüft der Auftraggeber, ob das geliefert wurde, was am Anfang des Projektes spezifiziert worden ist. Neben dem vertraglichen Aspekt des Akzeptanztests auf Erfüllung der Anforderungen liegt auch beim Testen ein Fokus auf die Akzeptanz aus der Sicht des

Endbenutzers und der Endbenutzerin. (Denert, 1992; Liggesmeyer, Sneed, & Spillner, 1992; Thaller, 1994; Hoffmann, 2008; Watkins, 2009; Spillner & Linz, 2012; Winter, Ekssir-Monfared, Sneed, Seidl, & Borner, 2012)

Wird die Software von einer großen Masse benutzt, bietet sich ein sogenannter *Feldtest* an, welcher sich wiederum in *Alpha*- und *Beta*-*Test* unterteilen lässt. Hierbei schaltet man einer ausgewählten, kontrollierten Gruppe diese Software frei, um sie schon vorab überprüfen zu können. Somit kann noch vor dem Go-Live ein Feedback eingeholt und eingearbeitet werden. (Hoffmann, 2008; Crispin & Gregory, 2009; Spillner & Linz, 2012)

Als Basis für den Akzeptanztest dienen die Akzeptanzkriterien der spezifizierten Anforderungen. Akzeptanzkriterien sind der Hauptbestandteil einer User Story und beschreiben detailliert die Anforderung des Auftraggebers. Diese können funktionale aber auch nicht funktionale Anforderungen enthalten. Die Akzeptanzkriterien bilden die Basis sowohl für den Systemtest, als auch für den Akzeptanztest, welche in Kapitel 6.2.1 beschrieben wurden und werden auch zum Erstellen von automatisierten Tests herangezogen. Anhand dieser Kriterien kann der Tester oder die Testerin, als auch der Auftraggeber überprüfen, ob die User Story die gewünschte Funktionalität erfüllt oder nicht. Akzeptanzkriterien haben demnach den großen Vorteil, dass potenzielle Interpretationsspielräume oder Missverständnisse weitgehend vermieden werden, da sie die Funktionalität der User Story genauestens definiert. Aus diesem Grund werden Akzeptanzkriterien vom Auftraggeber als Grundlage für das Akzeptieren eines Release genutzt. Ebenfalls werden unter anderem Akzeptanzkriterien vom QA Manager oder QA Managerin für die Freigabe eines Release genutzt. Anders gesehen könnte man mithilfe der Überprüfung der Umsetzung von Akzeptanzkriterien gewisse Features aus dem Release entnehmen, falls die Umsetzung nicht akzeptabel ist (Dustin, Rashka, & Paul, 2001; Crispin & Gregory, 2009; Becker, 2011; Hull, Jackson, & Dick, 2011).

6.2.2 Prüfkriterium

Die zweite Klassifizierung des Software-Tests ist das *Prüfkriterium*. Dieses behandelt die Art des Tests unabhängig davon, in welcher Prüfebene dieser stattfindet und lässt sich in drei Kategorien unterteilen: *Funktional, Operational* & *Temporal*. (Hoffmann, 2008)

Beim funktionalen Testing wird der Fokus auf die Anforderungen gelegt und zielt auf das Qualitätskriterium *Functional Suitability* ab. Der wichtigste und berühmteste Test unter dieser Kategorie ist der *Funktionstests*. Dieser Test bezeichnet die klassische Art des Testens, indem die Umsetzung der Anforderungen auf ihre Korrektheit geprüft wird. Ein weiterer Test unter dieser Kategorie ist beispielsweise der *Crashtests*, welcher im Unterschied zum Funktionstest, die zu testende Funktionalität ausreizen und das System somit „zum Absturz“ bringen will. Weitere Tests dieser Kategorie sind *Kompatibilitätstests*, *Zufallstests* oder auch *Trivialtests*. (Dustin, Rashka, & Paul, 2001; Hoffmann, 2008)

Die zweite Kategorie legt ihren Fokus auf den operativen Betrieb der Software. Es wird beispielsweise mithilfe des *Installationstests* die Inbetriebnahme simuliert und überprüft. Aber

auch die Aktualisierung der Software wird überprüft. Bei diesem Test wird versucht die Umgebung des Echtbetriebs so gut wie möglich nachzubauen. Des Weiteren wird unter dieser Kategorie die Usability des Systems durch sogenannte *Ergonomietests* geprüft und zu guter Letzt wird unter dieser Kategorie mit Hilfe von *Sicherheitstests* die Sicherheit und Vertraulichkeit des Systems verifiziert (Hoffmann, 2008; Winter, Ekssir-Monfared, Sneed, Seidl, & Borner, 2012). Diese Kategorie behandelt mehrere Qualitätskriterien zugleich. Diese wären *Usability, Security* und *Portability.*

Die letzte Kategorie *temporale Tests* fokussiert sich stark auf die Verfügbarkeit und Belastbarkeit des Systems. Dies spiegelt sich in den untergeordneten *Laufzeittests*, *Lasttests* und *Stresstests* wider. Gemessen wird vor allem, ob die gewünschten Ladezeiten eingehalten werden und wieviel Benutzerlast das System standhält (Hoffmann, 2008; Winter, Ekssir-Monfared, Sneed, Seidl, & Borner, 2012). Die Qualitätskriterien, welche hier eine Rolle spielen sind vor allem *Performance Efficiency* und *Reliability*.

6.2.3 Prüfmethodik

Zuletzt lassen sich Tests über ihre Methodik klassifizieren. Hierbei handelt es sich über die Techniken, welche beim Testen zum Einsatz kommen. Grundsätzlich unterscheidet man dabei zwischen zwei Kategorien: *Black-Box-Testing* & *White-Box-Testing*. (Hoffmann, 2008)

6.2.3.1 Black-Box-Testing

Das *Black-Box-Testing*, welches auch im lEEE Std 610.12 (1990) als *Functional Testing* bezeichnet wird, berücksichtigt nur die funktionalen Anforderungen für den Test und lässt die dahinterstehende Umsetzung außen vor (Winter, Ekssir-Monfared, Sneed, Seidl, & Borner, 2012). Der Grund dafür liegt in der Eigenschaft einer *Black-Box*. Diese fokussiert sich nur auf den Input und den Output einer Funktionalität, wobei die Implementierung dieser Funktionalität außer Acht gelassen wird (IEEE Std 610.12, 1990). Beim Black-Box-Testing wird auch versucht die nicht explizit spezifizierten Anforderungen zu testen, auch wenn dies keine einfache Aufgabe darstellt. Dabei orientiert man sich an dem für gewöhnlich erwarteten Verhalten des Systems, welches sich wiederum aus dem Kontext des Geschäftsbereichs ergibt (Graham, Van Veenendaal, Evans, & Black, 2008).

Im Gegensatz zum lEEE Std 610.12 (1990) wird Black-Box-Testing im Buch *Foundations of Software Testing* von Graham, Van Veenendaal, Evans, & Black (2008) die Bezeichnung *Functional Testing* bewusst nicht als Synonym verwendet, da Black-Box-Testing auch *Non-Functional Testing* beinhaltet. Non-Functional Testing fokussiert sich auf die Stabilität des Systems und dessen Verhalten, wie beispielsweise auf die Performanz oder Usability.

6.2.3.2 White-Box-Testing

White-Box-Testing oder auch *Structural Testing* behandelt das genaue Gegenteil des Black-Box-Testing. Es behandelt jene Aspekte, welche bewusst beim Black-Box-Testing ausgelassen werden, nämlich die hinter einer Funktionalität stehende Implementierung (Lewis, 2004; Hoffmann, 2008). Dazu wird der Quellcode der Software herangezogen, welcher für manche White-Box-Testing Verfahren auch graphisch aufbereitet wird (Winter, Ekssir-Monfared, Sneed, Seidl, & Borner, 2012). Die Verfahren dieser Prüfmethodik sind meist in den beiden Prüfebenen *Komponententest* und *Integrationstest* anzufinden (Graham, Van Veenendaal, Evans, & Black, 2008).

6.2.4 Zusammenwirken der Testing-Klassifikationen

Zum Abschluss soll in einer Tabelle (Tabelle 1), angelehnt an dem Beispiel von Hoffmann (2008) in seinem Buch *Software-Qualität*, gezeigt werden wie die zuvor erklärten Klassifikationen *Prüfebene, Prüfkriterium & Prüfmethodik* in Verbindung stehen. In der folgenden Tabelle 1 besitzt jede Spalte ein Prüfkriterium und die Zeilen beschreiben jeweils die Prüfebene und Prüfmethoden. Das Kreuz beschreibt, welches Prüfkriterium in welcher Prüfebene und mit welcher Prüfmethodik zur Anwendung kommt.

	Funktionstest	***Crashtest***	***Kompatibilitätstest***	***Installationstest***	***Ergonomietest***	***Sicherheitstest***	***Laufzeittest***	***Lasttest***	***Stresstest***
Komponententest-Ebene	***x***						***x***		
Integrationstest-Ebene	***x***		***x***				***x***		
Systemtest-Ebene	***x***	***x***	***x***	***x***	***x***	***x***		***x***	***x***
Akzeptanztest-Ebene	***x***	***x***	***x***	***x***	***x***	***x***		***x***	***x***
Black-Box-Testing	***x***	***x***	***x***	***x***	***x***	***x***		***x***	***x***
White-Box-Testing	***x***						***x***		

Tabelle 1 - Klassifikationen des Testings im Zusammenspiel, angelehnt an *Hoffmann (2008)*

6.3 Metriken aus der Qualitätssicherung

Metriken aus der Qualitätssicherung oder auch *Qualitätsmetriken* (*Quality Metrics*) sind essenziell für das Verfolgen des Qualitätsstands der Software oder eines bestimmten Software Release und für das Messen des Erreichens von Qualitätszielen (Dustin, Rashka, & Paul, 2001; Crispin &

Gregory, 2009). Diese können manuell oder über Ticket-Verwaltungstools ermittelt werden. In diesem Unterkapitel sollen einige solcher Metriken vorgestellt und beschrieben werden.

6.3.1 Aktuelle Qualitätsrate

Die *aktuelle Qualitätsrate* betrachtet den Erfolgsstatus aller Testverfahren eines Release: Wie viele Tests sind erfolgreich durchgelaufen im Vergleich zur Anzahl aller durchgeführten Tests? Die logische Konsequenz daraus ist es den Release nicht freizugeben, falls diese Rate weniger als 100 % beträgt. Falls jedoch alle nicht erfolgreichen Features vor der Freigabe des Release wieder korrigiert wurden und deren Tests erfolgreich durchgelaufen sind, könnte man diesen Release freigeben. Dies betrifft nicht nur die Akzeptanztests als letzte Ebene, sondern alle Prüfebenen des Testings wie beispielsweise Unit Tests oder Integrationstests. (Dustin, Rashka, & Paul, 2001)

Man könnte jedoch im Zuge dieser Metrik zusätzlich die *Reopen-Rate* einzelner Features betrachten, welche durch fehlgeschlagene Akzeptanztests verursacht wurde. Musste jedoch ein Feature beispielsweise drei Mal wiedergeöffnet (*reopened)* werden, bis der Fehler endgültig behoben wurde, dann könnte man dieses Feature dennoch aufgrund des erhöhten Risikos aus dem Release entnehmen. (Everett & McLeod, 2007; Crispin & Gregory, 2009)

6.3.2 Fehlerdichte

Diese Metrik beobachtet alle Defects, welche zu einem bestimmten Feature gefunden wurden. Die Fehlerdichte ergibt sich aus der Gesamtanzahl der gefundenen Defects geteilt durch die Defects, welche zu einem bestimmten Feature gefunden wurden. Ist diese Rate höher als die vorgegebene erlaubte Fehlerdichte, sollte jenes Feature aus dem Release ausgenommen werden. Man könnte auch alle Defects, welche zu Features einer bestimmten Kategorie gefunden wurden ermitteln und gegebenenfalls alle Features dieser Kategorie aus dem Release entnehmen. (Dustin, Rashka, & Paul, 2001)

7 RELEASE MANAGEMENT

Das *Release Management* bezeichnet den Prozess, welcher den Release an den Auftraggeber ausliefert, in der vereinbarten Art und Weise und zum festgelegten Zeitpunkt (Crispin & Gregory, 2009; Rossberg, 2014). Die Bezeichnung *Release* oder *Software Release* bezeichnet dabei ein Softwarepaket zu einem bestimmten Entwicklungsstand, welches ausgewählte Features beinhaltet und an den Auftraggeber ausgeliefert wird (Spillner & Linz, 2012; Gousset, Hinshelwood, Randell, Keller, & Woodward, 2014). Das Release Management bildet den letzten Schritt des SDLC und der Deployment Pipeline im Continuous Delivery Prozess.

Der Prozess des Release Managements stellt sicher, dass das Software Release vollständig getestet und von allen Verantwortlichen freigegeben wurde. Ein sogenannter *Release-Plan* regelt alle Schritte und Verantwortlichkeiten, sodass dieser Prozess in einer geregelten Ordnung abläuft. Ebenso wird darauf geachtet, dass der Release mit der korrekten Version versehen wird und alle für diese Version geplanten Features in diesem Release enthalten sind. (Humble & Farley, 2011; Winter, Ekssir-Monfared, Sneed, Seidl, & Borner, 2012)

7.1 Release Notes

Im Zuge des Prozesses des Release Managements ist es notwendig, vor der Auslieferung des Software Release, die *Release Notes* zu erstellen. Bei diesen Release Notes handelt es sich um ein Dokument, welches alle im Release inkludierten Features beinhaltet. Die Form dieses Dokuments, als auch dessen Detailgrad werden mit dem Auftraggeber ausgemacht. Die Release Notes können jedoch auch spezielle Anweisungen bezüglich des Deployments des Release oder angehängte Dokumente beinhalten, die als Hilfestellung dienen. Des Weiteren können Release Notes Defects beinhalten, welche im Release behoben wurden. Aber auch Defects die möglicherweise in diesem Release aufgetreten sind, aber noch behoben werden müssten, sollten dort festgehalten werden. Zusätzlich können Release Notes als Checkliste für den Release und dessen Features dienen. (Crispin & Gregory, 2009; Winter, Ekssir-Monfared, Sneed, Seidl, & Borner, 2012)

7.2 Versionsmanagement

Das *Versionsmanagement* ist ein Teil des Release Managements und stellt sicher, dass jeder Release eine eindeutige Version hat. Dies ist wichtig, um Releases eindeutig zu Kennzeichnen und Features ordnungsgemäß zuordnen zu können. Das Versionsmanagement wird auch für Auslieferungen von Hot-Fixes (wichtige Fehlerbehebungen) relevant und stellt somit sicher, dass eine eindeutige Zuordnung zum ursprünglichen Release gewährleistet ist und kein Durcheinander entsteht. Auch für den äußersten Fall, dass ein Deployment zurück zu einer vorherigen Version nötig wird, ist das Versionsmanagement erforderlich, um eine klare Trennung zwischen den Versionen und ihren Features zu haben. (Winter, Ekssir-Monfared, Sneed, Seidl, & Borner, 2012)

7.3 Einflussfaktoren aus dem Release Management

Im folgenden Unterkapitel sollen Metriken für das Release Management vorgestellt werden. Dabei handelt es sich nicht um Metriken, welche die Qualität des Release Managements an sich bewerten, sondern primär um Metriken, mit deren Hilfe das Release Management die Qualität des auszuliefernden Software Release bewerten kann. Folgende Metriken werden sowohl manuell als auch mithilfe von Ticket-Verwaltungstools ermittelt.

7.3.1 Definition of Done (DoD)

Die *Definition of Done (DoD)*, welche aus dem agilen Vorgehensmodell SCRUM entstammt, stellt die Qualität zum Abschluss der Entwicklungstätigkeiten sicher. Die DoD kann über mehrere Ebenen festgelegt werden, sowohl auf User Story Ebene, Sprint Ebene als auch auf Release Ebene. Es handelt sich, um eine Liste bestehend aus verschiedenen Kriterien, welche erfüllt sein müssen, damit die User Story, der Sprint oder der Release tatsächlich abgeschlossen (*done*) werden kann. Die DoD wird vom gesamten Team gemeinsam erstellt und genauso ist auch jedes Teammitglied verantwortlich für deren Einhaltung. Beispiele für Kriterien in der DoD könnten lauten: „Alle Akzeptanzkriterien der User Stories müssen erfüllt sein“ oder „Alle Komponententests sind fehlerlos durchgelaufen“. Im Laufe des Projektes sollte die DoD regelmäßig überprüft und gegebenenfalls aktualisiert werden. (Schwaber & Beedle, 2002; Panchal, 2008; Baumgartner, Klonk, Pichler, Seidl, & Tanczos, 2013)

Die DoD kann somit im Release Management als Grundlage und Referenz dienen, um feststellen zu können, ob der Release abgeschlossen ist und an den Auftraggeber ausgeliefert werden kann oder nicht. In SCRUM ist die DoD ein wesentlicher Bestandteil für die Gewährleistung der Qualität des Produkts. Jedoch findet dieser Begriff mittlerweile in anderen agilen Vorgehensmodellen ebenfalls seine Anwendung. (Dustin, Rashka, & Paul, 2001)

7.3.2 Anzahl der Defects per Release

Diese Metrik gibt Auskunft über die gefundenen Defects im Zuge des Sprints oder der Sprints eines Release. Die daraus resultierende Anzahl kann eine Qualitätsaussage über den Release geben und für die Freigabe von Releases herangezogen werden. Dabei ist es wichtig im Vorhinein einen Grenzwert festzulegen. Beispielsweise kann vorgegeben werden, dass ein Release nicht freigegeben werden kann, wenn mindestens fünf Defects gefunden wurden. (Craig & Jaskiel, 2002; Rossberg, 2014)

Zusätzlich ist eine Klassifikation durch den Schweregrad oder der Priorität des Defects möglich. Wie bei der vorherigen Metrik ist es auch hier wichtig Ziele vorzugeben. Man könnte beispielsweise vorgeben, dass keine Defects mit einem Schweregrad *Critical* und nicht mehr als drei Defects mit einer Priorität *High* in einem auslieferbaren Release enthalten sein dürfen. (Lewis, 2004; Ločmelis, 2016)

8 EINFLUSSFAKTOREN & MODELL

Das folgende Kapitel behandelt die aus dem SDLC ermittelten Einflussfaktoren und Metriken, welche in den Kapiteln zuvor beschrieben wurden und zeigt konkret wie diese mit den Features eines Release verknüpft und ergänzend zu den Testergebnissen des Akzeptanztests herangezogen werden können. Für jegliche verwendeten Faktoren oder Metriken im Modell wird nachfolgend der Begriff *Einflussfaktoren* als Sammelbegriff verwendet. Bevor das entwickelte Modell präsentiert wird, wird zunächst auf die Voraussetzungen zur Anwendung des Modells eingegangen. Darauf aufbauend wird das Modell mit den Verknüpfungen der Einflussfaktoren beschrieben und im Anschluss tabellarisch aufbereitet.

Durch den gesamten Continuous Delivery Prozess hindurch wurden folgende Einflussfaktoren ermittelt:

- *Volatility of Requirements*
- *Requirements Test Case Coverage*
- *Requirements Stability Metric*
- *Code Coverage*
- *Code Reliability*
- *Code Maintainability*
- *Code Security*
- *Cyclomatic Complexity*
- *Coupling*
- *Aktuelle Qualitätsrate*
- *Fehlerdichte*
- *Definition of Done (DoD)*
- *Anzahl der Defects per Release*

Diese Einflussfaktoren wurden aus den Schritten des SDLC ermittelt mit dem Ziel die Qualität eines Software Release zu steigern. Es wurden jene Einflussfaktoren ermittelt, mit welchen eine direkte Verbindung zu Features aus dem Release hergestellt werden kann. Im Mittelpunkt steht der Akzeptanztest, welcher durch dessen Testergebnissen und den in der Arbeit ermittelten Einflussfaktoren eine genauere Beurteilung der Qualität eines Software Release zulässt.

8.1 Voraussetzungen für die Anwendung des Modells

Für die Anwendung des Modells müssen zunächst einige Voraussetzungen erfüllt sein, um die Werte der einzelnen Einflussfaktoren erheben zu können. Diese Voraussetzungen werden im folgenden Kapitel näher beschrieben und zum Schluss nochmals zusammenfassend aufgezählt.

Die Basis für das zu entwickelnde Modell bildet der *Continuous Delivery Prozess*. Dieser Prozess muss in seiner Grundform praktiziert werden, um das Modell auch anwenden zu können. Dies impliziert auch, dass für die Anwendung dieses Modells ein gewisser Prozess für die Qualitätssicherung gelebt werden muss. Dabei spielt es keine wesentliche Rolle wie die einzelnen Schritte dieser Prozesse im Detail aufgesetzt sind, denn diese variieren von Entwicklungsteam zu Entwicklungsteam und werden je nach Bedürfnissen des Projekts und des Auftraggebers ausgerichtet. Letztendlich werden die vier Phasen *Requirements Engineering, Implementation Phase, Quality Assurance* und *Release Management* benötigt, um die Einflussfaktoren des Modells verwenden zu können. Innerhalb der *Quality Assurance* Phase werden auch die vier Prüfebenen des Testings aus dem Kapitel 6.2.1 benötigt.

Für die Ermittlung der einzelnen Werte der Einflussfaktoren des Modells müssen die entsprechenden Tools dafür vorhanden sein. Die Einflussfaktoren der Gruppen des *Requirements Engineerings*, der *Quality Assurance* oder des *Release Managements* benötigen zumindest ein *Issue-Tracking* System, eine *Collaborative Software* oder ein simplifiziertes *Content-Management* System, um darin Anforderungen festzuhalten, Features und Defects abzubilden und dementsprechende Verknüpfungen zwischen diesen realisieren zu können. Die Einflussfaktoren der Gruppe der *Implementation Phase* benötigen für die Erhebung der jeweiligen Werte ein entsprechendes Tool zur statischen Code-Analyse.

Wurden die Anforderungen und Features in einem ausgewählten Tool abgebildet, müssen sie als Nächstes mithilfe einer Technik priorisiert werden. Beispielsweise kann eine der in Kapitel 4.2 präsentierten Technik benutzt werden. Die Priorisierung hat in Folge dessen einen Einfluss auf die Priorisierung der erstellten Defect-Tickets. Diese werden in Abhängigkeit der jeweils zusammenhängenden Anforderungen priorisiert. Dies ist vor allem für den Einflussfaktor *Anzahl der Defects eines Schweregrades/einer Priorität* ausschlaggebend.

Aus diesem vorherigen Punkt ergibt sich unmittelbar die nächste Voraussetzung, nämlich die Sicherstellung der Verfolgbarkeit von Anforderungen. Diese muss für die Anwendung des Modells gegeben sein, da sie für einige Einflussfaktoren eine Voraussetzung darstellt. Beispielsweise müssen für den Einflussfaktor *Requirements Test Case Coverage* alle Test Cases mit den jeweiligen Anforderungen verknüpft sein, um diesen ermitteln zu können. Essenziell wird die Traceability auch für Defect-Tickets, welche ebenfalls zu den jeweiligen Anforderungen verlinkt werden müssen. Zuletzt ist es notwendig für die Anwendung des Modells alle Anforderungen miteinander zu verknüpfen, um mögliche Einflüsse feststellen zu können.

Die *Definition of Done* (*DoD*) ist im Vergleich zu den übrigen Einflussfaktoren nicht eindeutig definiert und ist stark vom Entwicklungsteam abhängig. Wie schon eingangs im Theorieteil erläutert (Kapitel 7.3.1) gibt es keine allgemeingültige Definition für die DoD, denn sie wird vom Entwicklungsteam selbst definiert. Deshalb ist es für die Anwendung des Modells wesentlich,

dass die DoD schon im Vorhinein definiert sein muss, um dann die beinhalteten Punkte dieser prüfen zu können. Je nachdem was der Inhalt der DoD ist, kann dieser Einflussfaktor für jedes Feature entweder manuell oder mithilfe eines Tools überprüft werden.

Wie bereits bekannt ist, wird zu jedem Einflussfaktor des Modells ein bestimmter Wert ermittelt. Um diesen Wert sinnvoll interpretieren zu können, braucht jeder Einflussfaktor, je nach dessen Kontext, einen Grenzwert oder einen Zielwert. Diese Werte müssen vor der Anwendung des Modells definiert sein, damit das Modell seinen Zweck erfüllen kann. Dieser Grenzwert/Zielwert wird für den Vergleich mit dem ermittelnden Wert des Einflussfaktors herangezogen. Je nachdem, ob der ermittelte Wert des Einflussfaktors über dem Grenzwert oder unter dem Zielwert liegt, wird eine Maßnahme bezüglich des betroffenen Features getroffen.

Die in den vorigen Absätzen beschriebenen Voraussetzungen werden noch einmal zusammenfassend aufgezählt:

1. Continuous Delivery Prozess
2. Tool(s) zum Abbilden von Anforderungen, Features und Defects
3. Tool(s) zur statischen Code-Analyse
4. Priorisierte Anforderungen
5. Traceability
6. DoD
7. Grenzwerte/Zielwerte für die einzelnen Einflussfaktoren

8.2 Modell

Im folgenden Kapitel wird das entwickelte Modell beschrieben und dargestellt, welches aus den ermittelten Einflussfaktoren besteht. Es soll dabei erklärt werden wie die Einflussfaktoren zur Anwendung kommen und welche Auswirkung sie auf den Release haben. Dieses Modell beinhaltet jene Einflussfaktoren, welche im Zuge der Literaturrecherche ermittelt wurden. Jene Einflussfaktoren wurden aufgrund ihrer Bekanntheit und Verbreitung ausgewählt. Das Modell darf jedoch nicht als statisch und unveränderbar betrachtet werden, sondern als Grundbasis für ein flexibles Qualitätsmodell mit Potenzial zur Erweiterung. Je nach Situation des Kundenprojektes, der Arbeitsprozesse des Entwicklungsteams und der technischen Möglichkeiten können dem Modell weitere Einflussfaktoren hinzugefügt, aber auch entnommen werden.

Das Modell wird als Tabellenform mit 3 Spalten: *Einflussfaktoren*, *Richtlinien & Maßnahmen*, wie an Tabelle 2 zu erkennen ist, dargestellt. In der Spalte *Einflussfaktoren* sind alle im Zuge dieser Arbeit ermittelten Einflussfaktoren aufgelistet, wobei sie farblich nach den Phasen des SDLC unterteilt sind, aus dem sie entstammen. Die Erklärung der farblichen Unterteilung kann der Legende unter der Tabelle entnommen werden.

Die Spalte Richtlinien beschreibt im Detail, welcher Bereich für den jeweiligen Einflussfaktor überprüft werden soll und welche Werte im Zuge dessen erhoben werden sollen. Es wird mithilfe

der Richtlinien überprüft, ob die ermittelten Werte den im Vorhinein festgelegten Grenzwert übersteigen oder nicht. Bei einigen Richtlinien wird überprüft, ob die ermittelten Werte ein im Vorhinein festgelegten Zielwert erreichen oder nicht. Dabei kommt es auch vor, dass für einen Einflussfaktor mehrere Bereiche überprüft werden, wie beispielsweise beim Einflussfaktor: *Aktuelle Qualitätsrate*. Dort werden die Qualitätsraten aller vier Prüfebenen ermittelt.

Wie am Beispiel des Einflussfaktors *Volatility of Requirements* zu erkennen ist, kann es auch sein, dass eine Richtlinie in zwei Schritte aufgeteilt ist. Im ersten Schritt wird die primäre Richtlinie herangezogen und es wird geprüft, ob der zu ermittelnde Wert über dem festgesetzten Grenzwert liegt. Stellt sich jedoch heraus, dass der ermittelte Wert dieser Richtlinie unter der festgesetzten Grenze liegt, dann wird als nächster Schritt die weiterführende sekundäre Richtlinie, oder gegebenenfalls mehrere sekundäre Richtlinien, überprüft. In diesem Fall wird bei der ersten Richtlinie des Einflussfaktors *Volatility of Requirements* die Änderungsrate überprüft. Übersteigt diese den festgesetzten Grenzwert, darf dieses Feature und möglicherweise zusammenhängende Features nicht im Release an den Auftraggeber ausgeliefert werden. Liegt die Änderungsrate jedoch unter dem festgesetzten Grenzwert, wird überprüft, ob das mit diesem Requirement zusammenhängende Feature genügend Test Cases besitzt. Liegt diese Anzahl ebenfalls unter dem festgesetzten Grenzwert, gelten die gleichen Maßnahmen wie für die erste Richtlinie. Ansonsten darf dieses Feature im Release enthalten sein und ausgeliefert werden.

Ein besonderer Einflussfaktor aus dem Modell ist die *Definition of Done (DoD)*. Im Vergleich zu den anderen Einflussfaktoren existiert kein Grenzwert oder Zielwert, gegen welchen überprüft werden soll. Dies resultiert daraus, dass es keinen konkreten Messwert gibt, welcher ermittelt werden könnte. Des Weiteren hat die DoD keine konkrete Richtlinie, da im Allgemeinen die DoD von jedem Entwicklungsteam in einem Softwareprojekt individuell definiert wird. Dennoch soll dieser Einflussfaktor im Modell aufgenommen werden, da die DoD einen großen Einfluss auf den Release ausübt. Innerhalb der Richtlinie dieses Einflussfaktors soll lediglich überprüft werden, ob die definierten Punkte der DoD für das spezifische Feature erfüllt worden sind oder nicht. Je nachdem, ob sie erfüllt wurden oder nicht, wird das jeweilige Feature aus dem Release genommen oder darf im Release verbleiben.

Die dritte Spalte *Maßnahmen* gibt an, welche Maßnahmen bei Nichterfüllung der Richtlinien getroffen werden soll. Wird eine Richtlinie nicht erfüllt, soll als erste Maßnahme das jeweilige Feature, auf welches dies zutrifft, aus dem Release entnommen werden. In weiterer Folge soll überprüft werden, ob das betroffene Feature eventuell negative Auswirkungen auf andere Features hat, welche mit diesem in Zusammenhang stehen. Ist dies der Fall, sollten die Features, auf denen dies zutrifft, ebenfalls aus dem Release entnommen werden. Durch diese Maßnahmen wird sichergestellt, dass der Auftraggeber ein Release mit zuverlässigen und weniger fehleranfälligen Features erhält.

