Kennzahlenbasiertes Qualitätsmanagement von Softwareplattformen im Testprozess

Dissertation

zur Erlangung des Doktorgrades Dr. rer. pol.

der Fakultät für Mathematik und Wirtschaftswissenschaften

der Universität Ulm

vorgelegt von Markus Prechtel

aus Heidenheim an der Brenz

Titel der Arbeit:	Kennzahlenbasiertes Qualitätsmanagement von Softwareplattformen im Testprozess
Verfasser:	Markus Prechtel
Amtierender Dekan:	Prof. Dr. Werner Kratz
1. Gutachter:	Prof. Dr. Franz Schweiggert
2. Gutachter:	Prof. Dr. Dieter Beschorner
3. Gutachter:	Prof. Dr. Dimitris Karagiannis
Tag der Promotion:	20.10.2009

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

In der Softwareentwicklung lässt sich der Trend beobachten, dass Softwareplattformen immer öfter als Mittel eingesetzt werden, um flexibel auf sich ändernde Kundenwünsche reagieren zu können. In einer Plattform werden für eine Reihe von im voraus noch nicht zwangsläufig bekannten Anwendungen unter anderem Funktionalitäten, Teilfunktionalitäten und Komponenten zusammengefasst. Im Folgenden werden kurz die Vor- und Nachteile der Verwendung einer Softwareplattform skizziert. Die Vorteile liegen darin begründet, dass verschiedene Anwendungen auf derselben Technologiebasis entstehen können und durch eine hohe Wiederverwendungsrate einiger Plattformbestandteile die Effizienz und die Geschwindigkeit der Entwicklung gesteigert werden können. Verglichen mit der direkten Anwendungsentwicklung erfordert die vorangestellte Entwicklung der Plattform zwar einen erhöhten Anfangsinvest und die ersten auf ihr basierenden Anwendungen werden den Markt später erreichen. Auf Basis einer bestehenden und gut funktionierenden Plattform ergeben sich bereits nach kurzer Zeit durch den Plattformansatz oft klare Vorteile. So wird etwa durch den Plattformeinsatz die Systemlandschaft in den Rechenzentren weiter konsolidiert, wodurch der Betrieb der Anwendungen vereinfacht wird. [Joh05, S. 15]

Die Entwicklung der Softwareplattform indes sieht sich Unterschieden zur Entwicklung kompletter Anwendungen konfrontiert, wodurch sich Nachteile gegenüber der direkten Anwendungsentwicklung ergeben können. So betreffen Entscheidungen bzgl. des Liefertermins, des Lieferumfangs oder der Qualität der Plattform stets eine Mehrzahl an Anwendungen. Beim Management der Plattformentwicklung muss somit beachtet werden, dass mittelbar auch die Entwicklung mehrerer Anwendungen von derartigen Entscheidungen betroffen sind. Dadurch erhält das Entwicklungs- und auch das Qualitätsmanagement von Softwareplattformen eine besondere Relevanz und muss umsichtiger behandelt werden.

Das Management der Entwicklung von Softwareplattformen gliedert sich dabei wie üblich in die folgenden Bestandteile [Kos76, S. 56 ff.][Ste78, S. 107 ff.][BP06a, S. 329 ff.]:

- Planung,
- Steuerung und
- Kontrolle.

Die Planung beinhaltet u. a. die Festlegung des Lieferumfangs, des Liefertermins und der Aufteilung der Entwicklung bzw. Weiterentwicklung in verschiedene Versionen. Die Steuerung betreut die Entwicklung und die Kontrolle schließlich umfasst u. a. den Test der fertigen Software, der Spezifikationen etc. [1]

Um von den oben genannten Vorteilen, der höheren Effizienz und Geschwindigkeit, profitieren zu können, muss die Planung der Softwareentwicklung die Unterschiede zur Entwicklung von Anwendungen berücksichtigen. Eine Möglichkeit hierzu besteht in der Auswertung der Daten, die durch die Kontrolle bereitgestellt werden. Dies kann beispielsweise anhand eines Testmetriksystems erfolgen. Hierunter versteht man ein kennzahlenbasiertes Managementwerkzeug, in dessen Fokus die Tests des Softwaresystems stehen. Diese quantitative Auswertung der Ergebnisse und sonstiger Artefakte der dynamischen Qualitätssicherung bildet oft die Grundlage für die Planung und Steuerung des Testprozesses. [SRWL06, S. 217]

1.1. Problemstellung

Der Ausgangspunkt der vorliegenden Arbeit war ein Problem in der Praxis der Softwareplattformentwicklung, das auch konzeptionell noch Hürden darstellt. In den folgenden Abschnitten wird näher auf die folgenden drei Sachverhalte eingegangen:

- Die Bewertung der Qualität der Softwareplattform, unter anderem um eine Auslieferentscheidung fundierter treffen zu können, erlangt durch die bei der Software-

[1] In diesem Umfeld werden sehr oft Kennzahlen eingesetzt. Diese sind meist betriebswirtschaftlicher Natur und dem Controlling zuzuordnen. Sie können dabei auch Teil einer sog. *Balanced Scorecard* sein. Die *Balanced Scorecard* ist ein Instrument, das verschiedene Kennzahlen zusammenfasst und zu einer monetären Spitzenkennzahl aggregiert. [HP07] Dies steht jedoch nicht im Mittelpunkt dieser Arbeit und wird nicht weiter verfolgt.

plattformentwicklung schwerwiegende Abwägung von Qualität und Liefertreue eine besondere Relevanz.

- Im Rahmen der Kontrolle, um die Qualität sicherzustellen, müssen Tests absolviert werden. Diese sind für das Management der Softwareplattform von hoher Bedeutung, gestalten sich durch den Plattformcharakter an mehreren Stellen jedoch schwierig.
- Um die Ergebnisse der Testdurchführung auch bereits während des Testprozesses für das Management der Softwareplattform zusammenzufassen, werden in vielen Fällen Testmetrikmodelle verwendet. Diese sind jedoch ebenfalls nicht an den Plattformcharakter angepasst.

1.1.1. Qualität von Softwareplattformen

Bei Softwareplattformen ergibt sich die Notwendigkeit, intensiver auf die Qualität zu achten als bei Anwendungen, da die Folgen schlechter Qualität weitreichender sind. Dies liegt darin begründet, dass auf Basis einer Plattform mehrere Anwendungen erstellt werden. Qualitative Mängel in der Plattform werden an mehrere, eventuell alle, Anwendungen weitergegeben. Von hoher Bedeutung ist das Qualitätskriterium der Stabilität, da auf Basis einer instabilen Plattform keine stabile Anwendung erstellt werden kann. [JWBH01]

Qualität ist immer als eine durch die Planung festzulegende Größe zu betrachten. Es besteht stets die Möglichkeit, Liefertermine oder Budgets in Abhängigkeit von Qualitätseinbußen einzuhalten. [SRWL06, S. 43] Diese Mittel werden insbesondere bei der Softwareplattformentwicklung oftmals eingesetzt, da diese einem erhöhten Termindruck ausgesetzt ist. Begründet wird dies durch die nachgelagerte Entwicklung mehrerer Anwendungen, da sich der Termindruck auf die einzelnen Anwendungen bei der Betrachtung der Plattform addiert. Beispielsweise kann es zu Vertragsstrafen kommen, falls eine Anwendung A nicht zum zugesagten Zeitpunkt ausgeliefert wird. Eine verspätete Einsatzbereitschaft einer Anwendung B kann die Produktion von Gütern verzögern oder verschieben, wodurch ebenfalls Kosten anfallen. Falls beide Anwendungen auf derselben Plattform basieren, würde eine Verzögerung der Auslieferung der Plattform sowohl die Vertragsstrafen als auch die durch den Produktionsausfall verursachten Kosten zur Folge haben.

Die Qualität der Plattform ist einerseits wichtig, andererseits kann es durchaus sinnvoll sein, Qualitätseinbußen zu akzeptieren, um einen Liefertermin einzuhalten. Eine fundierte Abwägung dieser Sachlage ist im Einzelfall unabdingbar und muss durch die ständige Kontrolle der Entwicklungsaktivitäten unterstützt werden.

1.1.2. Tests von Softwareplattformen

Eine Maßnahme zur Bewertung der Qualität liefert der Test der Software. Anhand von Testmetriken, wie beispielsweise der Testüberdeckung[2], kann eine qualitative Bewertung des Gesamtsystems vorgenommen werden.

Durch die oben beschriebene Abwägung der Qualität in vielen Fällen zu Gunsten der Termintreue verkürzt sich der Testzeitraum im Vergleich zum ursprünglich geplanten Umfang. Dadurch können zum Teil nicht alle Tests durchgeführt werden. Als Folge davon ist es wichtig, zu jeder Zeit innerhalb des Testzeitraums Aussagen über die bereits durchgeführten Tests, die Möglichkeit und die Risiken einer sofortigen Auslieferung und die Bewertung der noch offenen Fehler treffen zu können.[3]

Das Testen von Softwareplattformen findet nicht nur durch die oben beschriebenen Umstände in einem schwierigen Umfeld statt. Zusätzlich ist der Fokus und die Bewertung der Tests zum Teil anders gelagert. Der Fokus von Lasttests liegt bei Anwendungen in der Erfüllung von nichtfunktionalen Anforderungen (*non-functional requirements* – NFRs). Sie finden in einer der Produktionsumgebung identischen Umgebung statt, um eine Evaluierung dieser NFRs zu gewährleisten. Dies ist bei Plattformtests nicht möglich, da nicht nur eine Produktionsumgebung existiert, sondern für alle auf der Plattform zu entwickelnden Anwendungen eine. Dadurch können Lasttests nicht in derselben Weise wie bei Anwendungstests bewertet werden. Es ergibt sich ein gezwungenermaßen unterschiedlicher Fokus der Tests nicht auf absolute Messwerte, beispielsweise der Antwortzeit auf Anfragen an das

[2] Die Testüberdeckung gibt die Vollständigkeit der Tests an. Es gibt mehrere verschiedene Testüberdeckungsmaße, beispielsweise die *Statement*-Überdeckung, die durch den Quotienten aller in Tests ausgeführten Statements des Quelltexts und aller im Quellcode enthaltenen Statements errechnet wird. [FN99]

[3] Dies schneidet bereits den Bereich des Risikomanagements an, der in dieser Arbeit nicht weiter behandelt werden soll. Eine kurze Einführung aus dem Blickwinkel des Softwaretests befindet sich in [SRWL06].

System, sondern etwa auf die Skalierbarkeit und die Stabilität der Plattform und Vergleiche mit Vorgängerversionen der Plattform.

1.1.3. Testmetrikmodelle für Softwareplattformen

Eine Möglichkeit, um den Teststatus während der Testphase zu bewerten, liefern Testmetriken. Anhand von Testmetrikmodellen werden Kennzahlen erarbeitet, umgesetzt und ausgewertet, wodurch bereits frühzeitig Aussagen über die Qualität der Software gemacht werden können. [Sne03]

Speziell auf Softwareplattformen angepasste Modelle existieren bislang nicht. Bestehende Modelle können generell auch auf Softwareplattformen angewendet werden, da es für das Modell primär nicht entscheidend ist, welche Art von Software untersucht werden soll. Es ist jedoch möglich, bestehende Modelle und Verfahren anzupassen und somit für den Einsatz bei Softwareplattformen zu verbessern. Diese Anpassungen liegen in vielen Fällen nicht im Modell selbst, sondern am Kontext, unter dem es angewendet werden soll und wie die Datengrundlage beschaffen ist.

Viele Kennzahlen beziehen sich auf die Auswertung von gefundenen Fehlern. Diese werden mit den Anforderungen verknüpft, um dadurch erkennen zu können, welche Anforderungen (noch) nicht erfüllt sind. Ein solches Vorgehen ist auch bei Plattformen sinnvoll. Im Detail hängt die Art und Weise, wie eine solche Verknüpfung zustande kommt, sehr stark vom Entwicklungsmodell ab. Nicht nur für Plattformen eignet sich ein featurebasiertes Entwicklungsmodell, da für die Anwender die Features im Mittelpunkt stehen [Pal02; KCH+90]. Diese Modelle bieten jedoch noch keine Integration des Plattformbegriffes. Auch Anpassungen der Fehlerattribute an Plattformcharakteristiken, um den Einsatz aussagekräftigerer Kennzahlen zu ermöglichen, wurden in der Literatur bislang noch nicht behandelt.

1.2. Zielsetzung

Die Ziele der Arbeit sind, einen Prozess zu definieren, um ein Testmetriksystem zu erstellen und diesen anschließend umzusetzen. Dieses Testmetriksystem soll die Auslieferentscheidung und die Bewertung der Qualität während des Testprozesses erleichtern. Dadurch sollen die drei Problembereiche behandelt werden. Im Mittelpunkt steht hierbei das Testmetriksystem, anhand dessen zum einen die Bewertung der Qualität der Softwareplattformen erleichtert werden soll. Zum anderen soll es durch einen Informationsgewinn bei der Durchführung der Tests selbst von Nutzen sein. In der Literatur gibt es bislang kein speziell auf Softwareplattformen angepasstes Verfahren, um dieses Ziel zu erreichen.

Allgemeine Verfahren können zwar angewandt, an einigen Punkten durch ein spezialisierteres Vorgehen jedoch verbessert werden. Durch eine Anpassung von bestehenden Verfahren und des situativen Kontexts des Testens der Softwareplattformen wird den Besonderheiten der Softwareplattformentwicklung gegenüber der Anwendungsentwicklung begegnet.

1.3. Vorgehen und Aufbau der Arbeit

In dieser Arbeit werden speziell auf die Besonderheiten von Plattformen abgestimmte Testmetriken erarbeitet mit dem Ziel, das Qualitätsmanagement zu unterstützen und Entscheidungshilfen für die Freigabe der Plattform zu erhalten. Diese Testmetriken werden in einer Kennzahlsuite zusammengefasst und exemplarisch an einem Praxisbeispiel umgesetzt. Auch der Auswahlprozess, der über die Identifizierung der relevantesten Qualitätsattribute zu den Testmetriken führt, stellt einen zentralen Teil dar.

Der im Folgenden beschriebene Aufbau dieser Arbeit ist auch in Abb. 1.1 auf S. 10 dargestellt.

Kapitel 2 stellt die Grundlagen bereit, die für diese Arbeit relevant sind. Um zielgerichtet auf Softwareplattformen eingehen zu können, muss zunächst in Abschnitt 2.1 eine kurze Darstellung von Softwarekategorien und die Einordnung von Softwareplattformen darin gegeben werden. Abschnitt 2.2 beleuchtet die Zusammenhänge von Anforderungen

und Features und die Einordnung in einem Plattformentwickungsmodell. In Abschnitt 2.3 wird anschließend auf grundlegende Begriffe zur Softwarequalität eingegangen, bevor in 2.4 neben dem generellen Verständnis von Qualitätsmanagement, Normen und Standardverfahren dieses Bereichs erläutert werden. Die bisherigen Grundlagen werden daraufhin kombiniert durch die Vorstellung einer Untersuchung der Qualitätsattribute bei Softwareplattformen. Um auf diesen aufbauend eine Kennzahlsuite zur Unterstützung des Qualitätsmanagements zu errichten, wird in Abschnitt 2.5 auf die dynamische Qualitätssicherung, den Softwaretest, mit Fokus auf den Systemtest und im Speziellen auf Softwareplattformen, eingegangen. Der letzte Abschnitt dieses Kapitels, Abschnitt 2.6, gibt schließlich einen Überblick über Kennzahlen, Softwaremetriken und -maße.

Kapitel 3 fasst die Ausgangslage dieser Arbeit zusammen. Abschnitt 3.1 hebt die Besonderheiten der Qualitätsbewertung und des Qualitätsmanagements bei Softwareplattformen hervor. In Abschnitt 3.2 lassen sich der Stand der Technik und Wissenschaft zusammenfassen und alternative Ansätze diskutieren. Dies ermöglicht auch die Abgrenzung des eigenen Ansatzes, die ebenfalls in diesem Rahmen erfolgt. Gesondert dargestellt werden anschließend einige Beispiele für Kennzahlen in Abschnitt 3.2.5. Auf dieser Basis erfolgt in 3.3 die Zusammenfassung der Erkenntnisse.

Das beschrittene Vorgehen, dargelegt in Kapitel 4, sieht zunächst die Konzeption des Prozesses, der zur Erstellung und Anwendung von Testmetriken führt, vor, welcher in einem zweiten Schritt im Rahmen einer Einzelfallstudie realisiert wurde.

Der Prozess gliedert sich grob in drei Teile: Identifizierung von für Plattformen relevanten Qualitätsattributen, Entwicklung und Anpassung von Kennzahlen zu den jeweiligen Qualitätsattributen und Auswahl der umzusetzenden Kennzahlen.

Nach einem Überblick über den Auswahlprozess in Abschnitt 4.1 ist eine Anpassung des situativen Kontexts in Abschnitt 4.2 nötig, um den zu entwickelnden Kennzahlen die nötige Datengrundlage zu schaffen. Dies geschieht durch die Erstellung des Plattform-Feature-Modells, anhand dessen die featurebasierte Entwicklung den Plattformbegriff integriert. Diese Erweiterung des herkömmlichen Featuremodells liefert die Grundlage für die Rückverfolgbarkeit von Fehlern bis hin zu den Features und den Plattformen, auf die sie sich auswirken. Das featurebasierte Testen wird in Abschnitt 4.2.1 beschrieben. Durch die

Anpassung von bestehenden und die Definition von neuen Fehlerattributen und die Modifikation der Darstellung der Softwaretests im Vergleich zum jeweils üblichen Vorgehen wird in den Abschnitten 4.2.2 und 4.2.3 der Rahmen komplettiert. Als Standardvorgehen werden in der Arbeit die gängigen Normen zum Testprozess BS 7925 [BS798a], zur Klassifikation von Softwareanomalien IEEE 1044 [IEE93] und weitere Erkenntnisstände der Wissenschaft herangezogen.

Anschließend werden die Zusammenhänge von Qualitätsattributen, Test- und Qualitätsanforderungen vor diesem Hintergrund in 4.3 dargelegt. Die Priorisierung der Qualitätsattribute erfolgt in Abschnitt 4.4. Sie basiert auf einer Studie von Johansson [JWBH01] und Expertengesprächen im Umfeld des Plattformentwicklungsprojekts, in dessen Rahmen auch die spätere Fallstudie umgesetzt wurde.

Johansson untersuchte in seiner Studie [JWBH01] die einzelnen Qualitätsattribute der ISO 9126 [ISO01b]. Zwei Aspekte standen im Mittelpunkt: Erstens wurde festgestellt, dass es einen Unterschied gibt, wie verschiedene Interessengruppen die Qualitätsanforderungen bei der Entwicklung einer Softwareplattform bewerten. Der zweite Aspekt der Studie analysierte, ob es einen Unterschied gibt, welche Qualitätseigenschaften die verschiedenen Interessengruppen an einer Softwareplattform bevorzugen, wenn sie darauf eine Anwendung entwickeln sollen.

Unter Verwendung von Johanssons Datenbasis [JWBH01] wird in der eigenen Arbeit eine Priorisierung von Qualitätsattributen bezogen auf ihre Relevanz für Softwareplattformen erstellt. Dabei fließen auch Erkenntnisse aus Expertengesprächen im Umfeld der Softwareplattformentwicklung der späteren Fallstudie mit ein. Es entsteht eine Rangliste von Qualitätsattributen, geordnet nach ihrer Relevanz für Softwareplattformen.

Auf dieser Basis kann anschließend in Abschnitt 4.5 das zielgerichtete Auswählen von Kennzahlen behandelt werden. Exemplarisch werden in Abschnitt 4.6 einige Kennzahlen aufgelistet, erstellt und zum Beispiel durch Hinzunahme weiterer Eingangsparameter an Softwareplattformcharakteristiken angepasst. Als Grundlage wird überwiegend auf gängige Kennzahlen aus der Literatur zurückgegriffen. Die Auswahl, Anpassung und zum Teil Erstellung der Kennzahlen erfolgte zielgerichtet anhand der Priorisierung der Qualitätsattribute, wodurch sichergestellt wurde, dass den relevantesten Aspekten von Softwareplatt-

formen am meisten Beachtung geschenkt wird.

Die Validierung des Prozesses in der Praxis anhand der „Proaktiven Infrastruktur“ (PAI)[4] der Daimler AG[5] ist in Kapitel 5 beschrieben. Hier wird zunächst auf das Plattformentwicklungsprojekt PAI in Abschnitt 5.1 eingegangen. Die vorgefundenen und speziell angepassten Rahmenbedingungen und beispielsweise der spezifische Fehlerlebenszyklus werden anschließend erläutert. Der Abschnitt 5.3 stellt schließlich die Umsetzung der Kennzahlen und die Auswertung der Ergebnisse vor.

In Kapitel 6 werden abschließend Schlussfolgerungen gezogen sowie eine Zusammenfassung und ein Ausblick gegeben.

[4] PAI ist ein Projekt der DaimlerChrysler AG bzw. der Daimler AG, in dessen Rahmen eine Reihe von Middlewareplattformen entwickelt werden.

[5] Während der Umsetzung der Fallstudie entstand durch die Trennung von Daimler und Chrysler aus der DaimlerChrysler AG die Daimler AG und die Chrysler LLC. Im weiteren Verlauf dieser Arbeit wird ausschließlich die Firmenbezeichnung Daimler AG verwendet, auch wenn sich Teile des Textes ebenso auf die DaimlerChrysler AG beziehen.

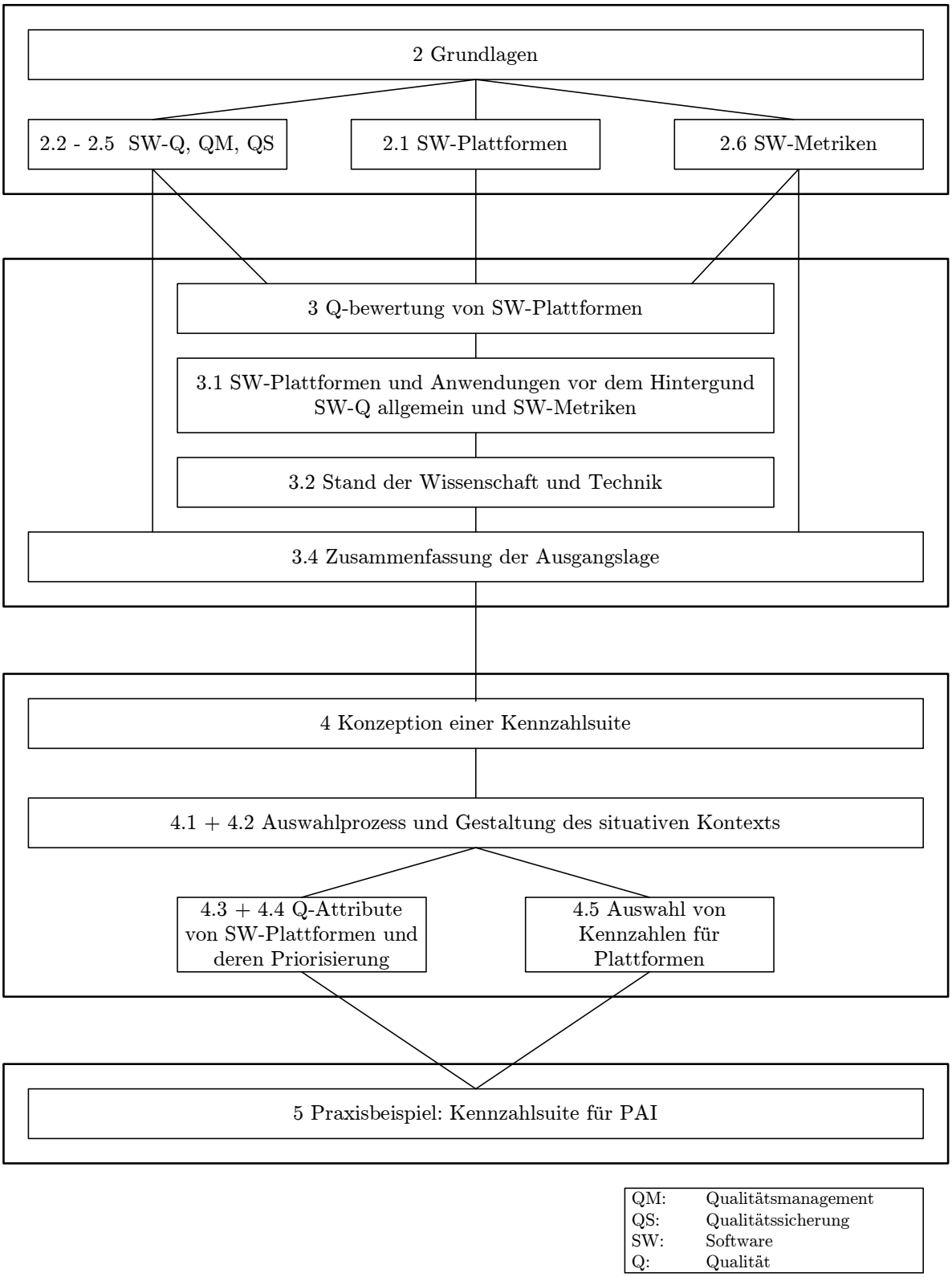

Abbildung 1.1.: Aufbau der Arbeit.

2. Grundlagen

In diesem Kapitel sollen allgemeine, grundlegende Begriffe aus verschiedenen Bereichen, die diese Arbeit betreffen, erläutert werden. Dadurch wird zum einen eine Basis geschaffen, auf deren Grundlage in den späteren Kapiteln das Testmetriksystem zur Unterstützung des Qualitätsmanagements erarbeitet werden kann. Des Weiteren wird der in diesem Abschnitt vorgestellte Stand der Wissenschaft und Technik in Kapitel 3.2 zusammengefasst und der eigene Ansatz davon abgegrenzt.

Zunächst wird der Begriff „Softwareplattform“ vor dem Hintergrund einiger anderer Softwarekategorien herausgearbeitet. Abschließend wird durch eine übergeordnete Einordnung der Begriffe verdeutlicht, in welchem Umfeld Softwareplattformen zum Einsatz gelangen und von welchem Standpunkt aus deren Charakteristiken zu bewerten sind.

Der zweite Abschnitt liefert einen kurzen Einblick in das Anforderungsmanagement, das wesentlichen Einfluss auf das Qualitätsmanagement hat. Im Vordergrund steht hierbei nicht die allgemeine Einführung in diesen Themenbereich, sondern vielmehr die Darstellung der Besonderheiten bei der Entwicklung von Plattformen. So wird ein bei Softwareplattformen zum Einsatz kommendes Modell, das Plattform-Feature-Modell, kurz erläutert und abschließend die besondere Bedeutung von nichtfunktionalen Anforderungen dargelegt.

Die Qualität von Software, deren Management, Sicherung und Bewertung, spielen eine zentrale Rolle in dieser Arbeit. Aus diesem Grund wird in Abschnitt 2.3 zunächst der allgemeine Qualitätsbegriff dargestellt und darauf aufbauend der Umgang mit Qualität in 2.4 erläutert werden. Um dem Ziel dieser Arbeit, das Qualitätsmanagement durch die weitere Auswertung von Testergebnissen zu unterstützen, näher zu kommen, wird anschließend das Qualitätsmanagement betrachtet. Hierzu werden relevante Normen, Standards und Verfahren des Qualitätsmanagements von Software dargestellt. Zusammenfassend werden

diese in 2.4.3 verglichen und kurz diskutiert.

Anschließend erfolgt die Hinführung zum Bereich des Softwaretests. In Abschnitt 2.5 wird dazu anfangs allgemein die Qualitätssicherung dargestellt, bevor in 2.5.2 grundlegende Begriffe des Softwaretests erläutert werden. Darauf aufbauend wird der Softwaretestprozess dargestellt, u. a. in Abschnitt 2.5.4 durch eine externe Betrachtung vor dem Hintergrund des Softwarelebenszyklus, wodurch eine Einordnung des Softwaretests in einen größeren Rahmen ermöglicht wird. Der Abschnitt 2.5.5 betrachtet im Gegensatz dazu die Methoden des Softwaretests und stellt dadurch eine interne Sichtweise des Testprozesses auf. Davon ausgehend können anschließend die Besonderheiten des Systemtests von Softwareplattformen dargestellt werden. In 2.5.7 werden die wesentlichen Aussagen zusammengefasst.

Für die Auswertung der Testergebnisse zu Qualitätsmanagementzwecken wird schließlich auf Qualitätsmetriken eingegangen. Abschnitt 2.6 stellt dazu den Bereich der Softwarekennzahlen vor. Hierzu werden nach einer Begriffsklärung die Ziele von Kennzahlen und Kennzahlensystemen dargestellt. Abschnitt 2.6.3 geht anschließend auf einige Kennzahlen ein und kategorisiert diese anhand des von ihnen zu bewertenden Attributs bzw. der von ihnen zu bewertenden Entität.

2.1. Softwarekategorien

Der folgende Abschnitt soll einen Überblick liefern über verschiedene Kategorien von Software, um Gemeinsamkeiten und Unterschiede zu Plattformen darzustellen. Dies erfolgt, um der Definition von Plattformen auch eine Abgrenzung zu erhalten, was eine Plattform ist und was sie nicht oder nicht zwingenderweise ist. Die meisten dieser Kategorien werden bzgl. ihrer Unterschiede zu „herkömmlichen Applikationen“ betrachtet, worunter man hierbei verschiedene Anwendungen verstehen kann. Als Beispiel könnte etwa eine Anwendung aus der Microsoft Office Produktfamilie aber auch ein Portal dienen. Eine genauere Definition und Abgrenzung wird im Folgenden behandelt.

Die Kategorien sind nicht als disjunkte Klasseneinteilung zu verstehen; vielmehr stellen sie eine lose Ansammlung von Begriffen dar, die in den letzten Jahren vermehrt in den Fokus wissenschaftlicher Betrachtungen gerückt wurden. Durch diese vom Betrachtungs-

winkel stark abhängige Einteilung lassen sich einige Softwareprodukte mehreren Klassen zuordnen.

2.1.1. Plattformen

In den meisten Fällen wird bzgl. der IT unter einer *Plattform* die Hardware-Basis eines Computersystems, ein Betriebssystem und grundlegende Softwareprodukte oder eine Kombination dieser Komponenten verstanden[1]. Beispielsweise gibt es Microsoft Word für jede Windows-Plattform, d. h. für jeden Rechner, auf dem ein Windows-Betriebssystem betrieben wird.

Oftmals bezeichnet eine Plattform auch ausschließlich ein Hardwaregerät, wie etwa einen PC, ein Mobiltelefon oder einen „Personal digital assistent“ (PDA) [TRK05]. Zusätzlich wird der Begriff auch auf andere Bereiche bezogen verwendet und kann in Form einer Kommunikationsplattform für einen Ort stehen, an dem Gedanken und Meinungen ausgetauscht werden können. Die folgende Definition ist eine Übersetzung der Begriffsdefinition der *Object Management Group* (OMG) und auch mit der in [JWBH01] gegebenen in Einklang zu bringen[2]. Dies ist notwendig, da die dort beschriebene Studie im weiteren Verlauf dieser Arbeit noch Verwendung finden wird.

Definition: Plattform

Eine Plattform besteht aus einer Menge an Subsystemen und Technologien, die eine zusammenhängende Menge an Funktionalitäten durch Schnittstellen und festgelegte Verwendungsanleitungen bereitstellen. Diese können von jeder von der Plattform unterstützten Anwendung verwendet werden, ohne dass diese sich um Detailfragen der Implementierung kümmern müssen. [Obj03]

Plattformen stellen grundlegende Funktionalitäten zur Verfügung, die anderen Softwareprodukten als Basis dienen. Eine Plattform ist so entworfen, dass ihre Dienste auch von

[1] Schneider und Werner definieren in [SW00, S. 731]: „Die [...] Plattform bezeichnet die Gesamtheit aus Hardware, Betriebssystem, Datenbanksystem bzw. Datenverwaltungssystem und Kommunikationssystem.“

[2] Johansson et. al. definieren in [JWBH01]: „A software platform is defined as the first in a product-line, which is a collection of functionality that a number of products is to be based on.“

sehr unterschiedlichen Softwareprodukten in Anspruch genommen und mit ihnen sehr unterschiedliche Anwendungen entworfen werden können. Oftmals stellt sie somit Dienste zur Verfügung, die Verallgemeinerungen spezialisierter Funktionalitäten anderer Softwareprodukte bilden.

Plattformen werden zur Verfolgung unterschiedlicher Ziele entwickelt. Meistens soll die Entwicklung von Anwendungen aufbauend auf der Plattform vereinfacht werden. Dies betrifft oft komplexe Zusammenhänge im Zusammenspiel einzelner Komponenten der Plattform. Durch die Bereitstellung einer abstrahierten Sichtweise für den Anwender erhält dieser eine vereinfachte Darstellung einer komplexen Basis. Einerseits kann er die Plattformfunktionalität als gegeben ansehen und in den meisten Fällen dadurch schneller und kostengünstiger entwickeln. Andererseits ist er durch die Vorgabe der Plattform eingeschränkt in der Architektur seiner Applikation.

Viele Plattformen haben ein weiteres Ziel bzw. einen weiteren Nutzen für die Anwender: Ohne sie wäre die Entwicklung bestimmter Anwendungen nicht möglich, d.h. erst durch sie können viele Applikationen realisiert werden. Beispielsweise bilden Betriebssystemplattformen die Basis für alle Applikationen wie Microsoft Office, PC-Spiele, etc. und sind dadurch die Grundlage ganzer Märkte der heutigen Wirtschaft.

Des Weiteren kann durch eine Plattform eine stabile Basis geschaffen werden, die darauf entwickelte Anwendungen damit auch tendenziell robuster macht. Hierbei ist zu beachten, dass die Entwicklung einer robusten Anwendung auf Basis einer instabilen Plattform so gut wie unmöglich ist. Diese Überlegung macht die Relevanz der Qualität bei der Plattformentwicklung deutlich. Ein Softwarefehler in einer Anwendung beeinflusst meist nur diese eine Anwendung. Ein Softwarefehler in einer Plattform beeinflusst in der Regel eine ganze Reihe von Anwendungen.

Der Einsatz von Plattformen ermöglicht die Vereinheitlichung des Betriebs, der Entwicklung und zu einem großen Teil auch der Wartung von Anwendungen. Die Entwickler, die Operatoren und Architekten müssen hauptsächlich im Umgang mit wenigen Plattformen geschult werden. Die einheitlicheren Entwicklungs- und Betriebsumgebungen ermöglichen darüber hinaus einen besseren Erfahrungsaustausch von Anwendungsprojekten, die dieselbe Plattform einsetzen.

2.1.2. Anwendungssoftware und Systeme

Die Begriffe *Anwendungssoftware* und *System* haben in der Informationstechnologie verschiedenste Bedeutungen und letzterer wird auch in anderen wissenschaftlichen Gebieten und umgangssprachlich verwendet. Um eine eindeutige Basis zu schaffen, werden diese beiden grundlegenden Ausdrücke hier kurz definiert.

Definition: Anwendungssoftware[3]

Unter *Anwendungssoftware* wird Software verstanden, „die Aufgaben des Anwenders mit Hilfe eines Computersystems löst.“ [Bal00, S. 24]

Anwendungssoftware führt damit eine oder mehrere Funktionen direkt für einen Anwender aus. Gemäß den an die jeweilige Applikation gestellten Anforderungen sollen Ergebnisse bzw. Rückmeldungen an den Benutzer zurückgeliefert oder bestimmte Aktionen ausgeführt werden. Diese Funktionen werden direkt durch den Benutzer initiiert. Die Aktionen der Anwendung können dann auch zeitverzögert oder scheinbar selbständig erfolgen.

Anwendungssoftware „setzt in der Regel auf der Systemsoftware [...] auf bzw. benutzt sie zur Erfüllung der eigenen Aufgabe“ [Bal00, S. 24]. Sie steht damit im Gegensatz zu Systemsoftware, welche „die grundlegenden Dienste für andere Programme zur Verfügung [stellt], insbesondere den Zugriff auf eine konkrete Rechnerplattform. Die zentralen Dienste der Systemsoftware werden zusammenfassend auch als Betriebssystem bezeichnet“ [HN01, S. 915], das dadurch den Betrieb von Anwendungen ermöglicht, dem Endanwender jedoch keinen direkten Nutzen bringt. Anwendungssoftware bildet somit einen Gegenpart zu Plattformen, wohingegen Plattformen in vielen Fällen durchaus als Systemsoftware bezeichnet werden können.

2.1.3. Frameworks

Der Begriff *Framework* wird zum Teil als Synonym von *Plattform* verwendet, hier jedoch von diesem unterschieden. Die wörtliche Übersetzung von Framework ist *Rahmen*, *Rahmenwerk*, *Skelett* oder *Gerüst*. Dadurch wird die Aufgabe eines Frameworks bereits gut

[3] Oft auch als *Anwendung* oder *Applikation* bezeichnet.

beschrieben: Es soll durch Vorgabe eines Rahmens entscheidenden Einfluss auf die Architektur einer Anwendung nehmen und die Wiederverwendung architektonischer Muster ermöglichen.

Eine exaktere Definition liefert R.V. Binder:

Definition: Framework

„A framework is an implementation of a reusable, domain-specific design.“ [4] [Bin00, S. 1089]

Ein Framework ist somit eine lose Zusammenstellung von Funktionseinheiten bzw. Diensten, die zusammen einen definierbaren Nutzen erbringen und je nach Blickwinkel den Rahmen, das Gerüst oder den Kern von verschiedenen Applikationen darstellen. Anwendungen können entsprechend darauf aufbauend oder davon eingegrenzt entwickelt werden.

Plattformen und Frameworks sind oft ähnlich, jedoch wird für die Verwendung des Begriffs *Framework* der Basischarakter der bereitgestellten Dienste nicht vorausgesetzt. Es kann auch lediglich einen groben Rahmen vorgeben, eine Plattform hingegen bildet immer eine solide Basis.

Die Benutzer einer Plattform können Menschen, andere Plattformen, Frameworks, Applikationen oder sonstige IT-Systeme sein, ein Framework wird in der Regel für die Benutzung durch einen Menschen entworfen. Die implementierten Schnittstellen unterscheiden sich dadurch entsprechend.

In dieser Arbeit haben die beiden Begriffe den Aspekt der Mehrfachverwendung gemeinsam. In manchen Definitionen in der Literatur wird auf diesen Punkt bei Frameworks jedoch verzichtet[5], bei Plattformen bildet er immer eine wesentliche Eigenschaft. Die Mehrfachverwendung stellt für Plattformen ein entscheidendes Entwicklungsziel dar.

[4] Binder bezieht sich dabei auf die folgende Definition von „domain“ bzw. Bereich:„The term domain is used to denote or group a set of systems or functional areas, within systems, that exhibit similar functionality.“ [Kea97]

[5] Siehe beispielsweise [PP04].

2.1.4. CBS – COTS- und komponentenbasierte Systeme

Die Abkürzung *CBS* hat in der Literatur verschiedene Bedeutungen. Sie steht bzgl. der IT sowohl für *Component Based Systems* oder *komponentenbasierte Systeme* als auch für *COTS Based Systems*. *COTS* ist die Abkürzung für *Commercial Off-The-Shelf* oder *Component Off-The-Shelf* [6]. Die beiden Begriffe können aber auch für dasselbe System zugleich zutreffen, indem ein komponentenbasiertes System nicht nur aus eigenentwickelten Komponenten besteht, sondern auch auf dem freien Markt verfügbare Komponenten, COTS-Komponenten, integriert. CBS weisen, wie unten erläutert, Gemeinsamkeiten mit Plattformen auf.

Komponentenbasierte Systeme

Die folgende Definition von komponentenbasierten Systemen ist Hansen und Neumann [HN01] entnommen, wo von komponentenorientierten Systemen die Rede ist:

> **Definition: Komponentenbasiertes System**
>
> „Unter einem komponentenorientierten System versteht man ein Softwaresystem, dessen Funktionalität auf klar abgrenzbare Komponenten verteilt wird, die jeweils eine bestimmte Teilfunktionalität zur Verfügung stellen.“ [HN01, S. 154]

Ein *komponentenbasiertes System* wird dabei direkt mit einem *monolithischen System* verglichen, worunter „ [...] man im Gegensatz dazu ein System [versteht], das nicht durch Komponenten aufgebaut ist“ [HN01, S. 154]. Die folgende Definition einer *Komponente* nach Binder passt dazu sehr gut:

> **Definition: Komponente**
>
> „Any software aggregate that has visibility in a development environment, for example, a method, a class, an object, a function, a module, an executable, a task, a utility subsystem, an application subsystem. This includes executable software entities supplied with an application programmer interface.“ [Bin00, S. 1080]

[6] Die wörtliche Übersetzung von „Off-the-shelf“ lautet „serienmäßig produziert“ oder „von der Stange“.

Diese Definitionen machen deutlich, dass der Umfang einer Komponente stark variieren kann (von einer Klasse bis zu einem Subsystem)[7]. Dadurch kann nicht ausgeschlossen werden, dass Plattformen als Komponenten betrachtet werden. Ein grundlegender Unterschied besteht jedoch darin, dass komponentenbasierte Systeme aus verschiedenen Komponenten zusammengesetzt und plattformbasierte Systeme auf in der Regel einer Plattform aufbauend entwickelt werden.

COTS-basierte Systeme

Bei vielen Software-Neuentwicklungen bieten Kombinationen von auf dem freien Markt erhältlichen Softwareprodukten dieselbe oder eine ähnliche Funktionalität wie das zu entwickelnde Programm. Nach Boehm und Abts ist der Einsatz von derartigen Produkten in vielen Fällen mittlerweile eine ökonomische Notwendigkeit [BA99]. Die Tage in denen größere Unternehmen und Regierungen ihre Datenbanken, Netzwerkinfrastrukturen usw. selbst entwickeln und warten konnten, seien vorbei.

> **Definition: COTS-Produkte (COTS – Commercial oder Component off-the-shelf)**
>
> „Mit dem Begriff COTS-Komponenten oder schlicht COTS werden kommerziell erwerbbare Softwarekomponenten bezeichnet. [...] Häufig ist auch das Adjektiv 'commercial' in diesem Akronym nur von zweitrangiger Bedeutung. Man versteht unter COTS dann jede Art von Standardsoftwarekomponenten, die über einen längeren Zeitraum von Dritten gepflegt werden, und – salopp formuliert – *von der Stange weg* eingesetzt werden können.“ [HN01, S. 159]

Der Einsatz von COTS-Produkten ist in vielen Fällen mit Schwierigkeiten verbunden. Insbesondere die Entwicklung von Softwaresystemen, die mehrere COTS-Produkte integrieren, bereiten erhebliche Aufwände, da oftmals der gesamte Entwicklungsprozess verändert werden muss [BOS00; TAB03]. Bereits die Auswahl und Evaluation passender COTS-Komponenten stellt einen erheblichen Aufwand dar [BMB96].

[7] Begründet durch verschiedene Sichtweisen auf komponentenbasierte Systeme sind auch verschiedene Definitionen von Komponenten in Verwendung. Für weitere Varianten wird an dieser Stelle auf Szyperski [Szy03] und Reussner [RPS03] verwiesen.

Definition: COTS-basierte Systeme

COTS-basierte Systeme sind Softwaresysteme, die mindestens eine COTS-Komponente beinhalten. Häufig werden mehrere verschiedene COTS-Produkte mit eigenentwickeltem Code zu einem System integriert. Dieser Code wird als *Glue-Code* bezeichnet. [BB01]

COTS-basierte Systeme weisen somit durchaus Parallelen zu plattformbasierten Systemen auf, da die Plattform als COTS-Produkt interpretiert werden kann. Einige Eigenschaften, die eine Plattform ausmachen, müssen bei COTS-Produkten jedoch nicht vorhanden sein, beispielsweise die Bereitstellung einer umfassenden Basis.

2.1.5. Middleware

Dieser Abschnitt erläutert den Begriff *Middleware*. Dies geschieht zum einen, da im späteren Praxisbeispiel Middlewareplattformen betrachtet werden. Zum anderen ergeben sich Gemeinsamkeiten zwischen Plattformen und Middleware.

Middleware bezeichnet anwendungsunabhängige Software, die Dienstleistungen zur Vermittlung zwischen Anwendungen anbieten, so dass die Komplexität der zugrundeliegenden Applikationen und Infrastruktur verborgen bleibt [RMB01]. Sie stellt Software für den direkten Datenaustausch zwischen Anwendungsprogrammen dar, die auch in heterogenen Netzen arbeiten kann [Bro05]. Dadurch bildet sie eine Vermittlungsebene zwischen verschiedenen Anwendungen oder Teilen von Systemen. Zu ihren Aufgaben gehört die Vereinfachung der Kommunikation und die Senkung des Komplexitätsniveaus innerhalb von Systemen aus Sicht der Anwendungsentwicklung [GB05]. Sie wird daher oft, wie Plattformen auch, als Teil des betrachteten Softwaresystems angesehen, ohne eine direkte Benutzerinteraktion anzubieten.

Middleware liegt aber nicht nur in verteilten Systemen *in der Mitte* zwischen verschiedenen Anwendungen oder Teilsystemen, sondern kann auch als Vermittlungsinstanz zwischen Anwendungen und Betriebssystem agieren. Dabei organisiert sie die Kommunikation auf einem höheren Niveau des Schichtenmodells – die unteren Schichten werden in der Regel bereits durch das Betriebssystem selbst bereitgestellt.

Meist wird Middleware im Kontext von Client-/Server-Systemen gesehen. Sie abstrahiert dabei vom eigentlichen Kommunikationsmechanismus und „realisiert eine einheitliche Schnittstelle, die für Client und Server quasi das Aussehen eines lokalen Prozedur- oder Methodenaufrufes hat“ [Sel00, S. 22]. Die Informationsübermittlung kann grob über einen *synchronen* oder einen *asynchronen* Nachrichtenaustausch realisiert werden. Eine der einfachsten und bekanntesten Formen von Middleware wird durch den *Remote Procedure Call* realisiert [Sel00]. Middleware-Produkte sind oftmals Frameworks, wie beispielsweise CORBA, .NET und J2EE, oder bauen auf diesen auf. Diese zum Teil proprietären Produkte setzen wiederum Standardtechnologien (XML, Web Services) und Protokolle (SOAP, HTTP, TCP/IP) zur Realisierung der Kommunikation ein.

Eine allgemein gehaltene und daher auch unspezifische Definition lautet:

Definition: Middleware

„Mechanismen und Techniken, die dazu dienen, die Interaktion zwischen getrennten Softwarekomponenten zu ermöglichen, werden als Middleware bezeichnet.“ [8] [HN01, S. 166]

2.1.6. Produktlinien

Der Begriff *Produktlinie*[9] hat eine besondere Relevanz für diese Arbeit, da Produktlinien stets eine Plattform zugrunde liegt und da er bereits vor Jahren vermehrt in den Fokus wissenschaftlicher Betrachtungen gerückt wurde. Auch Unternehmen führen vermehrt derartige Konzepte ein. Von der Wirtschaft und Instituten, wie etwa einigen Fraunhofer Instituten, geförderte Projekte zur Untersuchung verschiedenener Aspekte der Produktlinienentwicklung wurden ebenfalls bereits vor Jahren ins Leben gerufen. So wird in den ITEA[10]-Projekten ESAPS[11], CAFÉ[12] und FAMILIES[13] die europäische Forschung auf die-

[8] Die prägnanteste Beschreibung ist jedoch (frei nach [OHE99, S. 38]): Middleware is the slash (/) between client and server. It is the glue that lets a client obtain a service from a server.
[9] Oft auch als *Produktfamilie* bezeichnet.
[10] ITEA – „**I**nformation **T**echnology for **E**uropean **A**dvancement“.
[11] ESAPS – „**E**ngineering **S**oftware **A**rchitectures, **P**rocesse**s**“.
[12] CAFÉ – „From **C**oncept to **A**pplication in System-**F**amily **E**ngineering“.
[13] FAMILIES – „**Fa**ct-based **M**aturity through **I**nstitutionalisation **L**essons-learned and **I**nvolved **E**xploration of **S**ystem-family engineering“.

sem Gebiet gemeinsam vorangetrieben [Lin02; FAM].

In der Literatur werden die Begriffe *Produktlinie* und *Produktfamilie* meist unterschiedlich definiert: Oft steht eine Produktfamilie für eine Menge von Produkten, die einen gemeinsamen Markt haben und sich damit im Verwendungszweck ähneln[14]. Im Gegensatz dazu bestehen Produktlinien aus Produkten, die auf einer gemeinsamen Basis, auf gemeinsamen Artefakten, aufbauen. Hier werden die beiden Bezeichnungen in Anlehnung an [BG03] gleichbedeutend in letzterem Sinne, d. h. als Produkte mit einer gemeinsamen Basis, verwendet.

Bei der Produktlinienentwicklung wird die Wiederverwendung einiger Komponenten, Dokumente, etc. geplant. Dieser wiederverwendbare Teil der Software wird in einer sog. Plattform zusammengefasst. Dazu müssen mehrere Aspekte berücksichtigt werden; u. a. müssen der Entwicklungsprozess und die Organisation des Entwicklungsteams an die Gestaltung von zweigeteilten Produkten angepasst werden (siehe Abb. 2.1).

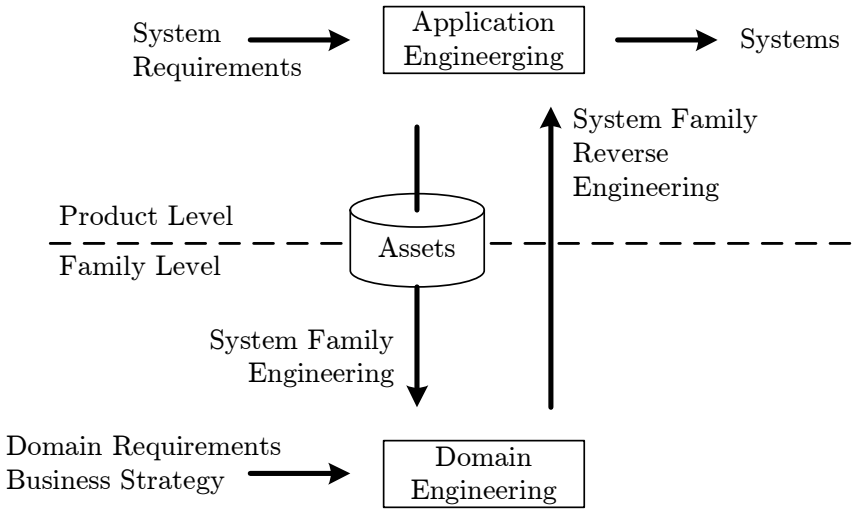

Abbildung 2.1.: Prozess-Referenz-Modell des CAFÉ-Projektes nach [BG03].

Im sog. *Scoping* werden in drei Schritten der Umfang und der grobe Aufbau der Produkte und der Produktlinie beschrieben. Im *Product Portfolio Scoping* werden die Produkte, die in der Produktlinie enthalten sein sollen, festgelegt. Im *Domain Scoping* wird ermittelt, welche Teile der einzelnen Produkte wiederverwendet werden können bzw. so gestaltet

[14] Als Beispiel kann hier die Microsoft Office Produktfamilie dienen.

werden können, dass sich eine hohe Wiederverwendungsrate realisieren lässt. Daraus ergibt sich dann eine allen Produkten der Produktlinie gemeinsame Plattform, die daraufhin im *Domain Engineering* entwickelt wird. Durch das *Asset Scoping* werden die Artefakte identifiziert, die zur Realisierung der Plattform nötig sind.

Die verschiedenen Produkte innerhalb einer Produktlinie unterscheiden sich dann an sog. *Variationspunkten*. Eine konkrete, applikationsspezifische Ausprägung in Bezug auf einen oder mehrere solcher Variationspunkte ergibt dann eine *Variante*. Die Varianten werden im *Application Engineering* entwickelt [BKP04].

Die Unterschiede der einzelnen Produkte einer Produktlinie können groß sein, meist überwiegen jedoch die Gemeinsamkeiten. In einem Beispiel in [NFLTJ03] besteht eine Produktfamilie aus verschiedenen Editionen eines Softwareprodukts[15].

Die Herangehensweise der Produktlinienentwicklung unterscheidet sich von der bisherigen Softwareentwicklung in einem entscheidenden Punkt: „Es werden nicht mehr einzelne Softwareprodukte *reaktiv* nach Markt- oder Kundenbedarf entwickelt, sondern der Fokus liegt auf der *proaktiven* Gestaltung einer gemeinsamen Plattform für eine Vielzahl von jetzigen und zukünftigen Produkten.“[BKP04]

Hierbei ist zu beachten, dass die bisherigen Entwicklungsmethoden hauptsächlich auf die Entwicklung einer Vielzahl von heutigen Produkten zugeschnitten ist. Dies wird an den verschiedenen Scoping-Begriffen deutlich, da zunächst die in der Produktlinie enthaltenen Applikationen beschrieben werden und anschließend deren Bestandteile jeweils in applikationsspezifische und in der Plattform enthaltene aufgeteilt werden. Die aktive Gestaltung zukünftiger, bisher noch nicht geplanter und somit unbekannter Applikationen und Projekte kann dadurch nur als untergeordnetes Entwicklungsziel angesehen werden. Dadurch unterscheidet sich die Entwicklung einer Plattform als Teil einer Produktlinie von der von Anwendungen zunächst losgelösten Entwicklung einer eigenständigen Plattform.

[15] *Demonstration Edition, Personal Edition, Enterprise Edition.*

2.2. Anforderungen und Features von Software

Quantitative Auswertungen, die ausschließlich die Testfälle eines Softwareprodukts bzw. einer Version desselben betrachten, sind nicht nur in der Plattformentwicklung wenig aussagekräftig. Sie müssen stets in einem breiteren Kontext betrachtet werden. Eine Möglichkeit dies umzusetzen bietet die Verknüpfung mit Anforderungen. Dadurch können detailliertere Aussagen getroffen und die Schwachstellen im Produkt identifiziert werden. Dies ist jedoch nur ein Grund, weswegen die Betrachtung von Anforderungen in dieser Arbeit nötig ist.

Ein weiterer Grund ist, dass nicht nur die Erarbeitung der Systemtestfälle meist auf den Anforderungen beruht. Dadurch sind die der Spezifikationsphase nachgelagerten Prozessschritte und ihre Verknüpfung mit den Anforderungen von Bedeutung für das Qualitätsmanagement.

In diesem Abschnitt wird auf das für die Plattformentwicklung erstellte Plattform-Feature-Modell eingegangen, das eine durchgängige Abbildung der Anforderungen ermöglicht und dadurch eine Basis schafft für Testmetriken, die auf die Anforderungen an die Software bezogen werden können. Zunächst wird jedoch eine allgemeine Einführung gegeben.

2.2.1. Durchgängigkeit von Anforderungen

Am Anfang des Prozesses der Softwareentwicklung steht in der Regel eine Idee. Dies kann auf der einen Seite die Idee eines Kunden sein, der sich an einen Softwarehersteller wendet, um ein Projekt abwickeln zu lassen. Auf der anderen Seite kann auch die Unternehmensführung eines Softwareunternehmens eine Vorstellung eines Produkts haben, das durch die Entwicklungsabteilungen umgesetzt werden soll.

In beiden Fällen haben die „Kunden der Entwicklungsabteilung“ bereits eine Vorstellung des Könnens eines Softwareprogramms. Die erste Hürde besteht aus der Spezifizierung der Anforderungen, der *Requirements*. Wie auch in [Rup04] festgestellt wird, kommt es dabei auf eine möglichst präzise Formulierung an. Man befindet sich hierbei immer auf einer Art Gratwanderung: Die Anforderungsspezifikation sollte weder zu vage, noch zu genau

sein. Der erste Fall kann zu einem von der Vorstellung des Anwenders stark abweichenden Produkt führen, wenn im Laufe des Entwicklungsprozesses Architekten, Designer und Programmierer nach eigenen Interpretationen der Spezifikation handeln. In letzterem Fall besteht die Gefahr, dass bei der Ausarbeitung der Spezifikation technisch so detaillierte Formulierungen verwendet werden, dass manch Anwender nicht alle Einzelheiten versteht. Auch auf diese Weise kann im Endeffekt ein Produkt entstehen, welches letztlich nicht in dieser Form gewünscht war. Dieser Sachverhalt wird auch in [Par98] unter dem Stichwort „Kommunikationsproblem“ behandelt. Um überhaupt eine Chance zu haben, dieses Problem in den Griff zu bekommen, müssen Anforderungsspezifikationen vollständig, eindeutig und konsistent sein [Rup04] und dabei dennoch verständlich formuliert bleiben.