Einflussfaktoren	Richtlinien	Maßnahmen
Volatility of Requirements	1) Änderungsrate eines Requirements übersteigt ein festgelegtes Limit	1) Feature aus dem Release entnehmen
	1.a) Das mit diesem Requirement zusammenhängende Feature hat ungenügende Test Cases	
Requirements Test Case Coverage	Das mit diesem Requirement zusammenhängende Feature hat keine Test Cases	
Requirements Stability Metric	Change Request Rate übersteigt ein festgelegtes Limit	
Code Coverage	Der mit einem Feature zusammenhängende Codeabschnitt hat keine oder ungenügende Unit Tests	
Code Reliability	Zu dem, mit einem Feature zusammenhängenden, Codeabschnitt wurde min. ein Bug identifiziert	
Code Maintainability	Zu dem, mit einem Feature zusammenhängenden, Codeabschnitt wurde min. ein Code Smell identifiziert	
Code Security	Zu dem, mit einem Feature zusammenhängenden Codeabschnitt, wurde min. eine Vulnerability identifiziert	
Cyclomatic Complexity	Das mit diesem Codeabschnitt zusammenhängende Feature hat keine oder ungenügende Test Cases	
Coupling	Das mit diesem Codeabschnitt zusammenhängende Feature hat keine oder ungenügende Test Cases	
Aktuelle Qualitätsrate	1) Erfolgsrate der Unit Tests des aktuellen Release ist unter einem festgelegten Zielwert	2) Das durch das entnommene Feature negativ beeinflusste Feature aus dem Release entnehmen
	2) Erfolgsrate der Integrationstests des aktuellen Release ist unter einem festgelegten Zielwert	

	3) Erfolgsrate des Systemtests des aktuellen Release ist unter einem festgelegten Zielwert	
	4) Erfolgsrate der erfüllten Akzeptanzkriterien der Akzeptanztests des aktuellen Release ist unter einem festgelegten Zielwert	
	4.a) Reopen-Rate eines Akzeptanztests eines Features aus dem Release übersteigt ein festgelegtes Limit	
Fehlerdichte	1) Anzahl der Defects zu einem bestimmten Feature übersteigen ein festgelegtes Limit	
	2) Anzahl der Defects zu einem bestimmten Topic/Requirement mehrerer Features übersteigen ein festgelegtes Limit	
Definition of Done (DoD)	Feature aus dem Release erfüllt DoD nicht (Build is stable, Test Documentations created, Code reviewed & passed, …)	
Anzahl der Defects per Release	1) Defects im aktuellen Release übersteigen ein festgelegtes Limit	
	2) Defects im aktuellen Release mit einem bestimmten Schweregrad übersteigen ein festgelegtes Limit	
	3) Defects im aktuellen Release mit einer bestimmten Priorität übersteigen ein festgelegtes Limit	

Requirements Engineering Implementation Phase Quality Assurance Release Management

Tabelle 2 - Modell

8.3 Anwendung des Modells

Das Kapitel zuvor beschrieb das Modell und dessen Aufbau und Funktionsweise, jedoch ohne dabei auf die konkrete Anwendung einzugehen. Die Tabelle 2 diente der Erklärung des Modells und dessen Richtlinien, um einen Überblick über die zu erhebenden Einflussfaktoren zu erhalten. Für die konkrete Anwendung des Modells wurde eine Bewertungsmatrix in Form einer Tabelle (Tabelle 3) entwickelt, mit dessen Hilfe konkrete Aussagen über einzelne Features getroffen werden können. Genauer gesagt wurde diese Bewertungsmatrix im Tabellenkalkulation-Programm *Excel* realisiert und diese Excel-Datei soll auch dementsprechend bei der Anwendung des Modells herangezogen werden.

Die Bewertungsmatrix besteht grundsätzlich aus den Einflussfaktoren und den Features, wobei die Einflussfaktoren die Zeilen und die Features die Spalten einnehmen. Jedes Feature durchläuft jede einzelne Richtlinie der Einflussfaktoren und wird dementsprechend bewertet. Zum Schluss werden die einzelnen Bewertungen summiert und auf Basis dieser Summe werden Maßnahmen bezüglich der jeweiligen Features getroffen.

Für die Bewertung der einzelnen Features wurde ein simples Bewertungsschema eingeführt. Die Anzahl der Gruppen der Einflussfaktoren diente dabei als Basis für das Punktesystem. Ein Feature kann dadurch 0 bis 4 Punkte erreichen. Erfüllt ein Feature die Richtlinie eines Einflussfaktors, werden dem Feature für jenen Einflussfaktor 0 Punkte vergeben. Wird die Richtlinie für das jeweilige Feature nicht erfüllt, können 1 Punkt bis 4 Punkte vergeben werden. Diese Punkte werden pro Richtlinie vergeben und zum Schluss für jedes Feature aufsummiert. Jene Features, welche in Summe 4 Punkte oder höher erzielen, werden aus dem Release entnommen.

Welche Punkteanzahl bei Nicht-Erfüllung einer Richtlinie vergeben wird, wird über eine Gewichtung festgelegt. Je nach Kritikalität ist die Gewichtung der Richtlinie höher oder niedriger. Beispielsweise erhalten jene Richtlinien, welche zu 100 % erfüllt sein müssen, eine Gewichtung von 4 Punkten. Dies soll sicherstellen, dass keine Features in den Release kommen, ohne kritische Richtlinien davor erfüllt zu haben. In der folgenden Bewertungsmatrix aus Tabelle 3 wurden Gewichtungen für jede Richtlinie auf Basis von Erfahrungswerten definiert, jedoch bleibt dies jedem Entwicklungsteam selbst überlassen, welche Gewichtungen pro Richtlinie definiert werden sollen.

Die Spalte der Einflussfaktoren wird nochmals in vier weitere Spalten unterteilt. Die erste Spalte auf der linken Seite bildet die Spalte für die Nummerierung, welche ähnlich aufgebaut ist wie in Tabelle 2. Darauf folgt eine Spalte für die Richtlinien, welche im Vergleich zu Tabelle 2 nochmals detaillierter formuliert sind. Zum Schluss beinhaltet die Bewertungsmatrix Spalten für den Grenzwert/Zielwert und für die Gewichtung pro Richtlinie. Die in der ersten Zeile angesiedelten Features bilden alle Features aus dem jeweiligen aktuellen Release. In Tabelle 3 wurden beispielhaft nur drei Spalten dargestellt, jedoch wird die Bewertungsmatrix bei jeder Anwendung mit allen Features des aktuellen Release befüllt. Am besten einigt sich die eindeutige Identifikationsnummer des jeweiligen Features dort einzutragen.

Einflussfaktoren / ***Features***				Feature 1	Feature 2	Feature 3
#	**Richtlinien**	**Grenzwert/ Zielwert**	**Gewichtung**			
Volatility of Requirements						
1)	Änderungsrate des ursprünglichen Requirements	**> …**	**1**			
1.a)	Anzahl der Test Cases des mit diesem Requirement zusammenhängenden Features	**< …**	**1**			
Requirements Test Case Coverage						
2)	Anzahl der Test Cases des mit diesem Requirement zusammenhängenden Feature	**< 1**	**4**			
Requirements Stability Metric						
3)	Change Request Rate	**> …**	**1**			
Code Coverage						
4)	Anzahl der Unit Tests des zusammenhängenden Codeabschnittes	**< …**	**1**			
Code Reliability						
5)	Zu dem zusammenhängenden Codeabschnitt identifizierte *Blocking* Bugs	**> 1**	**4**			
6)	Zu dem zusammenhängenden Codeabschnitt identifizierte *Critical* Bugs	**> …**	**2**			
7)	Zu dem zusammenhängenden Codeabschnitt identifizierte *Major* Bugs	**> …**	**1**			
8)	Zu dem zusammenhängenden Codeabschnitt identifizierte *Minor* Bugs	**> …**	**1**			
Code Maintainability						
9)	Zu dem zusammenhängenden Codeabschnitt identifizierte Code Smells mit einem *Technical Debt: 0.51 – 1*	**> 1**	**4**			
10)	Zu dem zusammenhängenden Codeabschnitt identifizierte Code Smells mit einem *Technical Debt: 0.21 – 0.50*	**> …**	**2**			

11)	Zu dem zusammenhängenden Codeabschnitt identifizierte Code Smells mit einem *Technical Debt: 0.11 – 0.20*	**> …**	**1**			
12)	Zu dem zusammenhängenden Codeabschnitt identifizierte Code Smells mit einem *Technical Debt: 0 – 0.10*	**> …**	**1**			
	Code Security					
13)	Zu dem zusammenhängenden Codeabschnitt identifizierte *Blocking* Vulnerabilities	**> 1**	**4**			
14)	Zu dem zusammenhängenden Codeabschnitt identifizierte *Critical* Vulnerabilities	**> …**	**2**			
15)	Zu dem zusammenhängenden Codeabschnitt identifizierte *Major* Vulnerabilities	**> …**	**1**			
16)	Zu dem zusammenhängenden Codeabschnitt identifizierte *Minor* Vulnerabilities	**> …**	**1**			
	Cyclomatic Complexity					
17)	Test Cases des mit diesem Requirement zusammenhängenden Features	**< …**	**2**			
	Coupling					
18)	Test Cases des mit diesem Requirement zusammenhängenden Features	**< …**	**2**			
	Aktuelle Qualitätsrate					
19)	Erfolgsrate der Unit Tests des aktuellen Release	**< 100%**	**4**			
20)	Erfolgsrate der Integrationstests des aktuellen Release	**< 100%**	**4**			
21)	Erfolgsrate des Systemtests des aktuellen Release	**< 100%**	**4**			
22)	Erfolgsrate der erfüllten Akzeptanzkriterien der Akzeptanztests des aktuellen Release	**< 100%**	**4**			
22.a)	Reopen-Rate eines Akzeptanztests eines Features aus dem Release	**> …**	**1**			
	Fehlerdichte					
	Fehlerdichte per Feature					
23.a)	Anzahl der offenen *Blocking* Defects zu einem bestimmten Feature	**> 1**	**4**			
23.b)	Anzahl der offenen *Critical* Defects zu einem bestimmten Feature	**> …**	**4**			
23.c)	Anzahl der offenen *Major* Defects zu einem bestimmten Feature	**> …**	**4**			
23.d)	Anzahl der offenen *Minor* Defects zu einem bestimmten Feature	**> …**	**4**			

Fehlerdichte per Topic/Requirement						
24.a)	Anzahl der offenen *Blocking* Defects zu einem bestimmten Topic/Requirement mehrerer Features	**> 1**	**4**			
24.b)	Anzahl der offenen *Critical* Defects zu einem bestimmten Topic/Requirement mehrerer Features	**> …**	**4**			
24.c)	Anzahl der offenen *Major* Defects zu einem bestimmten Topic/Requirement mehrerer Features	**> …**	**4**			
24.d)	Anzahl der offenen *Minor* Defects zu einem bestimmten Topic/Requirement mehrerer Features	**> …**	**4**			
Definition of Done (DoD)						
25)	Erfüllungsgrad der DoD des jeweiligen Features	**< 100%**	**4**			
Anzahl der Defects per Release						
26)	Anzahl aller offenen Defects im aktuellen Release	**> …**	**3**			
Anzahl der Defects per Release - Severity						
27)	Anzahl der offenen Defects im aktuellen Release mit dem Schweregrad *Blocker*	**> 1**	**4**			
28)	Anzahl der offenen Defects im aktuellen Release mit dem Schweregrad *Critical*	**> …**	**4**			
29)	Anzahl der offenen Defects im aktuellen Release mit dem Schweregrad *Major*	**> …**	**4**			
30)	Anzahl der offenen Defects im aktuellen Release mit dem Schweregrad *Minor*	**> …**	**4**			
Anzahl der Defects per Release - Priority						
31)	Anzahl der offenen Defects im aktuellen Release mit der Priorität *High*	**> 1**	**4**			
32)	Anzahl der offenen Defects im aktuellen Release mit der Priorität *Medium*	**> …**	**3**			
33)	Anzahl der offenen Defects im aktuellen Release mit der Priorität *Low*	**> …**	**3**			
			Summe ohne Verknüpfungen	0	0	0
			Summe mit Verknüpfungen	0	0	0
			Passed/Failed	Passed	Passed	Passed

0 = erfüllt	***1 – 4 = nicht erfüllt***
< 4 = Passed	***> 4 = Failed***

Tabelle 3 - Bewertungsmatrix

Die Richtlinien in der Bewertungsmatrix wurden für eine bessere Übersicht nach Einflussfaktoren gruppiert, wobei die Einflussfaktoren *Anzahl der Defects per Release* und *Fehlerdichte* nochmals in zwei Untergruppen aufgeteilt wurden, nämlich in den Gruppen: *Severity & Priority* und *Feature & Topic.* Im Vergleich zur Tabelle 2 des Modells, wurde eine Klassifizierung innerhalb einzelner Richtlinien erstellt, wie beispielsweise für die Richtlinie des Einflussfaktors *Code Reliability*, um eine detailliertere Überprüfung des jeweiligen Einflussfaktors zu erreichen.

Die Spalte der Grenzwerte/Zielwerte stellt einen Wert zur Verfügung, mit dessen Hilfe gesagt werden kann, ab wann eine Richtlinie erfüllt ist oder nicht. Diese Werte werden vom jeweiligen Entwicklungsteam individuell festgelegt und werden je nach Projektsituation und Anforderungen des Auftraggebers bestimmt. Aus diesem Grund sind auch in den meisten Feldern dieser Spalte der Tabelle 3 keine Werte eingetragen. Einzig bei jenen Richtlinien, welche, basierend auf den Empfehlungen der Literatur (Lewis, 2004; Muller & Friedenberg, 2011), zu 100 % erfüllt sein müssen, wurde ein Wert eingetragen.

Mit den Grenzwerten/Zielwerten soll die Überprüfung des Erfüllungsgrades der Richtlinien ermöglicht werden, jedoch wirkt sich nicht jede Richtlinie mit derselben Stärke auf den Release aus. Um dieses Problem zu lösen, wurde eine eigene Spalte für die Gewichtung eingeführt. Ähnlich der Grenzwerte/Zielwerte wurden hier Werte, basierend auf einer Einschätzung der Wichtigkeit der einzelnen Einflussfaktoren in der Literatur, eingeführt. Jedoch können diese ebenfalls wie die Grenzwerte/Zielwerte beliebig definiert werden.

Am Ende der Bewertungsmatrix befinden sich zwei Zeilen für die aufsummierten Punkte und darauffolgend eine für die darauf basierende Maßnahme pro Feature. Wie bereits anfangs erklärt wird bei einer Summe höher als 4 das Feature aus dem Release entnommen. Dies wird mit *Failed* gekennzeichnet, wie auch in der Legende am Ende der Tabelle 3 zu sehen ist. Andernfalls bekommt das Feature den Status *Passed* und darf im Release verbleiben. Das Vergeben der Status in der Bewertungsmatrix funktioniert automatisch mithilfe der *WENN*-Funktion von Excel. Wie an diesem Beispiel der Abbildung 8 zu sehen ist, ergibt die Summe für das erste Feature in der ersten Summenzeile 5, was zur Folge hat, dass das Feature den Status *Failed* erhält. Dieser Status wurde automatisch durch die *WENN*-Funktion vergeben, welche ganz oben in Abbildung 8 zu sehen ist, da die Summe dieses Features 4 überschritten hat.

E64 =WENN(E63<4; "Passed"; "Failed")

	A	B	C	D	E
1					***Features***
2		***Einflussfaktoren***			**Feature 1**
3	**#**	**Richtlinien**	**Grenzwert/Zielwert**	**Gewichtung**	
60	**32)**	Anzahl der offenen Defects im aktuellen Release mit der Priorität *Medium*	**> ...**	**3**	2
61	**33)**	Anzahl der offenen Defects im aktuellen Release mit der Priorität *Low*	**> ...**	**3**	3
62				**Summe ohne Verknüpfungen**	5
63				**Summe mit Verknüpfungen**	5
64				**Passed/Failed**	Failed

Abbildung 8 - Beispiel für die WENN-Funktion der Bewertungsmatrix

Um die zweite Maßnahme des Modells abbilden zu können, muss für die Bewertungsmatrix bereits im Vorhinein bekannt sein, welche Features voneinander abhängig sind. Somit kann jedes

Mal vor der Anwendung des Modells händisch im Excel die Verknüpfung in der zweiten Summenzeile „*Summe mit Verknüpfungen*“ realisiert werden. Dies kann über eine simple Addition der voneinander abhängigen Felder in der Excel-Datei erfolgen. Wie man an der Funktion in der Abbildung 9 erkennen kann, wird in der Funktion die Summe des ersten Features (das Feld *E62*) mit der Summe des zweiten Features addiert (das Feld *F62*). Daraus resultiert schließlich im Feld *F63* eine Summe von 5, obwohl die Werte der Spalte F eine Summe von 0 ergeben würden. Dies hat zur Folge, dass auch das zweite Feature den Status *Failed* erhält. Auf diese Art und Weise erfolgt die Verknüpfung von Features in der Bewertungsmatrix.

	A	B	C	D	E	F
1				**Features**	Feature 1	Feature 2
2		**Einflussfaktoren**				
3	#	Richtlinien	Grenzwert/Zielwert	Gewichtung		
62				Summe ohne Verknüpfungen	5	0
63				Summe mit Verknüpfungen	5	5
64				Passed/Failed	Failed	Failed

Abbildung 9 - Beispiel für die Verknüpfung von Features in der Bewertungsmatrix

Als letzten Schritt im Zuge der Anwendung der Bewertungsmatrix verbleibt die Interpretation der Ergebnisse. Grundsätzlich sieht das Modell vor, dass Features mit dem Status *Failed* aus dem Release entnommen werden und nur alle Features mit dem Status *Passed* verbleiben und ausgeliefert werden können. Bevor diese Maßnahmen jedoch sofort umgesetzt werden, sollte nochmals ein Blick auf die einzelnen Features geworfen werden. Beispielsweise sollte bei den Richtlinien des Einflussfaktors *Aktuelle Qualitätsrate* nochmals überprüft werden, aus welchem Grund die Tests der jeweiligen Prüfebene fehlgeschlagen sind. Eventuell schlagen die Tests wegen Problemen mit der Infrastruktur oder wegen Timeouts fehl. Unter diesen Umständen wäre es eventuell möglich zu entscheiden, dass das Feature dennoch ausgeliefert werden soll, obwohl dessen Staus auf *Failed* ist.

Selbiges gilt auch für Richtlinien, welche Klassifizierungen beinhalten. Ein Feature, zu welchem mehr als zehn Defects mit dem Schweregrad *Minor* verknüpft sind, bekommt laut der Bewertungsmatrix den Status *Failed*. Prüft man jedoch diese Defects nach, könnte man zu dem Schluss kommen, dass diese Defects so trivial sind, sodass dieses Feature dennoch ausgeliefert werden könnte.

Somit wird durch diese Beispiele ersichtlich, dass eine erneute Kontrolle der Ergebnisse der Bewertungsmatrix nötig ist, bevor Entscheidungen auf Basis der Status der einzelnen Features getroffen werden.

8.4 Praktisches Beispiel für die Anwendung des Modells

Im Anschluss an die Erläuterung der Verwendung des Modells, soll mit ausgewählten Tools ein konkretes praktisches Beispiel vorgeführt werden, in welchem gezeigt wird, wie jeder einzelne Einflussfaktor erhoben werden kann. Dadurch soll die praktische und technische Anwendbarkeit

des entwickelten Modells und dessen Bewertungsmatrix nachgewiesen werden. Zwei der Einflussfaktoren des Modells können jedoch aufgrund der fehlenden technischen Möglichkeiten der verfügbaren Tools nicht ermittelt werden: *Coupling* und *Anzahl der Defects per Release per Priority*.

Für das Aufzeigen der praktischen Anwendung wurden konkrete Tools, welche in einem tatsächlichen Projekt in Verwendung sind, ausgewählt. Requirements und Defect-Tickets wurden über das Verwaltungstool *Jira* abgebildet. Für Definitionen und andere Formulierungen wurde die Wiki-Software *Confluence* herangezogen. Der Quellcode wird über den Filehosting-Dienst *Bitbucket* abgebildet, welcher das eigentliche Versionsverwaltungssystem unterstützt. Automatisierte Integrations- und Akzeptanztests werden über das Tool *Jenkins* verwaltet, welches ein webbasiertes Tool zur Unterstützung von Continuous Integration darstellt. Zuletzt wird das Tool *SonarQube* zur statischen Code-Analyse verwendet.

8.4.1 Vorbereitung

Der erste notwendige Schritt, um alle Richtlinien aller Features eines bestimmten Release überprüfen zu können, ist nach den Features des Release zu filtern, um mit diesen weiterarbeiten zu können. Dies kann im Tool *Jira* über die simple Suche erreicht werden, wie in Abbildung 10 zu sehen ist. Dort wurden alle Tickets vom Typ Story und zu einem bestimmten Sprint gefiltert.

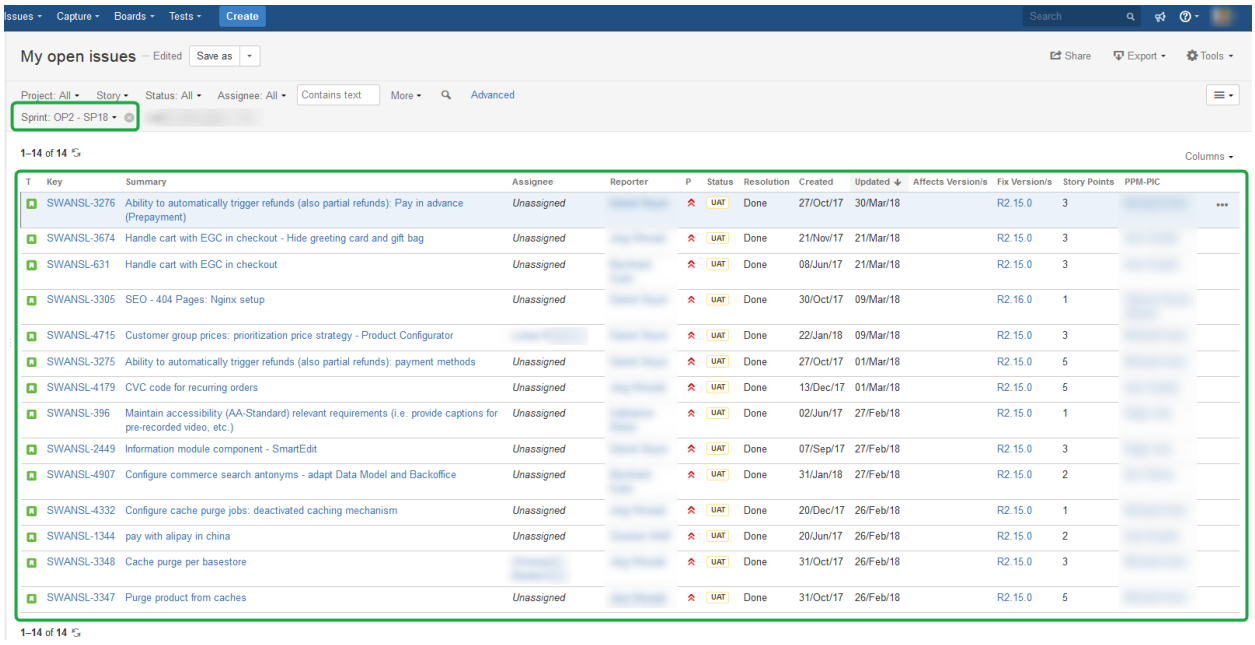


Abbildung 10 - Liste der Features eines bestimmten Sprints

Für jedes dieser Features wird eine Identifikationsnummer vom Tool vergeben. Diese Identifikationsnummern sollen anschließend in die Bewertungsmatrix in der Zeile der Features eingetragen werden. Im besten Fall könnte man diese mit einer Verlinkung auf das Feature im *Jira* Tool versehen, wie in Abbildung 11 zu sehen ist.

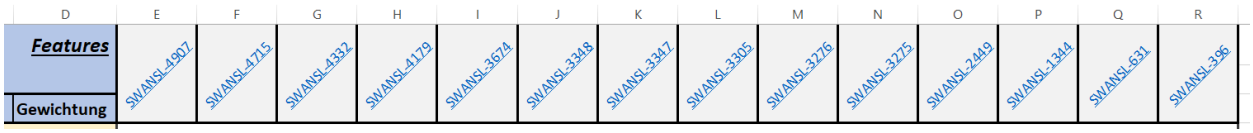


Abbildung 11 - Liste der eingetragenen und verlinkten Features des Release in der Bewertungsmatrix

Als letzten Schritt sollen alle zusammenhängenden Features in der Zeile „*Summe mit Verknüpfungen*“ verknüpft werden. Dies erfolgt nach demselben Prinzip, wie es bereits in Abbildung 9 illustriert wurde.

8.4.2 Ermittlung der Einflussfaktoren

Nachdem alle nötigen Vorbereitungen aus dem vorherigen Kapitel getroffen wurden, kann nun mit der Ermittlung der Einflussfaktoren begonnen werden. Es wird in diesem Kapitel jeder Einflussfaktor eigenständig behandelt und seine Ermittlung mit den anfangs erwähnten Tools demonstriert.

8.4.2.1 Volatility of Requirements

Wie bereits eingangs erwähnt werden Requirements über das Verwaltungstool *Jira* abgebildet. Dieses Tool unterstützt keine konkrete Funktion oder keinen konkreten Mechanismus zum Ermitteln der Änderungsrate, jedoch hält es jede Änderung zu jedem Ticket zur Nachvollziehbarkeit fest und fasst es in einer Historie zusammen. Dies erschwert zwar die Ermittlung der Änderungsrate eines Requirements, aber macht es dennoch möglich. Anhand der grün umrahmten Rechtecke in der Abbildung 12 sieht man wie diese Änderungen in *Jira* festgehalten werden.

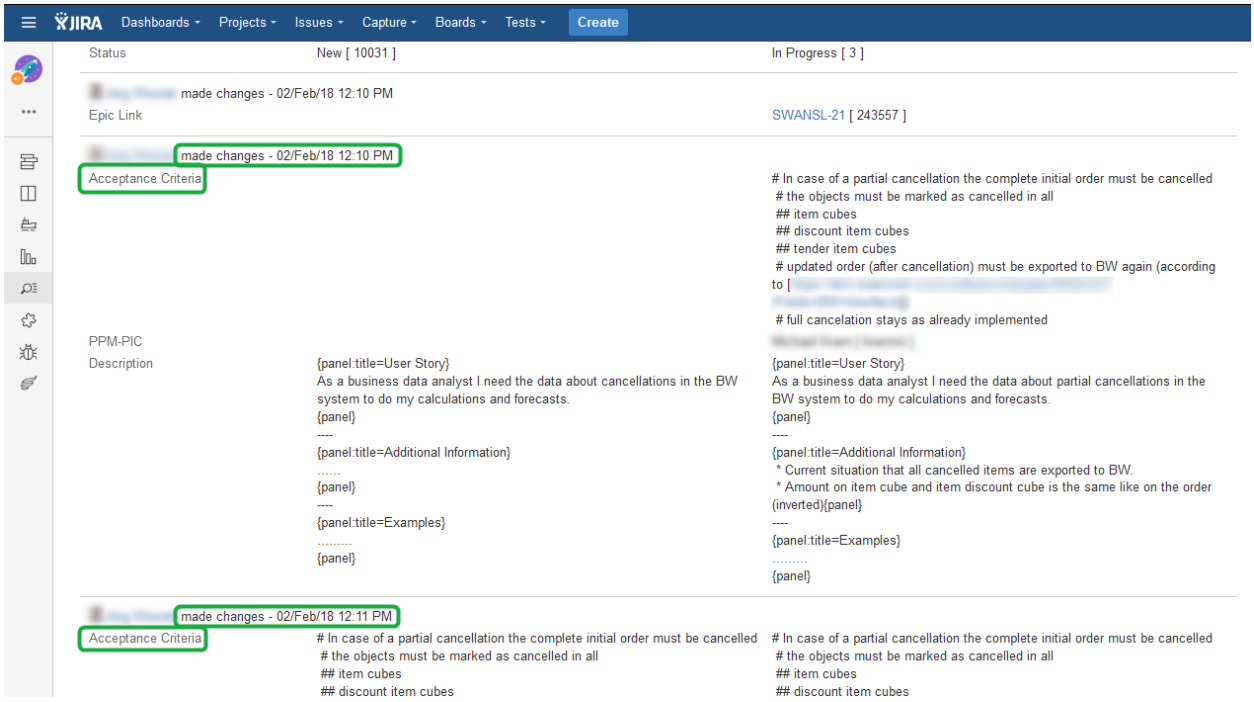


Abbildung 12 - Nachweis der Änderungen eines Requirements in Jira

In diesen beiden Rechtecken ist einerseits das Änderungsdatum abgebildet und andererseits sieht man, dass die Änderung die Akzeptanzkriterien betreffen. Anhand des Datums des Beginns der Entwicklung, welches ebenfalls am Ticket festgehalten wird, kann unterschieden werden, ob eine Änderung vor Beginn der Entwicklung geschehen ist oder erst danach. Im Fall, welcher in Abbildung 12 zu sehen ist, wurden die Änderungen vor Beginn der Entwicklung gemacht und somit wird die Richtlinie des Einflussfaktors erfüllt.

Um die zweite Richtlinie des Einflussfaktors *Volatility of Requirements* zu überprüfen, müssen die mit dem jeweiligen Requirement verknüpften Testfälle ermittelt werden. Dazu wird im Tool *Jira* das Plug-In *Zephyr* benutzt. Bei diesem Plug-In handelt es sich um ein Testmanagement Plug-In, mit welchem Testfälle erstellt und Testdurchläufe abgebildet werden können. Konkret werden diese Testfälle mit dem Requirement-Ticket verlinkt wie in Abbildung 13 zu sehen ist. In Abbildung 14 sieht man den Testfall und seine Testdurchläufe und kann somit feststellen, ob dieser Testfall erfolgreich durchgelaufen ist oder nicht. Wie in der zweiten grünen Markierung zu sehen ist, besitzt der Testfall gleichfalls eine Verlinkung zurück auf das Requirement. Für die zweite Richtlinie des Einflussfaktors bedeutet das, dass dieses Requirement genau einen Testfall besitzt.

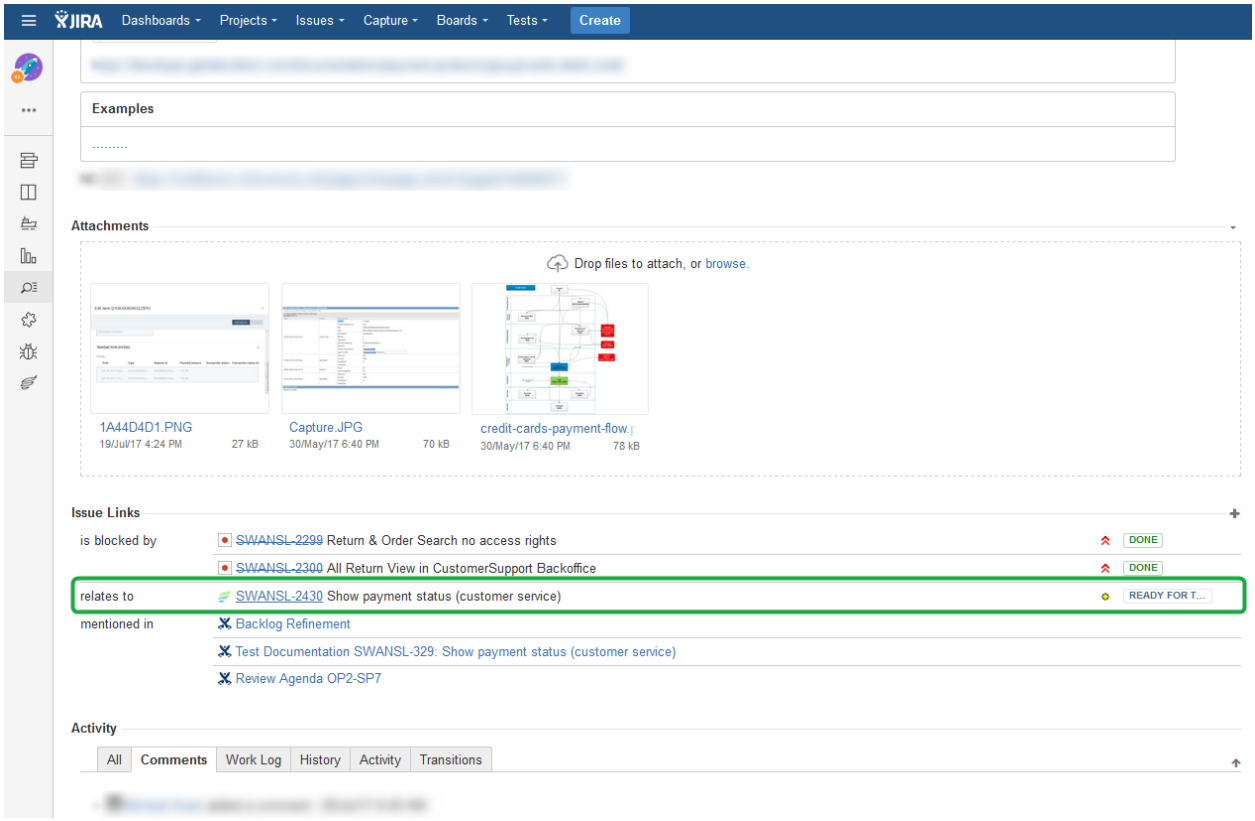


Abbildung 13 - Verlinkung von einem Requirement auf den dazugehörigen Test Case

8.4.2.2 Requirements Test Case Coverage

Bei diesem Einflussfaktor und seiner Richtlinie liegt der Fokus auf jenen Aspekt, welchen auch die zweite Richtlinie des vorherigen Einflussfaktors behandelt. In gleicher Weise wird hier die Anzahl der verknüpften Testfälle gesucht. Das Vorgehen dazu gleicht dem des vorherigen Einflussfaktors und kann ebenfalls der Abbildungen 13 & 14 entnommen werden.

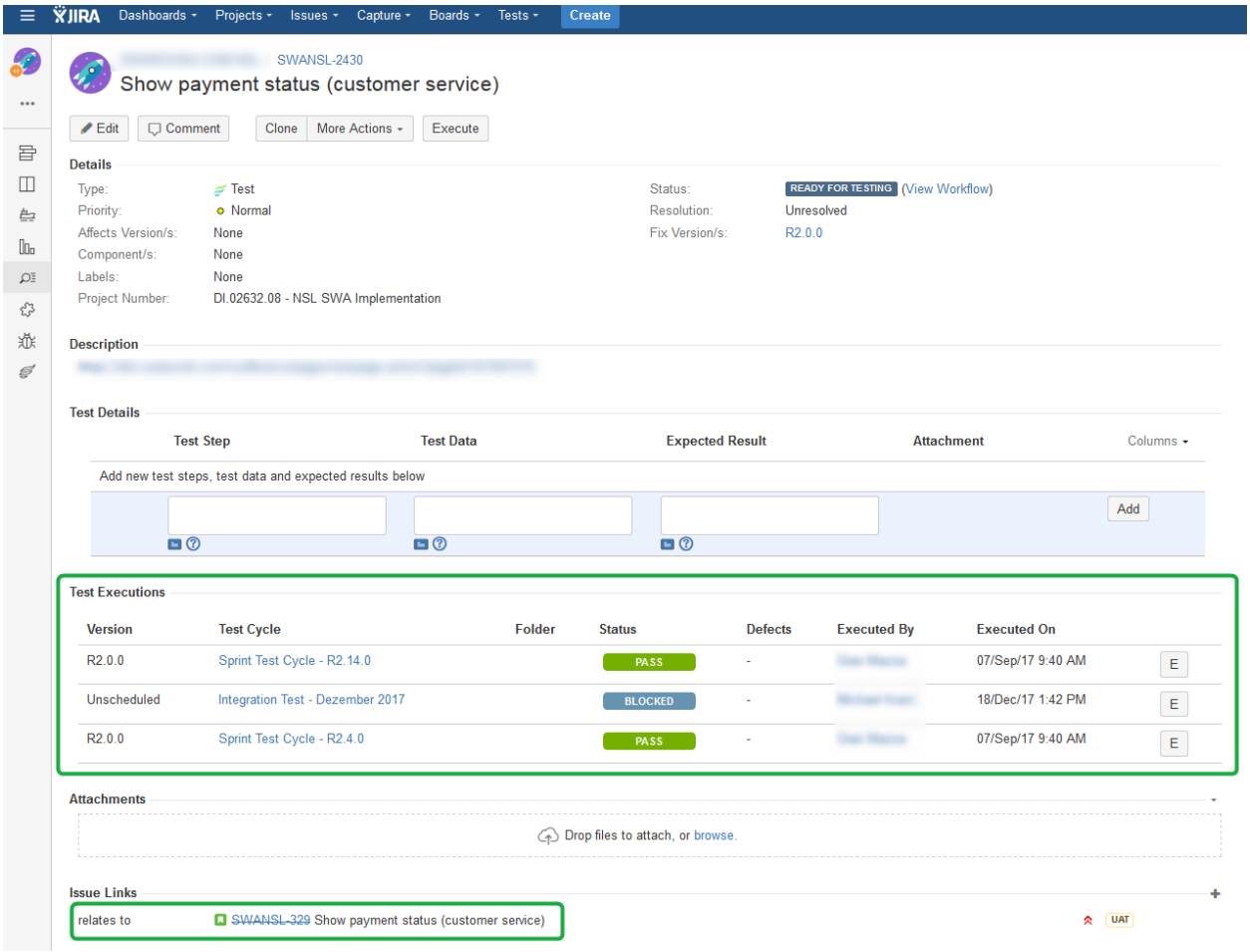


Abbildung 14 - Beispiel eines Test Case

8.4.2.3 Requirements Stability Metric

Für diesen Einflussfaktor wird in der dazugehörigen Richtlinie die *Change Request Rate* ermittelt. Dazu können im Tool *Jira*, wie in diesem Projekt konfiguriert wurde, Tickets vom Typ *Change Request* eingeführt werden. Diese können wie alle anderen Tickets im Jira mit dem Requirement verknüpft werden, wie auch in Abbildung 15 zu sehen ist. Beim Überprüfen der Features des Release, wie jene aus der Liste der Abbildung 10, kann man jedes Feature überprüfen und kontrollieren, ob oder wie viele Change Request Tickets zu dem jeweiligen Feature verlinkt sind. Im konkreten Beispiel aus Abbildung 15 kann man entnehmen, dass das als Beispiel gewählte Requirement genau einen Change Request besitzt.

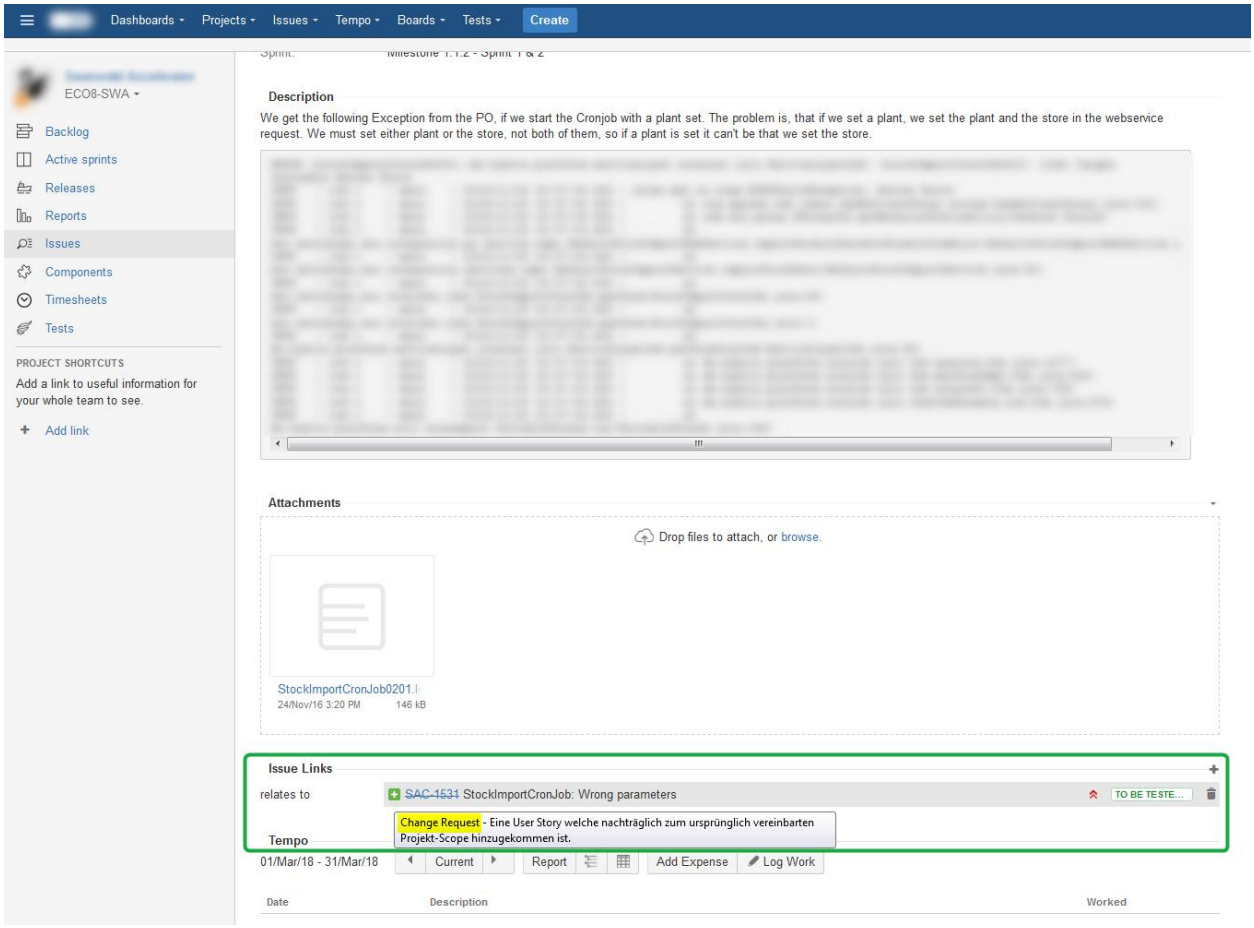


Abbildung 15 - Verlinktes Change Request Ticket zu einem Feature

8.4.2.4 Code Coverage

Für die nächsten Einflussfaktoren der Gruppe der *Implementation Phase* wird das Tool für statische Code-Analyse *SonarQube* verwendet. Die Richtlinie des ersten Einflussfaktors dieser Gruppe, *Code Coverage,* verlangt, dass die Anzahl der Unit Tests, des mit dem Requirement zusammenhängenden Codeabschnittes überprüft wird. Um diese Richtlinie zu überprüfen, benötigt man mehrere Schritte und muss auch mehrere Tools kombiniert nutzen, um an den gesuchten Wert zu gelangen.

Als erstes navigiert man über das Ticket des Requirements im *Jira* zu den Commits (Abbildung 16), welche vom jeweiligen Entwickler oder von der jeweiligen Entwicklerin erstellt wurden und über den Filehosting-Dienst *Bitbucket* zur Verfügung gestellt werden, wie in Abbildung 17 zu sehen ist.

In einem Commit können sowohl neu erstellte Dateien enthalten sein als auch Bearbeitungen existierender Dateien. Im Beispiel aus Abbildung 17 wurde eine Datei ausgewählt (gelb hervorgehoben), um deren Unit Test Coverage zu überprüfen.

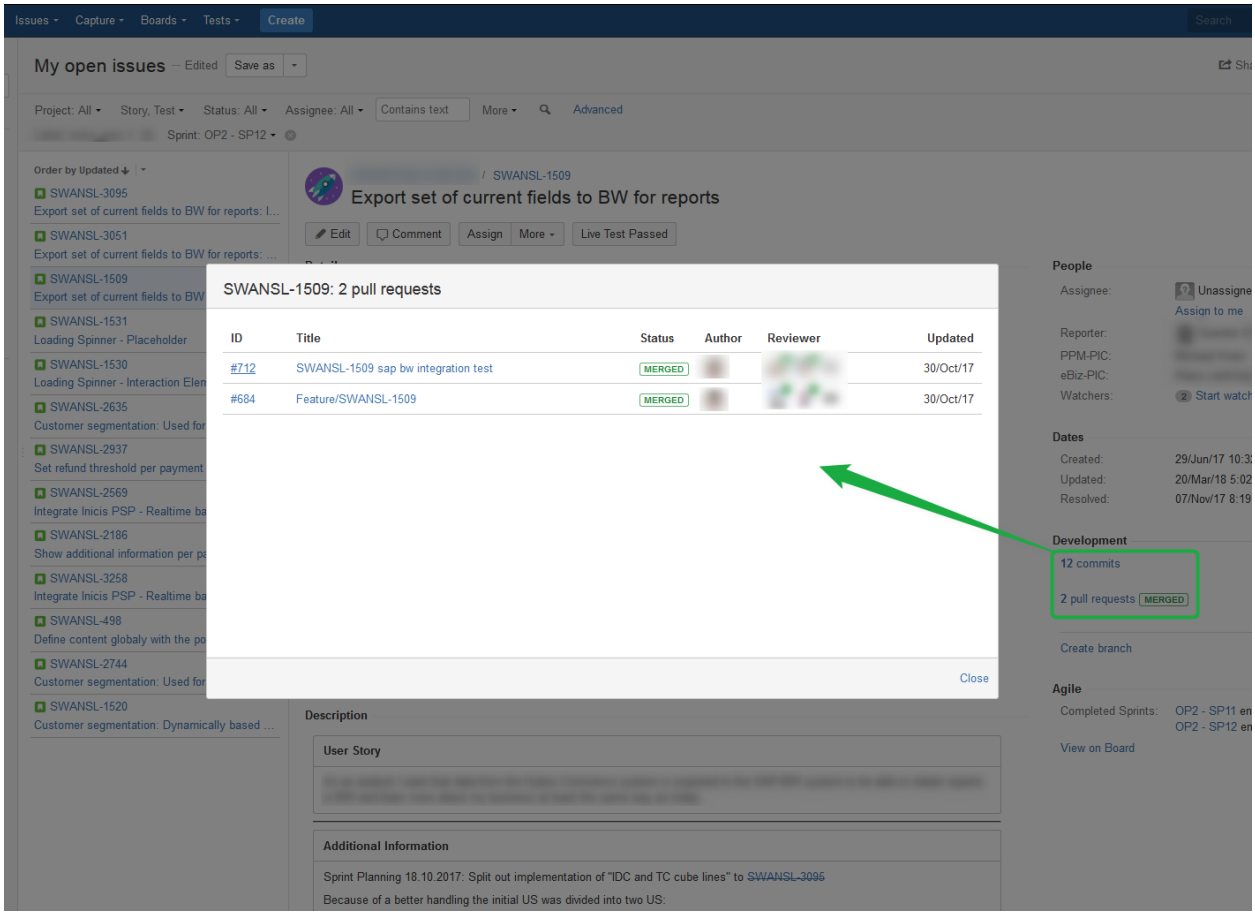


Abbildung 16 - Verlinkung vom Ticket im Jira zu den Commits im Bitbucket

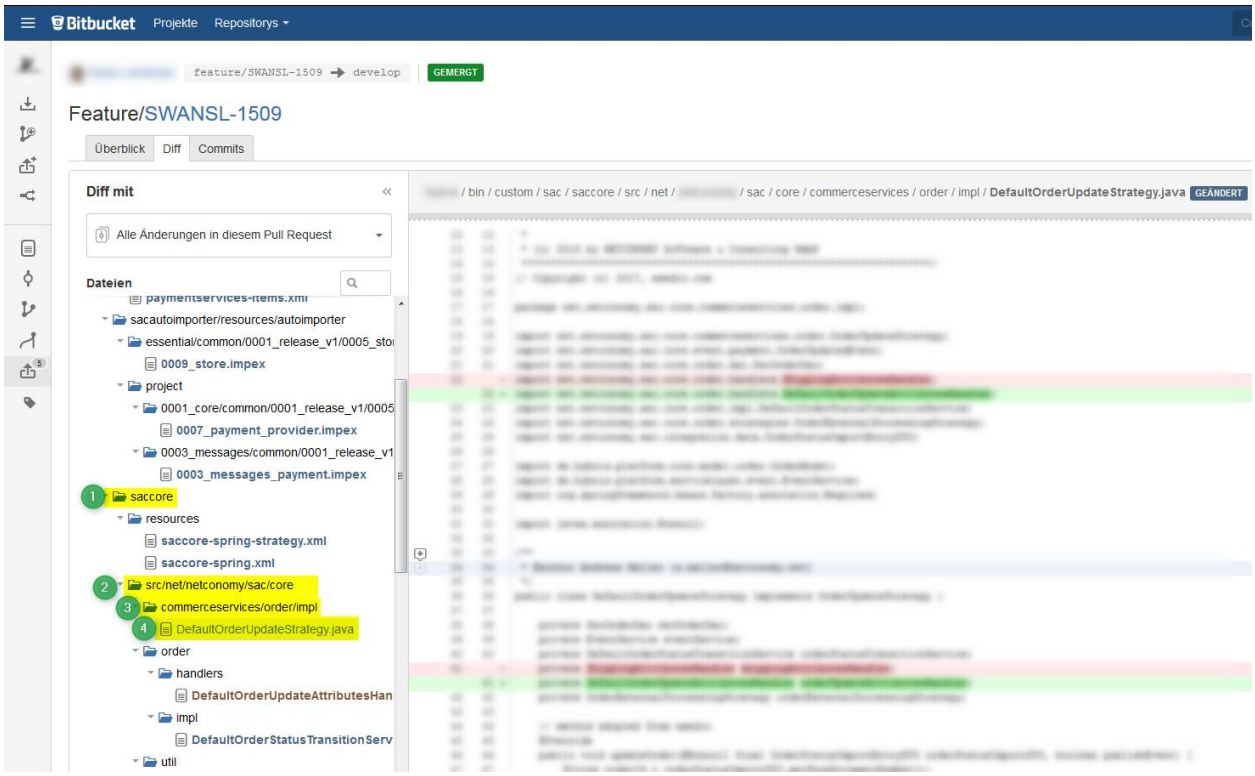


Abbildung 17 - Geänderte Dateien durch Commits basierend auf einem Requirement

Für das Überprüfen der tatsächlichen Code Coverage dieser Datei greifen wir nun auf das Tool *SonarQube* zu. Wie in Abbildung 18 zu erkennen ist, findet man unter dem Reiter *Measures* in der Navigation auf der linken Seite das Maß *Coverage*. Dort befindet sich ein Unterpunkt *Uncovered Lines*, welcher genutzt werden kann, um Dateien mit unzureichender *Code Coverage* zu finden.

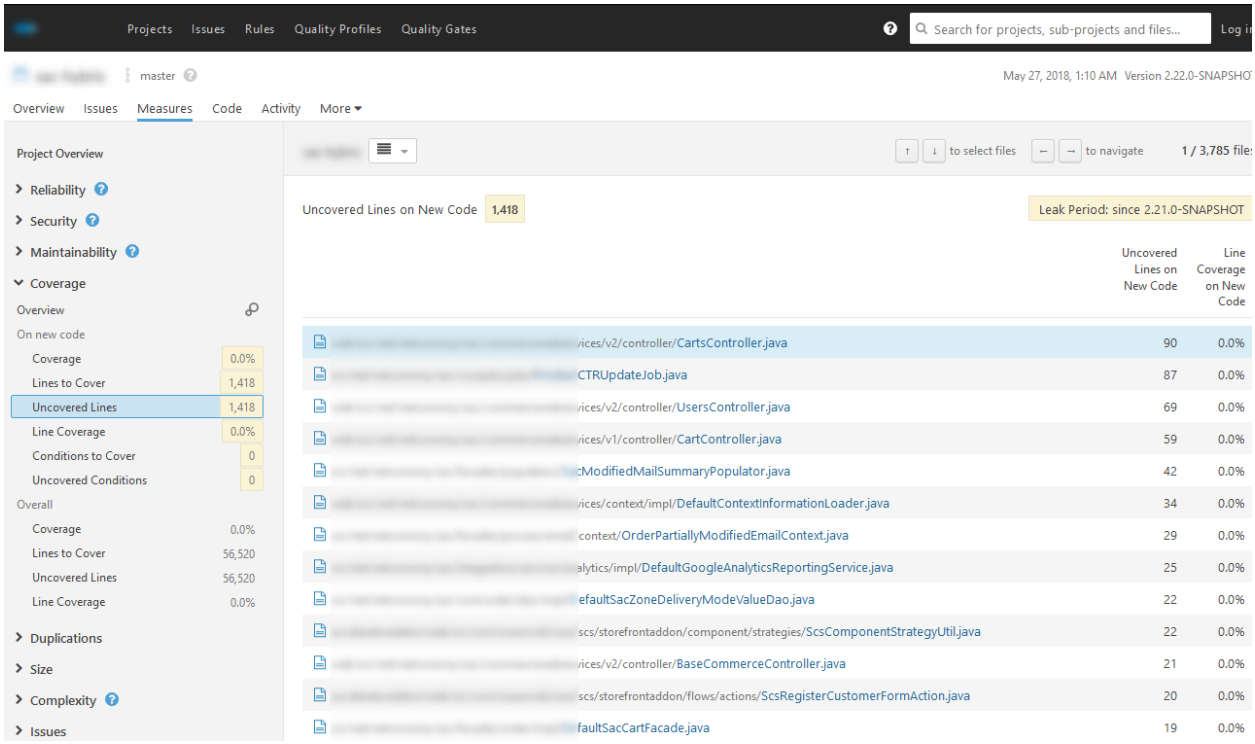


Abbildung 18 – Übersicht der nicht abgedeckten Codezeilen im SonarQube

Um zu der Datei aus dem Commit zu gelangen, kann man mithilfe der Dateistruktur, welche in Abbildung 17 nummeriert ist, bis zu dieser Datei navigiert werden. Diese Vorgehensweise kann jedoch eine Menge Zeit in Anspruch nehmen, je größer der Sourcecode und dessen Dateistruktur sind. Daher wird in diesem praktischen Beispiel für diesen und alle kommenden Einflussfaktoren, welche das Tool *SonarQube* benötigen, eine einfachere Vorgehensweise angewandt. Bei dieser Vorgehensweise wird nach der jeweiligen Datei in der Suchmaske im oberen rechten Bereich des Tools gesucht, in dem der Dateiname, welcher aus dem Commit im *Bitbucket* entnommen wurde, dort eingegeben wird.

Wie man aus der Abbildung 19 erkennen kann, resultiert die Suche in einem Popup unter dem Suchfeld mit einem exakten Ergebnis. Klickt man auf dieses Ergebnis gelangt man zu der gesuchten Datei. Dort kann schließlich die Code Coverage eingesehen werden. Das Beispiel aus Abbildung 20 zeigt, dass die Datei *DefaultOrderUpdateStrategy.java* eine Code Coverage von 95 % aufweist, wie in der grünen Markierung zu sehen ist. Eine detaillierte Ansicht der Code Coverage einer Datei erhält man, wenn man die Schritte, welche in der Abbildung 21 beschrieben sind befolgt und dabei auf die Option *Show Measures* klickt.