Wie im V-Modell (siehe Abb. 2.2) [VXT08] zu sehen ist, hat die Spezifikation der Anforderungen zwei nachgelagerte und darauf aufbauende Prozesse. Zum einen die Ableitung der Abnahmetestfälle und zum anderen die Entwicklung der Architektur und des Designs der Software.

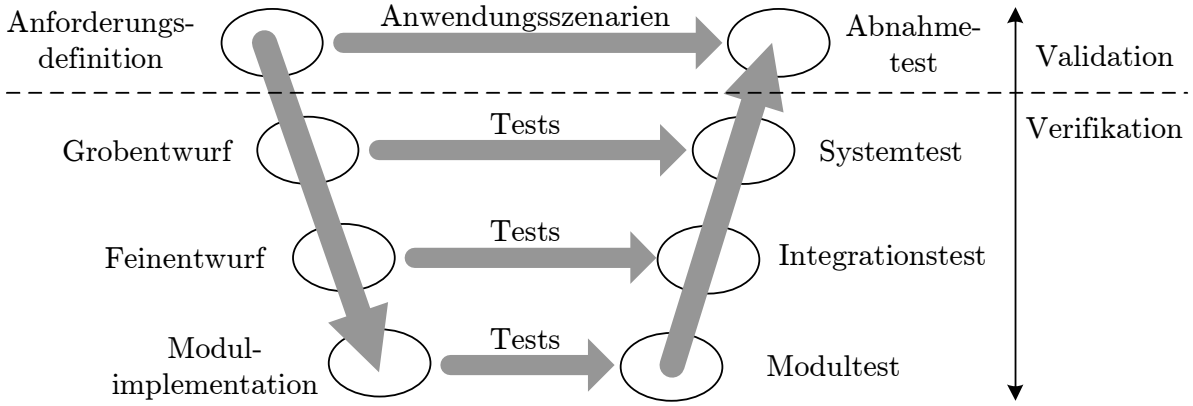

Abbildung 2.2.: V-Modell des Bundes und der Länder nach [Bal98].

Im ersten Fall bleibt man auf der obersten Ebene im V-Modell und validiert die erstellte Software gegen die Kundenanforderungen. Dies geschieht oft anhand von Anwendungsszenarien, die durch die Software abgedeckt werden sollen. Diese können anhand von *Anwendungsfalldiagrammen* (auch *Use case Diagramm*), eine Diagrammart, die in der Unified Modeling Language (UML) enthalten ist, dargestellt werden[16] [OMG].

[16] In Anwendungsfalldiagrammen wird im Allgemeinen das erwartete Verhalten des Systems in ausgewählten Situationen modelliert.

Im zweiten Fall folgt auf die Spezifikation der Anforderungen der grobe Entwurf der Software. Von diesem ausgehend werden wiederum einerseits die Systemtestfälle abgeleitet und andererseits das Design des Softwareprodukts verfeinert. Diese „Abzweigungsmöglichkeiten“ in Richtung Testfallentwicklung bestehen auf jeder Ebene des V-Modells bis hin zur Erstellung des Codes. Besonders wichtig ist bei allen daraus resultierenden Testfällen, die Rückverfolgbarkeit (oder *Traceability*) bis zu den Requirements durchgängig darzustellen. An dieser Stelle wird deutlich, dass die Möglichkeit, Zusammenhänge durchgängig nachzuvollziehen, bereits bei der Anforderungsspezifikation beginnt. Falls dies in frühen Phasen des Entwicklungsprozesses versäumt wird, ist es schwer, den Test des Gesamtsystems richtig zu bewerten. Eine Schwierigkeit, die noch nicht abschließend beseitigt ist und deren genaue Betrachtung den Rahmen dieser Arbeit sprengen würde, stellen Abhängigkeiten zwischen verschiedenen Anforderungen untereinander dar. Dies wird beispielsweise in [DP03] näher betrachtet.

In vielen Fällen werden die oftmals noch ungenau formulierten Requirements in sog. *Features* abgebildet. Diese dienen dann zum einen als Vertragsgrundlage mit den Kunden und sind dementsprechend unmissverständlich zu verfassen, zum anderen bilden sie die Grundlage für die Architektur, die Entwicklung und den Test des Softwareprodukts. In dieser Arbeit soll die folgende Definition nach [BP06b] gelten:

Definition: Feature

Ein Feature beinhaltet funktionale oder nichtfunktionale Leistungen des Systems, die gemäß den Anforderungen umgesetzt werden und klar voneinander abgegrenzt werden können.

Die erste Herausforderung besteht bei der o. g. Vorgehensweise darin, alle Anforderungen in Features umzuwandeln. Um eine vollständige Abdeckung aller Anforderungen durch Features zu erhalten, müssen auch „interne Anforderungen“, d. h. Anforderungen durch das Entwicklungsteam oder sonstige Randbedingungen, berücksichtigt werden (siehe Abb. 2.3). Im weiteren Verlauf der Entwicklung können dann alle Features, auch die aus internen Anforderungen resultierenden, wie beispielsweise die Vorgabe einer unternehmensweit standardisierten Authentifizierungsart oder Codeänderungen, um die Wartbarkeit zu erhöhen, in gleicher Form behandelt werden.

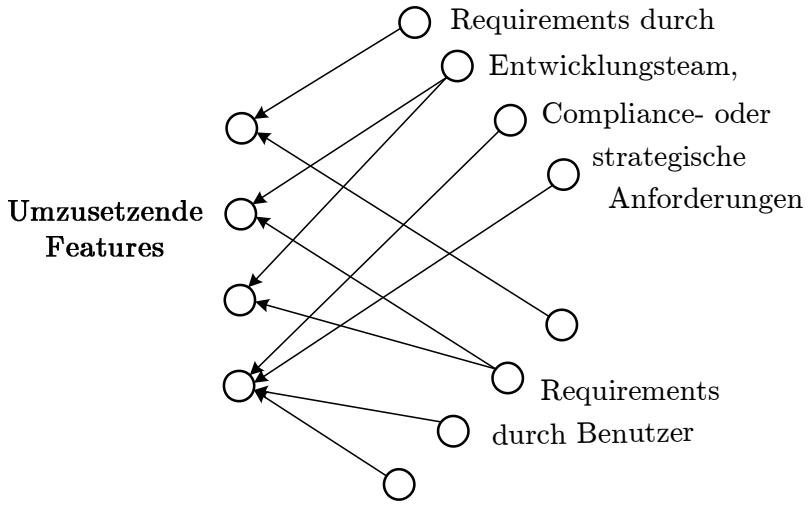

Abbildung 2.3.: Zusammenhang von Requirements und Features.

Im *Feature Driven Development* werden die Features in den Mittelpunkt gerückt und deren Entwicklung mit einigen Methoden des *Agile Development* umgesetzt [Pal02]. Dieser Ansatz kann auch weiter ausgebaut und die Traceability durchgängig anhand der Features dargestellt werden. Das Ziel eines solchen Vorgehens ist, zu jedem gefundenen Fehler die Kette aller Aktivitäten und Artefakte des Entwicklungsprozesses bis zum Requirement bzw. Feature zu verfolgen [BP06b].

Dies hat auch einige positive Auswirkungen auf den Systemtest. Hier sollen die Testfälle anhand der Spezifikationen erstellt werden. Dies kann anhand der Featurespezifikationen geschehen. Die Zuordnung von Features zu Testfällen ist im Einzelfall zwar wohl immer möglich, diesen Zusammenhang in der Breite mit geringem Aufwand transparent zu machen stellt jedoch bereits eine nicht zu unterschätzende Hürde dar. Diese Thematik wird in Kapitel 4.2.1 weiter erläutert.

Die Strukturierung des Produkts in einzelne Features zieht sich beim Feature Driven Development durch den gesamten Entwicklungsprozess, so dass es möglich ist, einzelne Codefragmente zum Teil mehreren Features zuzuordnen. Hierbei ist es notwendig für jede Entwicklungsebene entsprechende Abbildungen zu erstellen. Auch eine komponentenorientierte Entwicklung kann featurebasiert organisiert werden, indem Features und Komponenten aufeinander abgebildet werden [BP06b]. Auf diese Weise ist es möglich, jeden Testfall auf jeder Ebene des V-Modells, sei es im Unit-Test oder im Systemtest, den Features

zuzuordnen.

2.2.2. Plattformen und Features

Eine Herausforderung bei der Modellierung der Durchgängigkeit von Anforderungen ergibt sich durch Plattformkonzepte. Diese beruhen darauf, dass gleiche Teile verschiedener Applikationen nur einmal implementiert werden sollen[17]. Auch Produktlinienansätze berücksichtigen dieses Konzept. Der Wiederverwendungsgewinn und der Erfolg der Umsetzung des Plattformkonzepts wird durch eine möglichst langfristige Auslegung der Plattform maximiert. Dadurch ergeben sich vermehrt neue Kundenanforderungen, die in bereits bestehende Plattformen integriert werden sollen. Die in der Plattform enthaltenen Teile müssen nicht von allen auf ihr aufbauenden Applikationen eingesetzt werden, wodurch die meisten Applikationen einen unterschiedlichen Satz an Plattformfunktionalitäten einsetzen. Die Konsequenz ist, dass die Entwicklung der Plattform flexibel auf Änderungen reagieren und die Abwärtskompatibilität zu vorherigen Releases gewährleisten muss. Dies bedingt die Notwendigkeit, die Anforderungen durchgängig verfolgen zu können. [WN94]

Wie bereits erwähnt, ist es notwendig, die Durchgängigkeit von Anforderungen zu modellieren. Dazu werden häufig Feature-Modelle eingesetzt. Diese sind für Plattformen nicht nur wegen der Verfolgbarkeit der Anforderungen und Features gut geeignet, sondern auch weil sie bzgl. der Entwicklung die nötige Flexibilität bieten [SR02]. Die Integration von Features in Plattformen wird von den Modellen jedoch nicht berücksichtigt. Aus diesem Grund wird hier das erweiterte Feature-Modell nach [BP06b], das „Plattform-Feature-Modell“, und das dazugehörige Konzept zur Verfolgbarkeit eingeführt, das die Integration von Plattformen berücksichtigt. Darin spielen die Beziehungen zwischen verschiedenen Ebenen des Entwicklungsprozesses eine zentrale Rolle. Dieses bildet die Grundlage für die Erörterung der Auswirkungen der featureorientierten Entwicklung von Plattformen auf den Test in Kapitel 4.2.1.

Die featureorientierte Entwicklung birgt jedoch auch Risiken. Nachteile gegenüber herkömmlichen Entwicklungsmethoden können sich ergeben, wenn der Featurebegriff nicht

[17] Als Beispiel diene hier die Benutzer-Authentifizierung, die bei allen geschützten Applikationen eines Unternehmens gleich sein soll.

durchgängig eingesetzt wird. In diesem Fall besteht die Gefahr, durch die featureorientierte Sichtweise zum einen auf manchen Entwicklungsstufen mehr Zeit investieren zu müssen und keinen Nutzen daraus ziehen zu können. Im Gegenteil, in diesem Fall müssen zusätzliche Artefakte verwaltet und gepflegt werden. Zum anderen wird durch die zusätzliche Komplexitätsstufe, die sich durch die Features ergibt, leichter der Überblick über die Details verloren. Auch die interne Kommunikation zwischen Entwicklern, Testern, Architekten, etc. wird durch ein nicht durchgängig umgesetztes featureorientiertes Modell evtl. erschwert, da nicht zwangsläufig alle Beteiligten mit Features arbeiten. Somit ist es wichtig, auf die Durchgängigkeit bei der Einführung eines Feature-Modells zu achten.

Plattform-Feature-Modell

Das Plattform-Feature-Modell soll die Verbindung von Plattformen und Features ermöglichen. Features entstehen aus den Anforderungen, wobei jede Anforderung in mindestens ein Feature mündet, ein Feature jedoch auch mehrere Anforderungen abbilden kann. Die Features wiederum werden anschließend mindestens einer Plattform zugeordnet (siehe Abb. 2.4). Daraus ergeben sich verschiedene Featuretypen. Innerhalb der Plattformentwicklung, bei der der Einsatz mehrerer zu kombinierender Plattformen vorgesehen ist, wird zwischen den folgenden Featuretypen unterschieden:

- *Common Features* kommen in allen Plattformen vor,
- *Plattform Features* kommen in mindestens einer, aber nicht in allen Plattformen vor und
- *Shared Plattform Features* kommen in mindestens einer „Shared Plattform“ vor.

Das Plattform-Feature-Modell baut auf dem FODA-Verfahren[18] [KCH+90] auf und ermöglicht zum einen die Zuordnung der Features zu Plattformen. Zum anderen werden durch das Modell die Features in feinere Design-Artefakte zerlegt. Dies können Komponenten sein. Diese sind eingeteilt in einen festen Anteil, der in allen Plattformen enthalten sein muss und einen variablen Teil. Der feste Teil der Komponenten definiert ein *Core Set* von Komponenten, wodurch die einzelnen Plattformen selbst eine Art Plattform- oder

[18] FODA steht für Feature Oriented Domain Analysis.

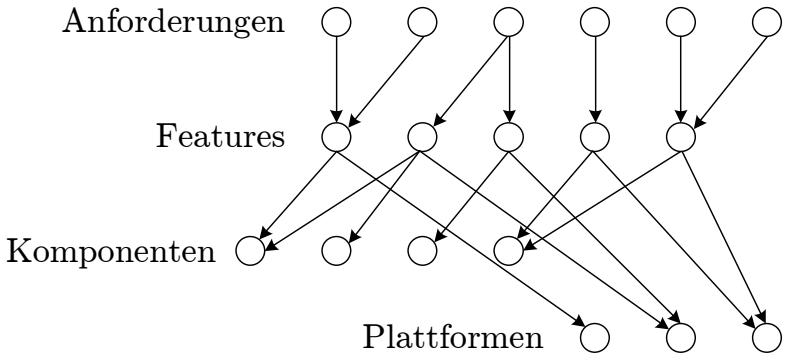

Abbildung 2.4.: Zusammenhänge zwischen Anforderungen, Features, Komponenten und Plattformen.

Produktliniencharakter bekommen. Dies spiegelt sich auch im Aufbau und der Einteilung der Features wider.

Da es vorkommen kann, dass ein Feature durch die Kombination mehrerer Plattformen realisiert wird, werden Traceability Links verwendet, um die Beziehungen zwischen Anforderungen, Features und Komponenten herzustellen [PRP04; Rie04]. Durch die Einführung der Featureschicht lässt sich die Kundensicht auf die Sichtweise der Entwickler, beschrieben durch die Komponenten, abbilden. Die Zusammenfassung von Features zu Plattformen ist hilfreich, um die Aufgaben von Komponenten untereinander abzugrenzen, da Features in verteilten Systemen oftmals Einfluss auf mehrere Komponenten haben.

Featureorientiertes Verfolgbarkeitskonzept

Die Grundidee des featureorientierten Verfolgbarkeitskonzepts nach [BP06b] ist die Darstellung des Entwicklungsprozesses auf verschiedenen Ebenen. Die Traceability Links zwischen verschiedenen Entwicklungsartefakten bewegen sich hierbei immer zwischen verschiedenen Ebenen. Als Startpunkt der Verfolgbarkeit werden wie in [Got95] Anforderungen in Form einer Anforderungsebene verwendet, um zwischen den verschiedenen Verfolgbarkeitsebenen nachvollziehbare Verbindungen herzustellen. Im Folgenden werden die für diese Arbeit relevantesten Ebenen kurz vorgestellt.

Auf der Ebene der Anforderungen werden die geforderten Funktionalitäten dokumentiert, strukturiert und zum Teil konsolidiert. Dies geschieht auf Basis von Priorisierungen und, im Produktlinienkontext, ersten Überlegungen, welche Funktionalitäten in der Platt-

form integriert werden sollen.

Auf der Ebene der Features entstehen aus den Anforderungen und Anwendungsfällen Features. Diese können präzise formuliert werden und stellen im weiteren Verlauf der Entwicklung die Grundlage der zu entwickelnden Software dar. Anhand der Featurespezifikationen können bereits System- bzw. Abnahmetestfälle erstellt werden.

Die tieferliegenden Ebenen (Plattform-Design, Komponenten, Implementation Unit) sind für diese Arbeit nur noch am Rande relevant. Sie beschreiben das Design und die Entwicklung der Software. Das Design eines Releases wird bestimmt durch die Identifikation der betroffenen Plattformen und weiter auch der Komponenten, die Bestimmung der Abhängigkeiten verschiedener Artefakte und schließlich die Feststellung der benötigten Änderungen bisheriger Software oder das Design neuer Komponenten. Bei der weiteren Entwicklung werden verschiedene Aktivitäten (Test, Entwicklung, Dokumentation, Build) unterschieden, bei deren Durchführung auf Abhängigkeiten untereinander und die Reihenfolge geachtet werden muss. Auf der untersten Ebene findet dann die Entwicklung der einzelnen Artefakte statt.

2.2.3. Nichtfunktionale Anforderungen

Der vorige Abschnitt bezog sich ganz allgemein auf Anforderungen aller Art. In der Literatur und in der Praxis wird jedoch unterschieden, ob es sich um funktionale oder nichtfunktionale Anforderungen handelt. Funktionale Anforderungen können in der Regel direkt umgesetzt und getestet werden. Die Behandlung nichtfunktionaler Anforderungen erfordert einen höheren Aufwand. Es ist zwar generell auch möglich, den im vorigen Kapitel beschriebenen Weg über Features und die daraus resultierende Durchgängigkeit zu gehen, man stößt dabei jedoch relativ bald auf Schwierigkeiten. Unter nichtfunktionalen Anforderungen, wie beispielsweise dem Erfüllen bestimmter Antwortzeiten für eine Webapplikation, kann man sich in den meisten Fällen etwas Handhabbares vorstellen, die Entwicklung des Designs und letztlich des Codes der betroffenen Features ist jedoch oft nicht offensichtlich. Auch in [Par98] wird dargelegt, dass es für die Beschreibung und weitere Behandlung nichtfunktionaler Anforderungen nur die Möglichkeit der textuellen Beschreibung gibt.

Die Erfüllung einiger NFRs wird durch die Qualitätssicherung verifiziert. Dies geschieht durch Lasttests, Stabilitätstests, Ausfalltests etc. und ist oftmals aufwändig. Es wird folglich im Einzelfall geprüft, welche NFRs für das bestimmte Softwaresystem von entscheidender Wichtigkeit sind und somit getestet werden sollten. Zu NFRs werden oftmals auch Rahmenbedingungen, wie beispielsweise Budget- oder zeitliche Vorgaben gezählt [Par98]. In dieser Arbeit bezieht sich der Begriff *NFR* jedoch ausschließlich auf Qualitätsattribute der gewünschten Funktionen.[19]

Für diese Arbeit ist die Behandlung von NFRs aus einem weiteren Grund von großer Bedeutung: Sie stellt einen zentralen Unterschied zwischen dem Testen von Anwendungen und Plattformen dar. Daher ist es wichtig, den Begriff „nichtfunktionale Anforderung“[20] genauer zu untersuchen. Nach [MCN92] gibt es keine formale Definition oder komplette Auflistung nichtfunktionaler Anforderungen, in [Rup04, S. 270] wird jedoch eine sehr treffende, simple Erklärung gegeben: „Nichtfunktionale Anforderungen, das sind zunächst einmal alle Anforderungen, die nicht funktionaler Natur sind.“

In der Praxis werden unter diesem Begriff Aspekte wie Verlässlichkeit, Sicherheit, Treffgenauigkeit, Gefahrlosigkeit, Performanz und Bedienbarkeit, jedoch auch organisatorische, kulturelle und politische Anforderungen zusammengefasst. Zusätzlich müssen auch rechtliche Fragestellungen beachtet werden. [CPL04]

Aufgrund dieses breiten Spektrums ist es unmöglich, alle nichtfunktionalen Anforderungen an eine Software zu definieren. Auf der anderen Seite ist es notwendig, sich über dieses Thema Gedanken zu machen und die zentralen Punkte immer im Auge zu behalten. Dies muss letztlich projekt- oder produktspezifisch erfolgen und kann nicht allgmeingültig durchgeführt werden.

[19] Diese ergeben sich als Antwort auf die Frage „Wie soll das geplante System die gestellten Aufgaben erfüllen?“ [Par98].

[20] In [MCN92] auch als Randbedingung, Ziel oder Qualitätsmerkmal bezeichnet.

2.3. Allgemeiner Qualitätsbegriff

Nach der Brockhaus-Enzyklopädie ist die Qualität im Allgemeinen die „Gesamtheit von charakteristischen Eigenschaften“ [Bro06]. Aristoteles sah in ihr die „wesentliche Eigenschaft eines Dings, die es zu dem macht, was es ist“ [Bro06]. Diese beiden Sichten zeigen, dass sich das allgemeine Qualitätsverständnis in den letzten 2400 Jahren zumindest nicht grundlegend verändert hat. Es wurde jedoch verfeinert und für viele Branchen und Lebenslagen ausgeweitet. In den letzten Jahrzehnten haben dann auch Verbände und Gesellschaften diesen Begriff geprägt. So wird in der DIN 55350 und der DIN EN ISO 8402 definiert [DIN95a; DIN95b] und auch für diese Arbeit festgelegt:

Definition: Qualität

„Qualität ist die Gesamtheit von Eigenschaften und Merkmalen eines Produkts oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse beziehen.“

Der IEEE Standard 610.12 (*Standard Glossary of Software Engineering Terminology*, [IEE90]) definiert hingegen:

„(1) The degree to which a system, component, or process meets specified requirements.
(2) The degree to which a system, component, or process meets customer or user needs or expectations.“

Diese Definition impliziert bereits durch die Einführung des Grads der Erfüllung der Anforderungen eine grundlegende Messbarkeit von Qualität. Inwiefern diese in der Praxis umsetzbar ist, ist natürlich im Einzelfall zu prüfen und in vielen Fällen wird man über eine Bewertung eines Qualitätsmerkmals anhand eines Bauchgefühls nicht hinauskommen. Doch auch in diesem Fall ließe sich anhand von Fragebögen eine objektivere Bewertung erstellen. Diese Definition setzt jedoch die Existenz von spezifizierten Anforderungen („specified requirements“) voraus oder, im zweiten Teil, beschränkt die Qualität auf die Kunden- oder Benutzersicht.

Das „Qualitätsmanagement“ umfasst sowohl das Management der Qualität der Produkte als auch das der internen Prozesse. Die in der Praxis daraus resultierenden Aufgaben

umfassen zunächst die Identifizierung der relevanten Qualitätsattribute und das Festlegen der zu erreichenden Qualitätsziele. Diese werden oftmals in Fundamental- und Instrumentalziele eingeteilt. Das Erreichen der untergeordneten Instrumentalziele stellt eine Möglichkeit für die Erfüllung der Fundamentalziele dar [Woh06]. Als Aufgabe für die Qualitätssicherung ergibt sich nun, die Erreichung dieser Qualitätsziele zu überprüfen und zu einem gewissen Grad sicherzustellen, unabhängig davon, ob es sich um Instrumental- oder Fundamentalziele handelt. Dies geschieht durch die Lokalisierung von „Qualitätslücken“, d. h. dem Auffinden von Unterschieden zwischen Soll- und Ist-Zuständen. Die Soll- und Ist-Zustände beziehen sich meist auf die Ausprägung einzelner Qualitätsattribute.

Die Qualitätslücken bilden dann die Grundlage für die „Qualitätsverbesserung“, deren Aufgabe die Ableitung geeigneter Maßnahmen für die Beseitigung der Lücken ist. Das Qualitätsmanagement betrachtet kontinuierlich die Ergebnisse der Qualitätssicherung und -verbesserung im Kontext der Gesamtqualität. Diese Zusammenhänge sind in Abb. 2.5 dargestellt.

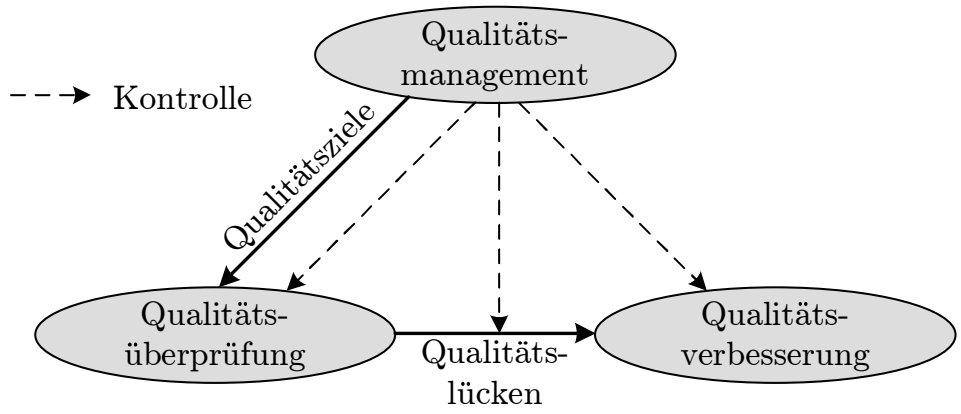

Abbildung 2.5.: 3-Teilung des Qualitätsmanagements.

In der Literatur wird dieses Modell zum Teil aus mehr als nur den drei genannten Aufgabenbereichen aufgebaut. In manchen Fällen beispielsweise bildet das Qualitätsmanagement den Oberbegriff für die Qualitätsplanung, -lenkung, -sicherung und -prüfung [Bal98]. In dieser Arbeit wird jedoch der obige Ansatz verwendet, da er die wichtigsten Aspekte klar strukturiert darstellt.

2.4. Qualitätsmanagement und -verbesserung

2.4.1. Aufgaben des Qualitätsmanagements

„Qualitätsmanagement befasst sich insbesondere mit organisatorischen Maßnahmen zur Erreichung und für den Nachweis einer hohen Qualität.“ [Lig02, S. 11]

Dies deckt sich auch mit dem Verständnis von Qualitätsmanagement in dieser Arbeit. Wie oben beschrieben, umfasst dies sowohl das Management der Qualität der Produkte als auch das der internen Prozesse. Die Qualität der Produkte wird in den meisten Fällen durch die Qualitätssicherung anhand von Tests und deren Auswertung ermittelt. Auf dieser Basis werden im Anschluss Strategien entwickelt, um die Qualität zu erhöhen.

Das Qualitätsmanagement gibt hierzu das Konzept vor. Aus ihm muss hervorgehen, „welche Stellen“ im Prozess bzw. am Produkt überprüft werden sollen. Ein Kernpunkt dieses Konzepts ist die Vorgabe der Qualitätsziele durch das Qualitätsmanagement. Diese sollten verständlich und eindeutig formuliert sein. Von großem Vorteil ist es außerdem, von vornherein darauf zu achten, dass das Erreichen gemessen werden kann. Ein Qualitätsziel in der Form „Es sollten genügend Tests ausgeführt werden, um die Kundenwünsche zu befriedigen.“ erfüllt dieses Kriterium nicht. Es mag trivial klingen, aber zum Aufstellen der Ziele ist es zunächst notwendig, sich darüber im Klaren zu sein, was genau erreicht werden soll. In obigem Beispiel ist dies noch nicht geschehen. Der Anwender hat zunächst nichts davon, wenn getestet wird – wenn das Produkt einwandfrei funktioniert, ist dies für ihn nicht relevant. Das Ziel sollte also eher folgendermaßen formuliert werden: „Die Tests sollen sicherstellen, dass alle Benutzerfunktionen in allen Konfigurationsmöglichkeiten korrekte Ergebnisse liefern.“ Darauf aufbauend kann eine entsprechende Menge an Tests erstellt und ausgeführt werden, um dieses Ziel zu erreichen.