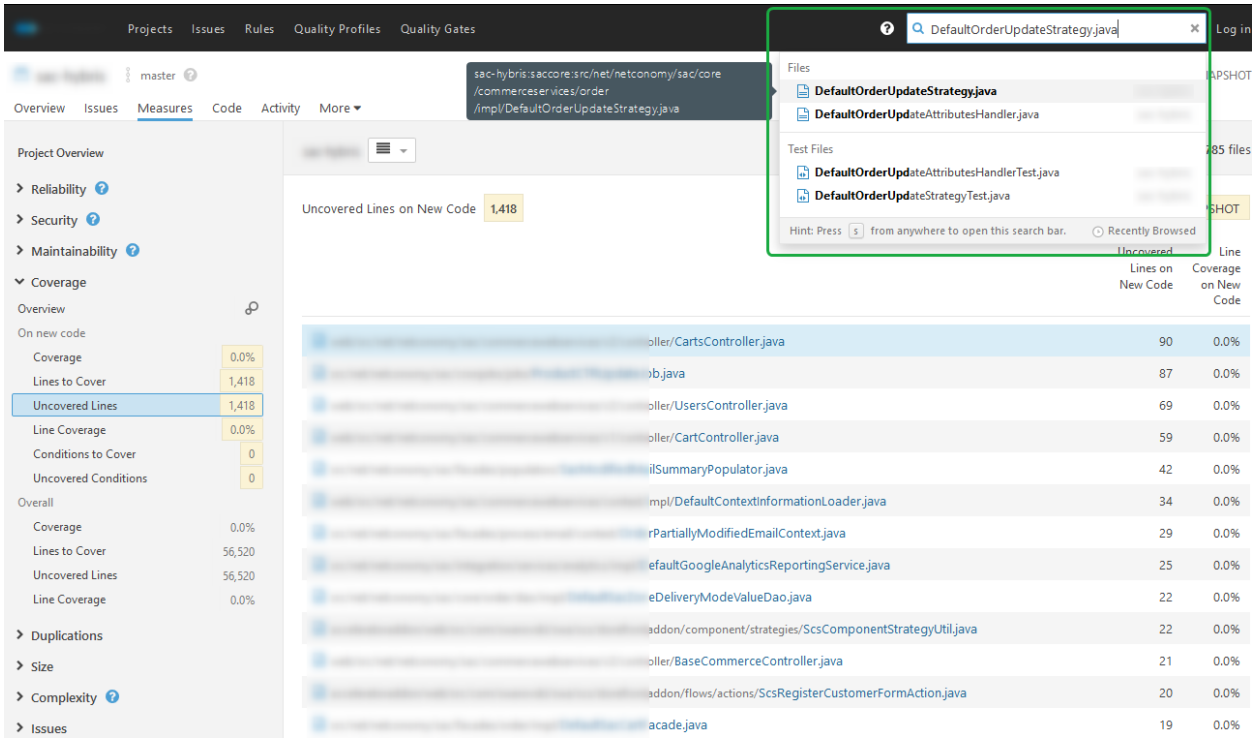


Abbildung 19 – Suchabfrage nach der gesuchten Datei im SonarQube

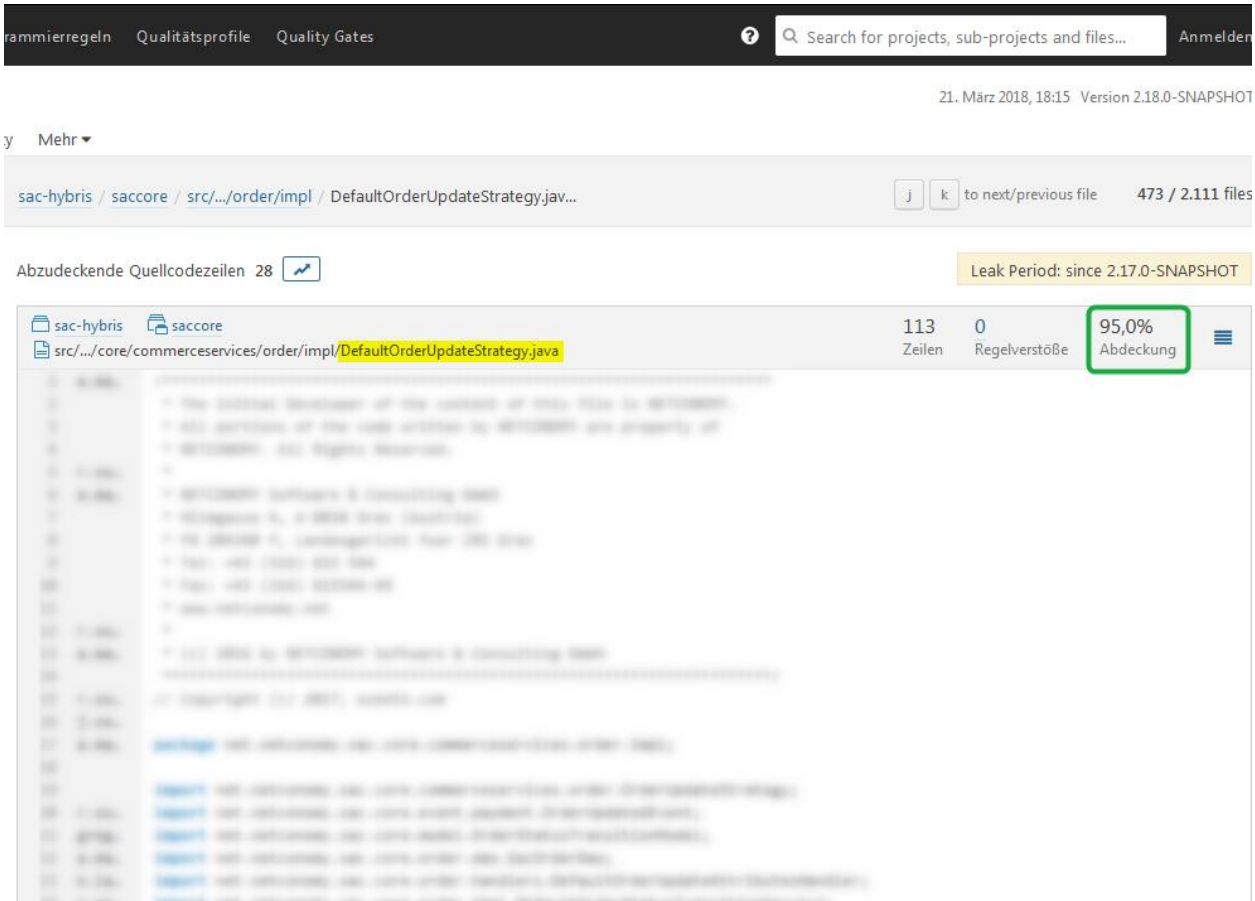


Abbildung 20 - Beispiel der Code Coverage einer bestimmten Datei

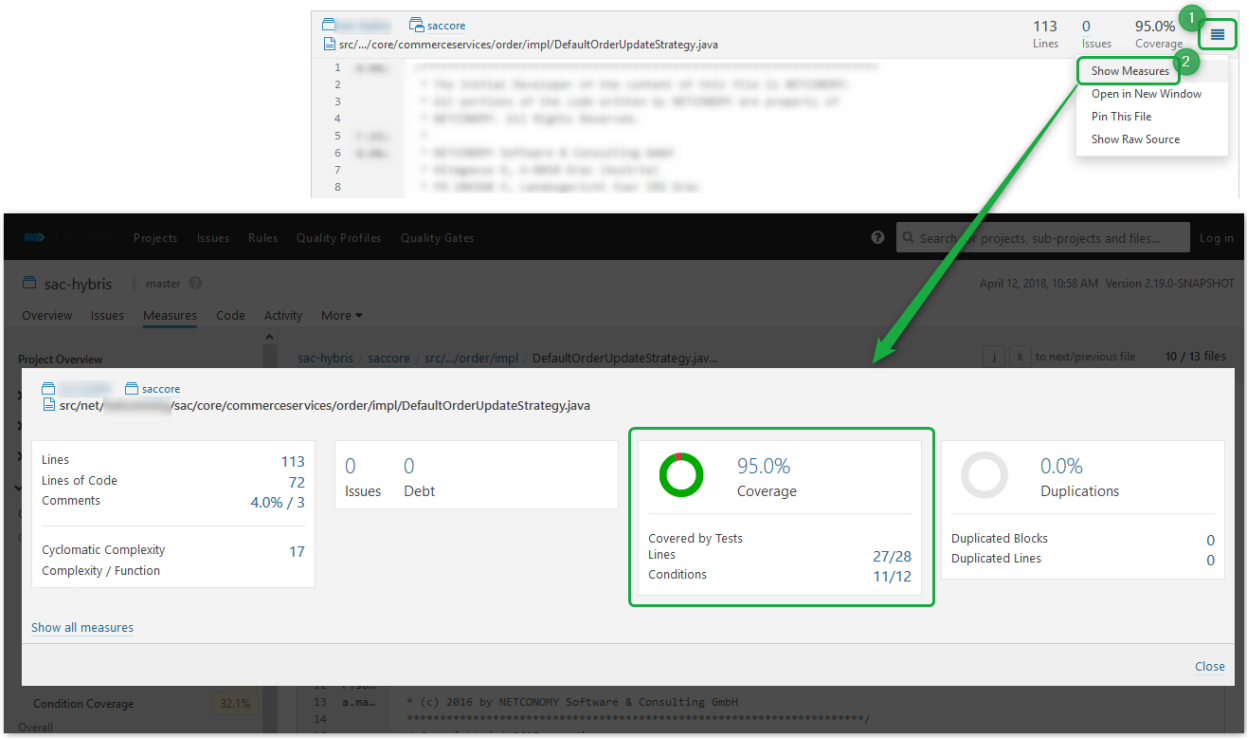


Abbildung 21 - Beispiel der detaillierten Ansicht der Code Coverage einer bestimmten Datei

8.4.2.5 Code Reliability

Der zweite Einflussfaktor der Gruppe der *Implementation Phase*, *Code Reliability*, ist im Unterschied zum vorherigen Einflussfaktor in vier weiteren Untergruppen nach der Anzahl der *Bugs* klassifiziert, welche in Richtlinien abgebildet sind. Die Richtlinien dieses Einflussfaktors werden jedoch mit denselben Tools und auf ähnliche Art und Weise überprüft wie die Richtlinie des Einflussfaktors *Code Coverage*.

Zunächst wird wie bei dem Einflussfaktor *Code Coverage* vom Ticket des Requirements zu den Commits navigiert (Abbildungen 16 & 17). Dort wird wiederum eine Datei gewählt und deren Dateistruktur für das Tool *SonarQube* herangezogen.

Als nächsten Schritt wird im *SonarQube* zum Reiter *Maße* navigiert und der Navigationspunkt *Reliability* auf der linken Seite geöffnet. Unter den Unterpunkten *Overall* und *Bugs* gelangt man zu einer Übersicht zu allen Dateien, welche einen Bug beinhalten, wie in Abbildung 22 zu sehen ist.

Eine weitere nützliche Übersicht über die vorhandenen Bugs mit deren Klassifizierung erhält man unter dem Reiter *Vorgänge*. Mit der Konfiguration *Display Mode: Issues* und *Type: Bug*, erhält man eine detailliertere Übersicht über die Klassifizierung des Fehlers und von welcher Codezeile dieser verursacht wurde, wie in der Abbildung 23 dargestellt ist.

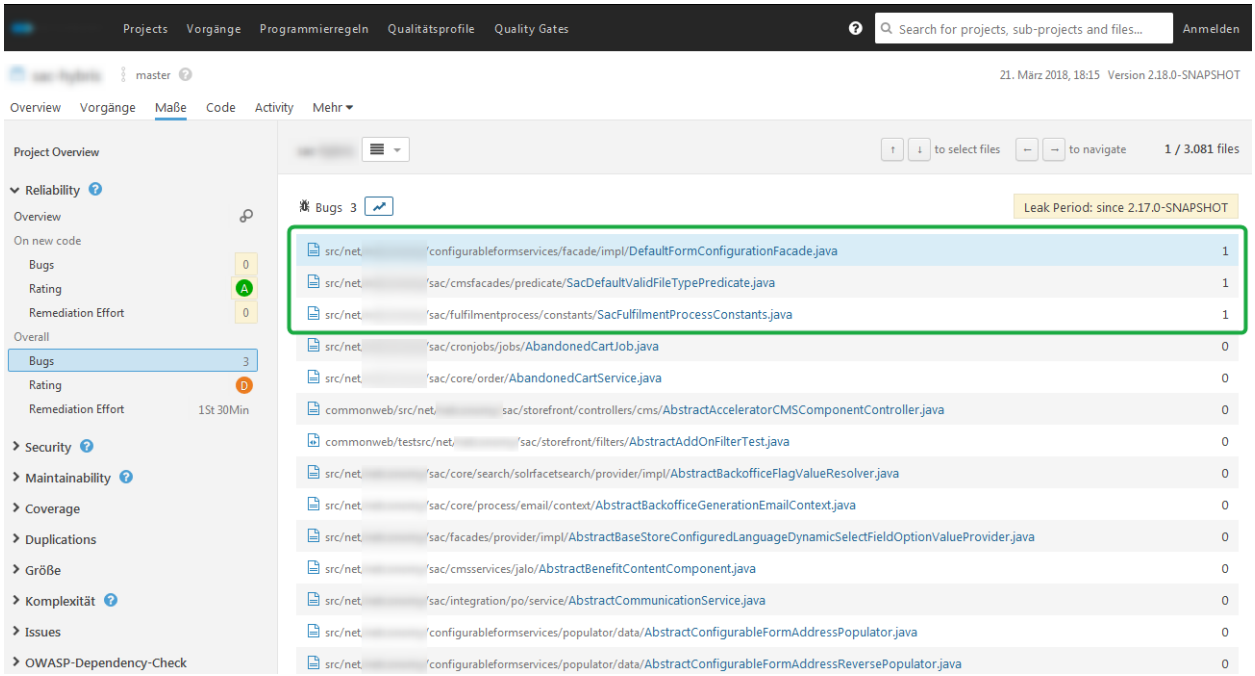


Abbildung 22 - Übersicht im SonarQube zu allen Dateien, welche einen Bug beinhalten

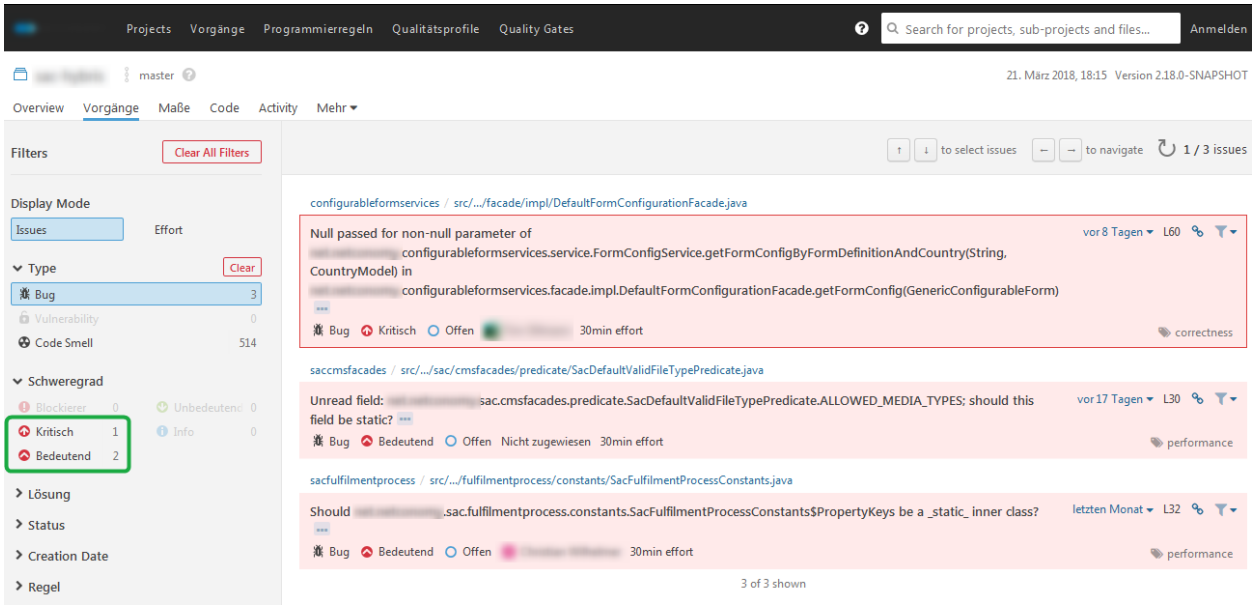


Abbildung 23 - Detaillierte Übersicht im SonarQube zu allen Bugs mit deren Klassifizierung

Für eine Überprüfung der Datei aus dem Beispiel des Einflussfaktors *Code Coverage* von zuvor, navigiert man zunächst zur Detailansicht *Show Meassures*, wie bereits in den Abbildungen 19, 20 & 21 demonstriert wurde. Wenn man sich nun diese Detailansicht ansieht, dann kann man erkennen, dass keine *Issues* und somit keine *Bugs* zu dieser Datei existieren, wie in Abbildung 24 zu sehen ist.

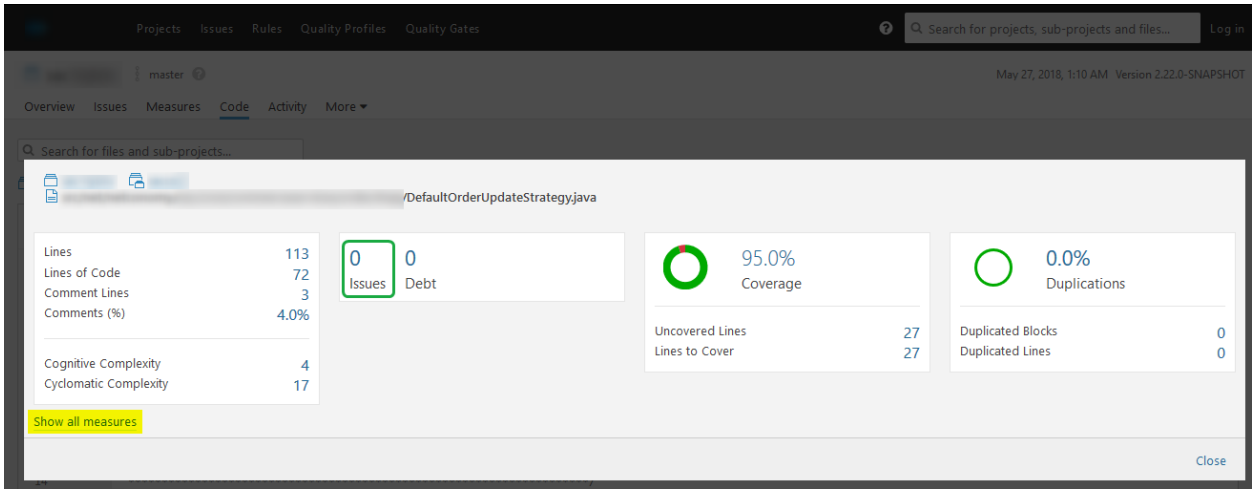


Abbildung 24 - Code Reliability der Datei aus den Beispielen

Eine klassifizierte Auflistung der *Issues* oder allen anderen *Measures* erhält man, indem man auf „*Show all measures“* klickt, welches links unten, wie in der Abbildung 24 zu sehen ist, positioniert ist. Dadurch wird das Popup vergrößert und mehr Details werden sichtbar, wie in der Abbildung 25 demonstriert wird.

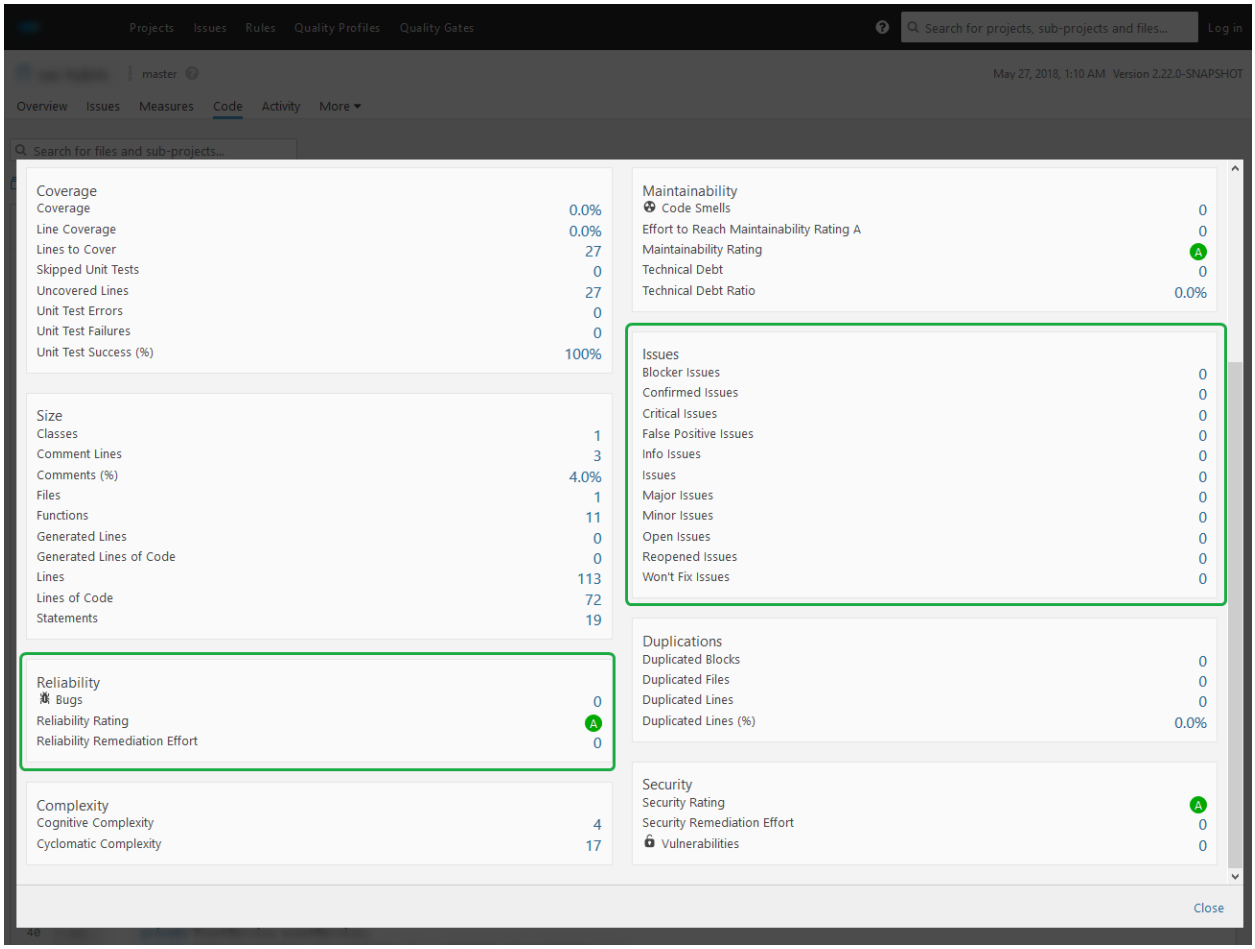


Abbildung 25 - Klassifizierte Ansicht der Code Reliability der Datei aus den Beispielen

8.4.2.6 Code Maintainability

Wie auch der Einflussfaktor *Code Reliability* zuvor, hat auch der dritte Einflussfaktor aus der Gruppe der *Implementation Phase, Code Maintainability,* eine Klassifizierung. Klassifiziert wurde der Einflussfaktor nach *Code Smells* und dessen Höhe der *technischen Schuld* (*Technical Debt*). Ebenso werden für diesen Einflussfaktor dieselben Tools verwendet wie bisher bei den Einflussfaktoren der Gruppe der *Implementation Phase*.

Dazu wird im *SonarQube* zum Reiter Maße navigiert und der Navigationspunkt *Maintainability* auf der linken Seite geöffnet, wie in Abbildung 26 zu sehen ist. Navigiert man durch die Unterpunkte *Overall* und *Code Smells* gelangt man zur Auflistung aller Dateien, welche *Code Smells* enthaltenen.

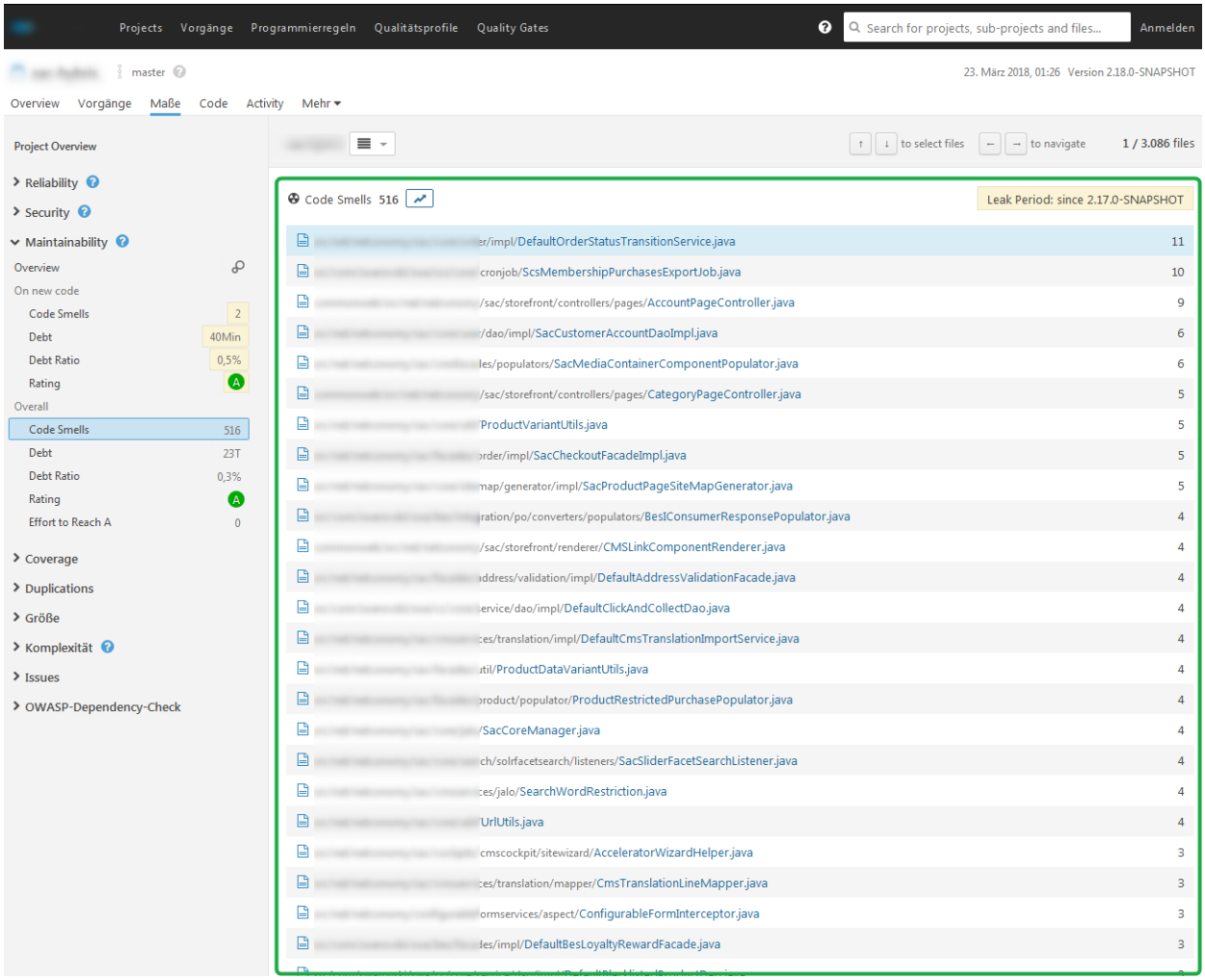


Abbildung 26 - Übersicht im SonarQube zu allen Dateien, welche Code Smells beinhalten

Wie beim vorherigen Einflussfaktor gibt es auch für *Code Maintainability* eine detaillierte Übersicht für *Code Smells*. Mit der Konfiguration *Display Mode: Issues & Type: Code Smell*, wird man über die Klassifizierung des *Code Smells* informiert und von welcher Codezeile diese verursacht wurden, wie in Abbildung 27 dargestellt wird.

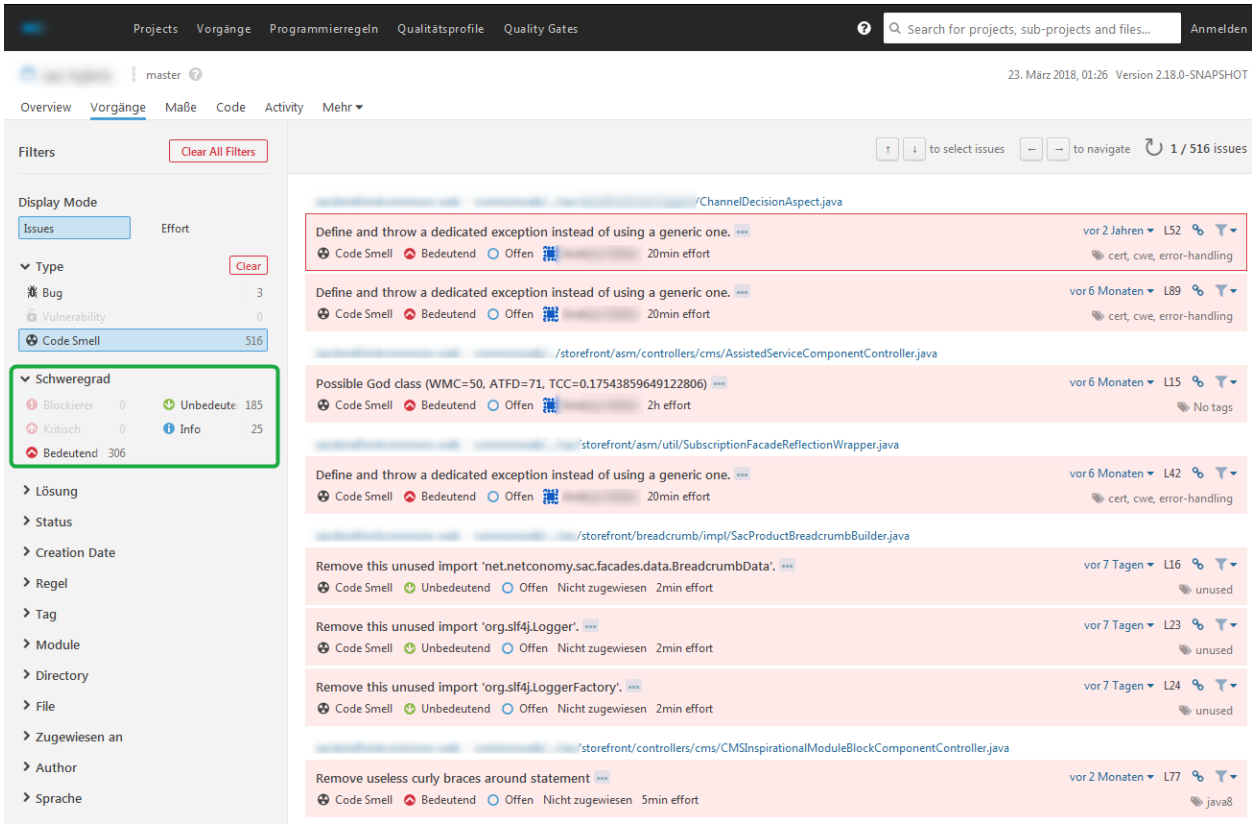


Abbildung 27 - Detaillierte Übersicht im SonarQube zu allen Code Smells mit deren Klassifizierung

Überprüft man nun die Datei aus den Beispielen zuvor, auf derselben Art und Weise wie in Abbildung 24, indem man die Schritte von Abbildung 16, 17 & 19 wiederholt, stellt man fest, dass die Datei keine *Technical Debts* aufweist und somit keine *Code Smells* verursacht (Abbildung 28).

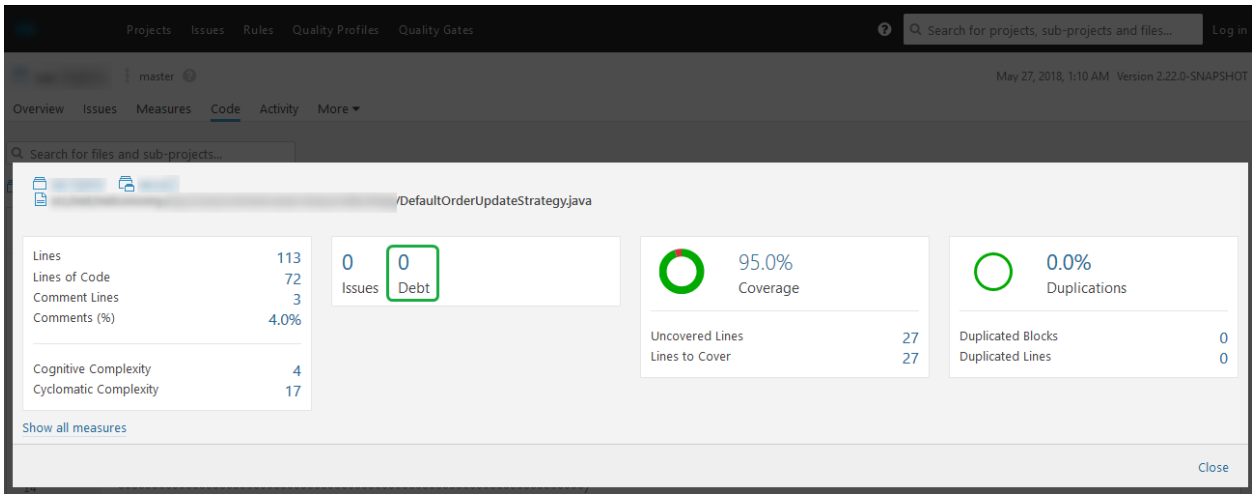


Abbildung 28 - Code Maintainability der Datei aus den Beispielen

8.4.2.7 Code Security

Der dritte Einflussfaktor der Gruppe der *Implementation Phase*, *Code Security*, wird über *Vulnerabilities* in vier Richtlinien klassifiziert. Es wird für diesen Einflussfaktor dieselbe Tool-Kombination benutzt, wie bei den Einflussfaktoren dieser Gruppe zuvor und die Schritte der Abbildungen 16 & 17 werden ebenfalls wiederholt.

Ist man nach diesen Schritten im *SonarQube* angelangt, wird zum Reiter Maße navigiert und der Navigationspunkt *Security* auf der linken Seite geöffnet. Anschließend navigiert man durch die Unterpunkte *Overall* und *Vulnerabilities* gelangt man zur Übersicht der Dateistruktur mit der Auflistung der enthaltenen *Vulnerabilities*, wie in Abbildung 29 zu sehen ist.