Eine Aufstellung der Qualitätsziele kann anhand der Norm ISO/IEC 25010 [ISO05b] erfolgen. Diese Norm bildet den Nachfolger des ersten Teils der Norm ISO/IEC 9126 [ISO01b]. In ihm wird das Qualitätsmodell aufgebaut durch Qualitätsattribute[21], eingeteilt in *internal*, *external* und *quality in use* Attribute. Die Liste der Qualitätsattribute soll hierbei erschöpfend sein bzw. jeder Qualitätsaspekt soll einem Attribut zuzuordnen

[21] Oft auch als Qualitätsmerkmale oder -charakterisitken bezeichnet.

sein [AKCK05]. Die Prozessqualität des Produkterstellungsprozesses wird in der Norm nicht betrachtet. Alle Qualitätsattribute werden dann in Teilattribute zerlegt und kurz definiert. In den letzten drei Teilen der vierteiligen Norm [ISO03a; ISO03b; ISO04] werden Metriken aufgelistet, die Aussagen über die Erfüllung der einzelnen Qualitätsattribute quantifizieren sollen. Die Metriken werden dabei eingeteilt nach den Ansatzpunkten der ihnen zugrundeliegenden Qualitätsattribute (*internal*, *external*, *quality in use*).

Die Norm wurde in der Literatur bereits zum Teil kritisiert und als unvollständig und schwer verständlich bezeichnet [KLPN97], zum Teil ignoriert [Som04], zum Teil aber auch als nützlich eingestuft [ABMD04a]. Als Ausgangspunkt für eigene Überlegungen, als Ideensammlung für Metriken und Auflistung einiger wichtiger Qualitätsattribute ist sie auf jeden Fall von großem Nutzen. Fraglich ist hierbei, ob die Norm als solche formuliert sein sollte. Dadurch besteht die Gefahr, dass sie von Unternehmen verpflichtend als Stand der Technik eingesetzt werden muss [AKCK05].

Anhand dieser Qualitätsmerkmale, seien sie nun der ISO/IEC 25010 entnommen oder nicht, kann die Qualität zumindest zu einem gewissen Grad greifbar gemacht werden. Durch das Herunterbrechen auf kleinere Teile bildet ein solches Vorgehen gleichzeitig auch eine Grundlage für die Qualitätsevaluierung, da Einzelattribute stets einfacher zu bewerten sind. Zu beachten ist hierbei, dass die Gesamtqualität des Systems nicht immer nur von der Qualität der Einzelattribute abhängt.

Die Festlegung der Qualität eines Systems erfolgt in der Praxis meist über die einzelnen Qualitätsattribute oder über die Bewertung der Tests. Da die Tests jedoch auf die Qualitätsattribute abgebildet werden können, sind die Ergebnisse dieser beiden Ansätze im Endeffekt identisch. Dadurch bildet die Betrachtung der Qualitätsmerkmale nach Liggesmeyer [Lig02] definitionsgemäß die Grundlage für alle Qualitätsmaße und somit für die meisten theoretischen Ansätze des Qualitätsmanagements.

Doch nicht erst die Bestimmung der Qualität des Produkts im Nachhinein sollte sich an den Qualitätsattributen orientieren. Vielmehr kann der Entwurf bereits die wünschenswerten Qualitätsattribute berücksichtigen. Im Idealfall werden Teilaspekte des Produkts also anhand der Qualitätsmerkmale bestimmt oder sogar spezifiziert. Die genaue Analyse der Qualitätsmerkmale, evtl. deren Priorisierung, ist demnach für das zu entwickelnde Produkt

von großer Wichtigkeit. Dies wird im Detail in Kapitel 2.4.3 dargestellt werden.

Doch auch die eigentlichen Qualitätsmerkmale werden oftmals nicht zur Genüge untersucht. Anhand der Produkt- oder Projektziele können Qualitätsziele eindeutig, teils sogar messbar, formuliert werden. Es ist beispielsweise möglich einen bestimmten Überdeckungsgrad von fehlerfrei bestandenen Tests zu fordern. Das Wissen um diese Qualitätsziele ist oftmals jedoch ausschließlich als implizites Wissen vorhanden. Die Aufstellung der Qualitätsziele sollte vom Qualitätsmanagement in Zusammenarbeit mit dem Projektmanagement erfolgen. Diese Ziele sollten im weiteren Verlauf Einfluss auf die Zusammensetzung, die Aufgaben und die Ausrichtung der einzelnen Teams und, wie oben beschrieben, auf das Produkt selbst haben. Diese Ansicht wird im „Total Quality Management“ (TQM) konsequent umgesetzt. [Kam03]

Erst im Anschluss an die Bewertung und Überprüfung der Qualität können durch die Qualitätsverbesserung entsprechende Maßnahmen definiert werden. Diese können im Anpassen des Entwicklungsprozesses, etwa durch hinzufügen zusätzlicher Quality Gates, in der Definition weiterer zu testender Bereiche des Produkts[22], der Einführung von Code-Reviews oder Ähnlichem liegen.

2.4.2. Normen, Standards und Verfahren

Wie bereits erwähnt, gibt es viele verschiedene Ansätze zum Umgang mit der Qualität, auch zur Verbesserung, Bewertung und des Managements aller Aspekte dieser Thematik. Viele dieser Verfahren wurden ursprünglich nicht speziell für die Softwareentwicklung entworfen, sondern für alle möglichen Bereiche, in denen Qualität von Bedeutung ist. Es ist aber oft möglich, die grundlegenden Ideen zu übertragen. Auf den folgenden Seiten sollen jedoch nur die für diese Arbeit wichtigsten Verfahren kurz vorgestellt werden, für eine umfangreichere Auflistung wird beispielsweise auf [Lig02] oder [Bal98] verwiesen.

Im ersten Abschnitt werden die Standards ISO/IEC 25000, 9126 und 14598, deren Zielsetzungen und Grenzen dargestellt, um in Kapitel 4 das Qualitätsmodell der Standards als Basis zu verwenden. Der Abschnitt über die ISO/IEC 15939 stellt das formale Vorgehen

[22] Gemeint ist hiermit beispielsweise die Einführung von Stabilitäts- oder Stresstests.

dar, aus einzelnen Metriken Informationen zu ziehen. Im Praxisbeispiel in Kapitel 5 wird dies jedoch größtenteils durch das eingesetzte Werkzeug abgedeckt. Der dritte Absatz beschreibt den „Goal Question Metric“-Ansatz, der die Auswahl von Maßen und Metriken unterstützen soll. Diesem Top-Down-Ansatz wird in Abschnitt 3.2 das eigene Vorgehen gegenübergestellt. Die nächsten beiden Abschnitte gehen auf zwei Verfahren ein, anhand denen der Gesamt- bzw. der Testprozess verbessert werden soll – das „Capability Maturity Model Integration“ (CMMI) und das „Test Process Improvement“ nach [PKS00]. Dies geschieht, da beide Verfahren oftmals zitiert werden, wenn es um den Einsatz von Qualitätsmetriken geht. Ebenfalls an dieser Stelle wird auf das *Application of Metrics in Industry* (ami) Verfahren eingegangen, dessen Ziel es ist, die Einführung und Anwendung von Kennzahlen zu unterstützen. Abschließend wird der Umgang mit Softwareanomalien mit Fokus auf deren Klassifikation, wie es der Standard IEEE 1044 vorsieht, beschrieben. Dies ist im Hinblick auf die für den Aufbau des Kennzahlsystems notwendigen Anpassungen im Umgang mit gefundenen Fehlern (siehe Abschnitt 4.2.2) sinnvoll.

International Standards ISO/IEC 25000, ISO/IEC 9126, ISO/IEC 14598

Bereits seit mehreren Jahren arbeitet die ISO an den Nachfolgestandards der ISO 9126 und ISO 14598 [AQDH05]. Diese werden in der Normenreihe ISO/IEC 25000 zusammengefasst und durch sie abgelöst. Der gemeinsame Titel der Reihe lautet *Software Product Quality Requirements and Evaluation* (SQuaRE) und diesem folgend geht die Reichweite über den Rahmen der ISO 9126 und 14598 hinaus. Die Reihe ist unterteilt in fünf Bereiche[23] [ISO05a]:

- *Quality management* Bereich (ISO 2500n)
- *Quality model* Bereich (ISO 2501n)
- *Quality measurement* Bereich (ISO 2502n)
- *Quality requirements* Bereich (ISO 2503n)
- *Quality evaluation* Bereich (ISO 2504n)

[23] Das n in der Nummerierung identifiziert die Version des eigentlichen Standards innerhalb der Reihe.

Eines der Hauptziele des neuen Standards ist die Abstimmung mit der ISO 15939 (siehe S. 41), welche auch im Rahmen der ISO 25000 Normenreihe durchaus noch ihren Sinn und Zweck hat [ISO05a]. Viele der generellen Konzepte der ISO 25000 sind bereits aus anderen Normen bekannt. Beispielsweise hat sich das Qualitätsmodell der ISO 9126 mit der Einführung der neuen Norm nicht geändert. Zusätzlich zu den o. g. Bereichen gibt es auch Erweiterungen der ISO 25000, die meist speziellere Themen behandeln. Als Beispiel kann hier die Norm ISO 25051 [ISO06] genannt werden, die Anforderungen an die Qualität und Anleitungen für den Test von COTS-Produkten behandelt. Im Detail wird an dieser Stelle nicht auf die gesamte ISO 25000 eingegangen werden, da dies zum einen den Rahmen dieser Arbeit sprengen würde und zum anderen nicht in Gänze für diese Arbeit relevant ist.

Der „International Standard ISO/IEC 25010: Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Quality Model“ [ISO05b] bildet den Nachfolger des ersten Teils der ISO 9126. Darin wird ein umfassendes Modell zur Spezifikation und Bestimmung der Qualität von Software-Produkten zur Verfügung gestellt. Der Entwicklungsprozess ist hierbei nicht enthalten; der Rahmen des Standards umfasst ausschließlich die Qualität des Produkts.

In ihm wird ein Qualitätsmodell erarbeitet, das auf verschiedenen Sichtweisen auf die Produktqualität basiert. Diese orientieren sich an drei verschiedenen Ebenen in der Sicht auf die Software. Daraus resultieren ebenso drei verschiedene Arten von Anforderungen: *User quality needs*, *External quality requirements* und *Internal quality requirements*.

Die *User quality needs* können dargestellt werden als Qualitätsanforderungen, die anhand von *Quality in use*-Metriken, externen oder zum Teil internen Kennzahlen definiert werden. Dem Standard nach sollten diese Anforderungen in Form von Kennzahlen als Validierungskriterien verwendet werden.

External quality requirements bilden die externe Sichtweise auf das Softwareprodukt ab und können somit zur Bestimmung des Niveaus der externen Qualität herangezogen werden. Sie stellen die Grundlage für die Validierung auf jeglichen Ebenen des Entwicklungsprozesses dar und sollten im Rahmen der Produktevaluierung überprüft werden.

Zunächst können diese externen Qualitätsanforderungen jedoch in *Internal quality requirements* abgebildet werden. Diese können dann als Grundlage für die Validierung von Entwicklungsartefakten, wie etwa Modellen, Dokumenten oder Quellcode, auf allen Entwicklungsstufen verwendet werden, auch als Grundlage zur Verifikation und zur Erstellung von Entwicklungsstrategien.

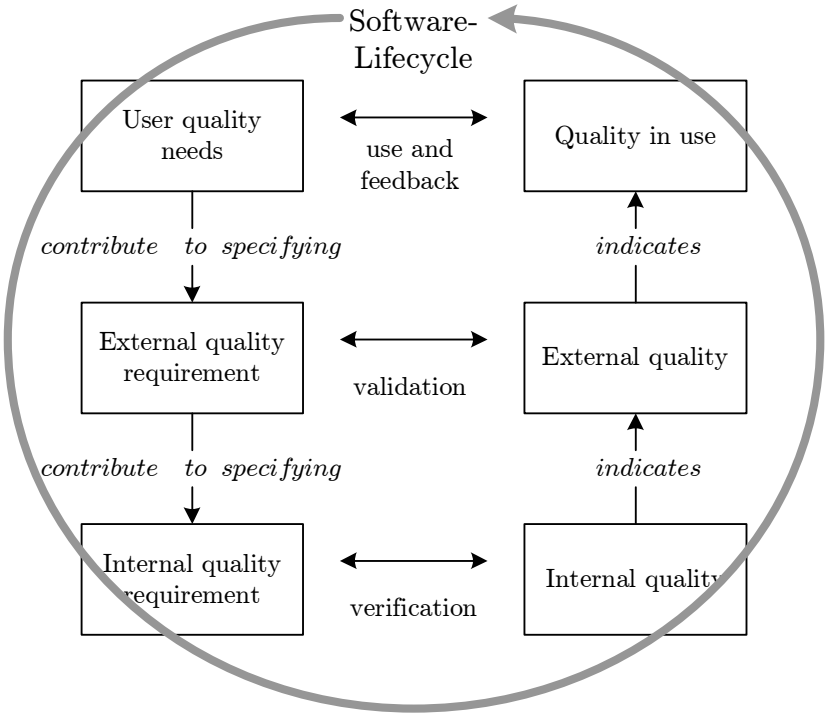

Abbildung 2.6.: Qualität von Software im Softwarelebenszyklus nach ISO 25010 bzw. 9126 [ISO01b; ISO05b].

Diese drei Arten von Anforderungen stehen in direkter Beziehung zu den entsprechenden Eigenschaften des Produkts (siehe Abb. 2.6). Um den Bezug herstellen zu können, muss die Granularität der Betrachtungsweise jedoch noch um eine weitere Stufe verfeinert werden. Diese Stufe wird durch die Qualitätsmerkmale aufgestellt und vervollständigt somit das Qualitätsmodell der ISO 25010. Im Anhang A werden die externen und internen, also beide direkt das Produkt betreffenden, Attribute und Subattribute dargestellt.

Die dritte Gruppe von Attributen bezieht sich auf die *Quality in use*, die „Qualität in Gebrauch“. Sie besteht aus den u. g. vier Attributen, die die Sicht der Benutzer auf die Auswirkungen der externen und internen Attribute widerspiegeln [AKCK05]:

- Effektivität

- Produktivität
- Gefahrlosigkeit[24]
- Zufriedenstellung[25]

Bereits an dieser Stelle gibt es in der Literatur auch andere Modelle. Barry W. Boehms Qualitätsmodell [BBL76] sieht ebenso wie das Modell von Rombach eine unterschiedliche Klassifikation der Qualitätsattribute vor [RB87]. Aber auch auf dem Qualitätsmodell der ISO/IEC 25010 bzw. der ISO/IEC 9126 basierende Standards und Verfahren sind weit verbreitet. In den letzten Jahren entstanden auch leichte Variationen und Anpassungen des Standards. Hervorzuheben ist an dieser Stelle die von Bertoa und Vallecillo erarbeitete Auswahl für Qualitätsattribute von COTS Komponenten [BV02]. Durch leichte Änderungen entsteht somit nicht ein weiterer Standard, sondern das bestehende Gerüst wird lediglich gedehnt. Dadurch behalten andere Standards im Umfeld der ISO 9126 auch weiterhin ihre Anwendbarkeit. So bleibt auch die sechsteilige ISO Norm 14598, „Information technology – Software product evaluation“ [26] , die auf der ISO 9126 aufbaut und sowohl das darin enthaltene Qualitätsmodell als auch die darauf zurückgreifenden externen Metriken verwendet, valide.

Die Normenreihe ISO/IEC 14598 [26] bietet parallel zur oben beschriebenen Norm ISO 9126 einen Überblick über den Prozess der Softwareevaluation. Die beiden Normen sollen ausdrücklich zusammen Anwendung finden und sind aufeinander abgestimmt.

Der erste Teil der Norm ISO/IEC 14598 [ISO99a], *General overview*, beschäftigt sich zunächst mit dem Zusammenspiel aller Dokumente der Standards ISO/IEC 9126 und ISO 14598. In der zweiten Hälfte des ersten Teils des Standards wird ein generischer Evaluationsprozess eingeführt, bevor in weiteren Teilen auf spezifische Blickwinkel eingegangen wird. Teil drei [ISO00b], *Process for developers*, behandelt den Prozess unter dem Gesichtspunkt der Eigenentwicklung der Software, Teil vier [ISO99b], *Process for acquirers*, im Falle des Einkaufs der Software[27] und Teil fünf [ISO98], *Process for evaluators*, aus

[24] Engl. „Safety“.
[25] Engl. „Satisfaction“.
[26] Siehe [ISO99a; ISO00a; ISO00b; ISO99b; ISO98; ISO01a].
[27] Hierunter fällt sowohl der Einkauf eines kompletten Systems, das ohne weitere Anpassung verwendet wird, als auch der Einkauf einer Komponente, eines Teilsystems oder eines fertigen Systems, das mit anderen Softwareprodukten zusammen im Zielsystem integriert wird.

Sicht der Evaluation der Software. Die Teile zwei [ISO00a], *Planning and Management*, und sechs [ISO01a], *Documentation of evaluation modules*, unterstützen die Planung, das Management und die Anwendung der Beschreibungen der Bewertungsprozesse aus den anderen Teilen [Kin03].

International Standard ISO/IEC 15939

Die Norm ISO 15939 [ISO02] definiert Aktivitäten und Aufgaben, die notwendig sind, um einen Softwaremessungsprozess zu implementieren. Der Prozess selbst wird festgelegt anhand seines Zwecks und des zu liefernden Ergebnisses. Das enthaltene Informationsmodell unterstützt die Auswahl der zu spezifizierenden Artefakte. Diese können während der Planungs-, der Durchführungs- oder der Bewertungsphase entstehen.

Auf Basis von zu bewertenden Attributen wird eine spezifisch auf das Attribut angepasste Messmethode angewandt, um Basismesswerte zu sammeln. Auf diesen aufbauend werden einer Berechnungsformel folgend, die eine Maßfunktion darstellt, spezifische Kennzahlen erstellt. Diese werden anschließend im Kontext eines Analysemodells verwendet, um einen Indikator zu erhalten. Diese Indikatoren sind Zahlenwerte, die zusammen auf die spezifischen Informationsanforderungen des Benutzers zugeschnitten sind. [AQDH05].

Goal Question Metric (GQM)

Das *Goal Question Metric*-Verfahren von Basili und Rombach [BR87] soll die Auswahl von einzusetzenden Kennzahlen unterstützen. Als Grundlage dient die Überlegung, dass es von wesentlicher Wichtigkeit ist, Kennzahlen immer für ein bestimmtes Ziel zu definieren [Lig02]. Aus diesem Grund soll einem Top-Down-Ansatz folgend zunächst das Ziel festgelegt werden. Dieses wird anschließend verfeinert, bevor letztlich die einzelnen Metriken aufgestellt werden.

Das GQM-Verfahren unterteilt sich somit in drei Abschnitte. Im ersten werden die Ziele (*Goal*) der Inititative festgelegt. Von ihnen abgeleitet werden die Fragen (*Question*), die im Rahmen des GQM-Verfahrens beantwortet werden sollen. Der dritte und letzte Schritt umfasst die Definition der Kennzahlen (*Metric*).

Das generelle Problem beim Einsatz von Kennzahlsystemen ist, dass es viele verschiedene Kennzahlen und Ziele gibt, die mit ihnen verfolgt werden können. Das GQM-Verfahren setzt genau an dieser Stelle an und versucht, durch die strukturierte Vorgehensweise die wichtigsten Kennzahlen zielgerichtet zu ermitteln. Das Ergebnis des GQM-Verfahrens ist sehr pragmatisch: Es werden nur die Maße gesammelt, deren Auswertung letztlich auch angewandt wird [SB97].

Die GQM-Modelle weisen damit eine hierarchische Struktur auf und werden auf diese Weise auch oftmals grafisch festgehalten (siehe Abb. 2.7) [BCR94]. In dieser Darstellungsform werden sie mitunter auch als Bäume bezeichnet, stellen im Allgemeinen jedoch keine dar, da die Blätter nicht zwangsläufig nur einen Vaterknoten haben müssen.

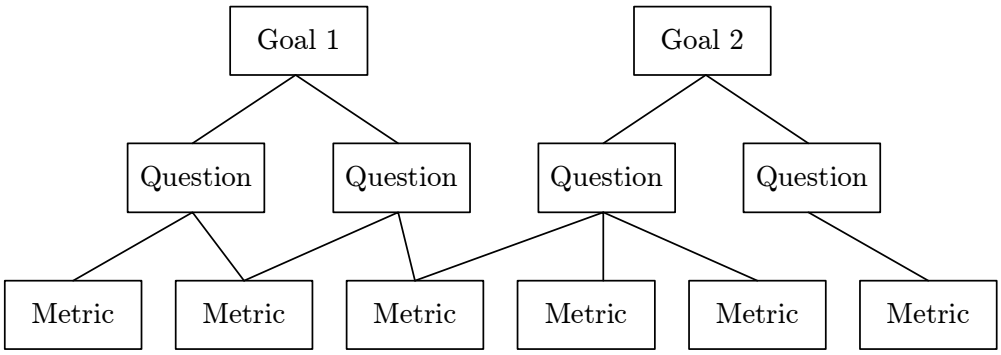

Abbildung 2.7.: *Goal Question Metric*-Modell.

Das GQM-Verfahren fand in den letzten Jahren auch Erweiterungen. Eine nennenswerte ist die Arbeit von van Solingen und Berghout [SB01]. Darin wird der Erfahrungsgewinn über Softwarequalität in den Mittelpunkt gestellt und als Konsequenz werden in Feedback-Sitzungen die Daten des Kennzahlsystems durch das Entwicklungsteam interpretiert. Des Weiteren wird betont, dass in den wenigsten Fällen Kosten-/Nutzenanalysen von Softwaremessprojekten erstellt werden.

Capability Maturity Model Integration (CMMI)

Das *Capability Maturity Model Integration (CMMI)* entstand als Fortführung aus verschiedenen *Capability Maturity Models (CMM)* am Software Engineering Institute (SEI) der

Carnegie Mellon University in Pittsburgh. Genau wie die Vorgänger ist es ein „process improvement maturity model“ [CMM06], ein Modell, das es ermöglicht, den Reifegrad eines Entwicklungsprozesses zu messen und darauf aufbauend auch kontinuierlich zu verbessern. Der Unterschied besteht darin, dass es verschiedenste Ansätze und Reifegradmodelle verbindet und integriert [CKS03; Som04]. Es stellt somit ein Modell für verschiedene Bereiche dar, was es v. a. auch in Industriezweigen beliebt macht, die auf der einen Seite Softwareentwicklungsprozesse, auf der anderen aber auch Entwicklungsprozesse anderer Produkte zu beherrschen haben [Som04]. Zu nennen ist an dieser Stelle beispielsweise die Automobilindustrie.

CMMI untergliedert sich zukünftig in die folgenden drei Modelle[28]: *CMMI for Development*[29] [CMM06], *CMMI for Acquisition*[30] [BGBW05] und *CMMI for Services*[31]. Seit seiner Freigabe im Jahr 2002 entwickelte sich das Modell aufgrund seiner Verbreitung im IT-Umfeld zum de facto Standard. Dies ist erstaunlich, da das Modell sehr komplex ist [Som04]. Ohne diese Komplexität wäre es jedoch nicht so breit gefächert einsetzbar. Die Komplexität des Modells verhindert eine detaillierte Darstellung in dieser Arbeit. Im Folgenden soll ein grober Überblick gegeben werden.

Das CMMI-Modell hat zwei verschiedene Ausprägungen: *Continuous* und *Staged*. Die „Staged“-Version ist identisch mit dem Software-CMM und ermöglicht eine Einstufung der Systementwicklungs- und -managementprozesse in Level 1 bis Level 5. Darin wird ein Projekt, ein Unternehmensbereich oder ein gesamtes Unternehmen einer Stufe zugeordnet. Die folgenden Stufen sind dabei möglich:

- Level 1: *Initial*
- Level 2: *Managed*
- Level 3: *Defined*
- Level 4: *Quantitatively managed*

[28] Zu entnehmen der Homepage von wibas, `http://www.wibas.de` , einem Partnerunternehmen des Software Engineering Institute der Carnegie Mellon University. *CMMI for Development* und *CMMI for Acquisition* sind bereits auf der CMMI Homepage des SEI `http://www.sei.cmu.edu/cmmi/models` erhältlich (zuletzt aufgerufen am 14.09.2008).
[29] CMMI für die Produktentwicklung.
[30] CMMI für Outsourcing-Projekte.
[31] CMMI für Servicedienstleister.

- Level 5: *Optimizing*

Die weniger weit verbreitete *Continuous*-Version erlaubt eine feinere Einteilung auf Stufen von 1 bis 6. Dabei wird nicht nur ein diskreter Wert ermittelt, sondern eine ganze Reihe an Werten für einzelne Prozesse oder Prozessgruppen. [CKS03]

Ein weiterer wichtiger Punkt bei CMMI ist die Möglichkeit der Zertifizierung. Einzelne Unternehmen, Unternehmensbereiche oder auch nur einzelne Projekte können sich in einem sog. *SCAMPI-Appraisal*[32] für eine Stufe zertifizieren lassen. Dabei werden durch die SCAMPI-Methodik die Prozesse auf die fünf bzw. sechs Stufen abgebildet. Für eine genauere Beschreibung wird an dieser Stelle auf die Literatur verwiesen [CKS03; SCA06]. Die Einstufung, auf welchem Level sich die Organisation befindet, sollte durch eine vom SEI autorisierte Person durchgeführt werden. Erst dann ist eine derartige Zertifizierung in der Industrie und vom SEI anerkannt.

Application of Metrics in Industry (ami)

Die ami Methode [PKCS95] entstand, um die Einführung von Kennzahlen in einem Unternehmen oder einem Projekt zu unterstützen. Sie besteht aus vier Aktivitäten, die sich an den Demingkreis[33] [Dem82] anlehnen und wiederum aus jeweils drei Schritten bestehen. Damit ergibt sich ein Verfahren, das die Umsetzung eines Metrikprogrammes in einer Organisation in zwölf Schritten darstellt.

Das Verfahren beinhaltet eine Verfeinerung des GQM-Verfahrens (siehe S. 41) und übernimmt dadurch seinen zielorientierten Top-Down-Charakter [PKCS95, S. 50]. Die ersten Schritte sehen eine Ist-Analyse und eine Definition der Ziele vor, die daraufhin verfeinert werden. Der anschließenden Kennzahldefinition und Datensammlung folgt eine Interpretationsphase und eine Feedback-Schleife, um Verbesserungsmöglichkeiten am Kennzahlsystem zu identifizieren. Besonderer Wert wird auf Schritte zur Validierung und Verifikation der Ziele, Kennzahldefinitionen und die gesammelten Daten gelegt.

Das ami-Verfahren beinhaltet außerdem sehr viele Querverweise auf CMM, den Vorgänger von CMMI, und empfiehlt den Einsatz eines CMM(I) Appraisals.

[32] SCAMPI steht für „Standard CMMI Appraisal Method for Process Improvement“.
[33] Auch bekannt als PDCA-Zyklus (**P**lan, **D**o, **C**heck, **A**ct).

Im Folgenden werden die oben angesprochenen zwölf Schritte kurz dargestellt.

- Assess
  - *Assess your environment*: Der erste Schritt besteht aus der Bewertung des Ist-Zustands, der Beschreibung der Entwicklungsprozesse und der Projektumgebung. Das ami-Verfahren empfiehlt, für diesen Schritt ein CMM(I) Appraisal (siehe S. 42) durchzuführen.
  - *Defining primary goals*: Auf Basis der vorangegangenen Analyse werden Primärziele definiert. Diese werden eingeteilt in *knowledge goals* oder *change or improvement goals*. Erstere werden hauptsächlich auf den unteren CMM(I)-Stufen angewandt, um mehr über das Projekt, das Produkt oder den Prozess herauszufinden und können somit als tiefergehende Ist-Analyse aufgefasst werden. Letztere finden auf höheren CMM(I)-Stufen Anwendung und beinhalten eine Veränderung oder Verbesserung des Projekts, des Produkts oder des Prozesses.
  - *Validating primary goals*: Für jedes Primärziel werden in diesem Schritt die folgenden Punkte geklärt: Relevanz, Detaillierungsgrad, Machbarkeit und der zeitliche Rahmen.
- Analyze
  - *From goals to sub-goals*: Anhand eines am GQM-Verfahren (siehe S. 41) angelehnten Vorgehens werden die Primärziele in Sub-Ziele verfeinert. Dies kann auch auf weitere Ebenen ausgeweitet werden, wodurch dann Sub-Sub-Ziele, etc. entstehen können. Unterstützt werden kann dieses Vorgehen durch die Auflistung der jeweils betroffenen Entitäten und deren Attribute.
  - *Verifying the goal tree*: Im folgenden Schritt wird der entstandene „Zielbaum“ überprüft. Dies geschieht mit Blick auf die Balance des Baums und die interne und externe Konsistenz. Interne Konsistenz bedeutet, dass die Sub-Ziele im Einklang mit ihren übergeordneten Zielen stehen und es keine widersprüchlichen Sub-Ziele in verschiedenen Zweigen gibt. Unter externer Konsistenz wird die Überprüfung der Relevanz und Widerspruchsfreiheit der Sub-Ziele bzgl. des jeweiligen Primärziels verstanden.

  - *From sub-goals to metrics*: Zu allen Blättern des Baums, d. h. allen Sub-Zielen auf der jeweils untersten Ebene, werden Kennzahlen erstellt. Hierbei können die vorher evtl. aufgelisteten Entitäten und Attribute von Hilfe sein.

- Metricate
  - *The measurement plan*: In einem von ami empfohlenen Template werden alle bislang gesammelten Informationen zusammengetragen und validiert. Dieser *measurement plan* enthält auch Angaben über die Datenverfügbarkeit und eine Auflistung aller zur Datensammlung benötigten Werkzeuge und Verfahren.
  - *Collecting the data*: Möglichst automatisiert werden nun die Basisdaten für die Kennzahlen gesammelt. Dieser Schritt kann sich über längere Zeit hinziehen, wobei bereits teilweise mit dem nächsten Schritt begonnen werden kann.
  - *Verifying the data*: Das Verifizieren der Daten beinhaltet zum größten Teil die Überprüfung, ob die Daten im richtigen Format, in der benötigten Granularität und korrekt gesammelt wurden.

- Improve
  - *Presenting and distributing the data*: Der Zielbaum wird nun von unten nach oben durchlaufen und auf jeder Stufe werden alle Daten mit allen vom jeweiligen Ziel Betroffenen durchgesprochen. Die Datenaufbereitung sollte hierbei grafisch und nicht mit statistischen Methoden geschehen.
  - *Validating the metrics*: In diesem Schritt sollen Fragen wie „Stimmen die Daten mit den Erwartungen überein?“ beantwortet und eventuellen Ungereimtheiten nachgegangen werden.
  - *Relating data to goals*: Schlussendlich erfolgt eine Analyse der Daten und die Ausarbeitung der Implikationen für das jeweilige Primärziel. Hierbei ist es von entscheidender Bedeutung, die Kennzahlen nicht isoliert zu betrachten, sondern stets auch deren Umfeld und alle betroffenenen Faktoren mit einzubeziehen.

Test Process Improvement (TPI)

Das Ziel des TPI-Modells der Sogeti-Unternehmensgruppe [Sog06] ist, wie der Name bereits sagt, die Verbesserung des Testprozesses. Hierzu soll zunächst der aktuelle Testprozess analysiert werden, um die Stärken und Schwächen offenzulegen. Darauf aufbauend kann eine Art Reifegrad des Testprozesses bestimmt werden.

Beim TPI-Modell werden 20 Kerngebiete betrachtet, denen eine besondere Aufmerksamkeit zukommen soll. Zu den Kerngebieten gehören klassische, auf den Prozess bezogene Aspekte, wie die Teststrategie oder das Phasenmodell, und technische Fragestellungen, wie die Auswahl von Kennzahlen und Testtools. Aber auch auf den ersten Blick weniger greifbare Dinge, wie das Engagement und die Motivation der Mitarbeiter oder die unternehmensinterne Kommunikation, fließen in die Bewertung mit ein.

Jedes dieser Kerngebiete wird einer bestimmten Stufe oder Ebene zugeteilt. Um zu bestimmen, auf welcher Ebene sich der Testprozess im entsprechenden Kerngebiet befindet, müssen bestimmte Anforderungen erfüllt werden. In manchen Fällen bedingen sich verschiedene Kerngebiete gegenseitig. Beispielsweise könnte eine Voraussetzung für das Erreichen von Ebene B in Kerngebiet X sein, zunächst in Kerngebiet Y auf Ebene C zu stehen.

Das „objektive Messinstrument“, anhand dessen die Ebenen bestimmt werden, sind als Fragen formulierte Anforderungen, sog. Kontrollpunkte. Um die jeweils nächsthöhere Ebene zu erreichen, müssen alle Fragen positiv beantwortet werden. Jede Ebene stellt eine Verbesserung im Vergleich zur darunterliegenden dar. Dadurch müssen für das Erreichen einer bestimmten Ebene auch alle Kontrollpunkte aller darunterliegenden Ebenen positiv bewertet werden. [Sog06]

Anschließend an die Bestimmung der Ebene jedes Kerngebiets werden die Schritte zur Verbesserung des Prozesses ausgewählt. Dazu wird zunächst die Entwicklungsmatrix ausgefüllt (siehe Tabelle 2.1). Vertikal sind die 20 Kerngebiete aufgeführt und horizontal 14 Entwicklungsskalen. Jede der drei oder vier Ebenen jedes Kerngebiets ist einer Entwicklugsskala zugeordnet. Beispielsweise entspricht die Ebene A im Kerngebiet Teststrategie der Entwicklungsskala 0. Zwischen den Ebenen der verschiedenen Kerngebiete existieren

einige Abhängigkeiten. Als Voraussetzung für das Erreichen einer Ebene, d. h. das Bestehen des Kontrollpunktes, ist oftmals die Erfüllung aller Kriterien eines Kontrollpunktes in einem anderen Kerngebiet gefordert. Die Ebene B ist beispielsweise im Kerngebiet Berichterstattung eine Voraussetzung für das Erreichen der Ebene A im Kerngebiet Metriken. [KP03]

Die in Tabelle 2.1 hellgrau hinterlegten Zellen spiegeln beispielsweise einen möglichen aktuellen Stand wider. Die dunkelgraue Fläche zeigt eine Verbesserungsmöglichkeit. Es gibt keine generelle Vorschrift, an welcher Stelle in welcher Situation das Erreichen der nächsthöheren Stufe anzustreben ist, in der Regel sollte jedoch von links nach rechts vorgegangen werden. Mit anderen Worten ist es meist empfehlenswert, zunächst eine Verbesserung der am wenigsten entwickelten Kerngebiete voranzutreiben [Sog06]. Dies liegt v. a. darin begründet, dass die Bereiche oftmals stark voneinander abhängen.

Skala → **Kerngebiet ↓**	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Teststrategie	A					B				C		D		
Metriken						A			B			C		D
Berichterstattung		A			B		C					D		

Tabelle 2.1.: Ausschnitt aus einer TPI-Entwicklungsmatrix

Ein Aspekt des TPI-Modells sind die sehr spezifischen Anleitungen, die für das Erreichen der Ebenen nötig sind. An dieser Stelle unterscheidet es sich vom CMMI, mit dem es oftmals verglichen wird (obwohl der Fokus von CMMI wesentlich breiter gefasst ist). Auch beim TPI-Modell ist eine Zertifizierung für die verschiedenen Ebenen möglich.

International Standard IEEE 1044

Die Dokumentation der während der Entstehung eines Softwaresystems auftretenden Abweichungen, beispielsweise der im Test entdeckten Fehler, wird durch den Standard IEEE 1044 („Standard Classification for Software Anomalies“ [IEE93]) beschrieben. Der Standard sieht für die Klassifikation vier nacheinander durchzuführende Schritte vor: Erkennung, Analyse, Behebung und Abschluss. In jedem dieser Schritte müssen drei Aufgaben

erledigt werden: Dokumentation des Ist-Zustands, Klassifikation und Bestimmung der Auswirkungen.

Die Dokumentation soll in jedem der vier Schritte genau den beobachteten Zustand beschreiben. Dies umfasst auch Aspekte wie beispielsweise eine Beschreibung der Umgebung auf der ein Testfall durchgeführt wurde, den Versionsstand des getesteten Systems, Zeitpunkte der Durchführung des Tests, etc. Die Klassifikation erfolgt bei allen vier Schritten anhand von im Standard definierten Kassifikationsschemata. Diese bestehen aus verpflichtenden und optionalen Attributen und Ausprägungen. Die Bestimmung der Auswirkungen erfolgt im ersten Schritt, bei der Erkennung der Anomalie, und wird in den darauffolgenden Schritten jeweils erneut überprüft und ggf. korrigiert.

Der Standard ist sehr umfassend, kann jedoch bei seinem Einsatz an die spezifischen Rahmenbedingungen des Projektes, des Produktes oder des Prozesses angepasst werden [SRWL06]. Die Anwendung wird durch einen „Guide to Classification for Software Anomalies“ erleichtert.

2.4.3. Zusammenfassung

Die in diesem Abschnitt vorgestellten Verfahren und Standards sind grundlegend verschieden. Sie verbindet jedoch stets das Ziel, die Qualität der Software zu verbessern oder zu bewerten. Dies kann direkt durch die Qualitätsmessung anhand von Normen wie der ISO 25010 geschehen oder indirekt durch die Verbesserung des Testprozesses nach TPI.

Die Normen ISO 9126 und ISO 14598 sind explizit dafür gedacht, in Kombination angewandt zu werden. Die ISO 9126 liefert dabei das Qualitätsmodell, aufgebaut aus einzelnen Qualitätscharakteristiken, und einem Satz an Kennzahlen, anhand derer die Qualität des Produkts aus verschiedenen Blickwinkeln bewertet werden kann. Die ISO 14598 geht auf den Prozess der Evaluation der Software ein und definiert diesen.

Nicht ohne Weiteres kombinierbar mit diesen beiden Standards ist die Norm ISO 15939, die den Softwaremessungsprozess vorgibt. Die Nachfolgenormenreihe der Standards ISO 9126 und ISO 14598, ISO 25000, beinhaltet nicht nur einige Erweiterungen der Ausgangsstandards, sondern die Kombinierbarkeit mit der ISO 15939 war eines der Hauptziele der

Entwicklung. Letztere Norm behält somit auch nach der Einführung der ISO 25000 im Gegensatz zur ISO 9126 und ISO 14598 ihre Gültigkeit.

Das GQM-Verfahren beschreibt einen Top-Down-Ansatz anhand dessen ein Kennzahlsystem aufzubauen ist. Mit Hilfe von gezielten Fragen sollen die auf die Ziele des Kennzahlsystems ausgerichteten Kennzahlen ausgewählt werden. Dieser Ansatz ist bedingt mit den ISO-Normen vereinbar, da diese einerseits nicht vorschreiben, auf welche Art und Weise die Metriken ausgewählt werden sollen, andererseits jedoch die ISO/IEC 25010 bzw. 9126 beispielsweise nicht berücksichtigen, nur einen Teil der Qualitätsattribute zu betrachten. Auch eine Priorisierung der verschiedenen Attribute sieht keines der hier vorgestellten Verfahren vor. Hervorzuheben beim GQM-Verfahren ist die Analyse der verschiedenen Qualitätsmerkmale. Diese ist oftmals von wesentlicher Bedeutung, da nur dadurch präzise festgelegt werden kann, welche Aspekte der Qualität von Bedeutung sind und explizit bewertet werden sollen.

Auch wenn beim Einsatz von Metriken oft die Abkürzung „CMMI“ ins Gespräch gebracht wird, hält sich das Reifegradmodell bedeckt, wenn es um die konkrete Auswahl von Metriken geht. Dies ist jedoch nicht der Kernpunkt des Modells. Vielmehr soll der Entwicklungsprozess im Allgemeinen bewertet werden. Ein Vorteil des oftmals relativ abstrakten Niveaus ist, dass CMMI nicht nur auf Softwareentwicklungsprozesse angewandt werden kann, sondern auch für die Herstellungs- und Entwicklungsprozesse von Gesamtsystemen und Hardware geeignet ist. Daraus ergibt sich die Möglichkeit, Prozesse zu bewerten, deren Endprodukt sowohl Software als auch andere Komponenten enthält. Dies ist beispielsweise in der Entwicklung von eingebetteten Systemen von großem Vorteil. Daraus resultiert unter Umständen auch die weite Verbreitung des Modells und seines Status als de facto Standard. Der Anwender wählt bei CMMI die für ihn relevanten Module aus.

Ähnliches gilt für TPI, das ebenfalls ein Verfahren darstellt, das den Einsatz von Metriken zwar miteinbezieht, diesen jedoch nicht als (einzigen) Kernpunkt verinnerlicht. Im Fokus liegen vielmehr die Bewertung und die Verbesserung des Testprozesses. Um dieses Ziel zu verfolgen, werden verschiedene Kerngebiete des Testprozesses bewertet und relativ konkrete Verbesserungsvorschläge gemacht. In die Bewertung der einzelnen Kerngebiete fließen auch Wechselwirkungen zwischen diesen mit ein.

Für die Dokumentation aller während der Entstehung und der Weiterentwicklung eines Softwaresystems aufgetretenen Anomalien stellt der IEEE Standard 1044 einen umfassenden Rahmen dar. Dieser umfasst nicht ausschließlich die Klassifikation, sondern beschreibt auch eine strukturierte Vorgehensweise beim Umgang mit gefundenen Anomalien.

2.5. Qualitätssicherung – Softwaretest

2.5.1. Aufgaben der Qualitätssicherung

In diesem Abschnitt soll auf die Softwarequalitätssicherung (QS) eingegangen werden. In der Praxis wird darunter im Wesentlichen oftmals der Softwaretest verstanden, obwohl dieser lediglich eine Möglichkeit ist, einen Teilbereich der QS abzudecken. Die Qualität zu sichern bedeutet zunächst die Qualität zu überprüfen. Und dies wird in der Regel anhand von Softwaretests umgesetzt. Hier soll ebenfalls nicht umfassend auf die QS eingegangen werden, sondern hauptsächlich auf den Softwaretest.

Die QS umfasst aber zusätzlich die Sicherung der Prozessqualität. Dies kann in der Regel nicht ausschließlich durch Softwaretests durchgeführt werden. Zusätzliche Aktionen sind hierfür notwendig. Der Aufgabenbereich der QS ist dadurch breiter, als er in vielen Fällen in der IT-Praxis umgesetzt wird.

Auch die Auffassung des Qualitätsbegriffs beeinflusst die QS. Ein breiter Qualitätsbegriff erweitert die QS z. B. durch die Interpretation der Planbarkeit oder der Einhaltung des Liefertermins als Qualitätsmerkmal.

Die QS kann anhand des Objekts, dessen Qualität es zu sichern gilt, eingeteilt werden in die QS von Prozessen und Produkten. In der Regel wird jedoch eine andere Klassifikation verwendet: die Aufteilung der QS anhand der Maßnahmen, die zu ihrem Zweck ergriffen werden. Dabei unterscheidet man im Allgemeinen zwischen konstruktiven und analytischen QS-Maßnahmen [Wal90]. Zu konstruktiven QS-Maßnahmen zählen Entwicklungsmethoden, wie das Prototyping, das V-Modell oder das Spiralmodell. Auch die Auswahl der Programmiersprache und -umgebungen und die Schaffung eines Entwicklungsrahmens durch das Software-Konfigurationsmanagement zählen zu den konstruktiven Maßnahmen. Analytische QS-Maßnahmen beinhalten statische und dynamische Prüfungen sowohl des Softwareprodukts als auch des Prozesses oder anderer Faktoren, die Einfluss auf die Entwicklung haben. Zu statischen Prüfungen zählen Reviews des Codes und aller anderer Artefakte, wie beispielsweise der Dokumentation oder der Entwicklungsprozessbeschreibungen [Sch03b]. Unter dynamischen Verfahren versteht man im Allgemeinen den Softwaretest.

2.5.2. Grundlagen des Softwaretests

Die Hauptaufgabe des Testens ist nicht, die Fehlerfreiheit eines Softwareprogramms zu gewährleisten bzw. die Abwesenheit von Fehlern zu zeigen, sondern vielmehr das Gegenteil. Ein Test wird durchgeführt, um Fehler zu finden. Bereits im Jahre 1970 stellte Dijkstra fest, dass das Testen niemals die Abwesenheit von Fehlern beweisen kann, sondern stets nur die Anwesenheit durch das Provozieren von Fehlfunktionen aufdecken kann [Dij70]. Dies ist auch ein grundlegender Fakt im Standardwerk „The Art of Software Testing“ von Myers aus dem Jahre 1979 [Mye79].

Eine sehr gute Beschreibung der Einstellung zum Testen liefert Myers in „Methodisches Testen von Programmen“ [Mye99]. Er vergleicht die Durchführung von Tests mit der Durchführung einer Untersuchug durch einen Arzt: Ein Test ist erfolgreich, wenn ein Fehler gefunden wurde, ebenso wie die ärztliche Untersuchung erfolgreich ist, wenn das Magengeschwür gefunden wurde. Dies mag auf den ersten Blick eigenartig erscheinen, aber nicht nur Myers widmet ein ganzes Kapitel der Psychologie des Testens [Mye79; SL04].

Damals wie heute ist das Testen fast schon eine der „dark arts“ der Softwareentwicklung [MSBT04]. Die Anzahl der gefundenen Fehler zeigt jedoch die Notwendigkeit, die erstellten Programme stets genauer unter die Lupe zu nehmen. Eine sehr „beliebte“ Aufgabe ist das Testen dennoch nicht. Der Grundantrieb ist destruktiver Natur, der Tester steht meistens als Überbringer schlechter Nachrichten da [SL04]. Doch bei genauerem Hinsehen ist es immer besser, wenn der Tester den Fehler findet als ein Anwender der Software.

Die Sichtweise auf die Tests und die Tester hat sich somit in den letzten Jahrzehnten nicht sehr verändert. Doch auch die damaligen Methoden zur Auswahl von Testfällen etc. finden heute noch Anwendung. Hinzugekommen ist in den letzten Jahren spezielle Literatur über das Testen bestimmter Systeme, Architekturen, etc. „Testing object-oriented systems“ von Binder [Bin00] und „Testmethoden für sequentielle und nebenläufige Software-Systeme“ von Riedemann [Rie97] sind zwei prominente Beispiele.

Die grundlegende Aufgabe des Testens ist es, Fehler zu finden. Bevor dies in Angriff genommen werden kann, muss zunächst geklärt werden, was unter einem Fehler verstanden wird. Im Englischen wird in der Literatur zum Thema Softwaretest meist nicht nur ein

Begriff verwendet, sondern mehrere – *bug, defect, error, failure, fault.* Auch die deutsche Literatur hat sich daran (sinnvollerweise) angepasst und verwendet verschiedene Begriffe, deren Unterschiede leider nicht bereits aus dem Sprachgebrauch deutlich zu erkennen sind. Die folgenden Definitionen richten sich nach internationalen Standards [IEE90; DIN95b] und sind auch im deutschsprachigen Raum verbreitet. Die hier dargestellten Definitionen sind Liggesmeyer [Lig02, S. 504–506] entnommen.

Definition: Fehlverhalten (*failure*)

„Ein Fehlverhalten oder Ausfall zeigt sich dynamisch bei der Benutzung eines Produkts. Beim dynamischen Test einer Software erkennt man keine Fehler, sondern Fehlverhalten bzw. Ausfälle. Diese sind Wirkungen von Fehlern im Programm. Bei Hardware sind z. B. Ausfälle durch Verschleiß möglich."

Definition: Fehler (*fault, defect*)

„Ein Fehler oder Defekt ist bei Software die statisch im Programmcode vorhandene Ursache eines Fehlverhaltens oder Ausfalls. Bei Hardware sind ebenfalls Konstruktionsfehler möglich (z. B. eine zu schwache Dimensionierung der Kühlung eines Leistungshalbleiters)."

Definition: Irrtum (error)

„Die Ursache eines Fehlers kann ein Irrtum des Programmierers oder ein einfacher Tippfehler sein. Ein Irrtum kann z. B. ein falsches Verständnis der Wirkung eines bestimmten Konstrukts der verwendeten Programmiersprache sein."

Diese drei Begriffe stehen oftmals in enger Beziehung zueinander. Ein während der Ausführung beobachtetes Fehlverhalten kann auf einer fehlerhaften Zeile im Code basieren. Diese wiederum kann durch einen Irrtum entstanden sein. Um das Fehlverhalten nun zu korrigieren, muss zunächst der Fehler gefunden und anschließend verbessert werden. Dies geschieht dadurch, dass der Irrtum beseitigt wird.

In der Praxis wird oftmals auch im deutschsprachigen Raum an Stelle des Begriffs „Fehler" der englische Ausdruck „Defect" verwendet. Auch in dieser Arbeit wird an einigen Stellen auf den Anglizismus zurückgegriffen.

Ein Fehler hat immer eine ganze Reihe an Attributen, die sich oftmals unterscheiden und bei deren Strukturierung dem Testmanagement meist freie Hand gewährt wird. Das wichtigste Attribut ist wohl die Schwere eines Fehlers, die aus Sicht des Anwenders anhand einer Klassifikation des gefundenen Problems den Grad der Beeinträchtigung des Produkteinsatzes darstellen soll. Mögliche Attributsausprägungen sind Klassifikationen der Art „Show Stopper“, „High“, „Medium“, „Low“. Wichtig ist an dieser Stelle eine Definition der Bedeutung einer entsprechenden Klassifizierung.

Dieses Attribut ist nicht zu verwechseln mit der Priorität eines Fehlers. Diese gibt an, wie wichtig und dringend die Behebung des Fehlers ist. Beispielsweise kann in einem Softwareprodukt zur Bearbeitung von Bestellungen die Freigabe durch den Einkauf oder den Vorgesetzten anhand einer Ampel geregelt werden. Ein nicht schwerwiegender Fehler höchster Priorität wäre das Vertauschen von rot und grün in der Darstellung der Ampel. Der Produkteinsatz wird dadurch nicht beeinträchtigt (das Softwareprodukt funktioniert an sich einwandfrei); die falsche Darstellung der Farben stellt jedoch ein hohes Risiko für den reibungslosen Arbeitsablauf beim Anwender dar, da Verwechslungen und Missverständnisse im wahrsten Sinne des Wortes vorprogrammiert sind.

Es gibt in den meisten Fällen eine ganze Reihe an weiteren Attributen, wie etwa Datum und Uhrzeit der Fehlerentdeckung, eine Beschreibung des aufgedeckten Problems oder der Name des Testers. Der Status eines Fehlers (*neu*, *in Bearbeitung*, *behoben*, etc.) hebt sich hierbei besonders ab, da er meist den Umgang mit Fehlern regelt und die im Einsatz befindlichen Merkmalsausprägungen eng mit dem Test- und Entwicklungsprozess verbunden sind.

2.5.3. Allgemeiner Testprozess

Das Testen von Software kann sehr unterschiedlich sein – bzgl. benötigter Gesamtzeit, Aufwand, Komplexität etc. Ein großes, verteiltes, Mehrbenutzer-Softwaresystem hat andere Anforderungen an den Test als ein Programm zur Passwortverwaltung, das jeweils nur von einer Person verwendet wird. Für Ersteres müssen auf jeden Fall Performance- und Stabilitätstests geplant werden. Auch komplexe Szenarien, in denen mehrere Benutzer gleichzeitig auf das System zugreifen, müssen berücksichtigt werden. Dies ist für das

zweite Programm nicht von derart großer Bedeutung. Hier darf jedoch eine ausgedehnte Überprüfung der Sicherheit nicht fehlen.

Durch die unterschiedlichen Testobjekte entstanden auch unterschiedliche Testarten und -verfahren. All diese Verfahren haben dennoch Gemeinsamkeiten. Der hier dargestellte Testprozess nach dem weit verbreiteten britischen Standard BS 7925-2 [BS798b] (siehe Abb. 2.8) trifft auf alle in gleicher Weise zu und jeder systematische Softwaretest wird so oder so ähnlich durchgeführt. Im Folgenden werden die einzelnen Phasen kurz vorgestellt.

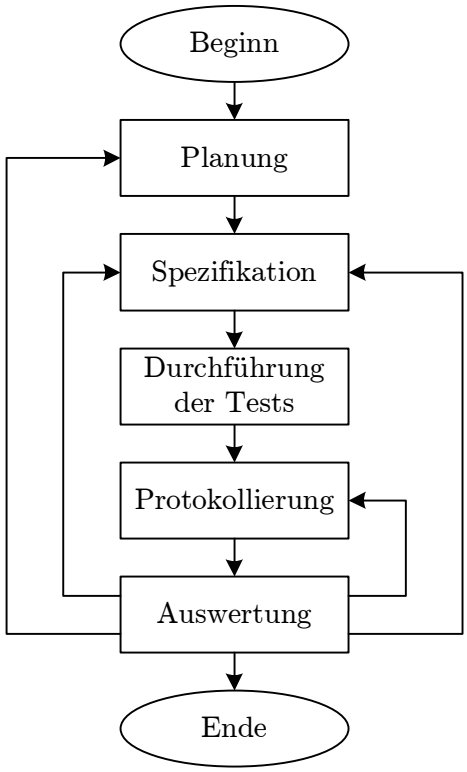

Abbildung 2.8.: Testprozess nach dem britischen Standard BS 7925-2[BS798b].

Planung

Bereits zu Beginn des kompletten Entwicklungsprozesses wird auch mit der Planung der Tests und anderer Maßnahmen zur Qualitätssicherung begonnen. Der grobe Entwurf wird in einem Qualitätssicherungsplan, beispielsweise nach der Norm IEEE 730 [IEE98b], festgehalten. Dieser enthält auch Angaben zu statischen Qualitätssicherungsmaßnahmen, dem generellen Umgang mit gefundenen Mängeln, der Organisationsstruktur, etc. [SL04]

Im Rahmen der Testplanung wird ein Testkonzept, etwa anhand der Norm IEEE 829 [IEE98c], erstellt. Dieses enthält zunächst die zu testenden Objekte und weitere Leistungsmerkmale des zu entwickelnden Produkts. Auch die Funtionalitäten oder Leistungsmerkmale, die nicht getestet werden, werden hier explizit genannt. In der Teststrategie wird darauf eingegangen, welche Teile mit welcher Intensität getestet werden sollen. Dies orientiert sich oftmals an einer Priorisierung der einzelnen Funktionalitäten. Es wird beispielsweise festgelegt, dass sicherheitskritische Teilbereiche umfassender auf einen möglichen Missbrauch überprüft werden müssen als andere Teilbereiche. Auch die Festlegung der Abnahmekriterien und der Testendekriterien sind Teil des Testkonzepts. Die Festlegung der Testumgebungen und der organisatorischen Rahmenbedingungen, der Festlegung der Verantwortlichkeiten, des Personals, des Zeit- und sonstigen Ressourcenbedarfs, etc. und eine Betrachtung des verbleibenden Risikos bilden den Abschluss des Testkonzepts.

Spezifikation

In der Spezifikationsphase wird das Testkonzept konkretisiert und durch das Erstellen von Testfällen mit Leben gefüllt. Die angewandten Testmethoden unterscheiden sich hierbei je nach Teststufe (siehe Abschnitt 2.5.5). Auf der Testbasis aufbauend, werden die Testfälle in Form von manuell auszuführenden Schritten oder später automatisch ablaufenden Skripten erstellt. Die Testbasis kann aus der Spezifikation (bei *Blackbox-Tests*) oder aus dem Quelltext bestehen (bei *Whitebox-Tests*). Auch Mischformen sind hierbei möglich.

Ein Testfall besteht immer aus mehreren Bestandteilen. Zum einen hat er bestimmte Testvoraussetzungen – beispielsweise muss das zu testende Objekt installiert und ausführbar oder bestimmte Ports müssen freigeschaltet sein. Es werden in der Regel nicht alle Testvoraussetzungen explizit aufgeschrieben, die meisten sind selbstverständlich[34]. Ebenfalls zur Testfallspezifikation gehören die *Testdaten*. In vielen Fällen kann durch eine Variation dieser ein neuer Testfall entstehen. Beispielsweise beim Test eines Logins bei einer Webapplikation kann durch Eingabe von gültigem Benutzernamen und Passwort, gültigem Benutzernamen und falschem Passwort, gültigem Benutzernamen, gültigem Passwort

[34] Hierunter fallen beispielsweise Voraussetzungen wie „Um den Test der Software durchführen zu können, muss der Rechner, auf dem diese installiert ist, eingeschaltet sein.“

und unzureichenden Benutzerberechtigungen, etc. eine ganze Reihe an Funktionalitäten getestet werden.