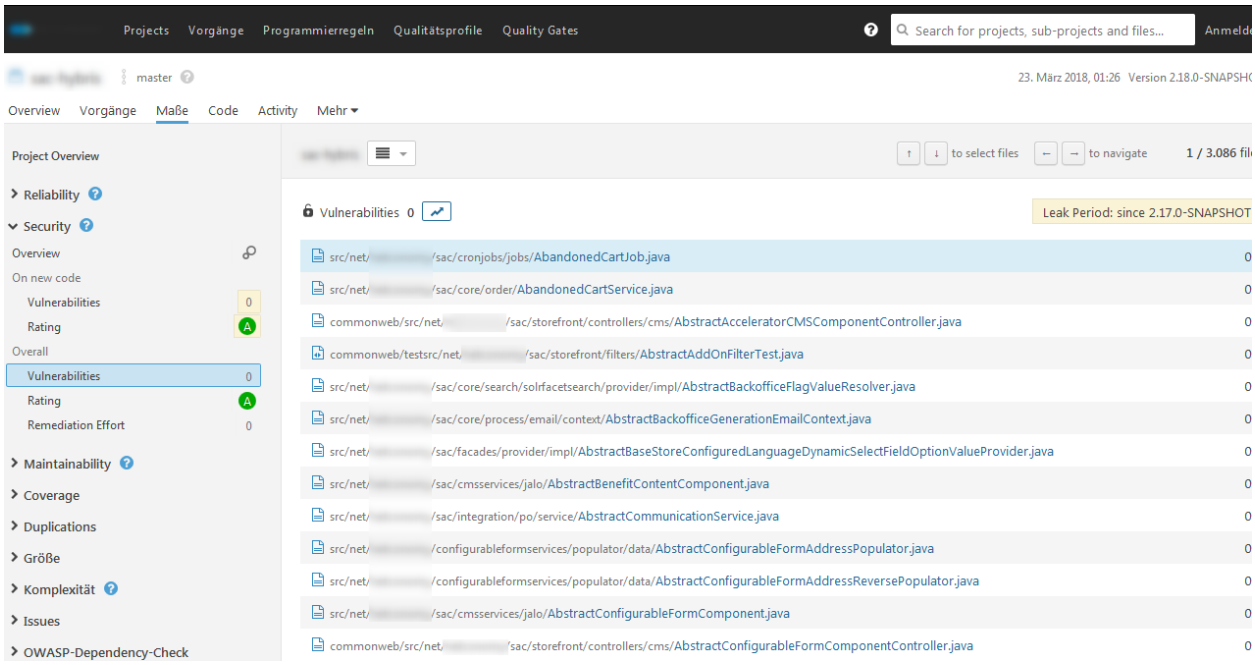


Abbildung 29 - Übersicht im SonarQube zu allen Dateien, welche Vulnerabilities beinhalten

Eine klassifizierte Ansicht über die *Vulnerabilities* ist auch für den Einflussfaktor Security verfügbar (Abbildung 25), jedoch konnten keine *Vulnerabilities* festgestellt werden, wie man aus der Abbildung 29 entnehmen kann, weswegen keine explizite Abbildung dafür bereitgestellt wird. Dies bedeutet automatisch für die Datei aus den bisherigen Beispielen, dass keine *Vulnerabilities* dort aufgetreten sind.

8.4.2.8 Cyclomatic Complexity

Im Unterschied zu den vorherigen Einflussfaktoren der Gruppe der *Implementation Phase*, wurde der Einflussfaktor *Cyclomatic Complexity* nicht klassifiziert. Auch liegt der Fokus bei diesem Einflussfaktor nicht direkt auf dem Grad der Komplexität eines Codeabschnittes, sondern auf den Umgang mit dieser in Form von Test Cases.

Für die Übersicht der *Cyclomatic Complexity* des Sourcecodes, werden die Schritte aus den Abbildungen 16 & 17 durchgeführt. Im *SonarQube* angelangt, wird zum Reiter Maße navigiert und der Navigationspunkt *Komplexität* und der Unterpunkt *Komplexität* auf der linken Seite geöffnet, wie in der Abbildung 30 zu sehen ist.

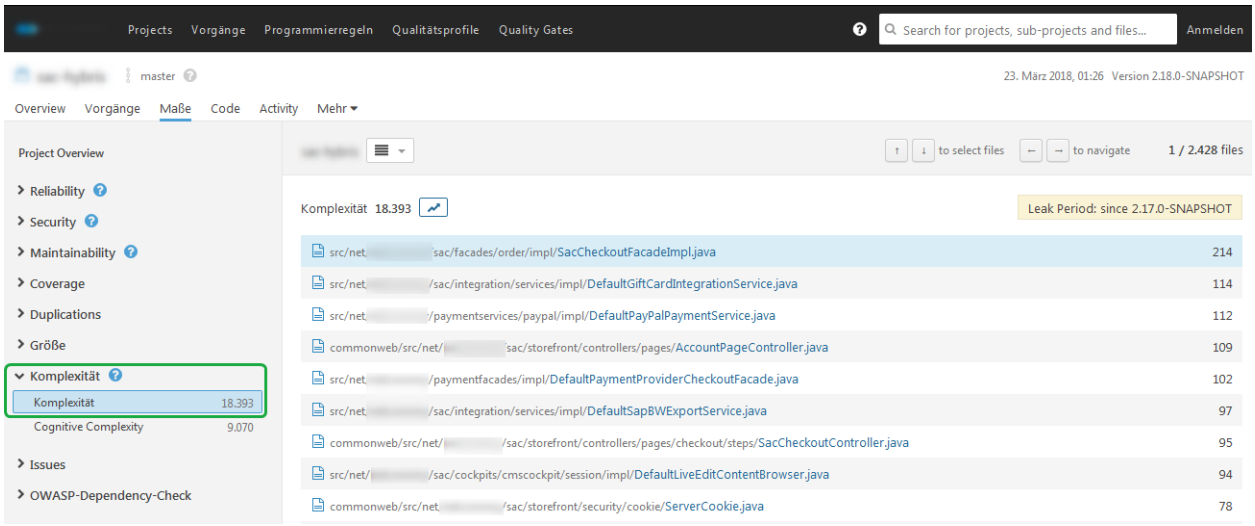


Abbildung 30 – Übersicht im SonarQube zur Komplexität aller Dateien

Um zu der *Cyclomatic Complexity* einer Datei im Code zu gelangen, sucht man nach der expliziten Datei im Suchfeld und navigiert über die Suchergebnisse dorthin, um dann die Komplexität aus der Detailansicht zu entnehmen. Überschreitet die Komplexität dieses Codeabschnittes einen gewissen vordefinierten Grenzwert, werden die Test Cases des zusammenhängenden Features überprüft. Liegt die Anzahl dieser unter einem bestimmten Zielwert aus der Bewertungsmatrix, wird das Feature aus dem Release entnommen.

Um ein konkretes Beispiel dafür vorzuzeigen, wird erneut die Datei aus den vorherigen Beispielen aufgegriffen und die Schritte ähnlich wie in den Abbildungen 19 & 21 wiederholt. In der Abbildung 31 wurde bereits zur Detailansicht navigiert und es kann daraus entnommen werden, dass diese Datei eine Komplexität von 17 aufweist.

Der Grenzwert für diesen Einflussfaktor ist in der Bewertungsmatrix zwar nicht definiert, da er projektbezogen und je nach Anforderungen definiert wird, jedoch nehmen wir in diesem Beispiel an, dass dieser überschritten wurde und somit die Richtlinie zum Einsatz kommt.

Daher muss als nächster Schritt überprüft werden, ob die Anzahl der Test Cases des zum Codeabschnittes zusammenhängenden Features den Zielwert dieser Richtlinie erreicht hat oder nicht. Dazu navigiert man wieder zurück zum Feature im Tool *Jira*, um dort die verlinkten Test Cases zu überprüfen, wie bereits bei den Einflussfaktoren der Gruppe des *Requirements Engineerings* (Abbildung 13) gemacht wurde. Überprüft man die verlinkten Tickets dieses Features, welches in Abbildung 32 zu sehen ist, kann man keine Verlinkung zu einem Test Case entdecken. Dies würde bedeutet, dass dieses Feature die Richtlinie nicht erfüllt und daher aus dem Release entnommen werden muss.

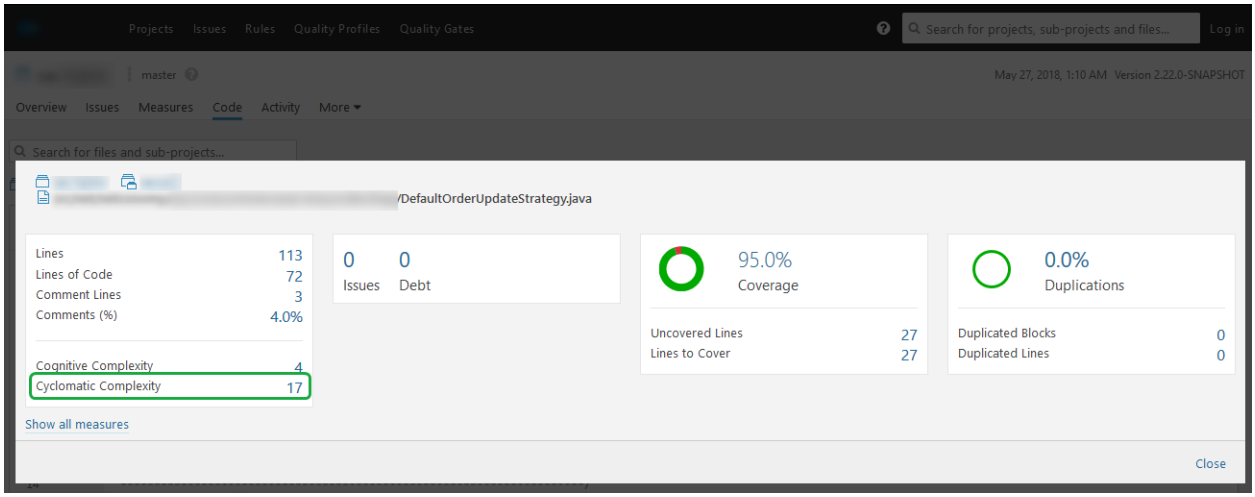


Abbildung 31 - Cyclomatic Complexity der Datei aus den Beispielen

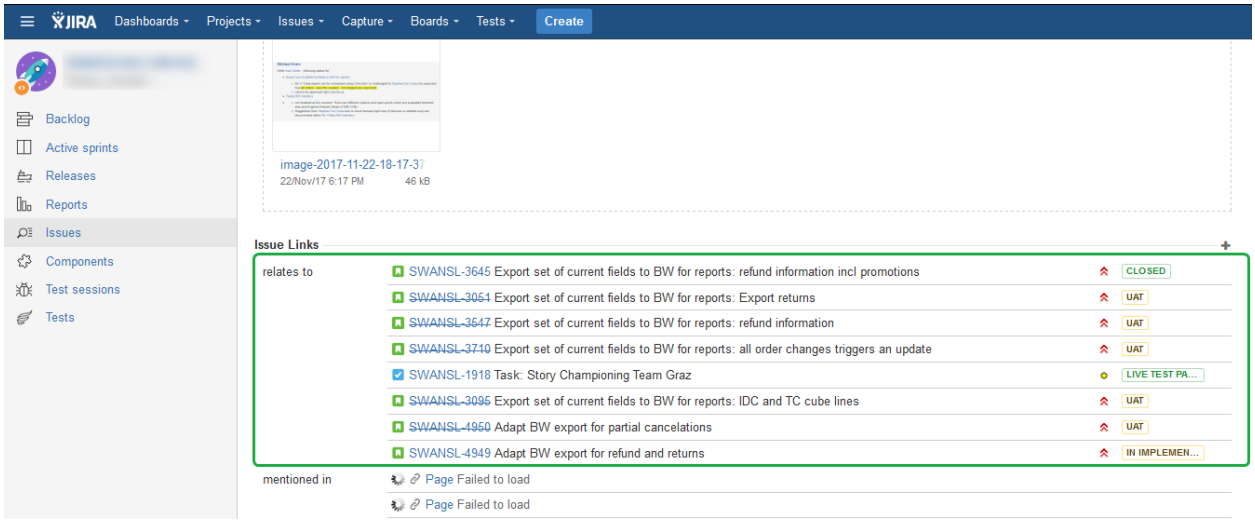


Abbildung 32 - Zu der Datei aus den Beispielen verlinkte Tickets

8.4.2.9 Aktuelle Qualitätsrate

Der folgende Einflussfaktor, *Aktuelle Qualitätsrate*, ist der erste Einflussfaktor aus der Gruppe der *Quality Assurance* und besitzt eine Klassifizierung anhand der Prüfebenen des Testings. Diese Klassifizierungen werden folgend in Unterkapiteln behandelt, da sie auf unterschiedliche Art und Weise ermittelt werden.

8.4.2.9.1 Komponententests

Die Erfolgsrate der Komponententests oder Unit Tests kann im Tool *SonarQube* auf einfache Weise eingesehen werden. Dabei wird, wie auf der Abbildung 33 zu sehen ist, zum Reiter Maße navigiert und der Navigationspunkt *Coverage* auf der linken Seite geöffnet. Navigiert man durch die Unterpunkte *Tests* und *Erfolg* gelangt man zur Übersicht der Erfolgsrate der existierenden

Unit Tests im Code. Wie in der Abbildung zu sehen ist, liegt die Erfolgsrate der Unit Tests bei 100 %.

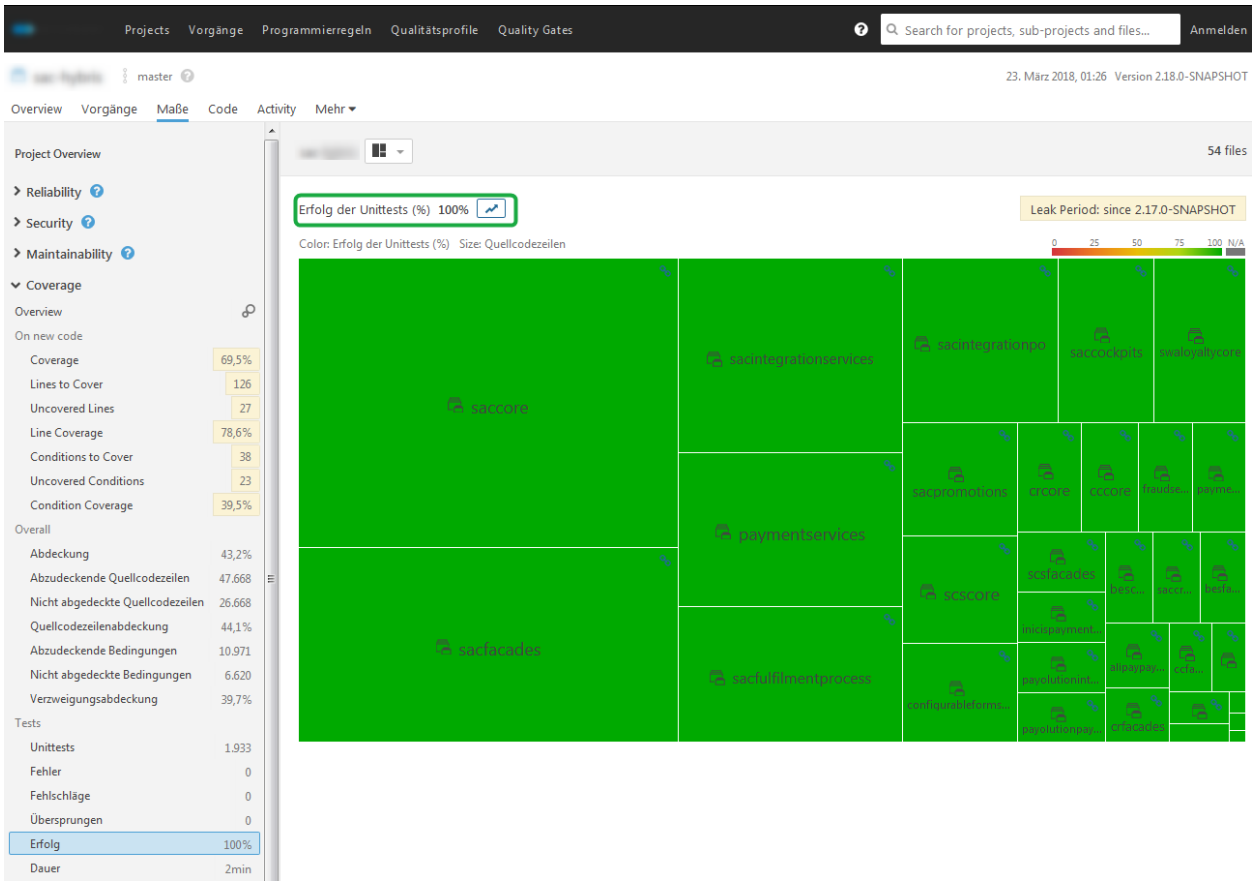


Abbildung 33 - Erfolgsrate der existierenden Unit Tests

Gäbe es jedoch fehlgeschlagene Unit Tests, müsste manuell kontrolliert werden, ob das jeweilige Feature, welches gerade überprüft wird, mit einem der fehlschlagenden Tests zusammenhängt. In dem man im *SonarQube* auf Listenansicht wechselt (grün umrahmtes Rechteck in Abbildung 34), sieht man alle Tests mit deren Namen und genauen Pfad, wobei die fehlgeschlagenen Tests zuerst aufgelistet sind, wie in Abbildung 34 zu sehen ist. Dadurch kann auf das Topic zurückgeschlossen werden, zu welchem der Test zugehörig ist und demnach mit dem Topic des Features abgeglichen werden. Auf Basis dieses Vergleichs wird dann eine Entscheidung für das jeweilige Feature getroffen.

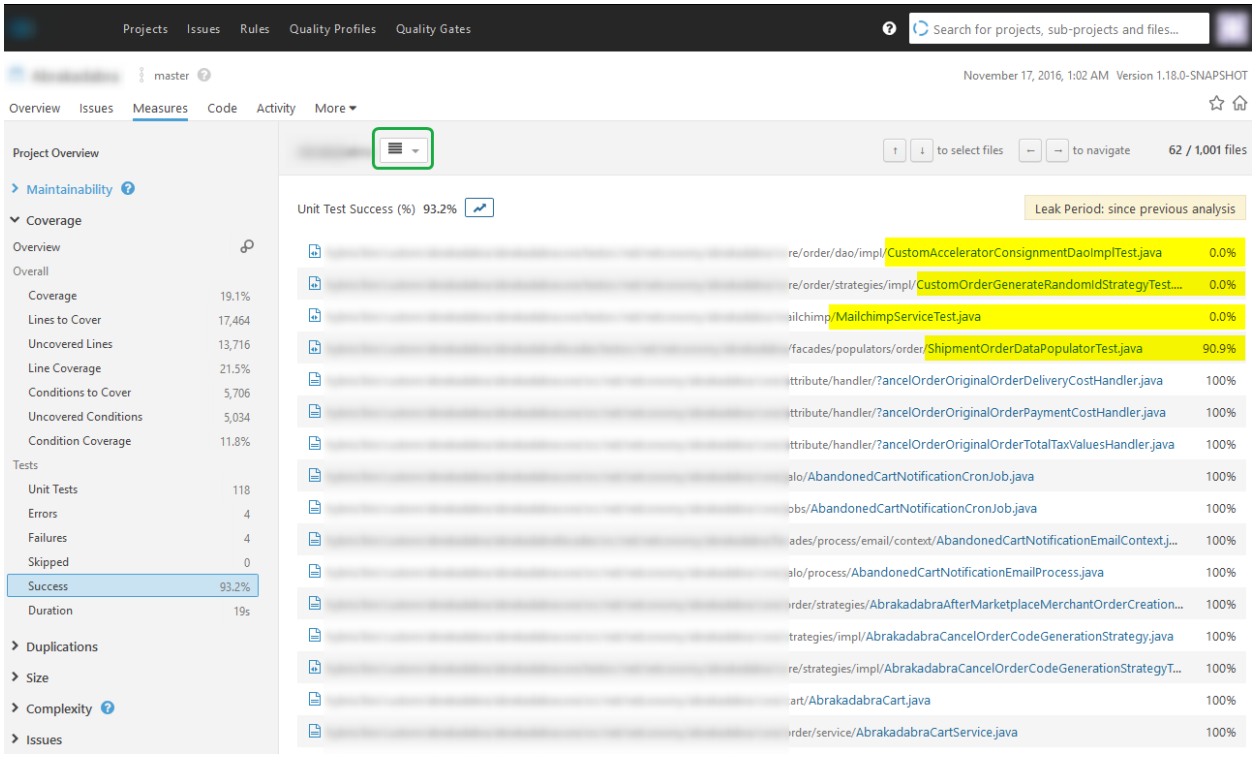


Abbildung 34 - Beispiel für fehlgeschlagene Unit Tests

8.4.2.9.2 Integrationstests

Für die Überprüfung der Erfolgsrate der Integrationstests wird das Tool für das Verwalten von automatisierten Continuous Integration Builds, *Jenkins*, herangezogen. Mithilfe dieses Tools kann eingestellt werden, dass die ganze Sammlung der Integrationstests nächtlich automatisiert ausgeführt wird, sodass kontinuierlich der Code überprüft wird und schnelles Feedback eingeholt werden kann.

Das Projekt im *Jenkins* besteht aus ausführbaren *Jobs*, welche Builds oder Tests automatisiert zu einem bestimmten Zeitpunkt ausführen. In Abbildung 35 sieht man den Job für den Nightly-Build, welcher den neuesten Codestand baut und darauf basierend die Integrationstests ausführt, um zu überprüfen, ob der neueste Codestand korrekt ist. Auf dieser Abbildung sieht man im linken untern Bereich den Build-Verlauf und auf der rechten Seite im grün umrahmten Rechteck das Testergebnis des letzten Builds. Klickt man auf den aktuellsten Build (grün umrahmtes Rechteck ganz links) gelangt man in eine Detailansicht der Testergebnisse dieses Builds, wie in Abbildung 36 demonstriert wird. In dieser Detailansicht sind alle Integrationstests aufgelistet, wobei hier eine Erfolgsrate von 100 % gegeben ist. In Abbildung 37 ist ein Beispiel eines Builds angeführt, welcher auch fehlgeschlagene Tests beinhaltet, um auch diesen Fall im Zuge dieses Einflussfaktors zu veranschaulichen.

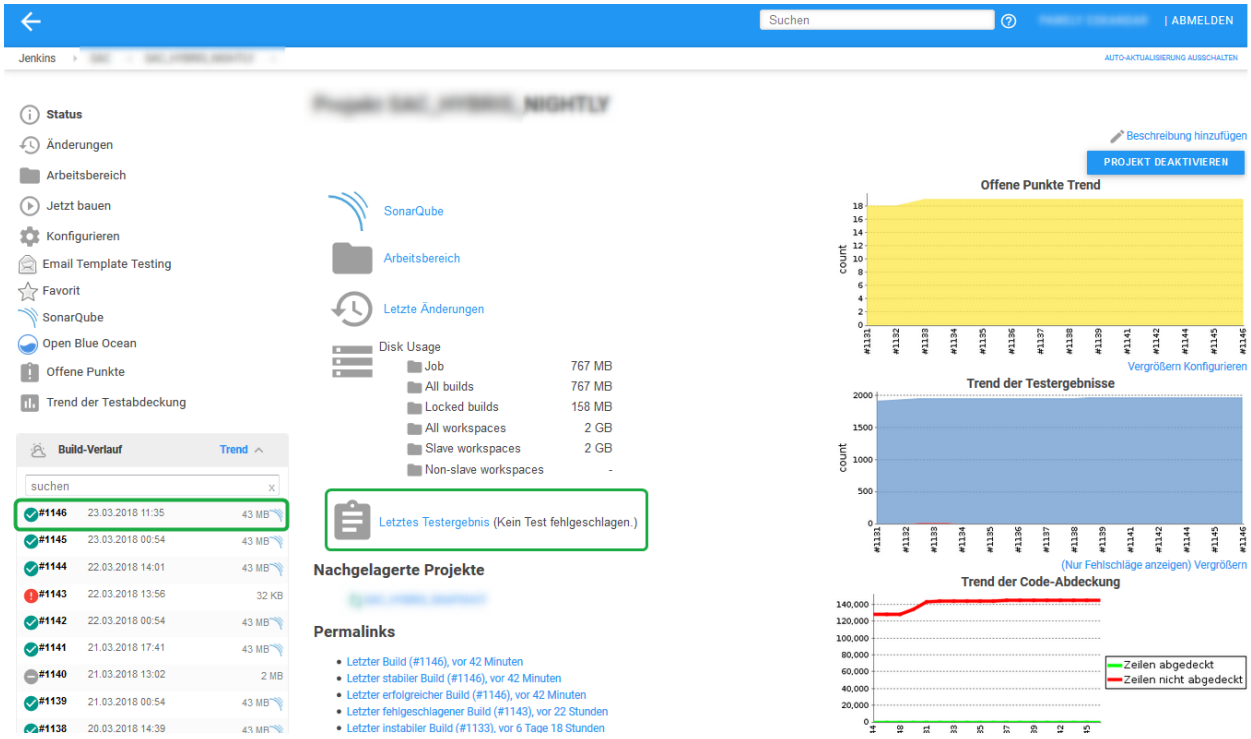


Abbildung 35 - Übersicht der Builds der Integrationstests

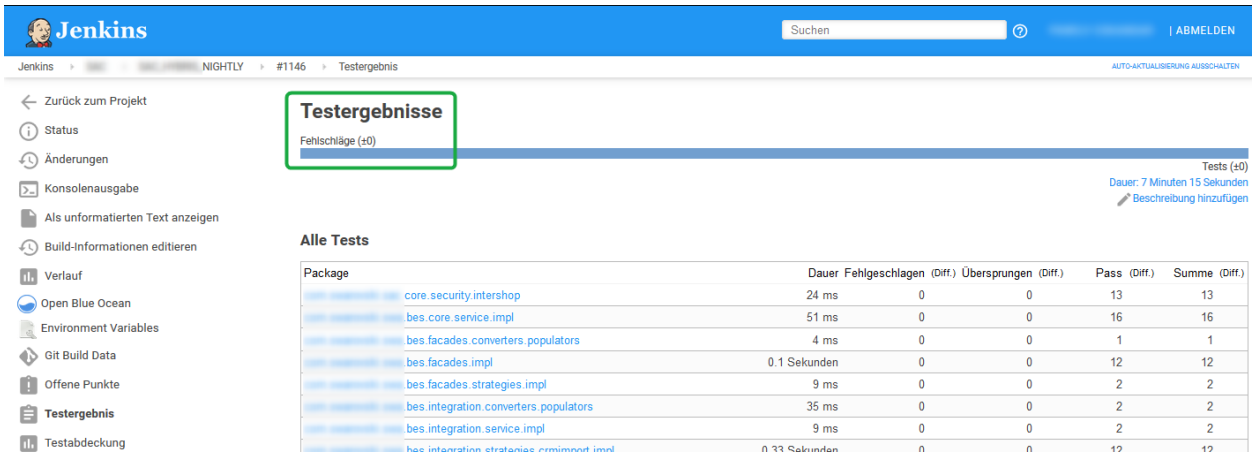


Abbildung 36 - Erfolgreicher Integrationstest-Build

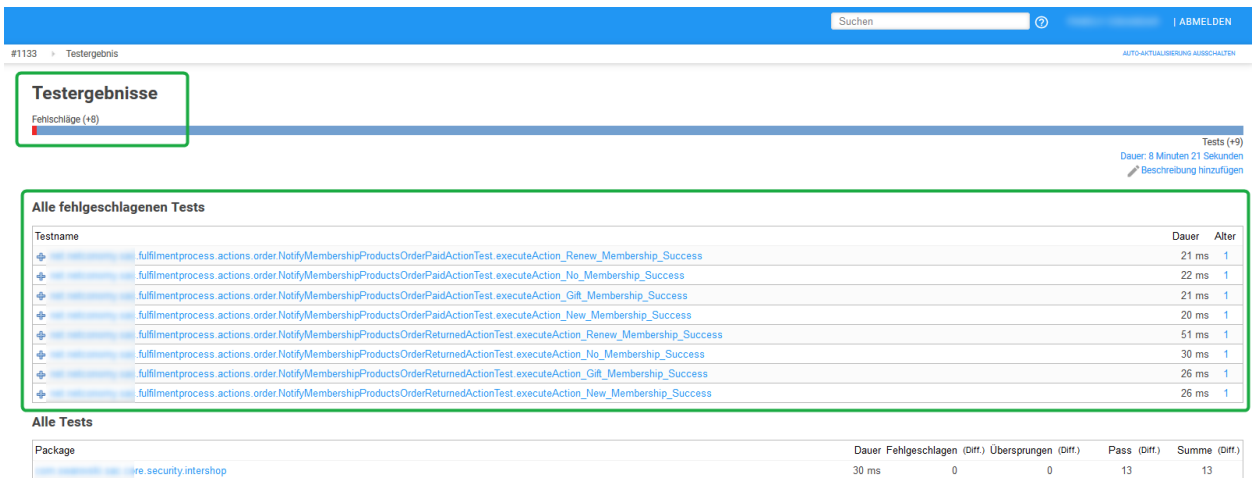


Abbildung 37 - Nicht erfolgreicher Integrationstest-Build

Eine direkte Verbindung eines Features mit einem spezifischen Test aus Abbildung 37, ähnlich wie bei den Einflussfaktoren der Gruppe der *Implementation Phase*, wird von diesem Tool nicht unterstützt. Um von einem Test auf ein Feature im Release zu schließen, muss man sich manuell ansehen, über welches Topic es sich bei jenem Test handelt (Registrierung, Checkout, etc.). Dieses Topic vergleicht man anschließend mit dem Topic des Features und kann somit eine Bewertung abgeben.

8.4.2.9.3 Systemtests

Der Systemtest wird in diesem Beispielprojekt in Form von Regressionstests abgebildet, welche automatisiert ausgeführt werden. Die Erfolgsrate der Regressionstests wird auf dieselbe Art und Weise ermittelt wie die der Integrationstests. Der Systemtest besitzt wie der Integrationstest einen eigenen Job, welcher ebenfalls nächtlich automatisiert ausgeführt wird. In Abbildung 38 sieht man die Übersicht des Jobs mit dem Build-Verlauf, dem Trend der Testergebnisse und eine Zusammenfassung der Testergebnisse des letzten Builds (grün umrahmtes Rechteck).