Ein weiterer Bestandteil der Testspezifikation können benötigte Testapplikationen oder Testtreiber sein. Diese sind in der Regel bei Tests einzelner Komponenten notwendig. Als Beispiel kann man sich die Tests einer Benutzeroberfläche vorstellen. Diese können auch ohne die Implementierung der darunterliegenden Anwendungslogik durchgeführt werden. Dazu können Testtreiber mit den Schnittstellen der Benutzeroberfläche kommunizieren und beispielsweise durch die Rückgabe von statischen Werten die Applikation simulieren.

Zuletzt muss auch das erwartete Verhalten des Testobjekts in der Testspezifikation festgehalten werden. Im Falle von automatisierten Tests wird dies oftmals im Skript codiert, bei manuellen Tests in Worten beschrieben. Bei Lasttests ist das erwartete Verhalten in vielen Fällen durch nichtfunktionale Anforderungen beschrieben. Diese sind in der Regel jedoch nicht umfassend, da das Systemverhalten bei der Testdurchführung nicht genau im voraus geplant werden kann.

Oftmals werden bei der Erstellung der Testspezifikation bereits Fehler in der Software oder deren Design gefunden, da dabei das Systemverhalten erneut gedanklich nachvollzogen werden muss.

Durchführung der Tests

Der Durchführung der Tests wird im British Standard BS 7925-2 [BS798b] genau dieser Satz gewidmet: „*Each test case defined in the component test specification shall be executed.*“ Es gibt hierbei jedoch noch weitere zu beachtende Punkte.

In der Literatur wird meist darauf hingewiesen, dass zumindest auf höheren Teststufen der Tester und der Entwickler des zu testenden Codes nicht ein und dieselbe Person sein sollten. Begründet wird dies meist dadurch, dass die wenigsten Menschen genügend Abstand zu ihrer eigenen Arbeit herstellen können, um deren Ergebnisse kritisch unter die Lupe zu nehmen [SL04].

Generell sollten die Tests so entwickelt worden sein, dass sie von „jedem“ ausgeführt werden können. Diese Regel hat jedoch die eine oder andere Ausnahme: Bei Last- und

Failovertests sollte der Tester stets wissen, was das Testskript genau macht bzw. welche Konsequenzen ein manueller Schritt möglicherweise hat.

Die Reihenfolge, in welcher die Tests ausgeführt werden, sollte im Testplan festgelegt sein. Es ist sinnvoll, zunächst die Testbarkeit des Systems festzustellen. Dies wird oft in sog. *Handovertests* gemacht. Anschließend werden grundlegende Funktionen zuerst getestet, da ohne deren korrekte Funktionsweise oftmals einige darauf aufbauende Funktionen und deren Tests nicht ausgeführt werden können. Die übrigen Tests können in der Reihenfolge ihrer Priorität durchgeführt werden.

Protokollierung

Jede Durchführung eines Tests muss genau, verständlich, nachvollziehbar und korrekt protokolliert werden. Zunächst muss anhand dieses Protokolls auch für nicht direkt an der Entwicklung beteiligte Personen der Test nachvollziehbar sein. Dies könnten Kunden sein; in manchen Fällen haben die Tests aber auch rechtliche Relevanz.

Die Aufzeichnungen für jeden Testfall sollen eindeutig das getestete Objekt und die Testspezifikation festhalten. Das tatsächliche Testergebnis sollte ebenfalls mit aufgezeichnet werden. Das Ziel ist, anhand eines dieser drei Artefakte die beiden anderen dazugehörigen identifizieren zu können. Mit anderen Worten soll beispielsweise anhand der Aufzeichnungen eines Tests festgestellt werden können, ob, wann, für welches zu testende Objekt und mit welchem Ergebnis ein bestimmter Testfall durchgeführt wurde. Nicht nur bei der Analyse gefundener Fehler können diese Informationen von großem Nutzen sein.

Das tatsächliche Ergebnis eines Tests wird stets mit dem erwarteten Ergebnis verglichen. Jede gefundene Abweichung wird aufgezeichnet und analysiert. Es soll festgestellt werden, wo der Fehler liegt und welches die frühestmögliche Testaktivität ist, die bei einem Re-Test ausgeführt werden muss. Letzteres geschieht vor dem Hintergrund, dass für jeden Fehler, der durch eine Änderung[35] korrigiert wird, der entsprechende Testfall erneut ausgeführt werden muss. Auf Re-Tests wird in Abschnitt 2.5.4 vertieft eingegangen.

Nach und während der Testdurchführungsphase ermöglicht eine genaue Protokollierung

[35] Diese Änderung kann sowohl im Code des zu testenden Objekts als auch im Test selbst durchgeführt werden.

der Tests eine Überprüfung, wie weit bzw. inwiefern die in der Teststrategie geplanten Testziele erfüllt werden und bereits wurden. Dies kann anhand der Auswertung der Testabdeckung geschehen. [BS798b]

Auswertung

Nach der Durchführung eines Testfalls wird das dazugehörige Testprotokoll analysiert und es wird überprüft, ob der Testfall bestanden ist oder nicht. Im Falle von Funktionaltests ist dies meist ohne weitere Schwierigkeiten möglich. Im Falle von Lasttests beispielsweise, müssen die Testergebnisse eingehender untersucht werden. Falls der Testfall nicht bestanden wurde, muss dies gesondert unter die Lupe genommen werden – ein „Defect muss aufgemacht werden“ [36]. Jeder Defect hat einige Attribute (siehe Abschnitt 2.5.2), von denen die meisten an dieser Stelle ausgefüllt werden und die zum Teil erheblichen Einfluss auf die Weiterbehandlung des Fehlers haben (beispielsweise die Schwere eines Fehlers). Falls der Testfall bestanden wurde, können im Falle eines bestandenen Re-Tests damit verbundene Defects geschlossen werden.

Nach Beendigung der Testaktivitäten und Durchführung aller geplanten Tests, kann geprüft werden, ob das Testendekriterium erfüllt ist. Bei diesem Kriterium handelt es sich in den meisten Fällen vielmehr um ein Auslieferungskriterium, da der Testumfang bereits im Testplan festgelegt wurde. Falls das Testendekriterium (oder die -kriterien) noch nicht erfüllt ist, muss nach einer entsprechenden Ausbesserung mit der ersten Testaktivität, die nötig ist, um die Kriterien zu erfüllen, erneut begonnen werden. [BS798b]

Abschließend ist es sinnvoll, an dieser Stelle den Testprozess zu untersuchen, zu bewerten und für das nächste Release oder Projekt Verbesserungen herauszuarbeiten. [SL04]

Der Prozessschritt der Testauswertung wird in der Literatur nicht immer als erwähnenswert erachtet. Myers behandelt relativ intensiv das Thema Debugging, das sich mit der Analyse eines durchgeführten Testfalls und der Entscheidung, ob der Test bestanden ist oder nicht, überschneidet, geht aber nicht explizit auf den o. g. Punkt ein [Mye79]. Auch

[36] Ein *Defect* ist in diesem Zusammenhang eine Beschreibung der Divergenzen zwischen erwartetem und tatsächlichem Ergebnis eines Testfalls. Oftmals werden sinnvollerweise auch die bei der weiteren Analyse anfallenden Informationen, Erkenntnisse, etc. an derselben Stelle protokolliert. Dies geschieht meist in speziell zu diesem Zweck entwickelten Werkzeugen, sog. *Bug Tracking Tools*.

die Auffassung, was genau ein Testfall ist und wie dieser zu handhaben ist, passt bei Myers nicht immer zu heutigen nichtfunktionalen Testszenarien[37]. Sommerville schenkt diesem Aspekt gar keine Beachtung [Som04]. Es hat sich in der Praxis jedoch gezeigt, dass v. a. bei nichtfunktionalen Testszenarien diese Phase einer erhöhten Aufmerksamkeit bedarf. Zum einen, da eine entsprechende Analyse meist sehr zeitaufwändig ist, zum anderen, da dadurch die Wahrscheinlichkeit sinkt, dass Defects zu unrecht geöffnet oder übersehen werden.

2.5.4. Testen im Softwarelebenszyklus

Im Laufe der Entwicklung eines Softwareprodukts werden auf verschiedenen Entwicklungsstufen unterschiedliche Tests durchgeführt. Dies bedeutet nicht nur, dass sich die Testfälle unterscheiden, sondern auch die Intention der Ausführung kann unterschiedlicher Natur sein. Es kann auch vorkommen, dass ein und derselbe Testfall mehrfach mit unterschiedlichen Absichten ausgeführt wird. Letzteres ist abhängig davon, ob ein Testfall erstmalig oder wiederholt im Rahmen eines Regressions- oder Re-Tests zur Ausführung gelangt.

In der Regel wird zwischen drei, vier oder fünf Teststufen unterschieden (vgl. [Bei90; Wal90; VXT08; Som04]):

- Modul- oder Unit-Test
- Komponententest
- Integrationstest
- Systemtest
- Abnahmetest

In der Literatur wird zum Teil bei dieser Betrachtung der Abnahmetest weggelassen (vgl. Beizer [Bei90]), teilweise auch auf den Komponententest verzichtet [Wal90]. In vielen Fällen gibt es in der Praxis dafür oftmals Zwischenstufen, wie Komponentenintegrationstests oder Teilsystemtests. Beizer beschreibt weiterhin, dass sich die Testphasen nicht dis-

[37] Myers [Mye79]: „A necessary part of a test case is a definition of the expected output or result. This obvious principle is one of the most frequent mistakes in program testing.“ Im weiteren Verlauf wird ausgeführt, dass ein Testfall immer nur die spezifizierten Attribute überprüft. Vor diesem Hintergrund ist sein Verständnis stimmig.

junkt unterteilen lassen, sondern in der Praxis stets der Gesamtprozess betrachtet werden muss, der etwa mit Unit-Tests beginnt und mit dem Test des gesamten, fertigen Softwareprodukts endet. Die grundlegend verschiedenen Intentionen lassen sich sehr gut an den im Folgenden dargestellten vier Stufen beschreiben, auf denen auch das V-Modell (siehe Abb. 2.2) aufbaut [Bal98]. Nach der Darstellung der einzelnen Teststufen, wobei dem Systemtest besondere Beachtung geschenkt wird, wird auf die Testwiederholungen in Form von Regressions- und Re-Tests eingegangen.

Modultest

Unter einem Modultest wird der Test kleiner, oft der kleinsten, Einheiten verstanden, die meist von einem einzelnen Entwickler erstellt werden. Module sind kompilier- oder ausführbar und werden meist anhand von Testtreibern oder -stubs getestet. Die Durchführung dieser Tests obliegt meist der Verantwortung des jeweiligen Entwicklers [Bei90]. Die Intention, mit der die Tests entworfen und ausgeführt werden, umfasst die Überprüfung der Funktionen der zu testenden Einheit, ihrer Struktur, ihres Verhaltens unter Ausnahmebedingungen (etwa bei einer vollen Festplatte) und ihrer Performance [Wal90]. Die Testvoraussetzungen sind das Vorhandensein des Moduls und seiner Spezifikation und aller ggf. benötigten Testtreiber. Für diese Teststufe gibt es einige Testtools, die die Durchführung und die Automatisierung der Testfälle unterstützen. Als Beispiel sei hier das *JUnit Framework* genannt [Bin00].

Integrationstest

Der Integrationstest betrachtet die Integration verschiedener Komponenten. Im Fokus steht dabei die Zusammenarbeit der Einzelteile, die Schnittstellen, das Behandeln von Fehlerzuständen und soweit möglich auch die Umsetzung von Funktionalitäten, die erst durch das Zusammenspiel der Einzelteile ermöglicht werden. Beim Integrationstest wird davon ausgegangen, dass die Einzelteile bereits getestet sind und ihre Funktion verifiziert wurde. Da auf dieser Teststufe bereits Teile des Gesamtsytems getestet werden, die ebenfalls als System zu betrachten sind, wird dieser Test auch Subsystemtest genannt [Wal90].

Bei der Frage danach, wer für den Integrationstest verantwortlich ist, die Entwickler selbst oder ein unabhängiges Testteam, gehen die Meinungen weit auseinander. Beizer formuliert treffend:

> „Integration is a no-man's land. [...] Where on the self-test / independent-test axis should integration testing be placed? I don't know."[Bei90, S. 430]

Beim Integrationstest gibt es grundlegend unterschiedliche Vorgehensweisen, in welcher Reihenfolge die einzelnen Komponenten integriert werden sollen und was zunächst getestet werden soll: Top-down (die Module der obersten Ebene zuerst), Hardest first (die komplexesten Systemteile zuerst), Bottom-up (die Module der untersten Ebene zuerst), funktionsorientiert (jeweils alle für je eine Systemfunktion benötigten Module), je nach Verfügbarkeit, Big-Bang (alle Module auf einmal), etc. [SL04]

Die meisten dieser Integrationsteststrategien werden als inkrementelles Testen bezeichnet (in obiger Aufzählung alle abgesehen vom Big-Bang-Testen). Welche Integrationsmethode am besten ist, kann allgemein nicht beantwortet werden und muss im Einzelfall entschieden werden.

Systemtest

Der Systemtest[38] soll Fehler aufdecken, die nicht der internen Funktionsweise einzelner Komponenten oder den Schnittstellen zwischen Komponenten zugeordnet werden können. Systemtests betreffen das Gesamtsystem oder zumindest einen großen Teil davon. Das Ziel ist, das Gesamtsystem auf die geforderten Anforderungen hin zu überprüfen und Fehler in ihrer Realisierung zu enthüllen. [Bei90; Wal90; Som04]

Es gibt in der Literatur auch andere Sichtweisen, die sich nur zum Teil mit der hier verwendeten, oben dargestellten in Einklang bringen lassen. Nach [ABMD04b] sollte auf dieser Ebene das Testen von nichtfunktionalen Anforderungen im Vordergrund stehen. Dies ist in vielen Fällen zwar der Fall, aber nicht der eigentliche Zweck des Systemtests. In [Lig02] bestehen Systemtests aus Testfällen, die in Funktionstest, Leistungstest, Stresstest, Beta- und Regressionstest eingeteilt werden. Auch dies ist in der Praxis zwar oftmals möglich, charakterisiert aber noch nicht den Kern der Systemtests.

[38] Auch als *System Integration Test* bezeichnet [Som04].

Ausgeführt und spezifiziert werden Systemtests in der Praxis und in der Literatur meist von einem unabhängigen Testteam [SBS06]. Bei vielen nichtfunktionalen Testszenarien beispielsweise ist eine enge Zusammenarbeit zwischen den Testern und den Entwicklern jedoch unabdingbar. Nach [PKS00] unterliegen die Systemtests aber oft auch als Black- oder vielmehr Grey-Box-Tests der Verantwortung des Entwicklers.

Nach [Wal90] liegen folgende Punkte im Fokus des Systemtests: Vollständigkeit, Volumen, Last, Handhabung / Benutzerfreundlichkeit, Sicherheit, Effizienz, Konfiguration, Kompatibilität / Datenkonversion, Dokumentation und Wartbarkeit. Diese Liste ist aus heutiger Sicht nicht vollständig und muss je nach Softwareprodukt um einige Aspekte wie Zuverlässigkeit, Ausfallsicherheit, Recovery-Fähigkeit, etc. ergänzt werden.

Die Systemtests bauen auf den Anforderungen an das Softwareprodukt auf und können grob in zwei Bereiche geteilt werden: funktionale und nichtfunktionale Tests, je nachdem, ob sie sich an einer funktionalen oder nichtfunktionalen Anforderung orientieren.

Abnahmetest

Der Abnahmetest stellt die letzte Teststufe vor dem Einsatz der Software dar. Das Hauptziel ist sicherzustellen, dass das Produkt den Kundenanforderungen gerecht wird. Diese sind meist Bestandteil des Vertrages zwischen Auftragnehmer und -geber. Falls die Testfälle keine Fehler aufdecken, wird das Produkt abgenommen.

Im Allgemeinen wird der Abnahmetest direkt beim und oftmals vom Benutzer der Software durchgeführt. Er sollte immer in der späteren Betriebsumgebung durchgeführt werden. [BM04]

Testwiederholungen – Regressions- und Re-Tests

Ein Testfall wird in der Regel nicht nur einmal durchgeführt, sondern mehrfach. Dies kann grundsätzlich aus zwei verschiedenen Gründen geschehen. Zum einen in Form eines Re-Tests, unter dem man die erneute Ausführung eines zuvor nicht bestandenen Testfalls versteht. Dieser Test hat das Ziel, einen gefundenen Fehler nach dessen Behebung schließen zu können. Zum anderen werden Testfälle in einem Regressionstest erneut ausgeführt.

Re-Tests sind ebenfalls Teil des Regressionstests, dessen Ziel es ist, die korrekte Funktionsweise der Software nach einer Änderung erneut zu überprüfen. Eine solche Änderung kann beispielsweise bei der Entwicklung einer neuen Version oder der Veröffentlichung eines Patches entstehen.

Die Auswahl der Testfälle für den Regressionstest ist bereits seit einigen Jahren ein Thema in der Forschung. Die einfachste Möglichkeit ist, alle Testfälle in einem Regressionstest erneut auszuführen. Diese „Auswahl“ jedoch kann etliche ungültige Testfälle beinhalten, da durch die neue Softwareversion auch die Funktionalität geändert worden sein kann. Des Weiteren kann, abhängig vom Verhältnis der Größe des gesamten Softwareproduktes zur Größe der geänderten Teile, ein hoher Blindleistungsanteil entstehen.

Andere Ansätze beinhalten feinere Selektionsstrategien für den Regressionstestumfang und dessen Zusammensetzung. Hierbei müssen mehrere Aspekte beachtet werden. Je geringer die Anzahl der Testfälle ist, die für den Regressionstest ausgewählt wurden, umso höher ist die Wahrscheinlichkeit, dass ein Fehler, der sich durch die Softwareänderung ergeben hat, übersehen wird. Andererseits wird der Aufwand zur Ausführung der Testfälle umso größer, je mehr Tests im Regressionstest ausgeführt werden. In vielen Fällen lohnt sich eine genauere Analyse, wobei der Aufwand der Selektion der Testfälle ebenfalls betrachtet werden muss.

Die heutigen Selektionsverfahren sind vielfältig und umfassen Minimierungsverfahren, datenflussbasierte Verfahren, Zufallsverfahren, architekturbasierte Verfahren und sog. sichere Verfahren. [GHK+01; MDR05]

2.5.5. Testmethoden

Es gibt eine Fülle von verschiedenen Testmethoden, worunter Methoden verstanden werden, wie die Testfälle entworfen werden. Diese unterscheiden sich durch die Teststufen auf denen sie ausgeführt werden. Die Teststufen sind in Abschnitt 2.5.4 beschrieben und orientieren sich oft am V-Modell (siehe Abb. 2.2). Auch Sommerville teilt die Tests nach einem dem V-Modell ähnlichen Schema ein (siehe Abb. 2.9).

Grundlegend gibt es zwei verschiedene Methoden der Testfallbestimmung und damit der

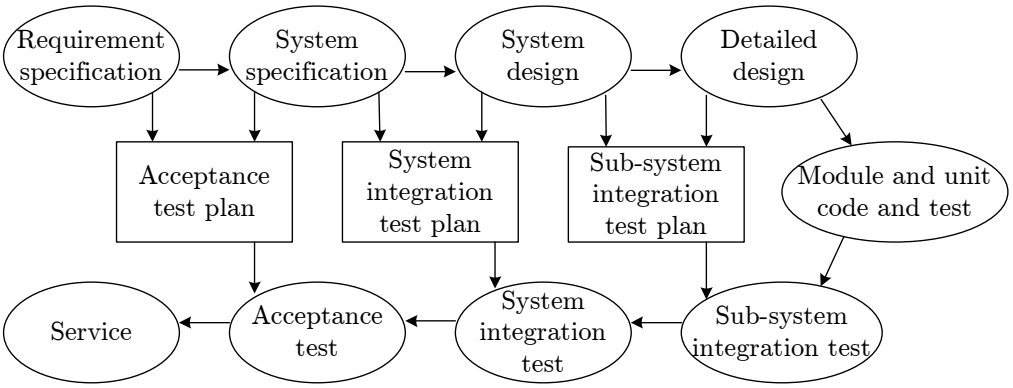

Abbildung 2.9.: Überblick über den Test- und Entwicklungsprozess nach Sommerville [Som04].

Testzuordnung: Whitebox- und Blackbox-Tests. Bei Blackbox-Tests betrachtet der Tester die zu testende Software als Blackbox, d. h. ohne die interne Struktur zu kennen oder zu beachten. Die Testfälle und Testdaten entstehen auf Basis von Anforderungsdokumenten und Spezifikationen. Im Gegensatz dazu stehen die Whitebox-Tests, die die innere Struktur und die Implementierung des Softwaresystems als Ausgangspunkt heranziehen. Whitebox-Tests werden auch strukturelle Tests, Glassbox-Tests, implementierungsbasierte Tests oder Clearbox-Tests genannt [Bin00]. Teilweise wird der Begriff Greybox-Tests verwendet für Tests, die sich sowohl an der inneren Struktur des Softwaresystems als auch an Anforderungen bzw. Spezifikationen orientieren [Bei90]. Dazu zählen Tests auf Ebene der Integrations- oder Teilsystemtests.

Für die Erstellung von Whitebox-Testfällen, die hauptsächlich auf der Stufe der Modul- und Integrationstests stattfinden, gibt es eine ganze Reihe an Vorgehensweisen. Die Testfälle zielen meist auf die Ausführung aller Befehle, Entscheidungen, Bedingungen, etc. ab, versuchen alle Pfade durch das Programm abzubilden oder alle möglichen Datenflüsse zu überprüfen. Die Wahl eines passenden Kriteriums ist abhängig von mehreren Faktoren, wie etwa der Programmiersprache [Bin00].

Die Blackbox-Testfälle basieren auf den vom Softwaresystem angebotenen Funktionen und den möglichen Eingabewerten. Hierbei findet oftmals eine Äquivalenzklassen- oder Grenzwertanalyse Anwendung, die den Eingabebereich analysiert und aufteilt. Diese Tests werden in der Regel auf Systemtestebene und zum Teil auf Integrationstestebene durchgeführt. Sie orientieren sich an den Spezifikationen und Anforderungsdokumenten und

führen somit oftmals Tests aus Anwendersicht durch.

Weitere Darstellungen von Testmethoden, Methoden zur Auswahl von Testfällen, zur Reihenfolge der Durchführung etc. würden den Rahmen dieser Arbeit sprengen, hier konnten nur die grundlegenden Konzepte vorgestellt werden.

2.5.6. Systemtest von Plattformen

Allgemein

Der Systemtest von Plattformen gestaltet sich anders als der Systemtest von (End-)Anwendungen. Es gibt mehrere Punkte, die unterschiedlich zu behandeln sind. Zunächst entsprechen die Plattform-Systemtests aus Sicht der Anwendungsentwicklung Teilsystem- bzw. Integrationstests, da von ihrem Standpunkt aus betrachtet noch nicht das gesamte System integriert ist – die Anwendung selbst fehlt noch. Aus Sicht der Plattformentwicklung ist das Gesamtsystem jedoch fertiggestellt und wird nach bestandenem Test ausgeliefert.

In vielen Fällen ist es nicht ohne Weiteres möglich, anhand von Testfällen die Plattform direkt zu testen. Dies wird in Abschnitt 3.1 näher erläutert. Eine Möglichkeit der Lösung dieses Problems stellt der Einsatz von Test- oder Beispielanwendungen dar (siehe auch Abschnitt 3.1). Diese bilden eine Zwischenschicht zwischen Plattform und Testfällen. Durch diese Einführung einer weiteren Stufe der Indirektion werden auch die Tests indirekter. Daraus ergeben sich zwei Punkte, die besonderer Beachtung bedürfen: Die Güte der Tests hängt von der Validierung der Testapplikationen ab und ein gefundener Fehler kann auch in der Testapplikation stecken, wodurch sich zunächst keine Aussage über die Plattform machen lässt. Ein Vorteil des Einsatzes von Testapplikationen ist, dass dadurch die Tests als herkömmliche Anwendungstests erscheinen und durchgeführt werden können.

Letzteres trifft jedoch nicht auf die Bewertung von nichtfunktionalen Testfällen zu. Um diese zu bewerten bzw. in bestanden und nicht bestanden einzuteilen ist ein Vergleich mit nichtfunktionalen Anforderungen notwendig. Wie in 3.1 gezeigt wird, ist dies bei Plattformtests nicht ohne Weiteres möglich.

Test von Produktlinienplattformen

Der Test von Produktlinienplattformen bringt dieselben Herausforderungen mit sich wie der Test von Plattformen im Allgemeinen. Ein Unterschied besteht im Anspruch bei der Produktlinienentwicklung, auch die Testfälle wieder zu verwenden.

Es gibt grundlegend drei verschiedene Möglichkeiten, Produktlinien zu testen. Die einfachste und aufwändigste sieht vor, jede abgeleitete Applikation isoliert zu betrachten und zu testen. Eine zweite Möglichkeit besteht darin, bereits für eine abgeleitete Applikation entwickelte Testfälle wieder zu verwenden. Somit wird die erste Anwendung „normal“ getestet, als ob es sich nicht um einen Teil einer Produktlinie handeln würde. Doch alle weiteren Anwendungen können von den bereits bestehenden Testfällen profitieren und diese ggf. nutzen. Dabei besteht die Gefahr, dass zu wenig auf die spezifischen Eigenheiten der zweiten und aller späteren Anwendungen eingegangen wird. Der dritte Weg umgeht dieses Problem durch die Erstellung von wiederverwendbaren Testfällen im Domain Engineering. Der Testprozess wird hierbei in zwei Teile aufgeteilt. Im Domain Engineering werden die Plattformbestandteile anhand auch in der späteren Anwendungsentwicklung wiederverwendbarer Testfälle getestet. Der spätere Test der abgeleiteten Anwendungen geschieht meist durch Anpassungen der Domänentestfälle [BKP04]. Diese Methode hat den Nachteil, dass bei den Plattformtests zu viel Wert auf eine möglichst hohe Wiederverwendung gelegt wird und nicht die Güte der Plattformtests an sich im Mittelpunkt steht.

Doch auch dieses Ziel, ein stimmiges Konzept für den Test der gesamten Produktlinie mit einem hohen Anteil an effizient wiederverwendeten Testartefakten zu entwerfen, ist nicht einfach. Bertolino und Gnesi stellen in [BG03] eine Methodik vor, die auf beiden Ebenen, dem Domain und dem Application Engineering, arbeitet. Anhand von Use Cases, die um Variationsmöglichkeiten erweitert wurden, werden im Domain Engineering variable Testfallbeschreibungen erstellt. Aus diesen entsteht dann im Application Engineering für mehrere (im Idealfall alle) Anwendungen je ein Testfall. Im Domain Engineering wird bei diesem Vorgehen, wenn überhaupt, dann nur mit eigens erstellten Testfällen getestet.

Auch Nebut et. al. benutzen Use Cases im Domain Engineering. Diese sind parametrisiert und drücken die Requirements der Produktlinie aus. Aus den Use Cases werden anschließend Testszenarios abgeleitet, wobei die Parametrisierung und damit die Variabi-

lität beibehalten wird. Erst im Application Engineering werden daraus Testfälle generiert. [NFLTJ03]

Die ScenTED-Methode [RKPR05] der Arbeitsgruppe Software Systems Engineering der Universität Duisburg-Essen verfeinert die Use Cases durch einen szenariobasierten Ansatz. Die Szenarien sind exemplarische Abläufe und Bestandteil des Use Cases. Diese werden um Testinformationen ergänzt, wodurch die resultierenden Testfälle über die Szenarien mit den verknüpften Applikationen auf Basis der Produktlinienplattform in Verbindung stehen. Dabei wird die in den Use Cases enthaltene Variabilität auf die Testfälle übertragen.

All diese Verfahren haben gemeinsam, dass die Ausführung der Testfälle erst im Application Engineering erfolgt. Eine Möglichkeit, dies zu umgehen, ist die Entwicklung sog. Beispiel- oder Testanwendungen im Domain Engineering [RKPR05]. Dies wird auch in dieser Arbeit vorgesehen und in Abschnitt 3.1 näher erläutert.

2.5.7. Zusammenfassung

In diesem Abschnitt wurde eine Einführung in einen Teil der QS, den Softwaretest, gegeben. Zunächst wurde hierzu die Grundidee des Testens erläutert. Es wurde festgestellt, dass Softwaretests nicht die Fehlerfreiheit eines Softwareprogramms beweisen können, sondern lediglich das Gegenteil – durch eine gezielte Ausführung werden Fehlfunktionen provoziert.

Die Definition, was genau unter einem Fehler, einem Fehlverhalten und einem Irrtum verstanden wird, bildete den Einstieg in die genauere Beschreibung, was einen Fehler ausmacht und wie mit diesen verfahren werden sollte. Eine kurze Darstellung möglicher Fehlerattribute komplettierte dieses Thema.

Der Testprozess nach dem British Standard BS 7925-2 [BS798b] lieferte den Rahmen für die Beschreibung der verschiedenen Aktionen, die beim Test durchgeführt werden müssen, von der Testplanung, über die Testfallspezifikation, die Testdurchführung und deren Protokollierung bis hin zur Auswertung der Testergebnisse.

Anschließend wurden anhand der Betrachtung des Testens als Teil des Softwarelebenszyklus verschiedene Teststufen vorgestellt. Auch die Auswahl von Regressionstests wurde in diesem Umfeld behandelt. Die Darstellung verschiedener Testmethoden fand im Anschluss

daran statt, da sie meist abhängig von der Teststufe ist.

Zuletzt wurde der Systemtest von Plattformen genauer erläutert und auf besondere Gegebenheiten eingegangen. Auch die Betrachtung der Tests von Softwareproduktlinien bzw. den im Domain Engineering entwickelten Plattformen fand in diesem Kontext statt.

2.6. Softwaremaße, -metriken und -kennzahlen

2.6.1. Begriffsdefinitionen

Die Begriffe *Softwaremetrik* und *-maß* sind bereits seit einigen Jahrzehnten Gegenstand von wissenschaftlichen Untersuchungen. Und genauso lange ist deren Einsatz umstritten. Die Befürworter zitieren W. Edwards Deming: „In God we trust. All others bring data.“ und argumentieren, dass es unumgänglich ist, Messwerte zu ermitteln, um ein Projekt zu planen, die Entwicklung eines Produkts zu steuern oder um Prozesse, Projekte und Produkte zu verbessern. Die Gegner von Maßprogrammen, Metriksystemen, Kennzahlenprojekten usw. halten diese für großen zusätzlichen Aufwand, der in keiner Beziehung zum Nutzen steht. Außerdem seien sie leicht zu manipulieren und schwer zu interpretieren. Wie eine derartige Diskussion umgangen bzw. wie evtl. Schwierigkeiten in diesem Umfeld vorgebeugt werden kann, wird in Abschnitt 2.6.2 beschrieben.

Es gibt jedoch noch ein weiteres grundlegenderes Problem beim Einsatz von Kennzahlen: die Begrifflichkeiten. Für die Ausdrücke „Maß“ und „Metrik“ (abgeleitet vom griechischen metron = Maß [Dud07]) existieren mehrere Erläuterungen bzgl. dem Messen von Software und zusätzlich auch eine mathematische Definition. In der Praxis werden die Begriffe meist synonym verwendet [ED96]. Bereits 1984 wies Schmidt im aktuellen Schlagwort des Informatik Spektrums auf diese Problematik hin [Sch84]. In dieser Arbeit soll hauptsächlich der Begriff *Kennzahl* verwendet werden.

Definition: Kennzahl

Eine Kennzahl ist eine quantitative Bewertung eines Artefakts, eines Prozesses oder eines Zustands.

Diese eher allgemein gehaltene Begriffsklärung spiegelt den unterschiedlichen Einsatz von Kennzahlen in der Praxis wider und vereint die gegensätzlichen Ansichten in der Literatur. Der Begriff Kennzahl wird hauptsächlich in den Wirtschaftswissenschaften verwendet.

Die beiden Begriffe Maß und Metrik kommen auch in anderen Bereichen vor, beispielsweise in der Biometrie, die in den letzten Jahren vermehrt in den Fokus öffentlichen Interesses gerückt wurde. Dabei werden Körper- oder Verhaltensmerkmale gemessen, „um

die Identität einer Person zu ermitteln (Identifikation) oder die behauptete Identität zu bestätigen oder zu widerlegen (Verifikation)“ [Bun]. Auch dieser Bereich des Messens wird durchaus kontrovers diskutiert [Pfi06]. In der Mathematik kommen die beiden Begriffe ebenfalls vor und finden entsprechend folgender Definitionen Anwendung (frei nach [BSMM99; Els99]):

Definition: Metrik

Für jede Menge A ist eine Metrik eine Abbildung $d : A \times A \rightarrow \mathbb{R}$, so dass gilt:

1. $d(a, b) \geq 0 \ \forall a, b \in A, \ d(a, b) = 0 \Leftrightarrow a = b$,
2. $d(a, b) = d(b, a) \ \forall a, b \in A$,
3. $d(a, c) \geq d(a, b) + d(b, c) \ \forall a, b, c \in A$.

Definition: Maß

Eine auf einer σ-Algebra $\mathcal{A}$ defininierte Funktion $\mu : \mathcal{A} \rightarrow \bar{\mathbb{R}}_+$ heißt Maß, wenn

1. $\mu(A) \geq 0 \ \forall A \in \mathcal{A}, \ A \neq \emptyset$,
2. $\mu(\emptyset) = 0$,
3. $\mu(\bigcup_{n=1}^{\infty} A_n) = \sum_{n=1}^{\infty} \mu(A_n) \ \forall A_1, A_2, ... \in \mathcal{A}, \ A_i \neq A_j \ \forall i \neq j$.

In vielen Fällen wird versucht, diese beiden mathematischen Begriffe mit einer praxisnahen Sicht der Softwareentwicklung in Einklang zu bringen. Weit verbreitet sind beispielsweise folgende Definitionen nach Dumke [Dum03]:

„Eine Software-Metrik ist gemäß der Maßtheorie eine Abstandsfunktion, die Attributen von Software-Komponenten Zahlen(-bereiche) zuordnet.“

„Ein Software-Maß ist gemäß der Messtheorie eine mit einer Maßeinheit versehene Skala, die in dieser Form ein Software-Attribut bewertet bzw. messbar macht.“

Liggesmeyer geht in [Lig02] ebenfalls auf diese unterschiedlichen Begriffsauffassungen ein und argumentiert, dass der mathematischen Sichtweise nach in der Informatik so gut wie ausschließlich Maße verwendet werden (die einer Menge von Software-Attributen einen mit einer Skala versehenen Zahlenwert zuordnen). Metriken stellen dahingegen eine Abstandsfunktion dar, die jeweils zwei konkreten Ausprägungen eines Attributs eine Zahl zuordnen und bzgl. der Software selten vorkommen.

Zusätzlich zu den o. g. Definitionen gibt es zwei unterschiedliche IEEE Standards, die den Begriff *Metric* auf zwei weitere Arten festlegen:

> „Software quality metric: 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.“[IEE98a]

> „Quality metric:
>
> 1. A quantitative measure of the degree to which an item possesses a given quality attribute.
> 2. A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute.“[IEE90]

Im Zusammenhang mit Kennzahlen fällt oftmals auch der Begriff *Key Performance Indicator* (KPI). Hierunter wird eine auf die Bewertung eines bestimmten Ziels fokussierte Kennzahl verstanden. KPIs sind oftmals Kennzahlen für die Steuerung und Optimierung von Prozessen und werden in Unternehmen häufig in Verbindung mit Scorecards eingesetzt. [Hor06]

In dieser Arbeit werden die Begriffe *Softwaremetrik* und *Softwaremaß*, wie in der Praxis üblich, als Synonyme von Kennzahl verwendet, obwohl es auch mit der mathematischen Definition weitaus besser in Einklang zu bringende Begriffsklärungen gibt. Bewusst wird auf den Gebrauch der Begriffe *Maß* und *Metrik* als nicht zusammengesetzte Wörter verzichtet[39].

Ein wichtiger Punkt, den man bei der Arbeit mit Kennzahlen stets beachten muss, sind die verschiedenen *Skalentypen*. Unter einer Skala ist eine oft mit einer Maßeinheit versehene Bezugsgröße zu verstehen, die Vergleiche ermöglicht oder verbietet und diese somit in geordnete Bahnen lenkt. Als Beispiel kann man sich die Länge eines Fahrzeugs in Metern oder die Temperatur in ° Celsius vorstellen. Da aufgrund der unterschiedlichen Skalentypen auch unterschiedliche Operationen auf den Daten erlaubt sind, gilt es hier, Vorsicht walten zu lassen. In Anhang B werden die verschiedenen Skalentypen dargestellt.

[39] Begriffe wie *Softwaremaß*, *Qualitätsmetrik*, etc. sind, wie bereits erwähnt, als entsprechende *Kennzahlen* zu verstehen.

2.6.2. Anforderungen und Ziele von Kennzahlen

Die meisten theoretischen Ansätze und Verfahren zur Etablierung eines Kennzahlenprogramms heben hervor, dass zunächst die Ziele der Kennzahlen festgelegt werden sollten [Lig02; JA97]. Auch das GQM-Verfahren baut darauf auf (siehe Abschitt 2.4.2). Dies schließt mit ein, dass zunächst die Verwendung und die Verbreitung der Daten festzulegen ist. Dies liegt darin begründet, dass Kennzahlen zum einen oft auch Rückschlüsse auf die Arbeitsqualität der Mitarbeiter zulassen und zum anderen auch oft verfälscht werden können.

Bei der Auswahl von Kennzahlen spielen somit nicht nur fachliche Gründe eine Rolle – es muss auch auf andere Rahmenbedingungen eingegangen werden. Auch eine ganze Reihe technischer Voraussetzungen müssen erfüllt sein, um Kennzahlen sinnvoll nutzen zu können. In [Lig02] werden folgende Forderungen an Softwaremaße genannt: Einfachheit, Eignung, Stabilität, Rechtzeitigkeit, Analysierbarkeit und Reproduzierbarkeit.

Ein weiterer Punkt, der im Vorfeld geklärt werden sollte, ist die Bestimmung des Personenkreises, dem die Ergebnisse zur Verfügung gestellt werden sollen. Der Ansatz von van Solingen und Berghout sieht vor, dass das Entwicklungsteam über die Verwendung der Daten entscheidet[40]. Dem ist hinzuzufügen, dass es in vielen Fällen sinnvoll ist, auch den Kunden einige (nicht alle!) Kennzahlen zur Verfügung zu stellen.

In manchen Fällen ist es sinnvoll, für einen Teil der Kennzahlen feste Zielgrößen vorzugeben. Dies ist beispielsweise bei nichtfunktionalen Anforderungen an Anwendungen der Fall, aber oftmals auch bei internen Werten, wie dem Testautomatisierungsgrad oder Ähnlichem, denkbar. Diese Zielgrößen müssen wohlüberlegt, sinnvoll und realistisch sein, da sie ansonsten kontraproduktiv wirken können.

Die Ziele, die mit der Einführung eines Kennzahlsystems erreicht werden sollen, können sehr unterschiedlich sein. Im Mittlepunkt steht im weitesten Sinne der Erkenntnisgewinn über die Qualität des Produkts und des Entwicklungsprozesses. Hierbei gibt es allgemeine quantitative Untersuchungen von Fehlern bzw. Fehlerhäufigkeiten, um Aussagen über deren zukünftige Entwicklung machen zu können [FO00; Neu05; GKMS00], Kennzahlen, um

[40] „Advised is to stay with the rule that the software development team ‘owns’ the measurement data and therefore decides on distribution of both data and reports to, for example, management.“ [SB01]

die Effizienz des Softwaretests zu bewerten [Per95; CPR04], aber auch Kennzahlen, die spezielle Arten von Software, beispielsweise auf objektorientierten oder Logik-basierten Programmiersprachen basierende Programme, untersuchen [Mel95]. Viele der spezifischeren Untersuchungen spezialisieren sich auf implementationstechnologische Unterschiede, wie etwa objektorientierte Software. Es gibt jedoch, wenn auch nur wenige, Veröffentlichungen zu Kennzahlen bei eingebetteten Systemen [CT04], verteilten Systemen [MCM+99], etc.

2.6.3. Kategorisierung von Softwaremetriken

Eine Kategorisierung von Softwaremetriken kann anhand verschiedener Kriterien erfolgen. Die gängigste, grobe Einteilung bezieht sich zunächst auf das zu bewertende Objekt. Dabei gibt es Prozess-, Produkt- und Projektkennzahlen. Dies deckt sich auch mit der grundlegenden Voraussetzung, sich zunächst darüber bewusst zu werden, welches die Attribute und Entitäten sind, die man zu messen beabsichtigt. Die dritte Gruppe (Projektkennzahlen) nennt Fenton in [Fen91] *Resource Measures*. Darunter werden Bewertungen des Personals, Zusammensetzungen und Aufstellungen von Teams, Bewertungen von eingesetzten Software-Tools, von eingesetzter Hardware, der Büroräume etc. verstanden.

Sobald die Entität identifiziert ist, deren Attribute bewertet werden soll, kann nach Fenton eine weitere Kategorisierung erstellt werden. Die Attribute können je nach Kontext in interne und externe Attribute eingeteilt werden. Interne Attribute ergeben sich durch das alleinige Betrachten der Entität, zu der sie gehören. Externe Attribute müssen im Kontext ihrer Umgebung betrachtet werden und stellen beispielsweise die Kommunikation zwischen verschiedenen Entitäten in den Mittelpunkt. Letztere können nach Fenton nicht direkt gemessen werden. Einige Beispiele für Entitäten, ihre Zuordnungen und interne und externe Attribute befinden sich in Tabelle 2.2.

In dieser Arbeit sind hauptsächlich die Produktmetriken und zum Teil die Prozessmetriken von Interesse. Im Folgenden wird daher nicht weiter auf Projektmetriken eingegangen. Verwendung finden oft auch die Begriffe Codemetriken, Systemmetriken und Testmetriken, die ebenfalls im Folgenden kurz erläutert werden.

ENTITÄTEN	ATTRIBUTE	
	Intern	**Extern**
Produkt		
Spezifikation	Größe, Wiederverwendung, Modularität, Redundanz, ...	Konsistenz, Verständlichkeit, ...
Design	Größe, Modularität, Funktionalität, ...	Qualität, Komplexität, ...
Code	Größe, Wiederverwendung, Datenfluss-Struktur, ...	Zuverlässigkeit, Wartbarkeit, ...
...	...	...
Prozess		
Design	Dauer, Aufwand, # gefundener Spezifikationsfehler im Systemtest, ...	Qualität, Kosten, Stabilität, ...
Testphase	Dauer, Aufwand, # gefundener Fehler, ...	Kosteneffektivität, Stabilität, ...
...	...	...
Projekt / Ressourcen		
Personal	Alter, Kosten, ...	Produktivität, Erfahrung, ...
Teams	Größe, Aufbau, ...	Produktivität, Qualität, ...
Büroräume	Lichtverhältnisse, Größe, Klimatisierung...	Komfort, Qualität, Erreichbarkeit ...
...	...	...

Tabelle 2.2.: Entitäten und deren Attribute, typische Ansatzpunkte von Kennzahlen nach [Fen91]

Prozessmetriken

Unter den Oberbegriff der Prozessmetriken fallen je nach Ansicht verschiedene Kennzahlen. Wallmüller beispielsweise fasst in [Wal90] unter Prozessmaßen alle Kennzahlen zusammen, die quantifizierbare Attribute des Entwicklungs-(Pflege-)prozesses oder der Entwicklungs-(Pflege-)umgebung bewerten. Als Beispiele werden u. a. Kennzahlen angeführt, „die die Erfahrung der Entwickler (etwa die Anzahl der Jahre an Programmiererfahrung) und die Kosten des Entwicklungsprozesses beschreiben“ [Wal90, S.33]. Diese Sichtweise steht im Gegensatz zu der Sichtweise von Fenton und Liggesmeyer und ergibt sich aus der ausschließlichen Einteilung in Prozess- und Produktmetriken bei Wallmüller. Das o. g. Beispiel ist der Dreiteilung bei Fenton und Liggesmeyer nach den Projektmetriken bzw. den Ressourcenmetriken zuzuordnen.

Für Prozessmetriken gibt es nur eine eingeschränkte Anzahl an direkt messbaren Attributen. Im Großen und Ganzen sind dies:

- Zeit, meist die Dauer des Prozesses oder von Teilprozessen,
- mit dem Prozess in Verbindung stehender Aufwand und
- Anzahl an Vorfällen bestimmter Ereignisse innerhalb des Prozesses (hierzu zählen beispielsweise die Anzahl an Spezifikationsfehlern, die während der Spezifikationsphase gefunden wurden).

Dies deckt sich auch mit der Darstellung von Liggesmeyer in [Lig02]. Dort wird zusätzlich hervorgehoben, dass Maße in vielen Fällen auf ein Produkt, einen Prozess oder ein Projekt zu beziehen sind. Als Beispiel wird die Testüberdeckung angeführt, die auf ein Produkt bezogen als Produktmaß dienen kann. Im Falle der Existenz eines Zielwerts ist sie jedoch zum Abschluss der Testphase zu den Prozessmaßen zu zählen und zur Überprüfung des Projektfortschritts kann sie als Projektmaß eingesetzt werden.

In vielen Fällen wird die grundlegende Annahme getroffen, dass die Qualität des Entwicklungsprozesses direkt die Qualität des hergestellten Produkts beeinflusst. Auf dieser Basis könnten Prozesskennzahlen als indirekte Kennzahlen für das Produkt dienen. Dieser Zusammenhang besteht jedoch nicht notwendigerweise – ein sehr guter Prozess muss nicht zwangsläufig ein sehr gutes Produkt hervorbringen. Erfahrungsgemäß hat die Prozessqua-

lität jedoch einen signifikanten Einfluss auf die Qualität der Software. [Som04]

Produktmetriken

Produktmetriken sind Kennzahlen, die Attribute des Produkts bewerten. Diese können weiter unterteilt werden anhand der Ebene des V-Modells, auf der sie zur Anwendung kommen bzw. anhand des Artefakts, das genauer betrachtet wird. Diese Unterteilung ist in der Literatur nicht explizit gebräuchlich, findet ungeplant jedoch oftmals Anwendung, da verschiedene Teams auf verschiedenen Ebenen agieren. Ebenenübergreifende Kennzahlen orientieren sich meist am Prozess und sind somit zu den Prozessmetriken zu zählen. Unter „Produkt“ wird hierbei nicht nur das Endprodukt, die Software an sich, verstanden, sondern auch Zwischenprodukte, wie etwa Spezifikationsdokumente.

Auf den oberen Ebenen stehen die Bewertung von Anforderungs-, Spezifikations- und Designdokumenten im Mittelpunkt. Dabei werden oftmals Reviews und Inspektionen bzw. deren Ergebnisberichte als Datengrundlage verwendet [Sch03b]. Anforderungsdokumente werden meist als Textdokumente festgehalten. Hier kommen textbasierte Kennzahlen zum Einsatz. Beispielsweise die Eindeutigkeit ist ein wichtiges Qualitätskriterium von Anforderungen. Darunter wird verstanden, dass eine Anforderung nur auf eine Weise interpretiert werden kann und ihre Aussage somit eindeutig ist. Die Eindeutigkeit einer Anforderung basiert auf der Abwesenheit sog. sprachlicher Defekte[41]. Damit ergibt sich für die Eindeutigkeit folgende Kennzahl:

$$\frac{\#(\text{Anforderungen ohne Defekte})}{\#(Anforderungen)}$$

Als weiterer Indikator für das Vorhandensein sprachlicher Defekte wird oft der Anteil an Passivsätzen ermittelt. Die Validität dieser Korrelation wurde in mehreren Studien bereits nachgewiesen. Eine Übersicht befindet sich in [RR06].

Die am meisten verbreiteten Produktmetriken befinden sich auf der Code-Ebene. Die meisten der sog. Codemetriken beziehen sich hierbei auf den Umfang bzw. die Größe des Softwareprodukts. Beispiele hierfür sind die *Lines of code* (LOC) oder die *Function Points*

[41] Unter sprachlichen Defekten werden sowohl grammatikalische Fehler oder Rechtschreibfehler als auch unklare Formulierungen verstanden.

[Fen91]. LOC geben die Anzahl der Zeilen des Quellcodes an. Genau definiert werden muss, welche Zeilen hierbei in die Kennzahl mit einfließen (Kommentarzeilen, Leerzeilen, etc.). Wie dies gehandhabt wird, ist bei ausschließlich interner Verwendung irrelevant, muss jedoch im Vorfeld festgelegt und konsequent umgesetzt werden. Sollen auch Vergleiche mit bereits existierenden Daten durchgeführt werden, ist es wichtig, auf den Einsatz derselben Festlegung zu achten. Beim Einsatz von *Function Points* kann der Anwender zwischen vier verschiedenen ISO-Standards[42] wählen. Die Anwendung ist in jedem Fall aufwändiger als bei LOC, kann jedoch schon vor der Codierung durchgeführt werden und liefert bereits früh relativ gute Schätzwerte. [Feh05]

Auch Komplexitätsmetriken, wie die von Halstead oder McCabe, finden verbreitet Anwendung. McCabe's zyklomatische Komplexität beispielsweise ermittelt graphentheoretisch die Anzahl an Pfaden durch ein Codefragment auf Basis dessen Kontrollflussgraphen. Mit ihr lassen sich komplexe Programmteile identifizieren – unnötig komplexe Abschnitte können daraufhin vereinfacht werden, an sich komplexe Vorgänge oder algorithmisch zwingenderweise komplex darstellbare Abschnitte werden zumindest explizit identifiziert. [GC87]

Dem V-Modell folgend können bei der Integration verschiedener Klassen und Komponenten ebenfalls Kennzahlen Anwendung finden. Auf Code- oder Klassenebene werden oft auch speziell für die objektorientierte Entwicklung Kennzahlen erstellt. Die Kopplung kann hier als Beispiel dienen [Bin00]. Dabei wird die Anzahl der gegenseitigen Aufrufe verschiedener Klassen ermittelt. Dies kann zum einen durch entsprechende Protokollierung während der Tests erfolgen oder anhand der Anzahl möglicher Aufrufe bei der Analyse des Codes bzw. der Klassendiagramme.

Auf Systemebene können ebenfalls Produktmetriken zum Einsatz gelangen. Diese werden dann als Systemmetriken bezeichnet. Ein Beispiel ist die Untersuchung einzelner Use Cases und die Bestimmung der Anzahl Klicks, die zu dessen Abarbeitung nötig sind. Hierzu ist jedoch nicht das Gesamtsystem nötig – dies kann auch auf Basis der Spezifikation erfolgen. Die meisten Kennzahlen auf dieser Ebene sind in dieser Arbeit den *Testmetriken* zuzuordnen und werden gesondert betrachtet.

[42] ISO/IEC 20926, ISO/IEC 20968, ISO/IEC 24570, ISO/IEC 19761.

Testmetriken

Nach [SL04] dienen Testmetriken der Überwachung und der Erfolgskontrolle der laufenden Tests. Diese sollten bereits im Testkonzept vereinbart werden, wobei darauf zu achten ist, dass, wie bei anderen Kennzahlen auch, die einzelnen Testmetriken regelmäßig, einfach und zuverlässig zu messen sind. In [ED96] werden Testmetriken auch zur Auswahl von Testfällen herangezogen. Als Beispiel diene hier das Zufallstesten. Dabei wird der Eingabedatenbereich mit einer Wahrscheinlichkeitsverteilung versehen und daraus entsprechende Testfälle automatisch zufällig erzeugt. Dies fällt in dieser Arbeit nicht unter den Begriff *Testmetriken*.

Testmetriken können auf zwei unterschiedliche Weisen weiter unterteilt werden. Zum einen können sie anhand ihrer Aufgabe eingeteilt werden in Metriken, die die Testeffektivität messen und Metriken, die die Testergebnisse bewerten und somit beispielsweise die Qualität der Software messen [Kan03]. Zum anderen können sie eingeteilt werden je nach Ausgangspunkt der Messung. Hierbei werden nach [SL04] drei verschiedene Kategorien unterschieden: fehlerbasierte Metriken, testfallbasierte Metriken und testobjektbasierte Metriken.

Die beiden Kategorien aus der ersten Unterteilung können nach Kan [Kan03] auch kombiniert werden und dadurch weitere Rückschlüsse zulassen. Das Projekt A in Abb. 2.10 ist charakterisiert durch einen hohen Testaufwand und viele gefundene Fehler. Nach Kan wird diese Kategorie in der *Effort-Output-Matrix* als *good / not bad* eingestuft, was insofern zutrifft, dass man durch den hohen Aufwand abschätzen kann, was noch zu tun ist, um ein gutes Produkt zu entwickeln. In Projekt B lassen die vielen gefundenen Fehler bei geringem Testaufwand auf eine niedrige Qualität bzw. insgesamt viele Fehler in der Software schließen. Dies ist nach Kan der *worst case*. Projekt C wird als *unsure* klassifiziert. Es ist keine Aussage bzgl. der Ursache der geringen Anzahl entdeckter Fehler möglich. Es kann nicht entschieden werden, ob nur wenige Fehler gefunden wurden, weil nur wenige Fehler in der Software enthalten sind, oder weil nicht ausreichend getestet wurde. Das Projekt D stellt den aus Sicht der Qualitätssicherung anzustrebenden Fall dar. Die Kombination aus hohem Testaufwand und wenigen gefundenen Fehlern bietet die höchste Sicherheit für ein qualitativ hochwertiges Softwareprodukt.

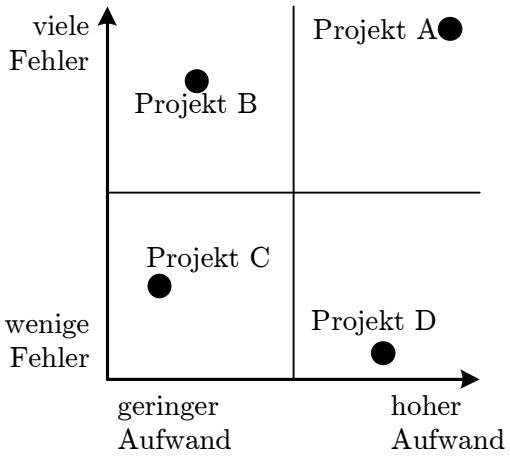

Abbildung 2.10.: Effort-Output-Matrix nach [Kan03].

Die zweite Unterteilung betrachtet die Datenbasis bzw. das Untersuchungsobjekt als wesentliches Kriterium. Als Beispiele für fehlerbasierte Kennzahlen können die Anzahlen gefundener Fehler pro Release, pro Iteration, pro Schwere oder eingeteilt nach anderen Fehlerattributen dienen. Auch zeitliche Aspekte können hierunter fallen, indem beispielsweise die durchschnittlich benötigte Zeit zur Behebung eines gefunden Fehlers und der Durchführung eines Re-Tests analysiert wird. Testfallbasierte Kennzahlen orientieren sich an den Testfällen und betrachten deren Status, die Anzahl geplanter und durchgeführter Tests, den Prozentsatz Fehler aufdeckender Testfälle etc. Daraus lassen sich Rückschlüsse auf den Testfortschritt und die Testgüte ermitteln und je nach Verfeinerung eine ganze Reihe weiterer Informationen gewinnen. Die Kategorie der testobjektbasierten Kennzahlen bezieht sich meist auf den Code in Kombination mit den Testfällen. Daraus lassen sich verschiedene Testüberdeckungsmaße ableiten. Auch der Prozentsatz abgedeckter Installationsvarianten beispielsweise kann analysiert werden.