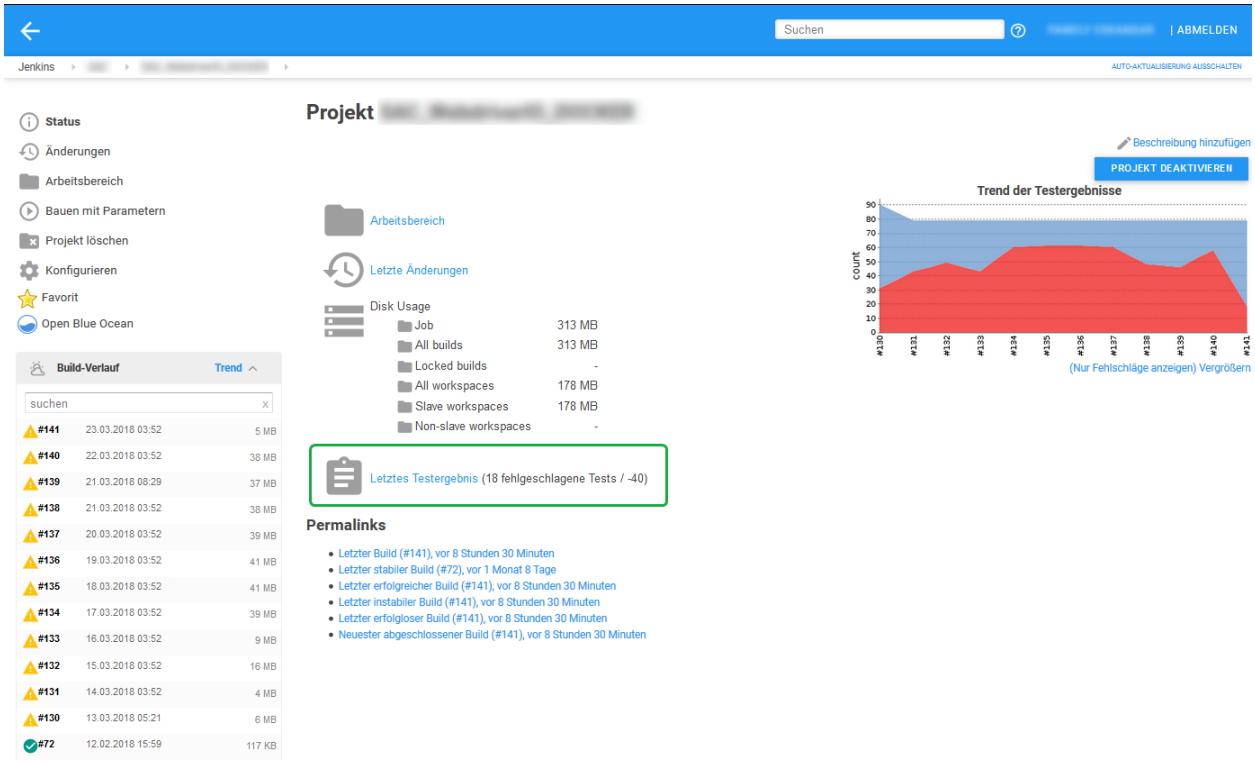


Abbildung 38 - Übersicht der Builds der Akzeptanztests

Navigiert man zu den Testergebnissen des letzten Builds, erhält man eine Übersicht über die fehlgeschlagenen Tests, wie in Abbildung 39 zu sehen ist. In diesem Fall sind 18 von 40 Regressionstests fehlgeschlagen.

Ähnlich wie bei den Integrationstests muss man auch bei den Regressionstests manuell vom Test auf das Feature rückschließen. Dabei geht man auf dieselbe Art und Weise vor und ermittelt

zuallererst das Topic, um welches es sich bei dem Test handelt und vergleicht dieses dann mit dem Topic des Features und kann somit eine Bewertung für das Feature abgeben.

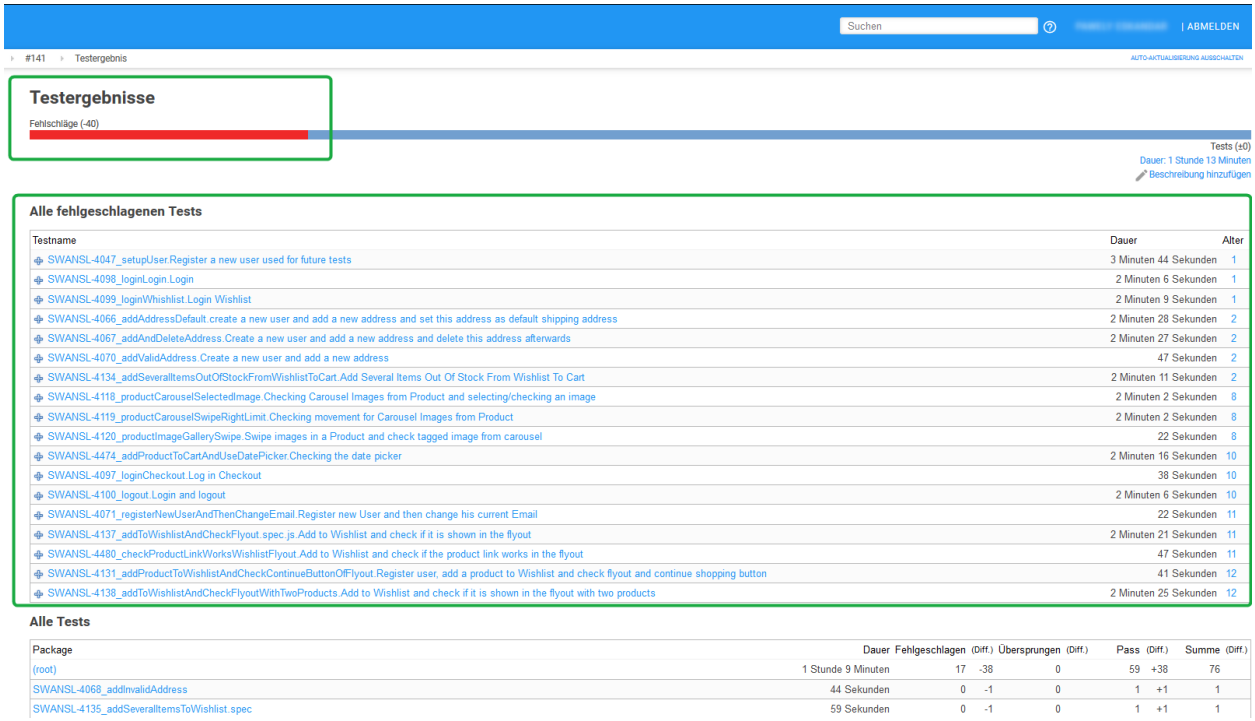


Abbildung 39 - Nicht erfolgreicher Akzeptanztest Build

8.4.2.9.4 Acceptance Tests

Akzeptanztests können im Allgemeinen automatisiert oder manuell durchgeführt werden, wobei automatisierte Akzeptanztests, wie schon in Kapitel 3.3.2 beschrieben wurde, vorteilhafter sind. In diesem Beispielprojekt sind, mangels Zeit und Möglichkeiten, keine automatisierten Akzeptanztests möglich, sondern eine manuelle Überprüfung der Akzeptanzkriterien aller Features des Release wird durchgeführt.

Bei jedem manuellen Test ist es erforderlich eine Testdokumentation zu erstellen, welche unter anderem auch die Testergebnisse beinhaltet. Wählt man nun ein Feature aus dem Release des Beispielprojektes aus der Abbildung 10 aus, dann findet man dort eine Verlinkung auf eine Testdokumentation, wie in Abbildung 40 im grün umrahmten Rechteck zu sehen ist.

Diese Testdokumentation jenes Features beinhaltet unter anderem eine Auflistung der Akzeptanzkriterien und einen Testerfolgsstatus, wie aus den grün umrahmten Rechtecken aus der Abbildung 41 entnommen werden kann. Im folgenden Beispiel wird ersichtlich, dass der Testerfolgsstatus dieses Features erfolgreich war und alle Akzeptanzkriterien erfüllt wurden.

Einen negativen Testerfolgsstatus einer Testdokumentation eines Features, erkennt man daran, dass der Testerfolgsstatus mit *Not Successful* und einem roten Hintergrund gekennzeichnet wird, wie in Abbildung 42 illustriert wird. Im Bereich der aufgelisteten Akzeptanzkriterien werden jene Akzeptanzkriterien mit einem roten Kreuz markiert, welche zu einem negativen Testerfolgsstatus

geführt haben. Ein Feature mit einem solchen Testerfolgsstatus wird laut den Richtlinien der Bewertungsmatrix aus dem Release entnommen.

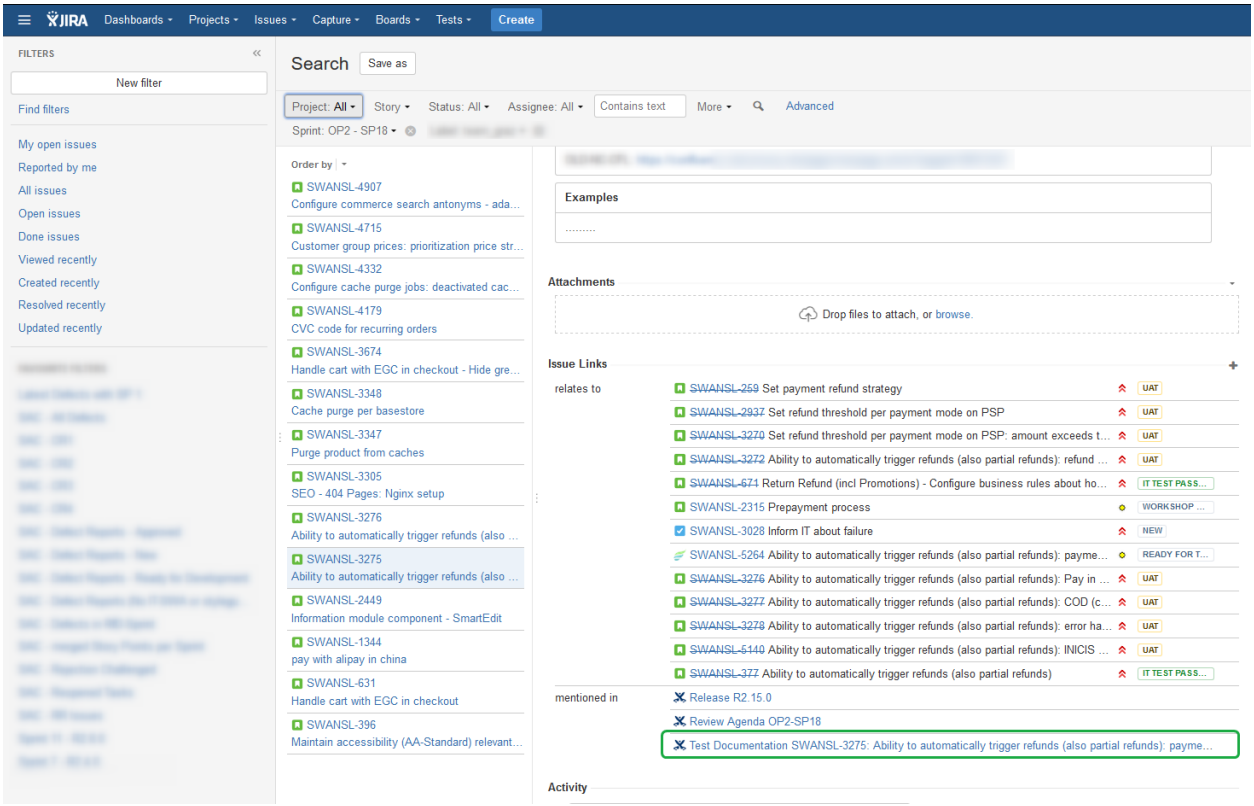


Abbildung 40 - Verlinkung von einem Feature auf dessen Testdokumentation

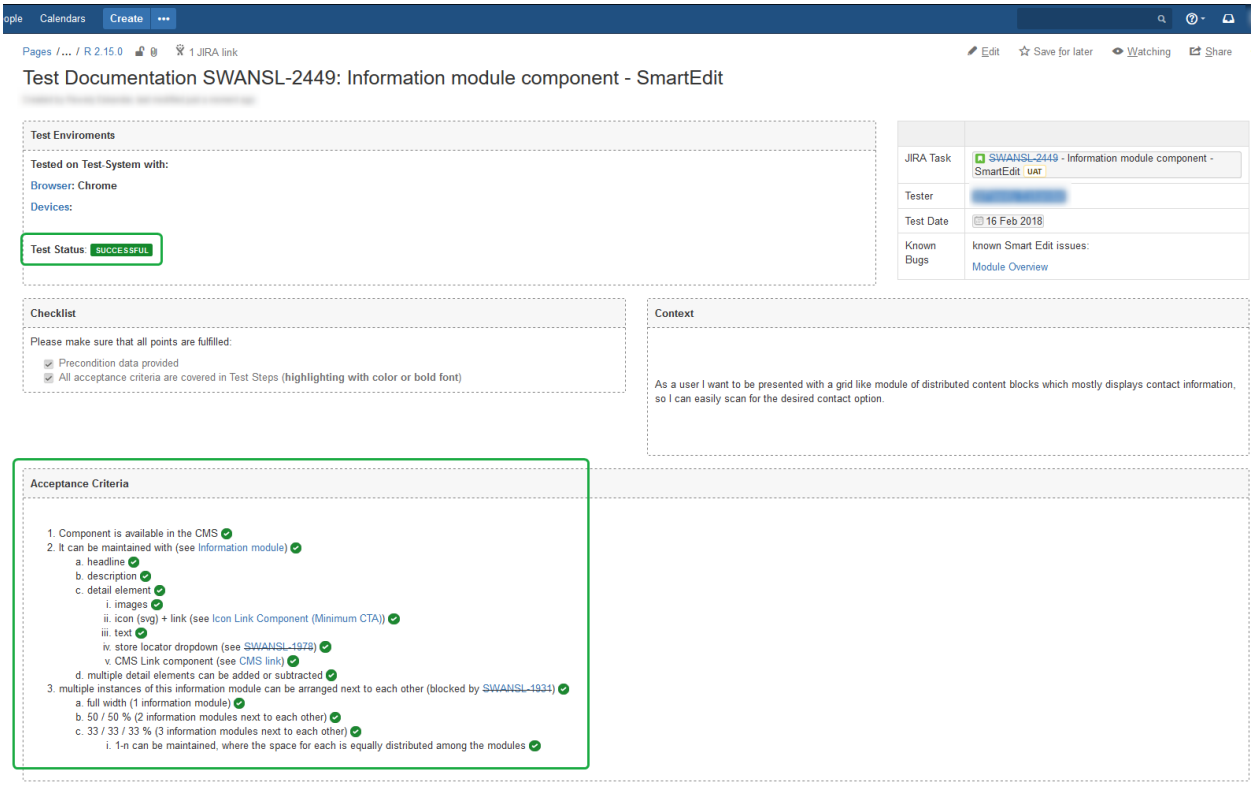


Abbildung 41 - Beispiel einer Testdokumentation mit erfolgreichem Testerfolgsstatus

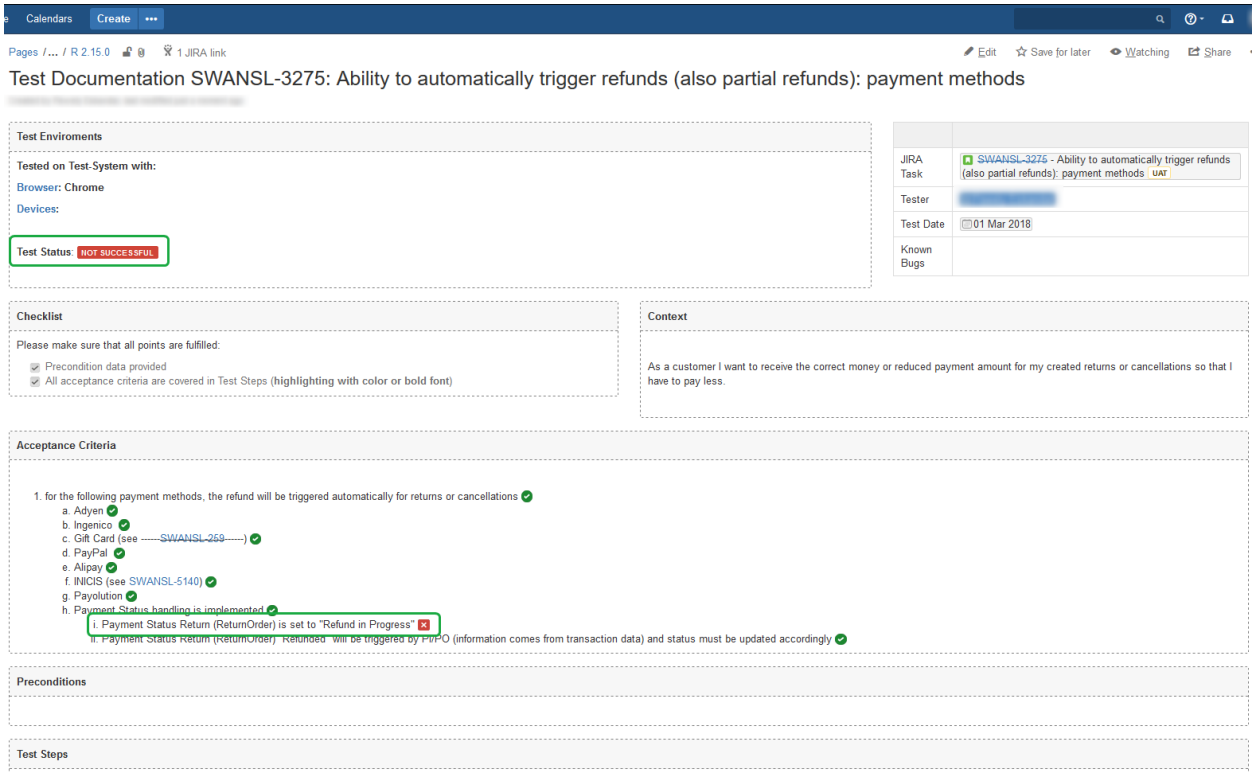


Abbildung 42 - Beispiel einer Testdokumentation mit negativem Testerfolgsstatus

Das Modell sieht für die vierte Richtlinie des Einflussfaktors *Aktuelle Qualitätsrate* eine weiterführende Richtlinie vor, welchen die *Reopen-Rate* eines Features überprüfen soll. Diese Richtlinie sieht vor, dass ein Feature, welches zwar einen erfolgreichen Testerfolgsstatus besitzt, jedoch um diesen zu erreichen mehrere Anläufe gebraucht hat, eventuell aus dem Release entnommen werden soll, da es ein hohes Risiko in sich birgt. Die Anzahl dieser Anläufe nach der dieses Feature gemessen wird, ist nicht in der Bewertungsmatrix definiert und kann beliebig je nach Projektsituation oder Häufigkeit der Vorkommnisse von „Reopens“, gesetzt werden.

Um dies zu überprüfen, wird zuallererst im Tool *Jira* die Liste aus allen Features des Release, wie aus Abbildung 10, vorbereitet. Darauffolgend kann im Jira mit einer genauen Suchabfrage, wie sie in Abbildung 43 im oberen grün umrahmten Rechteck beschrieben wird, jene Features gefunden werden, welche im Zuge des Testings wieder aufgemacht (reopened) worden sind.

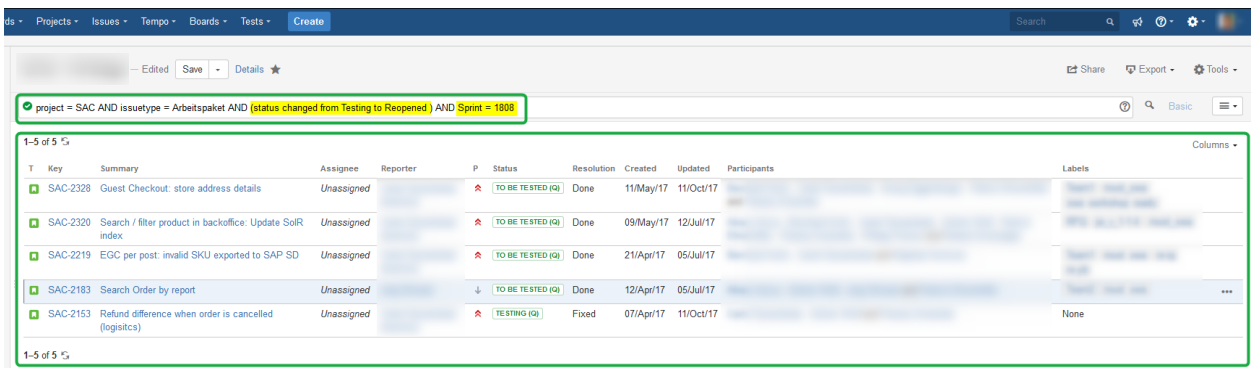


Abbildung 43 - Suchabfrage für die Reopen-Rate im Jira

Die Liste, welche man durch diese Suchabfrage bekommt, zeigt lediglich, welche Tasks grundsätzlich reopened wurden, jedoch nicht wie häufig dies geschehen ist. Für diese Zahl muss man genauer in das Ticket des Features hineinsehen und die Übergänge (Transitions) des Tickets überprüfen, wie in Abbildung 44 gezeigt wird. Dort sind alle Übergänge des Tickets aufgelistet inklusive des Übergangs von *Testing* zu *Reopened*.

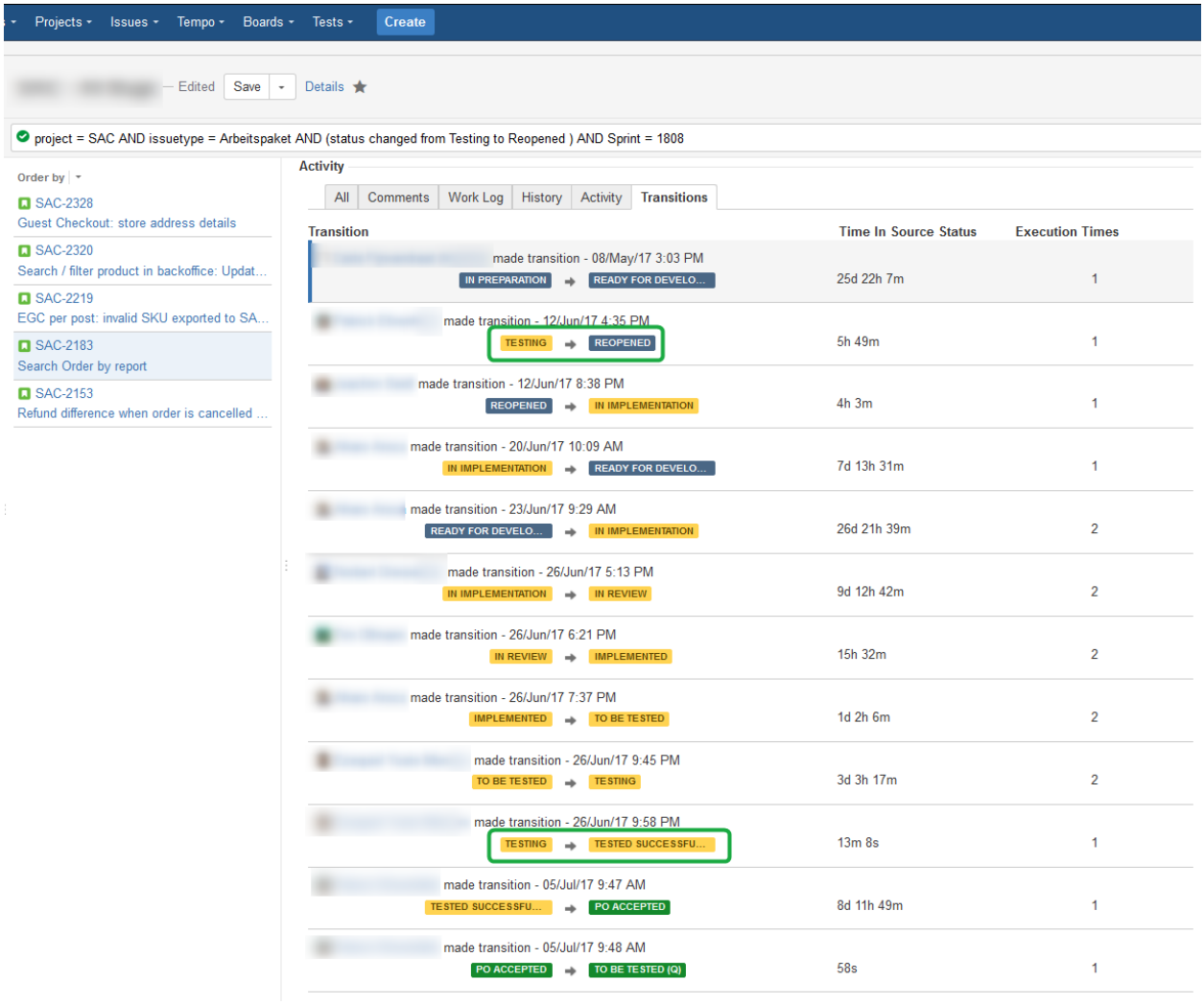


Abbildung 44 - Transitions des Tickets eines Features im Jira

8.4.2.1 Fehlerdichte

Der Einflussfaktor *Fehlerdichte* ist der zweite Einflussfaktor der Gruppe der *Quality Assurance* und beinhaltet zwei Gruppen von Richtlinien, *Feature* und *Topic*, welche nach dem *Schweregrad* in weitere Richtlinien klassifiziert wurden. Bei diesen Richtlinien geht es darum herauszufinden, ob ein Feature aus dem Release mehrere zusammenhängende Defects besitzt, welche die Funktionalität des Features beeinträchtigen könnten und somit das Risiko erhöhen einen mangelhaften Release auszuliefern.

Um dies herauszufinden benötigt man ein weiteres Mal die Liste der Features, welche in der Abbildung 10 dargestellt ist. Mit dem Tool *Jira* ist es möglich Defect-Tickets auf einfachen Weg mit Tickets von Features zu verknüpfen. Überprüft man nun die verlinkten Tickets eines Features wie in Abbildung 45, findet man sofort unter dem Punkt „*has defect*“ die verlinkten Defect-Tickets mit ihrem Schweregrad und ihrem aktuellen Status.

Das selbe Prinzip für die verknüpften Defect-Tickets von Features gilt auch für Topics. Im Unterschied zu den Features bedarf es mehr Aufwand, um die verknüpften Defect-Tickets herauszufinden.

Jedes Feature- oder Defect-Ticket im *Jira* hat unter seinen Details im Jira einen Verweis auf das Topic (im Tool *Jira* wird es *Epic* genannt) zu welchem das Feature zugehörig ist, wie in Abbildung 46 gezeigt wird. Für das Finden der Defect-Tickets, welche zu einem bestimmten Topic dazugehören, ist es nötig eine Suchabfrage durchzuführen. In diesem Fall wird nach allen Defect-Tickets gesucht, welche zum Topic „*Content Management*“ zugehörig sind und einen Status besitzen, welcher darauf hindeutet, dass der Defect noch offen und nicht gelöst oder getestet wurde. Wie in Abbildung 47 demonstriert wird, resultieren daraus 8 Defect-Tickets. Die Anzahl pro Schweregrad wird nun mit den Grenzwerten der jeweiligen Richtlinien verglichen und basierend darauf wird eine Bewertung abgegeben.

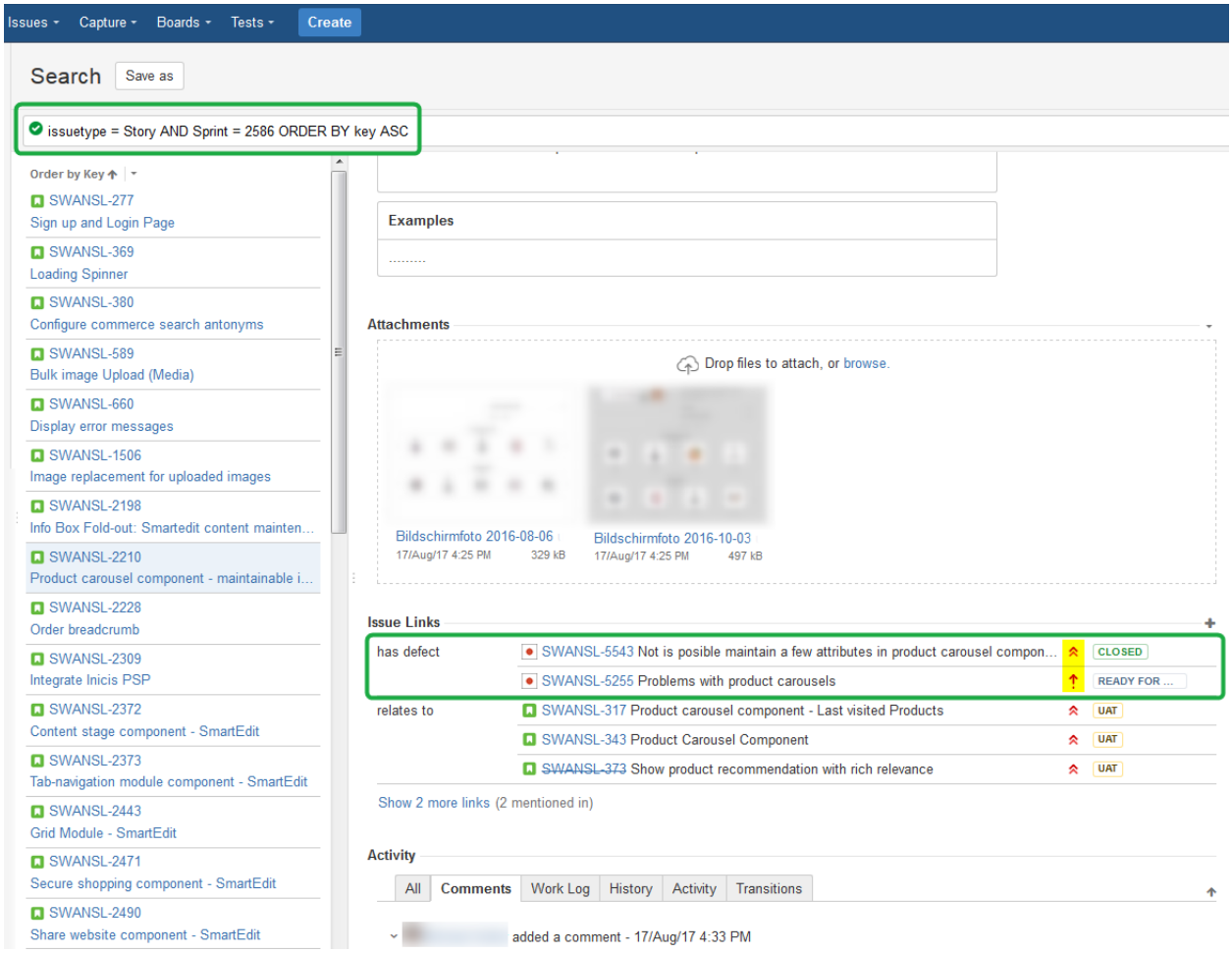


Abbildung 45 - Mit einem Feature verknüpfte Defect-Tickets im Jira

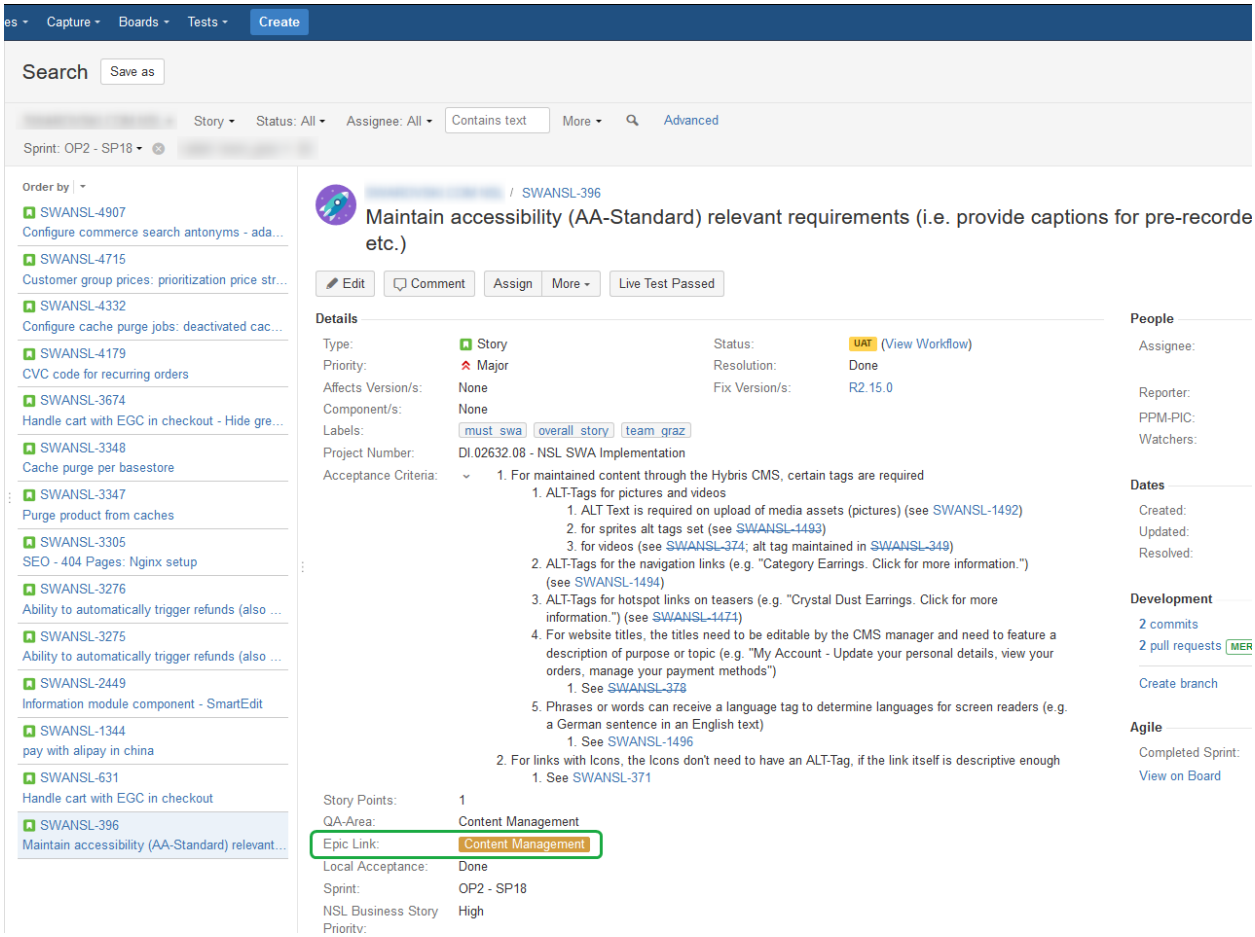


Abbildung 46 - Verweis auf das zugehörige Topic eines Feature-Tickets im Jira

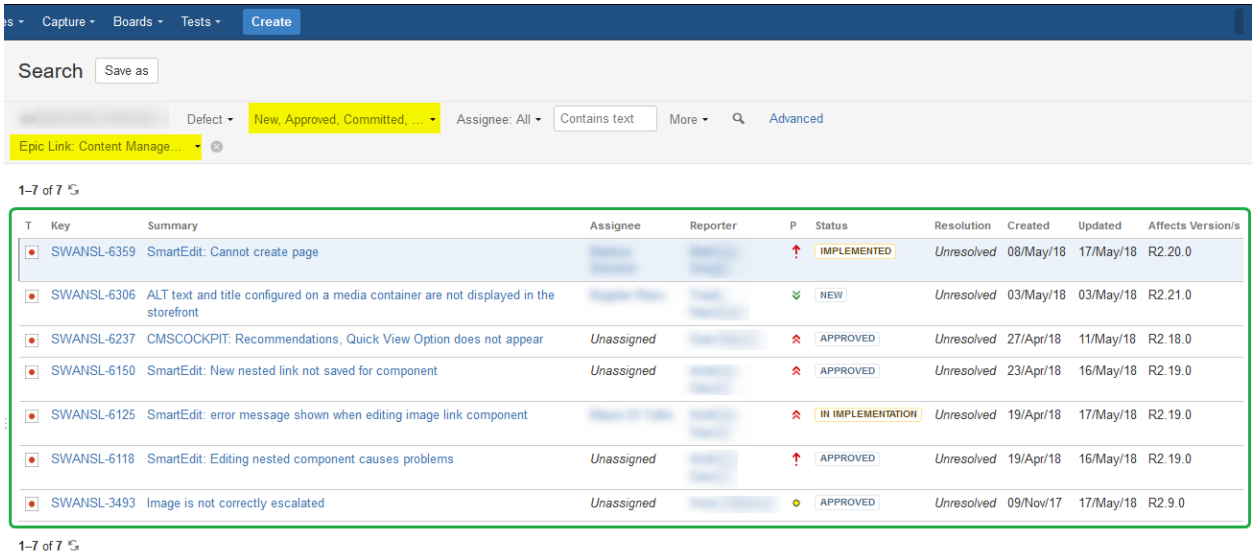


Abbildung 47 - Suchabfrage für alle zu einem Topic zugehörigen Defect-Tickets im Jira

8.4.2.2 Definition of Done (DoD)

Die *DoD* ist ein Einflussfaktor, welcher zur Gruppe des *Release Managements* zählt. Dabei geht es darum festzustellen, ob jedes Feature eines Release alle Punkte erfüllt, welche in der *DoD*

festgelegt wurde. Wie bereits im Kapitel 7.3.1 näher erläutert wurde, gibt es keine allgemeine Auslegung einer *DoD*, da sie von jedem Entwicklungsteam individuell nach ihren Bedürfnissen erstellt wird. Das aufgezeigte Beispielprojekt besitzt daher ebenfalls eine selbst definierte *DoD*, welche im Tool *Confluence* festgehalten wird. In Abbildung 48 wird diese DoD dargestellt.

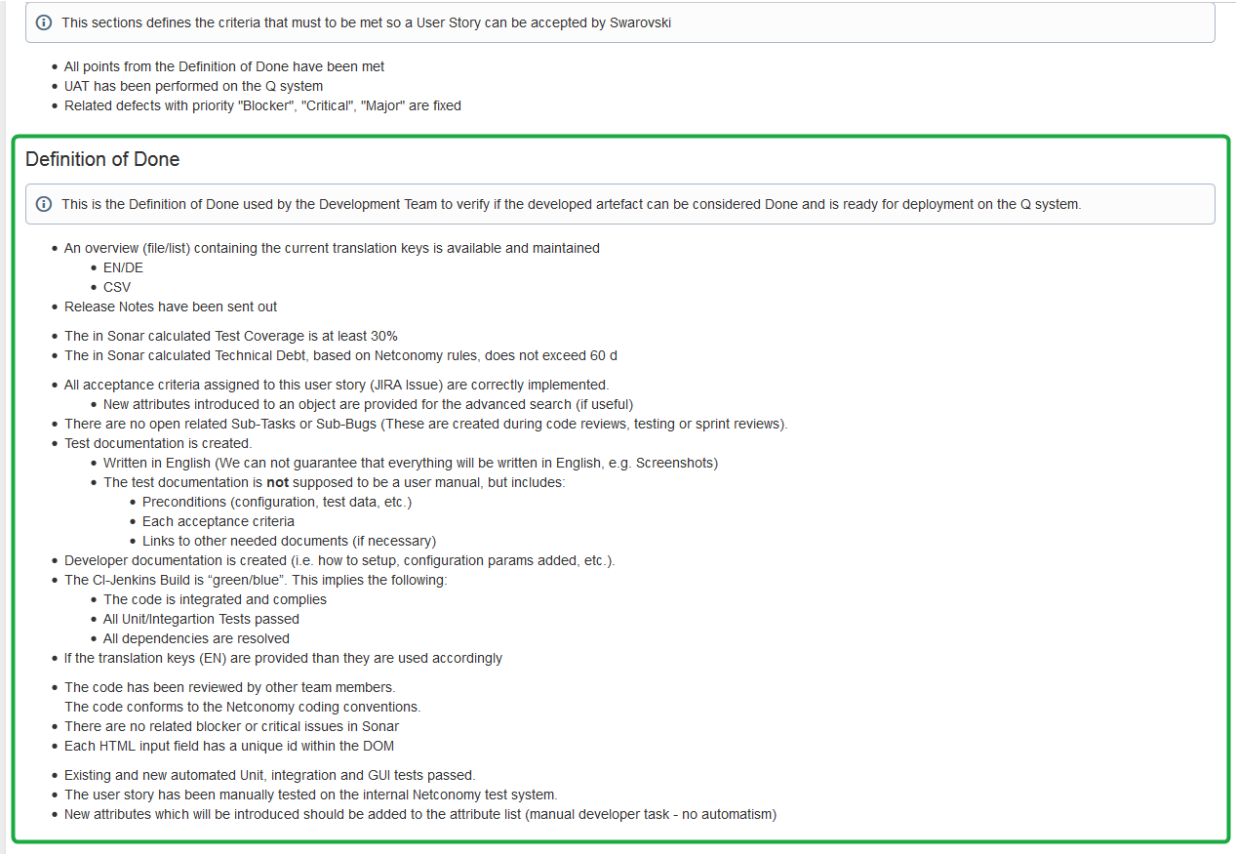


Abbildung 48 - DoD des Beispielprojekts

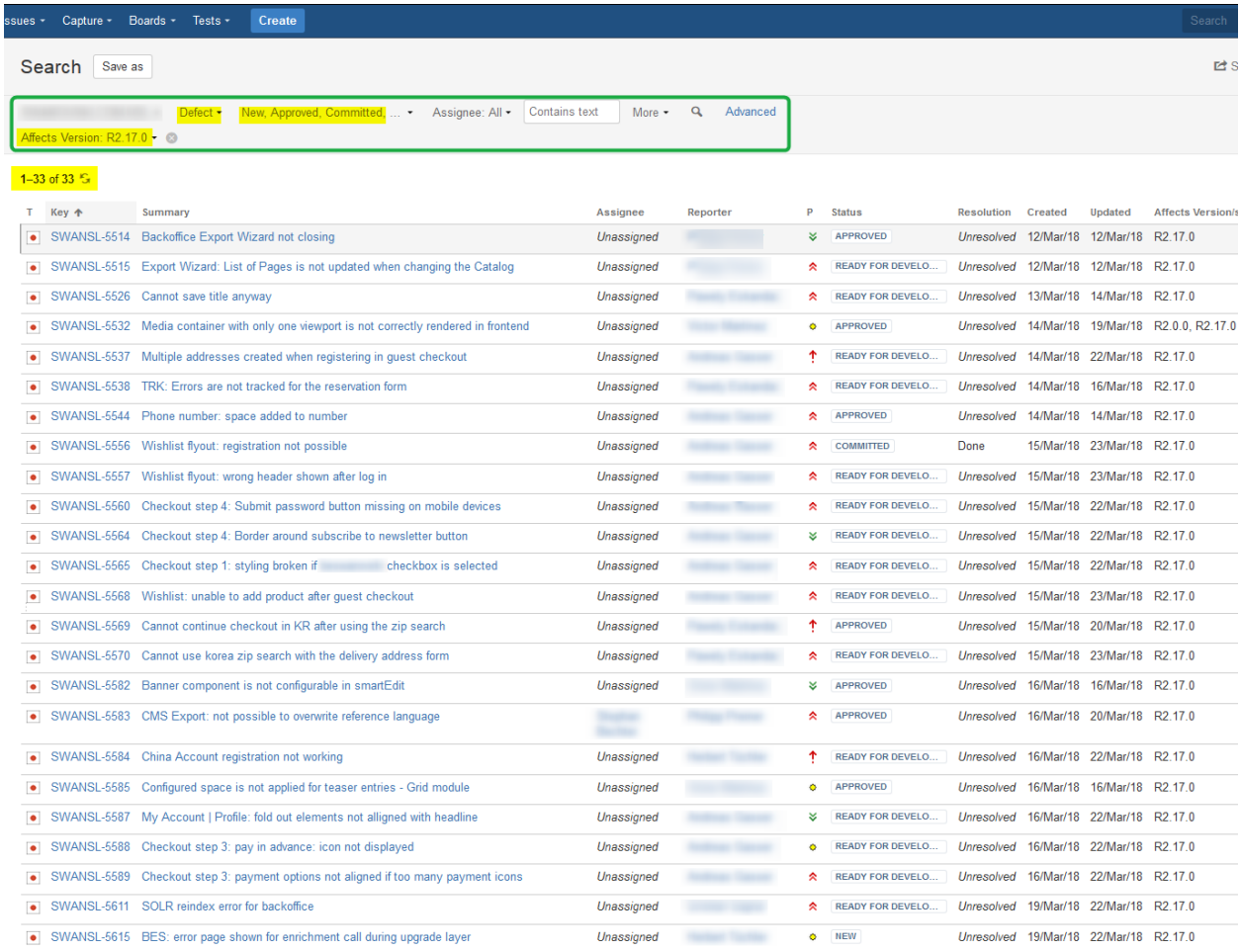
Die aufgelisteten Punkte werden nacheinander für jedes Feature durchgegangen und überprüft. Da die *DoD* beschreibt, wann ein Feature wirklich fertig (*done*) ist, muss dieser Einflussfaktor für jedes Feature zu 100 % erfüllt sein.

8.4.2.3 Anzahl der Defects per Release

Anzahl der Defects per Release ist der zweite Einflussfaktor der Gruppe des *Release Managements* und der letzte Einflussfaktor des Modells. Dieser Einflussfaktor ist in drei Gruppen aufgeteilt: eine allgemeine, den Release betreffende Gruppe und in den beiden Gruppen *Priority* und *Severity*. Wie schon die Namen der zwei letzten Gruppen darauf hindeuten, sind deren Richtlinien nach Priorität und Schweregrad klassifiziert. Im folgenden Beispiel wird jedoch die Gruppe *Priority* ausgelassen, da sie im folgenden Beispielprojekt nicht verwendet wurde.

Die erste allgemeine Gruppe ermittelt alle offenen Defects im Release und versucht einen gewissen Grenzwert nicht zu überschreiten, da ansonsten das Risiko für den Release zu hoch

wird. Diese Anzahl kann mithilfe einer Suchabfrage im *Jira* ermittelt werden. Dazu wird nach allen Defect-Tickets aus einem bestimmten Release gesucht, welche noch einen offenen Status besitzen, wie in Abbildung 49 zu sehen ist. Aus dieser Suchabfrage resultiert, dass in diesem Release 33 Defect-Tickets enthalten sind.



T	Key	Summary	Assignee	Reporter	P	Status	Resolution	Created	Updated	Affects Version/s
	SWANSL-5514	Backoffice Export Wizard not closing	Unassigned			APPROVED	Unresolved	12/Mar/18	12/Mar/18	R2.17.0
	SWANSL-5515	Export Wizard: List of Pages is not updated when changing the Catalog	Unassigned			READY FOR DEVELO...	Unresolved	12/Mar/18	12/Mar/18	R2.17.0
	SWANSL-5526	Cannot save title anyway	Unassigned			READY FOR DEVELO...	Unresolved	13/Mar/18	14/Mar/18	R2.17.0
	SWANSL-5532	Media container with only one viewport is not correctly rendered in frontend	Unassigned			APPROVED	Unresolved	14/Mar/18	19/Mar/18	R2.0.0, R2.17.0
	SWANSL-5537	Multiple addresses created when registering in guest checkout	Unassigned			READY FOR DEVELO...	Unresolved	14/Mar/18	22/Mar/18	R2.17.0
	SWANSL-5538	TRK: Errors are not tracked for the reservation form	Unassigned			READY FOR DEVELO...	Unresolved	14/Mar/18	16/Mar/18	R2.17.0
	SWANSL-5544	Phone number: space added to number	Unassigned			APPROVED	Unresolved	14/Mar/18	14/Mar/18	R2.17.0
	SWANSL-5556	Wishlist flyout: registration not possible	Unassigned			COMMITTED	Done	15/Mar/18	23/Mar/18	R2.17.0
	SWANSL-5557	Wishlist flyout: wrong header shown after log in	Unassigned			READY FOR DEVELO...	Unresolved	15/Mar/18	23/Mar/18	R2.17.0
	SWANSL-5560	Checkout step 4: Submit password button missing on mobile devices	Unassigned			READY FOR DEVELO...	Unresolved	15/Mar/18	22/Mar/18	R2.17.0
	SWANSL-5564	Checkout step 4: Border around subscribe to newsletter button	Unassigned			READY FOR DEVELO...	Unresolved	15/Mar/18	22/Mar/18	R2.17.0
	SWANSL-5565	Checkout step 1: styling broken if checkbox is selected	Unassigned			READY FOR DEVELO...	Unresolved	15/Mar/18	22/Mar/18	R2.17.0
	SWANSL-5568	Wishlist: unable to add product after guest checkout	Unassigned			READY FOR DEVELO...	Unresolved	15/Mar/18	23/Mar/18	R2.17.0
	SWANSL-5569	Cannot continue checkout in KR after using the zip search	Unassigned			APPROVED	Unresolved	15/Mar/18	20/Mar/18	R2.17.0
	SWANSL-5570	Cannot use korea zip search with the delivery address form	Unassigned			READY FOR DEVELO...	Unresolved	15/Mar/18	23/Mar/18	R2.17.0
	SWANSL-5582	Banner component is not configurable in smartEdit	Unassigned			APPROVED	Unresolved	16/Mar/18	16/Mar/18	R2.17.0
	SWANSL-5583	CMS Export: not possible to overwrite reference language				APPROVED	Unresolved	16/Mar/18	20/Mar/18	R2.17.0
	SWANSL-5584	China Account registration not working	Unassigned			READY FOR DEVELO...	Unresolved	16/Mar/18	22/Mar/18	R2.17.0
	SWANSL-5585	Configured space is not applied for teaser entries - Grid module	Unassigned			APPROVED	Unresolved	16/Mar/18	16/Mar/18	R2.17.0
	SWANSL-5587	My Account | Profile: fold out elements not alligned with headline	Unassigned			READY FOR DEVELO...	Unresolved	16/Mar/18	22/Mar/18	R2.17.0
	SWANSL-5588	Checkout step 3: pay in advance: icon not displayed	Unassigned			READY FOR DEVELO...	Unresolved	16/Mar/18	22/Mar/18	R2.17.0
	SWANSL-5589	Checkout step 3: payment options not aligned if too many payment icons	Unassigned			READY FOR DEVELO...	Unresolved	16/Mar/18	22/Mar/18	R2.17.0
	SWANSL-5611	SOLR reindex error for backoffice	Unassigned			READY FOR DEVELO...	Unresolved	19/Mar/18	22/Mar/18	R2.17.0
	SWANSL-5615	BES: error page shown for enrichment call during upgrade layer	Unassigned			NEW	Unresolved	19/Mar/18	22/Mar/18	R2.17.0

Abbildung 49 – Suchabfrage für alle offenen Defects eines Release im Jira

Die Richtlinie der ersten Gruppe dieses Einflussfaktors berücksichtigt nicht den Status dieser Defects, sondern fokussiert sich lediglich auf die Anzahl. Die Richtlinien der Gruppe *Severity* fokussieren sich auf die Anzahl der Defects eines bestimmten Schweregrads.

Um diese Anzahl herauszufinden wird bei der bestehenden Suchabfrage zusätzlich nach den Schweregraden gefiltert, wie in den Abbildungen 50 – 53 demonstriert wird. Es zeigt sich, dass keine offenen Defect-Tickets in diesem Release mit dem Schweregrad *Blocker* existieren, jedoch 4 mit dem Schweregrad *Critical*, 19 mit dem Schweregrad *Major* und 4 mit dem Schweregrad *Minor*.

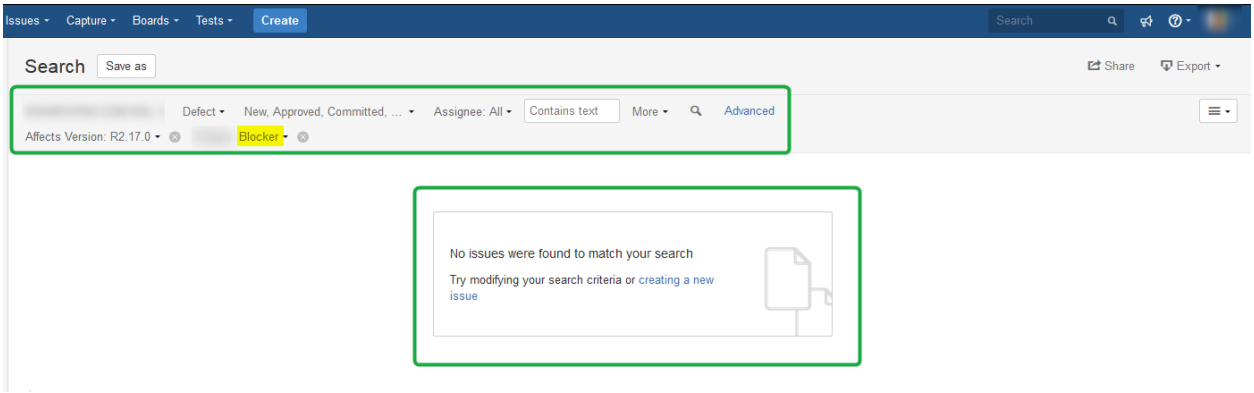


Abbildung 50 - Suchabfrage nach allen Defect-Tickets mit dem Schweregrad Blocker im Jira

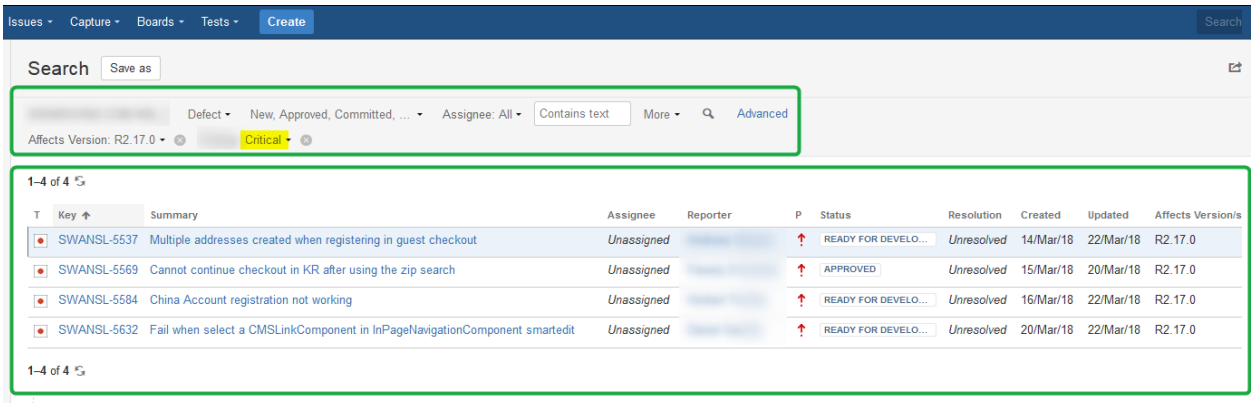


Abbildung 51 - Suchabfrage nach allen Defect-Tickets mit dem Schweregrad Critical im Jira

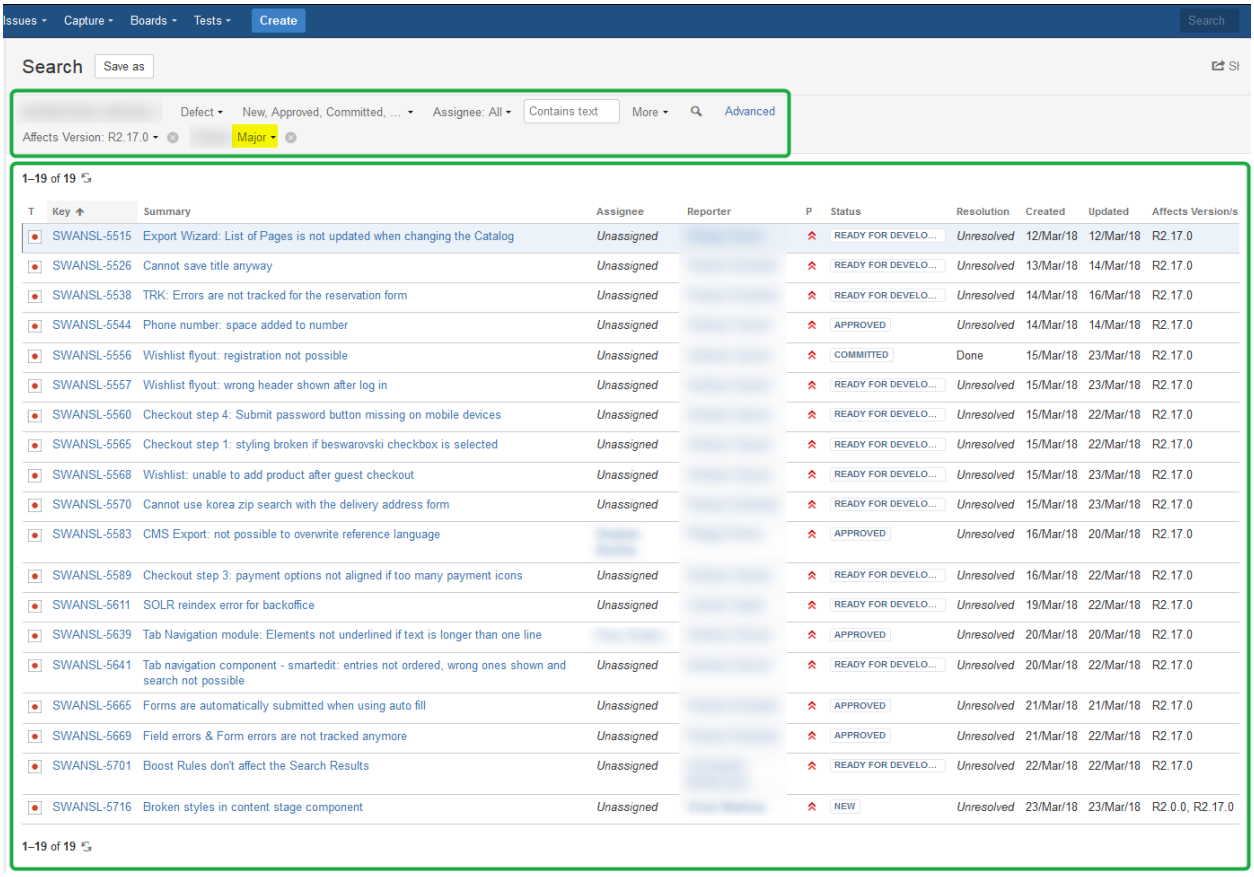


Abbildung 52 - Suchabfrage nach allen Defect-Tickets mit dem Schweregrad Major im Jira

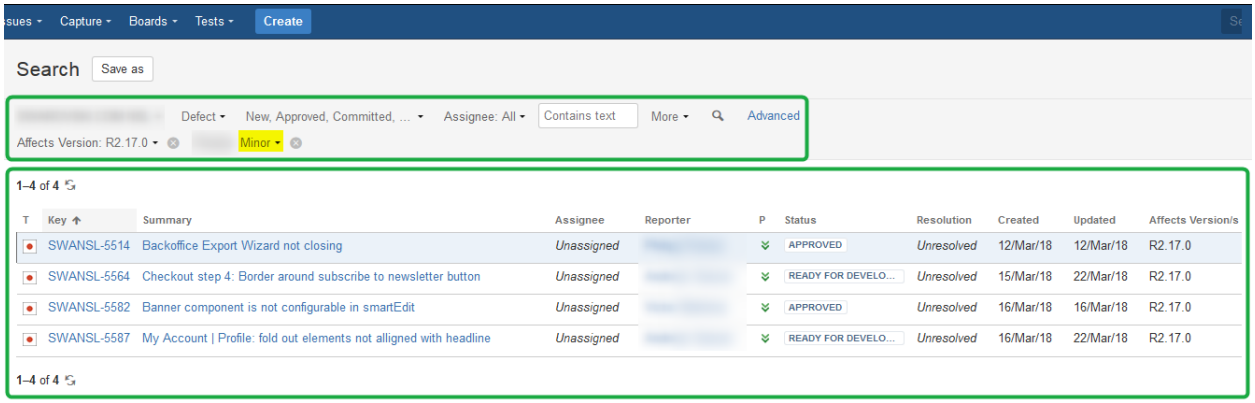


Abbildung 53 - Suchabfrage nach allen Defect-Tickets mit dem Schweregrad Minor im Jira

8.4.3 Ergebnisse

In den Beispielen zu den Einflussfaktoren des vorherigen Kapitels wurden Beispiele aus dem Release genommen, welcher in der Abbildung 10 zu sehen ist. Dabei wurden nur einzelne Features zu Demonstrationszwecken ausgewählt. Für manche Beispiele im vorigen Kapitel mussten sogar Features als Beispiele verwendet werden, welche nicht aus dem Release aus Abbildung 10 entstammen, da keines davon für die Demonstration des jeweiligen Einflussfaktors geeignet war. Dies war deshalb möglich, da die Beispiele nur der Demonstration der Ermittlung der Einflussfaktoren galten und nicht Teil des Experiments waren.

Im Anschluss an der beispielhaften Demonstration der Ermittlung der Einflussfaktoren im vorherigen Kapitel, wurde das Modell und dessen Bewertungsmatrix auf den Release aus Abbildung 10 angewandt, um eine tatsächliche Anwendung des Modells aufzuzeigen. Dieses konkrete Experiment sollte das Modell auf ihre Anwendbarkeit validieren. Die Ergebnisse dieses Experiments sind in Tabelle 4 dargestellt. Die Spalte mit den Bezeichnungen der Richtlinien wurden aus der Tabelle aufgrund des Platzmangels entfernt und kann der Tabelle 3 mithilfe der Nummerierungen aus der ersten Spalte entnommen werden.

Im Zuge des Experiments wurde jedes der 14 Features nacheinander überprüft. Dabei wurden die Ergebnisse in die Bewertungsmatrix im Excel-File eingetragen. Für die Anwendung des Modells an dem Beispiel-Release wurden Grenz- & Zielwerte definiert, welche für das Projekt sinnhaft erschienen. Im Anschluss wurden die folgenden drei zusammenhängenden Features miteinander verknüpft:

- *SWANSL-3674* ⟷ *SWANSL-1344*
- *SWANSL-3348* ⟷ *SWANSL-3347*
- *SWANSL-3275* ⟷ *SWANSL-3276*

Das tatsächliche Ergebnis der Bewertungsmatrix für jedes Feature kann der letzten Zeile der Tabelle 4 entnommen werden. Wie man daran erkennen kann, erhielten acht Features den Status *Passed* und sechs Features den Status *Failed*. Dies würde bedeuten, dass die sechs Features

mit dem Status *Failed* aus dem Release entnommen werden müssten, da sie ein hohes Risiko für die Qualität des Release darstellen.

Die Ergebnisse zeigen, dass vier Richtlinien wesentlich zum Ergebnis beigetragen haben, da sie alle jeweils drei oder vier Mal nicht erfüllt wurden. Diese vier Richtlinien entstammen den Einflussfaktoren aus allen vier Gruppen des Modells. Es sind die Richtlinien 2, 17, 22 & 25 aus der Tabelle 4. Bis auf die Richtlinie 17, hatten alle eine Gewichtung von 4 und somit große Auswirkung auf die Summen der jeweiligen Features. Die Richtlinie 4 hatte zwar keine wesentliche Auswirkung auf das Ergebnis, wurde aber dennoch zwei Mal nicht erfüllt.

Ebenfalls lässt sich aus den Ergebnissen ableiten, dass drei der sechs Features mit dem Status *Failed* diesen Status durch die drei selben Richtlinien erhalten haben (2, 22 & 25). Ein Feature erhielt diesen Status aufgrund von zwei dieser 3 Richtlinien (22 & 25). Lediglich ein Feature der Sechs (*SWANSL-3276*) erhielt den Status *Failed* durch fünf verschiedene Richtlinien (4, 8, 12, 17 & 23.a), welche sich mit den drei zuvor erwähnten Richtlinien unterscheiden.

Fünf der Features mit dem Status *Failed* haben eine relativ hohe Summe erreicht, weswegen keine Zweifel an ihrem Risiko für die Qualität des Releases besteht. Ganz besonders die zwei verknüpften Features dieser Fünf (*SWANSL-3348* & *SWANSL-3347*) zeigen aufgrund ihrer Summe nach der Verknüpfung, dass sie unbedingt aus dem Release entnommen werden müssen. Eines der Features mit dem Status *Failed* zeigt jedoch, dass es lediglich aufgrund seiner Verknüpfung mit einem zusammenhängenden Feature diesen Status erhalten hat. Dies trifft auf das Feature mit der Identifikationsnummer *SWANSL-3275* zu, welches in der ersten Summenzeile ohne Verknüpfung eine Summe von 0 aufweist. Dieses Feature Bedarf somit einer manuellen Überprüfung, bevor es zwangsläufig aus dem Release ausgenommen wird.

Zuletzt zeigen die Ergebnisse, dass nur drei von acht der Features mit dem Status *Passed* eine Summe von 0 aufwiesen. Die restlichen fünf Features hatten jedoch Summen größer als 0, wobei keines dieser fünf Features die Summe von 2 überschritt. Diese Ergebnisse werden anschließend im darauf folgenden Kapitel kritisch behandelt und es werden daraus Rückschlüsse gezogen.

Einflussfaktoren / Features	#	Grenzwert/ Zielwert	Gewichtung	SWANSL-4907	SWANSL-4715	SWANSL-4332	SWANSL-4179	SWANSL-3674	SWANSL-3348	SWANSL-3347	SWANSL-3305	SWANSL-3276	SWANSL-3275	SWANSL-2449	SWANSL-1344	SWANSL-631	SWANSL-396
Volatility of Requirements																	
	1)	> 3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1.a)	< 2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Requirements Test Case Coverage																	
	2)	< 1	4	0	0	4	0	0	4	4	0	0	0	0	0	0	0
Requirements Stability Metric																	
	3)	> 2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Code Coverage																	
	4)	< 1	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0
Code Reliability																	
	5)	> 1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	6)	> 3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	7)	> 5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	8)	> 10	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
Code Maintainability																	
	9)	> 1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	10)	> 3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0

11)	> 5	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
12)	> 10	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
Code Security																
13)	> 1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14)	> 3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15)	> 5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16)	> 10	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Cyclomatic Complexity																
17)	< 2	2	2	0	0	2	0	0	0	0	2	0	0	2	0	0
Aktuelle Qualitätsrate																
19)	< 100%	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20)	< 100%	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21)	< 100%	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22)	< 100%	4	0	0	4	0	0	4	4	4	0	0	0	0	0	0
22.a)	> 2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Fehlerdichte																
Fehlerdichte per Feature																
23.a)	> 1	4	0	0	0	0	0	0	0	0	4	0	0	0	0	0
23.b)	> 3	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23.c)	> 5	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23.d)	> 10	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Fehlerdichte per Topic/Requirement																
24.a)	> 1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24.b)	> 3	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0

24.c)	> 5	4	0	0	0	0	0	0	0	0	0	0	0	0	0
24.d)	> 10	4	0	0	0	0	0	0	0	0	0	0	0	0	0
Definition of Done (DoD)															
25)	< 100%	4	0	0	4	0	0	4	4	4	0	0	0	0	0	0
Anzahl der Defects per Release															
26)	> 20	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Anzahl der Defects per Release - Severity															
27)	> 1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28)	> 3	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29)	> 5	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30)	> 10	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Summe ohne Verknüpfungen			2	0	12	2	1	12	12	8	9	0	0	2	1	0
Summe mit Verknüpfungen			2	0	12	2	2	24	24	8	9	9	0	2	2	0
Passed/Failed			Passed	Passed	Failed	Passed	Passed	Failed	Failed	Failed	Failed	Failed	Passed	Passed	Passed	Passed

Tabelle 4 - Bewertungsmatrix der Features des Release aus dem praktischen Beispiel

9 DISKUSSION

Das durchgeführte Experiment im Kapitel 8.4 bestand aus einem praktischen Beispiel zur Anwendung des entwickelten Modells und dessen Bewertungsmatrix. Die Wahl dieses Vorgehens lag dem Zeitmangel zugrunde, weshalb ein Experiment in einem realen Projekt über die Zeit und über mehrere Iterationen hinweg nicht möglich war. Von daher prüft dieses Experiment lediglich die grundsätzliche Anwendbarkeit des Modells und dessen Bewertungsmatrix und kann zwar Interpretationen zulassen, jedoch verfügen diese über keine Vergleichsbasis und haben daher keine zuverlässige Aussagekraft oder hohen Grad an Signifikanz.

In den Ergebnissen, welche in Kapitel 8.4.3 zu finden sind, ging hervor, dass 8 der 14 Features des Release aus dem praktischen Beispiel den Status *Failed* erhielten und in Folge dessen aus dem Release entnommen werden müssen. Die verbleibenden sechs erhielten den Status *Passed* und stellen daher kein Risiko für die Qualität des Releases dar.

Das Experiment hat aufgezeigt, dass das Modell und dessen Bewertungsmatrix grundsätzlich anwendbar sind, jedoch kann die Ermittlung für manche Einflussfaktoren, besonders die der Gruppe der *Implementation Phase*, zeitaufwendig sein. Der Grund dafür liegt in der Größe des Projektes und des Sourcecodes. Je größer das Projekt und der Sourcecode, desto mehr Tickets und Sourcecode-Dateien müssen überprüft werden. Diese Schlussfolgerung basiert jedoch nur auf die im praktischen Beispiel ausgewählten und verwendeten Tools. Das Modell müsste mit verschiedenen anderen Tools überprüft werden, um schließlich über den durchschnittlichen zeitlichen Aufwand der Anwendung des Modells urteilen zu können.

Ein wesentlicher Punkt in den Ergebnissen ist, dass vier der sechs Features mit dem Status *Failed* unter anderem an der Richtlinie 22, welche sich mit dem Akzeptanztest beschäftigt, gescheitert sind. Dies hatte automatisch zur Folge, dass auch die Richtlinie 25 nicht erfüllt werden konnte, da die *Definition of Done (DoD)* einen erfolgreichen Akzeptanztest pro Feature verlangt. Gleichzeitig existiert auch ein gewisser Zusammenhang zwischen der Richtlinie 2 und der Richtlinie 22, da der Akzeptanztest eines Features unmöglich erfolgreich sein kann, wenn keine Test Cases zu diesem Feature existieren. Diese Erkenntnisse deuten auf einen gewissen Zusammenhang oder Verknüpfung mancher Einflussfaktoren hin.

Aus dieser Erkenntnis kann jedoch im Umkehrschluss abgeleitet werden, dass zwei der sechs Features mit dem Status *Failed* nicht aus dem Release entnommen worden wären, falls nur auf den Akzeptanztest dieser Features geachtet worden wäre. Dieser Rückschluss wird besonders durch das Feature *SWANSL-3276* deutlich, welches bereits im Ergebniskapitel erwähnt wurde. Dieses Feature erhielt den Status *Failed* aufgrund von fünf Richtlinien, welche allesamt in keinem Zusammenhang zum Akzeptanztest stehen. Dieses Beispiel liefert eine gewisse Bestätigung der aufgestellten Hypothese dieser Arbeit.

Eine weitere Erkenntnis aus den Ergebnissen ist der Unterschied der Summen mit und ohne Verknüpfung der zusammenhängenden Features. Würde die Summe der Features vor der Verknüpfung herangezogen werden, müssten nur fünf Features aus dem Release entnommen

werden, statt den tatsächlichen sechs. Dies trifft auf das Feature mit der Identifikationsnummer *SWANSL-3275* in der Bewertungsmatrix aus Tabelle 4 zu. Bei diesen Verknüpfungen werden jedoch keine Unterschiede in der Stärke des Zusammenhangs gemacht. Diese Tatsache erfordert daher eine manuelle Überprüfung, insbesondere bei Feature *SWANSL-3275*, welches ohne Verknüpfung eine Summe von 0 aufweist und somit gegen keine einzige Richtlinie verstoßen hat. Dies deutet darauf hin, dass die Verknüpfung zusammenhängender Features zwar essenziell ist, da Zusammenhänge dadurch berücksichtigt werden, welche ein Risiko für Qualität des Releases darstellen könnten, jedoch trotzdem manuelle Überprüfungen nötig sind, da die Stärke des Zusammenhangs von Features im Vorhinein nicht bekannt ist. Wobei diese manuelle Überprüfung nur nötig wird, falls sie einen wesentlichen Einfluss auf den Status des Features haben kann.

Im Zuge des Experiments ist ebenfalls aufgefallen, dass der Einflussfaktor DoD viele Punkte enthalten kann und aufgrund dessen vieles überprüft werden muss. Teilweise sind in der DoD Punkte aufgelistet, welche allgemein für das Entwicklungsteam gelten, jedoch nicht explizit auf jedes Feature anwendbar sind. Daher wäre es angebracht die DoD für die Anwendung des Modells dementsprechend zu filtern und dies als Punkt für die Voraussetzungen des Modells zu inkludieren.

Für das Experiment wurden die Grenz- & Zielwerte selbstständig definiert, da wie bereits in Kapitel 8.2 erklärt, diese Werte vom Projekt abhängig sind und von jedem Entwicklungsteam selbst definiert werden müssen. Im Rahmen des Experiments wurde festgestellt, dass nicht jeder Grenz- & Zielwert verhältnismäßig zu jedem Feature passend ist. Grenz- & Zielwerte, welche für verhältnismäßig kleine Features Sinn ergeben haben, waren wiederum unpassend für größere Features, welche ein höheres Risiko beherbergen. Für dieses Problem könnte man wiederum die manuelle Überprüfung im Anschluss heranziehen, um gegebenenfalls inkorrekte Bewertungen in der Bewertungsmatrix, welche durch diese Unverhältnismäßigkeiten entstanden sind, anzupassen.

Dieses Problem gilt ebenfalls für die Gewichtungen der Richtlinien der Bewertungsmatrix. Es ist in der Bewertungsmatrix nicht möglich auf Basis der Kritikalität eines Features die Gewichtung anzupassen. Beispielsweise könnten fünf *Major Vulnerabilities* für ein Feature kritischer sein als für ein anderes. Zurzeit würden jedoch beide gleichermaßen behandelt und die Gewichtung von 1 der Richtlinie 15 für die Bewertung herangezogen werden. Ebenso wäre hier die anschließende manuelle Überprüfung der Bewertungen der einzelnen Features ein möglicher Lösungsweg.

Ebenso kann das eingeführte Punktesystem der Bewertungsmatrix kritisch betrachtet werden. Dieses Punktesystem wurde ganz simpel gehalten und basiert lediglich auf die Anzahl der Gruppen der Einflussfaktoren des Modells. Es wäre aber durchaus denkbar, dass für die Bewertungsmatrix ein Punktesystem entwickelt wird, welches auf ein erprobtes Schema oder System aufbaut und gleichzeitig an der Kritikalität des Projektes angepasst ist. Eine weitere Möglichkeit ist das Punktesystem für die Bewertungsmatrix der freien Gestaltung des Entwicklungsteams zu überlassen und somit die Flexibilität und Integrierbarkeit des Modells zu steigern.

Die vorangegangenen Argumentationen lassen den Schluss zu, dass das Modell und dessen Bewertungsmatrix durchaus eine genauere Beurteilung der Qualität eines Software Release ermöglichen. Zugleich wurde die Anwendbarkeit des Modells und dessen Bewertungsmatrix anhand des praktischen Beispiels demonstriert. Jedoch ist zu bedenken, dass wie bereits zu Anfang erwähnt, die durchgeführte Evaluierung keine zuverlässigen Rückschlüsse zulassen, da das Modell weder in einem realen Umfeld geprüft, noch über die Zeit hinweg angewendet wurde und somit auch keine Gegenüberstellungen möglich waren.

10 RESÜMEE

In der vorliegenden Arbeit wurde basierend auf einer deduktiven Herangehensweise nach Einflussfaktoren aus dem *Software Development Life Cycle (SDLC)* geforscht, welche eine genauere Beurteilung der Qualität eines Software Release rund um den Akzeptanztest ermöglichen sollen. Diese Einflussfaktoren wurden in einem Modell integriert und darauf basierend eine Bewertungsmatrix zur Anwendung des Modells entwickelt. Das Modell und dessen Bewertungsmatrix wurden anschließend in Form eines praktischen Beispiels im Rahmen eines Experiments überprüft.

Die Ergebnisse dieser Untersuchung zeigen, dass die ausgewählten Einflussfaktoren durchaus eine genauere Beurteilung der Qualität eines Software Release ermöglichen und die aufgestellte Hypothese stützen. In der Diskussion der Ergebnisse hat sich herausgestellt, dass das Modell eine genauere Beurteilung ermöglicht, als wenn lediglich der Akzeptanztest für die Beurteilung herangezogen wird. Des Weiteren wird im praktischen Beispiel die Anwendbarkeit des Modells und dessen Bewertungsmatrix ausführlich demonstriert. Darüber hinaus musste festgestellt werden, dass obwohl das Modell den gesamten *SDLC* berücksichtigt, dennoch manuelle Überprüfungen nach der Anwendung dessen nötig sind. Diese Tatsache wurde anhand einiger Beispiele im vorherigen Kapitel erläutert.

Andererseits, wie schon im Diskussionskapitel dargelegt worden ist, mangelt es den Rückschlüssen der Ergebnisse, aufgrund der suboptimalen Bedingungen des Experiments, an Signifikanz. Ebenfalls wurden im Zuge des Experiments zwei Einflussfaktoren nicht verwendet, da diese vom Beispielprojekt nicht unterstützt wurden. Von daher hat die vorliegende Arbeit Potenzial für zukünftige Studien, welche die Schwächen und Nachteile des durchgeführten Experiments aufheben könnten, indem das Modell in einem realen Umfeld, mit unterschiedlichen Tools und über mehreren Iterationen hinweg angewendet wird. Des Weiteren ist es möglich das Modell mit weiteren Einflussfaktoren aus den Phasen des SDLC zu erweitern und diese anschließend ebenfalls in einem Experiment zu validieren. Es wäre auch denkbar das Modell mit weiteren Prüfkriterien, wie beispielsweise Performance, Usability oder Security, zu erweitern. Somit wird ersichtlich, dass die vorliegende Arbeit einige Möglichkeiten für ergänzende Untersuchungen anbietet.

ABKÜRZUNGSVERZEICHNIS

D

DoD

Definition of Done *46*

M

MoSCoW *26*

Q

Quality Assurance

QA *35*

R

RS

Requirements Specification *29*

S

SDLC

Software Development Life Cycle *8, 16*

U

UAT

User Acceptance Test *40*

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LITERATURVERZEICHNIS

Akula, V. (2017, Dezember 16). *Continuous Testing at the Core of Frequent and Faster Software Releases*. Retrieved Juni 20, 2018, from DZone.com: https://dzone.com/articles/continuous-testing-at-the-core-of-having-frequent

Ariola, W., & Dunlop, C. (2014). *Continuous Testing.* CreateSpace Independent Publishing Platform: USA.

Auerbach, A. (2015, August 3). *Part of the Pipeline: Why Continuous Testing Is Essential*. Retrieved Oktober 4, 2017, from TechWell - Software Conferences, Training, & Resources: https://www.techwell.com/techwell-insights/2015/08/part-pipeline-why-continuous-testing-essential

Baumgartner, M., Klonk, M., Pichler, H., Seidl, R., & Tanczos, S. (2013). *Agile Testing.* München: Hanser.

Beck, K., Beedle, M., Bennekum, v., Cockburn, A., Cunningham, W., Fowler, M., . . . Thomas, D. (2001, Februar). *Manifesto for Agile Software Development*. Retrieved September 02, 2017, from http://agilemanifesto.org/principles.html

Becker, A. (2011). Akzeptanzkriterien im klassischen und agilen Testumfeld. *Online-Themenspecial Testing 2011*, p. 4.

Berenbach, B., Paulish, D. J., Kazmeier, J., & Rudorfer, A. (2009). *Software & Systems Requirements Engineering: In Practice.* New York: McGraw-Hill Education.

Brennan, K. (2009). *A Guide to the Business Analysis Body of Knowledge.* Toronto: International Institute of Business Analysis.

Britton, C. (2017, Oktober 19). *Mastering Corporate Reputation Management in a Social Media World*. Retrieved November 14, 2017, from RockDove Solutions: https://www.rockdovesolutions.com/blog/mastering-corporate-reputation-management-in-a-social-media-world

Chatterjee, S. (2017, März 31). *Importance Of Quality Engineering In Consumer Electronics*. Retrieved November 14, 2017, from EFY Group: http://electronicsforu.com/electronics-projects/electronics-design-guides/importance-quality-engineering-consumer-electronics

Chemuturi, M. (2013). *Requirements Engineering and Management for Software Development Projects.* New York: Springer.

Chen, L. (2015). Continuous delivery: Huge benefits, but challenges too. *IEEE Software, 32*(2), pp. 50-54.

Cohn, M. (2010). *Succeeding with Agile: Software Development Using Scrum.* Boston: Pearson Education.

Craig, R. D., & Jaskiel, S. P. (2002). *Systematic Software Testing.* London: Artech House.

Crispin, L., & Gregory, J. (2009). *Agile Testing: A Practical Guide For Testers And Agile Teams.* Boston: Pearson Education.

Denert, E. (1992). *Software-Engineering: Methodische Projektabwicklung.* Heidelberg: Springer.

Dumke, R., & Zuse, H. (1994). *Theorie und Praxis der Softwaremessung.* Wiesbaden: Springer.

Dustin, E., Rashka, J., & Paul, J. (2001). *Software automatisch testen.* Heidelberg: Springer-Verlag.

Ebert, C., & Dumke, R. (1996). *Software-Metriken in der Praxis: Einflihrung und Anwendung von Software-Metriken in der industriellen Praxis.* Heidelberg: Springer.

Everett, G. D., & McLeod, R. J. (2007). *Software Testing: Testing Across the Entire Software Development Life Cycle.* New Jersey: John Wiley & Sons.

Fenton, N., & Bieman, J. (2015). *Software Metrics: A Rigorous and Practical Approach.* Florida: CRC Press.

Fowler, M. (2006, Mai 1). *Continuous Integration*. Retrieved 11 03, 2017, from martinfowler.com: https://martinfowler.com/articles/continuousIntegration.html

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code.* Massachusetts: Addison Wesley.

Fuchs, E., Fuchs, K. H., & Hauri, C. H. (2002). *Requirements-Engineering in IT effizient und verständlich.* Wiesbaden: Vieweg+Teubner Verlag.

Gärtner, M. (2012). *ATDD by Example: A Practical Guide to Acceptance Test-Driven Development.* Westford, Massachusetts: Addison-Wesley.

Gotel, O. C., & Finkelstein, A. C. (1994, April 18-22). An Analysis of the Requirements Traceability Problem. *Proceedings of IEEE International Conference on Requirements Engineering*, pp. 94–101.

Gousset, M., Hinshelwood, M., Randell, B. A., Keller, B., & Woodward, M. (2014). *Professional Application Lifecycle Management with Visual Studio® 2013.* Indianapolis: Wrox.

Graham, D., Van Veenendaal, E., Evans, I., & Black, R. (2008). *Foundations of Software Testing: ISTQB Certification.* London: Cengage Learning EMEA.

Hamill, P. (2004). *Unit test frameworks.* Sebastopol CA: O'Reilly.

Hoffmann, D. W. (2008). *Software-Qualität* (2 ed.). Heidelberg: Springer.

Hull, E., Jackson, K., & Dick, J. (2011). *Requirements Engineering.* London: Springer.

Humble, J., & Farley, D. (2011). *Continuous Delivery.* Boston: Pearson Education, Inc.

IEEE Std 610.12. (1990). *IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990).* Los Alamitos: IEEE Computer Society.

IEEE Std. 1061. (1998). *IEEE Standard for a Software Quality Metrics Methodology, Std. 1061-1998.* New York: Institute of Electrical and Electronics Engineering.

ISO/IEC 25010. (2011). *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models.* Genf: ISO/IEC.

ISO/IEC 9126-1. (2001). *Software engineering - Product quality - Part 1: Quality model.* Genf: ISO/IEC.

ISO/IEC, & IEEE. (2011). International Standard ISO/IEC/IEEE 29148: Systems and software engineering - Life cycle processes - Requirements engineering. Genf: IEEE.

Kano, N., Seraku, N., Takahashi, F., & Tsuji, S. (1984, April). Attractive Quality and Must-be Quality. *Journal of the Japanese Society for Quality Control, 14*(2), pp. 39-48.

Kaur, G., & Bahl, K. (2014, Mai). Software Reliability, Metrics, Reliability Improvement Using Agile Process. *IJISET - International Journal of Innovative Science, Engineering & Technology, 1*(3), pp. 143-147.

Krypczyk , V. (2014, Mai 20). *Softwarequalität – so misst und verbessert man Software*. Retrieved Jänner 17, 2018, from entwickler.de: https://entwickler.de/online/agile/softwarequalitaet-so-misst-und-verbessert-man-software-114867.html

Langer, A. M. (2012). *Guide to Software Development.* London: Springer.

Lewis, W. E. (2004). *Software Testing and Continuous Quality Improvement.* Boca Raton Fla.: Auerbach Publications.

Liggesmeyer, P. (2009). *Software-Qualität* (2 ed.). Heidelberg: Spektrum.

Liggesmeyer, P., Sneed, H., & Spillner, A. (1992). *Testen, Analysieren und Verifizieren von Software.* Heidelberg: Springer.

Linz, T. (2013). *Testen in Scrum-Projekten: Leitfaden für Softwarequalität in der agilen Welt.* Heidelberg: dpunkt.verlag.

Ločmelis, D. (2016, November 24). *7 software quality KPIs favorable in agile development projects*. Retrieved from TestDevLab Blog: https://www.testdevlab.com/blog/2016/11/7-software-quality-kpis-favorable-in-agile-development-projects/

Malhotra, C., & Chug, A. (2013). Agile Testing with Scrum-A Survey. *International Journal of Advanced Research in Computer Science and Software Engineering, 3*(3), 452–459.

Mark, A. (2017, Januar 10). *When should you involve QA in the Software Development Life Cycle (SDLC)?* Retrieved November 14, 2017, from Bonzai Engineering: https://blog.bonzaiengineering.com/when-should-you-involve-qa-in-the-software-development-life-cycle-sldc-52fbd4c05ad8

McCabe, T. J. (1976, Dezember). A Complexity Measure. *IEEE Transactions on software Engineering, SE-2*(4), pp. 308-320.

McConville, D. (2017, Februar 03). *Quality in the World of Software Development*. Retrieved November 14, 2017, from Seamgen: Top Web and Mobile Application Development Company: http://www.seamgen.com/blog/quality-software-development/

Minduca, A. (2014, März 7). *Quality assurance in software development: When should you start the testing process?* Retrieved November 14, 2017, from https://arthurminduca.com/2014/03/07/quality-assurance-in-software-development-when-should-you-start-the-testing-process/

Muller, T., & Friedenberg, D. (2011). *Certified tester foundation level syllabus.* Journal of International Software Testing Qualifications Board.

Myers, G. J. (1975). *Reliable Software through Composite Design.* New York: Petrocelli/Charter.

Panchal, D. (2008, September 3). *What is Definition of Done (DoD)?* Retrieved September 19, 2017, from SCRUM ALLIANCE: https://www.scrumalliance.org/community/articles/2008/september/what-is-definition-of-done-(dod)

Philipp-Edmonds, C. (2014, Dezember 5). *The Relationship between Risk and Continuous Testing: An Interview with Wayne Ariola* . Retrieved Oktober 4, 2017, from StickyMinds - Software Testing & QA Online Community: https://www.stickyminds.com/interview/relationship-between-risk-and-continuous-testing-interview-wayne-ariola

Pohl, K., & Rupp, C. (2015). *Basiswissen Requirements Engineering: Aus- und Weiterbildung nach IREB-Standard zum Certified Professional for Requirements Engineering Foundation Level.* Heidelberg: dpunkt.verlag.

Pryce, T. (2017, Februar). *Continuous Testing for Continuous Delivery: What Does It Mean in Practice?* Retrieved Oktober 4, 2017, from CA Technologies: Architecting the Modern Software Factory: https://www.ca.com/us/collateral/white-papers/continuous-testing-for-continuous-delivery--what-does-it-mean-in.html

Rady, B., & Coffin, R. (2011). *Continuous Testing with Ruby, Rails, and JavaScript.* Dallas: Pragmatic Bookshelf.

Rentrop, C. (2017, Juni 14). *Software-Qualität sicherstellen und messen*. Retrieved Jänner 17, 2018, from Dev-Insider: https://www.dev-insider.de/software-qualitaet-sicherstellen-und-messen-a-609609/

Rossberg, J. (2014). *Beginning Application Lifecycle Management.* New York: Apress.

Saleem, R. M., Qadri, S., Hassan, I. u., Bashir, R. N., & Ghafoor, Y. (2014, November 2). Testing Automation in Agile Software Development. *International Journal of Innovation and Applied Studies, 9*(2), pp. 541-546.

Schilling, W. W., & Alam, M. (2006, November 1). *Integrate static analysis into a software development process.* Retrieved Februar 09, 2018, from Embedded Systems Design: https://www.embedded.com/design/prototyping-and-development/4006735/Integrate-static-analysis-into-a-software-development-process

Schmitz, P., Bons, H., & Megen, R. (1983). *Software-Qualitätssicherung* (2 ed.). Braunschweig: Friedr. Vieweg & Sohn.

Schwaber, K., & Beedle, M. (2002). *Agile software development with Scrum.* Uper Saddle River: Prentice Hall.

Seidl, R., Bucsics, T., Gwihs, S., & Baumgartner, M. (2015). *Basiswissen Testautomatisierung: Konzepte, Methoden und Techniken.* Heidelberg: dpunkt.verlag.

Spillner, A., & Linz, T. (2012). *Basiswissen Softwaretest.* Heidelberg: dpunkt.verlag.

Thaller, G. E. (1994). *Verifikation und Validation.* Braunschweig: Friedr. Vieweg & Sohn.

Tian, J. (2005). *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement.* New Jersey: John Wiley & Sons, Inc.

Wagner, S. (2013). *Software Product Quality Control.* Heidelberg: Springer.

Watkins, J. (2009). *Agile Testing.* Cambridge: Cambridge University Press.

Wiegers, K., & Beatty, J. (2013). *Software Requirements.* Washington: Microsoft Press.

Winter, M., Ekssir-Monfared, M., Sneed, H. M., Seidl, R., & Borner, L. (2012). *Der Integrationstest.* München: Hanser.

Wolff, E. (2016). *Continuous Delivery: Der pragmatische Einstieg.* Heidelberg: dpunkt.

Young, R. R. (2004). *The Requirements Engineering Handbook.* Norwood: Artech House.