Eine weitere Einteilung von Testmetriken ergibt sich aus der Beschreibung der Ansatzpunkte für die Kennzahlen. Dies ist in Abb. 2.11 dargestellt. Zunächst, siehe *1* in Abb. 2.11, kann die Testphase als Teil des Gesamtentwicklungsprozesses aus der Vogelperspektive betrachtet werden. Von Interesse sind hier Kennzahlen, die die zeitliche Projektplanung unterstützen oder die benötigten Ressourcen modellieren. Die Verteilung der Anzahl Tests pro Testrequirement steht bei *2* im Mittelpunkt und sollte die Priorisierungen der Testrequirements widerspiegeln. Eine wichtige Testanforderung sollte durch eine entsprechend ho-

he Anzahl an Testfällen abgedeckt sein. *3* bezieht sich auf die Testdurchführungen und die Testergebnisse. Auswertungen an dieser Stelle können der Bewertung des Testfortschritts oder der Qualität des Produkts dienen. Eine gängige Metrik in diesem Bereich ist die Darstellung des Teststatus in sog. S-Kurven [ED96]. Dabei wird der Status der Testfälle, meist *passed*, *failed* oder *not executed*, in einem Diagramm dargestellt. Die Anzahl der durchgeführten Tests folgt einem natürlichen Wachstum, welches in einem S-förmigen Schaubild resultiert. Unter *4* fallen Fehleranalysen – deren Verteilung auf verschiedene Testanforderungen, Testfälle, Arten von Testfällen, deren Bearbeitungszeiten, deren Verteilung bzgl. Status oder Schwere, etc.

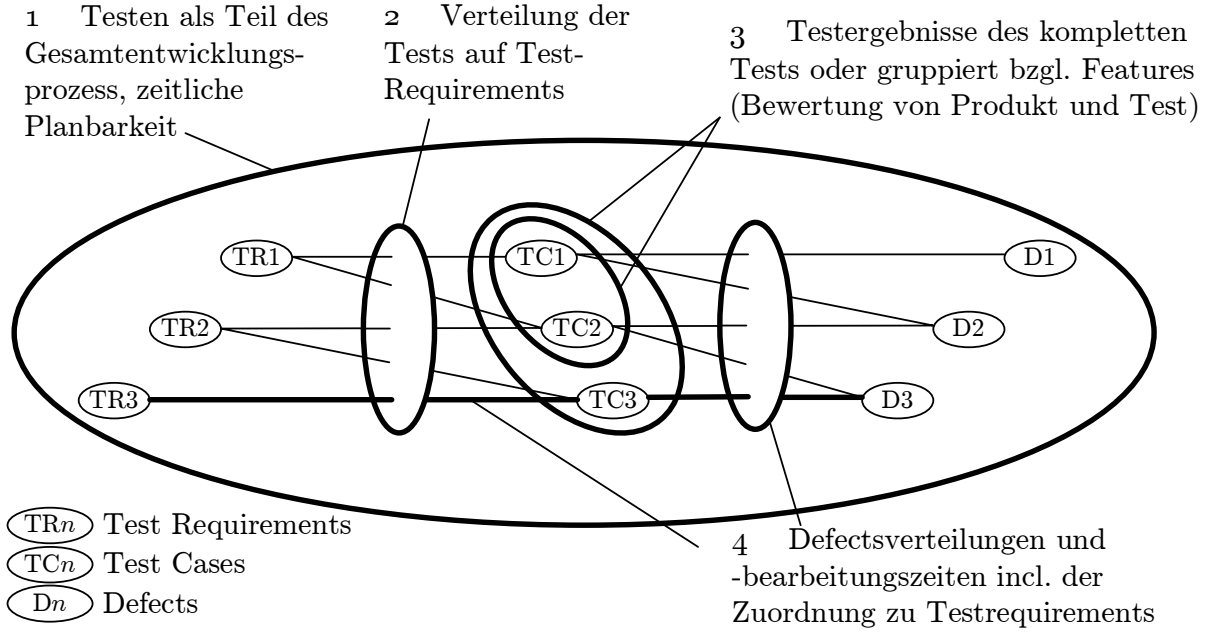

Abbildung 2.11.: Kategorisierung von Testmetriken anhand der von ihnen betrachteten Artefakte.

2.6.4. Zusammenfassung

In diesem Abschnitt wurde auf Softwaremetriken, -maße und -kennzahlen eingegangen. Dabei wurden zunächst verschiedene Sichtweisen und Definitionen erläutert, deren Auswirkung auf die Praxis und die dort zu erreichenden Ziele meist jedoch nur von untergeordneter Bedeutung sind.

Anhand verschiedener Kategorien von Kennzahlen wurden verschiedene Einsatzmöglichkeiten deutlich. Nach den Gruppen der Prozess-, Projekt- und Produktmetriken wurden die Testmetriken eingehender betrachtet. Diese wurden anhand der Ansatzpunkte und der zu analysierenden Artefakte eingeteilt. Exemplarisch wurden jeweils spezifische Vertreter vorgestellt. Der Bereich der Testmetriken stellte sich als ein weites Feld dar, wobei viele der Kennzahlen nur in manchen Projekten oder für manche Produkte sinnvoll sind. Eine generelle Anwendung von Testmetriken scheint jedoch aufgrund der vielfältigen Einsatzmöglichkeiten in den meisten Projekten durchaus angebracht zu sein.

3. Die Qualitätsbewertung von Softwareplattformen

In diesem Kapitel werden zunächst die Probleme, die sich bei der Qualitätsbewertung von Softwareplattformen ergeben, dargestellt. Diese ergeben sich meist aus Unterschieden und Schwierigkeiten beim Testen einer Softwareplattform im Vergleich zu herkömmlichen Anwendungen, aber auch Schwierigkeiten bei der Entwicklung erschweren die durchgängige Modellierung der Anforderungen und somit die Zuordnung von Fehlern zu bestimmten Plattformfunktionalitäten. In Abschnitt 3.2 wird anschließend der Stand der Wissenschaft und Technik, der in den Grundlagenkapiteln dargestellt wurde, diskutiert. In diesem Rahmen findet auch eine Abgrenzung des eigenen Ansatzes statt. Der Darstellung einiger auch in der Plattformentwicklung einsetzbarer Qualitätskennzahlen wird ein eigener Abschnitt gewidmet. An dieser Stelle lassen sich viele Ideen aus der Anwendungsentwicklung übernehmen. Den Abschluss dieses Kapitels bildet in 3.3 die Zusammenfassung der Ausgangslage.

3.1. Plattformen und Anwendungen – Gemeinsamkeiten und Unterschiede

- Metrikeinsatz

  Der Einsatz von Metriken unterscheidet sich bei Plattformen und Anwendungen im Grunde genommen nicht voneinander. Beides sind Softwaresysteme, deren Design zunächst in Architekturdokumenten entworfen wird und die anschließend entwickelt, integriert und getestet werden. Es können auf allen Ebenen mehr oder weniger dieselben

Metriken angewendet werden. Es ist jedoch sinnvoll, bei der Auswahl von Metriken speziell auf Plattformen einzugehen. Dies liegt darin begründet, dass bei Plattformen verschiedene Faktoren und Entwicklungsziele anders gewichtet sind. Auch die Umsetzung der Metriken sollte diese unterschiedlichen Profile sinnvollerweise berücksichtigen. Dadurch ergeben sich Metriken, die in ähnlicher Weise bei Anwendungen eingesetzt werden, bei Plattformen jedoch beispielsweise einen zusätzlichen Parameter besitzen oder sich in anderen Punkten leicht unterscheiden. Die Punkte, die Variationen beim Metrikeinsatz nach sich ziehen können, werden im Folgenden beschrieben.

- Entwicklungsziele

  Wie bereits angedeutet, haben Plattformen in aller Regel andere Entwicklungsziele als Anwendungen. Zunächst werden Plattformen im Gegensatz zu Anwendungen nicht für den direkten Einsatz durch Endanwender entwickelt. Der Fokus liegt vielmehr auf dem Einsatz in Anwendungen bzw. in der Entwicklung von Anwendungen. Die Benutzer der Plattform sind damit Entwickler und Anwendungen oder andere Softwaresysteme. Dementsprechend werden auch die Schnittstellen gestaltet. Beispielsweise gibt es in der Regel keine grafische Benutzeroberfläche.

  Die Wiederverwendung des Codes für mehrere Anwendungen stellt einen zentralen Punkt bei der Entwicklung der Plattform dar. Dies kann erreicht werden durch eine hohe Flexibilität und ggf. durch verschiedene Varianten der Plattform. Auch bei Anwendungen ist meist ein weit verbreiteter Einsatz von großem Interesse. Dieser wird jedoch nicht nur durch eine hohe Flexibilität (sondern beispielsweise auch durch ein geschicktes Marketing) erreicht, wodurch die zielgerichtete Entwicklung eines flexiblen Systems nicht derart im Mittelpunkt steht.

  Auch die Durchsetzung von Standards, Sicherheitsrichtlinien, die Verwendung von Standardprodukten, Standardbetriebsszenarien, etc. ist oftmals ein Argument für die Entscheidung, eine Plattform zu entwickeln. Dieser Punkt steht im Widerspruch zu einer hohen Flexibilität, da diese hierbei bewusst eingeschränkt wird. Es ergibt sich somit eine Nebenbedingung für den o. g. Punkt der Flexibilität.

- Plattformsystemtest und Anwendungsintegrationstest

  Beim Systemtest von Plattformen werden aufgrund der bis dahin noch fehlenden Appli-

kationen so gut wie alle Testfälle indirekt über Testapplikationen durchgeführt. Diese sind notwendig, da im Falle der direkten Kommunikation über die Plattformschnittstellen die Testfälle jeweils nur als Whitebox-Tests einzelne oder kleine Gruppen von Komponenten testen könnten. Die Sicherstellung der korrekten und performanten Funktionsweise ganzer Features wäre auf dieser Basis nur durch den Einsatz umfangreicher und zum Teil komplexer Testfälle und Szenarien möglich. Der Nachteil des Gebrauchs der Testapplikationen ist, dass dadurch die Systemtestfälle nur indirekt auf die Plattform zugreifen.

Der Systemtest von Anwendungen gestaltet sich grundlegend anders (ohne Testapplikationen). Verglichen werden können jedoch der Plattformsystemtest und der Anwendungsintegrationstest. Bei letzterem wird nicht das gesamte System, sondern nur einzelne Komponenten oder Subsysteme getestet. Die jeweils benachbarten (und für den Test benötigten) Komponenten und Subsysteme werden durch Stubs und Mock-Objekte ersetzt. Diese sind durchaus mit Testapplikationen zu vergleichen. Ein entscheidender Unterschied ergibt sich jedoch auch hier: Bei der Entwicklung der Stubs bei Anwendungsintegrationstests ist das zu simulierende Subsystem bereits bekannt (da es bereits spezifiziert wurde). Beim Einsatz der Testapplikationen bei den Plattformsystemtests sind die zu simulierenden Systeme, die auf der Plattform aufbauenden Anwendungen, noch nicht bekannt.

- Systemtests von Produktlinienplattformen

  Einen Spezialfall bilden die Systemtests von Produktlinienplattformen. Hier besteht oftmals der Anspruch, die Testfälle bei den Tests der Anwendungen, die auf Basis der Plattform entwickelt werden, wieder zu verwenden. Dies stellt eine besondere Herausforderung an das Design der Testfälle, die bei diesem Vorgehen ähnlich flexibel (bzw. parametrisierbar) sein müssen, wie die Plattformen selbst.

- Variation der Umgebung des Zielsystems

  Im Idealfall werden die Systemtests auf einer mit dem Produktivsystem identischen Umgebung (bzgl. Hardware und benachbarter Softwaresysteme) durchgeführt. Dies führt bereits bei Anwendungen oft zu Schwierigkeiten, da die Variationsvielfalt dieser Faktoren immens groß sein kann. Bei Plattformsystemtests wird diese Größe noch

beeinflusst durch die Anzahl an Anwendungen als Multiplikator, da theoretisch alle Variationsmöglichkeiten aller Anwendungen auf der Plattform betrachtet werden müssen. Dies trifft jedoch nicht in vollem Umfang zu, da durch Einschränkungen für die Plattform, beispielsweise die Entscheidung, dass die Plattform nur auf einem bestimmten Betriebssystem eingesetzt werden soll, auch die Einsatzmöglichkeiten von Anwendungen eingeschränkt werden. Die Sachverhalte bei Plattformen und Anwendungen sind an dieser Stelle somit mehr oder weniger vergleichbar.

- Zuverlässigkeitstests und NFRs

  Die Zuverlässigkeit ist ein für die Plattformqualität außerordentlich wichtiger Punkt. Dessen Überprüfung wird meist im Rahmen des Systemtests anhand von nichtfunktionalen Tests umgesetzt.

  In der Regel werden für Anwendungen NFRs angegeben, anhand derer die nichtfunktionalen Tests aufgestellt und bewertet werden. Als Beispiel kann hier die Antwortzeit von Anfragen an ein Internet-Portal dienen. Zumindest für größere Softwareprodukte werden viele der Testfälle direkt bezogen auf NFRs definiert. Das Problem bei Plattformen ist nun, dass es während der Entwicklung und des Tests noch keine auf der Plattform aufbauende zu testende Applikation gibt. Diese soll erst später von den Anwendern der Plattform erstellt werden. Und im weiteren Verlauf wird es auch nicht nur eine Applikation geben, die diese Plattform nutzt, sondern mehrere, die unter Umständen für sehr unterschiedliche Aufgaben eingesetzt werden[1].

  Ohne eine konkrete Applikation können in der Regel auch keine NFRs für die Endanwender festgelegt werden. Bei Plattformen muss an dieser Stelle mit Erfahrungs- und Schätzwerten gearbeitet werden. Da die NFRs für die im weiteren Verlauf auf Basis der Plattform entstehenden Anwendungen indes im Nachhinein für die Plattform gelten, müssten bei dieser Schätzung alle Einsatzszenarien aller erwarteter Anwendungen mit einbezogen werden. Dieser Multiplikator stellt ein großes Hindernis für die Praktikabilität eines solchen Vorgehens dar, da die Menge an (potentiellen) NFRs unüberschaubar groß wird. Daraus ergeben sich Probleme bei der Auswahl der Testfälle und der zu betrachtenden Messwerte bei Performance- und Stabilitätstests. Da über zu erwartende

[1] Ansonsten hätte sich die Entwicklung der Plattform nicht gelohnt.

Anwendungsszenarien der Plattform in vielen Fällen nur Mutmaßungen angestellt werden können und die Gewichtung der zumindest theoretisch in Tests prüfbaren Aspekte im voraus nicht bekannt ist, ist somit auch die Auswahl der nichtfunktionalen Testfälle schwierig.

Diese theoretisch unüberschaubare Anzahl an geschätzten NFRs hat nicht nur Auswirkungen auf die Auswahl der Testfälle, sondern auch auf die Bewertung, ob ein durchgeführter, nichtfunktionaler Testfall bestanden ist oder nicht. Es kann zwar beispielsweise die Antwortzeit eines Aufrufs der Plattform auf einer bestimmten Hardware gemessen werden, die Aussage, ob der Wert gut genug ist, kann jedoch nicht sicher getroffen werden. Die Bewertung der Testergebnisse ist somit an dieser Stelle unscharf.

- Einsatz von Testapplikationen

  Ein Nachteil, der sich aus dem Einsatz von Testapplikationen ergibt, ist die erschwerte Lokalisation eines gefundenen Fehlers. Dieser kann stets auch in der Testapplikation selbst seinen Ursprung haben. Dadurch lässt sich zunächst keine sichere Aussage über die Korrektheit der Funktionsweise der Plattform machen. Bezogen auf die Qualitätsbewertung der Plattform muss ein aufgrund einer fehlerhaften Testapplikation nicht bestandener Testfall als nicht durchgeführt bewertet werden. Dieser Punkt trifft in ähnlicher Weise auch bei Tests von Anwendungen zu, da auch hier die Testdaten, das Testskript, bei Anwendungsintegrationstests die Stubs, usw. fehlerhaft sein können. Tendenziell trifft dieser Punkt bei Plattformen jedoch häufiger zu, da nahezu alle Systemtests anhand von Testapplikationen (und zusätzlich Testdaten, Testskripten usw.) durchgeführt werden.

  Auch ein bestandener Testfall muss immer vor dem Hintergrund bewertet werden, dass die Testapplikation evtl. nicht valide ist. Der Testfall würde in diesem Fall seinen Zweck nicht erfüllen können. Dieses Problem ist beim Softwaretest im Allgemeinen nichts Neues. Auch die Testfälle selbst können immer fehlerhaft bzw. die falschen sein. Doch auch hier entsteht durch die Indirektion der Testapplikationen tendenziell eine weitere Unsicherheit. Dieser Umstand hat weitere Folgen. Er erschwert beispielsweise die Bewertung der Testabdeckung, da Fehler aufdeckende Testfälle nicht als nicht bestanden, sondern als nicht durchgeführt einzustufen sind, falls der Fehler in der Testapplikation liegt.

Durch die o. g. Punkte ist nicht nur die Bewertung der Testfälle[2], sondern auch die der gefundenen Fehler schwieriger. Die Fehlerquellen sind vielseitig und es kann tendenziell vermehrt kein Rückschluss auf das Produkt gezogen werden. Die Bewertung der Schwere gefundener Fehler wird stets beeinflusst durch die Anzahl an betroffenen Einsatzszenarien. Auch an dieser Stelle wirkt bei Plattformtests die Anzahl an Anwendungen als Multiplikator, da wieder alle Einsatzszenarien aller auf der Plattform basierenden Anwendungen betrachtet werden müssen.

- Freigabeentscheidung

  Die oben beschriebene Wirkung der Anzahl an Anwendungen auf Basis der Plattform als Multiplikator betrifft auch die Freigabeentscheidung. Diese wird in der Regel auf Basis der Anzahl und der Schwere noch offener Fehler und der erzielten Testabdeckung getroffen. Wie oben beschrieben, sind diese beiden Punkte bei Plattformen durch den beschriebenen Multiplikator erschwert. Zusätzlich verschärft wird diese Situation noch durch die Wichtigkeit dieser Entscheidung, da von ihr nicht nur die Anwender der Plattform, sondern auch die Anwender der auf der Plattform basierenden Anwendungen betroffen sind. Die Variation des Releaseumfangs ist hierbei bei Plattformen ein oft eingesetztes Mittel, um den Releasetermin zu halten.

- Kombination verschiedener Plattformen

  Eine Besonderheit der Plattformen, die im Fallbeispiel in Kapitel 5 eingesetzt werden, ist, dass einige Features erst durch die Kombination verschiedener Plattformen realisiert werden. Dies kann auch bei Anwendungen der Fall sein, beispielsweise bei Plugin-basierten Anwendungen, und muss nicht bei allen Plattformen auftreten. Tendenziell ist dies jedoch ein Punkt, der bei Plattformen durch die hohe Flexibilität im Gegensatz zu fertigen Endanwendungen vermehrt auftritt. Das in 2.2.2 dargestellte Plattform-Feature-Modell adressiert diesen Punkt jedoch zur Genüge.

[2] Was ergibt sich aus der Aussage „Testfall nicht bestanden“?

3.2. Diskussion des Stands der Technik und Wissenschaft und Abgrenzung des eigenen Ansatzes

Dieser Abschnitt diskutiert den Stand der Technik und Wissenschaft. Die relevantesten Verfahren, Konzepte, Ideen, etc. aus dem einführenden Kapitel 2 werden vor dem Hintergrund des Ziels dieser Arbeit, der Unterstützung des Qualitätsmanagements durch Testmetriken, dargestellt. Bereiche, die diese Arbeit nur tangieren und für den Kern nicht notwendig sind, werden nicht erneut aufgegriffen.

Im Rahmen der Darstellungen der einzelnen Bereiche findet auch die Abgrenzung des eigenen Ansatzes statt und zu jedem der vorgestellten Bereiche soll kurz erläutert werden, inwiefern er im eigenen Ansatz Verwendung findet oder worin die Unzulänglichkeiten liegen und eine Anpassung auf Plattformen notwendig ist. Vorweg kann festgestellt werden, dass bislang zu keinem der behandelten Bereiche eine Spezialisierung auf Plattformen existiert. In einigen Fällen ist es jedoch auch möglich, bestehende Verfahren und Methoden direkt ohne Modifikation zu verwenden.

In Kapitel 2.1 wurde ein grober Überblick über Softwarekategorien und die Einordnung der Plattformen gegeben. Diese sind noch nicht lange verbreitet und es besteht noch kein einheitliches Bild, welche Softwaresysteme zu Softwareplattformen zu zählen sind und welche nicht. Auch die in dieser Arbeit verwendete Definition der OMG hat sich noch nicht durchgesetzt. Es ist jedoch abzusehen, dass eine zumindest ähnlich formulierte Definition zum Einsatz kommen wird, weswegen in dieser Arbeit die in den Grundlagen gegebene Definition zur Anwendung kommen kann.

3.2.1. Anforderungen und Features von Software

Im Grundlagenkapitel befindet sich ein Abschnitt über Anforderungen und Features von Software (siehe 2.2). Diese beiden Begriffe werden durch das eigenentwickelte Plattform-Feature-Modell nach [BP06b] miteinander verbunden und aufeinander abgebildet. Die wissenschaftliche Untersuchung der Anforderungen wird bereits seit einigen Jahren stark vorangetrieben. Auch in vielen heutigen Ansätzen zur Softwareentwicklung, wie beispielsweise

dem V-Modell oder der modellgetriebenen Entwicklung, wird das Thema Anforderungen und Anforderungsmanagement angesprochen. Letzteres beinhaltet die Formulierung, die Modellierung, die Abstimmung mit Kunden oder dem Marketing, die Anpassung bei gefundenen Fehlern, die Planung auf Basis von Aufwandschätzungen etc.

Diese Fülle an Informationen, an Modellen und Vorgehensweisen steht im Gegensatz zum Mangel an Beschreibungen, wie mit Features umgegangen werden soll. Die featureorientierte Softwareentwicklung bildet noch einen relativ jungen Wissenschaftszweig, findet jedoch immer mehr Anwendung. Die Featureorientierung findet indes an zwei Fronten statt. Zum einen können Features Auswirkungen auf eine oder mehrere Komponenten haben und müssen an diese darunterliegende Design-Schicht angepasst sein. Zum anderen muss die Integration von Features in Plattformen geplant werden.

Bisherige Feature-Modelle unterstützen die Zuordnung zwischen Anforderungen, Features und Komponenten. Diese Zusammengehörigkeit wird mit Hilfe von Traceability-Links erreicht [SR02]. Die Integration in Plattformen wird dadurch jedoch noch nicht dargestellt. Diese ist aber ein wichtiges Glied in der Kette zwischen Kunden und Entwicklungsteam, da „sie eine Brücke bauen zwischen einer externen Sicht auf das ausgelieferte Produkt im Kontext des Features und einer internen komponentenbasierten Sicht des Entwicklungsteams“ [BP06b].

Es ergibt sich somit aus featureorientierter Sichtweise die Notwendigkeit, sich über die Kopplung von Features und Plattformen Gedanken zu machen. Daraus entstand das eigenentwickelte Plattform-Feature-Modell, um die Lücke zwischen der Anwendersicht auf die Features und der Entwicklersicht auf die Plattformanforderungen zu schließen.

Fazit für Plattformen

Das Plattform-Feature-Modell bietet die Möglichkeit, die Durchgängigkeit von Anforderungen in der Plattformentwicklung zu modellieren. Es bezieht hierbei bereits die zur Realisierung mancher Features notwendige Kombination verschiedener Plattformen mit ein. In der Literatur gab es bislang kein Modell, das dies gezielt ermöglicht.

3.2.2. Qualitätsattribute

Weitere Normen, die das Kerngebiet dieser Arbeit zumindest anschneiden, sind die Standards im Umfeld der ISO/IEC 25000. Die ISO/IEC 25010 bzw. 9126 [ISO05b; ISO01b] stellt ein grundlegendes Qualitätsmodell bereit und definiert für die einzelnen Qualitätsattribute eine Reihe an möglichen Kennzahlen. Obwohl die Norm ein anderes Vorgehen vorsieht, liefert das Qualitätsmodell und die dazugehörigen Kennzahlen eine gute Basis, die durch entsprechende Veränderungen an die für Softwareplattformen spezifischen Anforderungen angepasst werden können. Dieses Vorgehen wird in Kapitel 4 beschrieben.

An der ISO 9126 gibt es jedoch Kritikpunkte, die auch mit der Einführung der ISO/IEC 25010 noch Bestand haben. Nach Johansson [JWBH01] sind die Definitionen der Qualitätsanforderungen in der Norm schwer zu verstehen und sie erlauben einen gewissen Interpretationsspielraum. Sie ohne Modifikation einzusetzen würde somit ein erhebliches Risiko implizieren.

Johansson untersucht weiterhin die einzelnen Qualitätsattribute der ISO 9126[3] auf zwei Aspekte hin. Es wird grundlegend davon ausgegangen, dass in jedem Unternehmen verschiedene Interessengruppen existieren, was bei der Bestimmung der gewünschten Qualitätseigenschaften zu einer fehlerhaften Balance zwischen den einzelnen Attributen führen kann. Dies soll in seiner Studie überprüft und damit die Notwendigkeit gezeigt werden, sich über die verschiedenen Anforderungen der verschiedenen Interessengruppen an Plattformen im Klaren zu sein.

Zum einen wird untersucht, ob es einen Unterschied gibt, wie verschiedene Interessengruppen die Qualitätsanforderungen bei der Entwicklung einer Softwareplattform priorisieren. Der zweite Aspekt, der untersucht werden soll, ist, ob es einen Unterschied gibt, welche Qualitätseigenschaften die verschiedenen Interessengruppen an einer Softwareplattform bevorzugen, wenn sie darauf eine Anwendung entwickeln sollen.

Die Zuverlässigkeit ist in der Studie immer als wichtiger Aspekt priorisiert worden, obwohl sie immer teuer sei. Johansson zieht daraus den Schluss, dass eine Plattform nie Teil eines eher marktorientierten Projektes sein solle, da dieses harte Lieferzeit- und Bud-

[3] Wie bereits erwähnt, werden die Attribute von Johansson leicht verändert.

getbedingungen haben könne. Falls eine Plattform dennoch in einem derartigen Umfeld entwickelt wird, sei es wahrscheinlich, dass sie entweder nicht verlässlich sein wird oder ein anderer Aspekt der Plattform darunter leidet.

Die Ergebnisse der Studie legen nahe, dass die Auswirkungen von Qualitätsanforderungen auf die Kosten und den Lieferzeitpunkt der Plattform besser bekannt sind, als die auf die Anwendungen beim Einsatz der Plattform. Dies führt die Autoren zu dem Schluss, dass die falschen Kennzahlen für die Analyse der Qualitätseigenschaften von Plattformen eingesetzt würden, da für die Endbenutzer und den Unternehmenserfolg die Anwendungen die relevanten Softwaresysteme seien und nicht die Plattformen. Die Autoren schließen weiterhin, dass die Interessengruppen nicht wüssten, welche Qualitätsattribute bei der Plattformentwicklung von Relevanz sind.

Eine Möglichkeit dem entgegenzuwirken ist die Etablierung eines Entwicklungsprozesses, der den Einfluss von verschiedenen qualitativen Faktoren bereits im voraus transparent macht. Hierbei sollten auch die Wünsche der unterschiedlichen Interessengruppen berücksichtigt werden. Des Weiteren sollte eine Feedbackschleife vom Einsatz der Softwareplattform zurück an das Entwicklungsteam der nächsten Plattformversion etabliert werden.

Diese Gedanken werden bei der Konzeption der Kennzahlsuite in Kapitel 4 berücksichtigt. Die Daten der Studie finden außerdem in der Priorisierung der Qualitätsattribute in Abschnitt 4.4 Anwendung.

Fazit für Plattformen

Die ISO/IEC 25010 stellt ein grundlegendes Modell von Qualitätsattributen bereit, das auch bei Plattformen zum Einsatz kommen kann. Leichte Modifikationen speziell für Plattformen bieten sich dennoch an.

Die Untersuchung von Johansson stellt durch die Zuordnung von Prioritäten zu Qualitätsattributen von Plattformen bereits eine gute Grundlage bereit, auf deren Basis ein Auswahlverfahren für Kennzahlen aufgebaut werden kann.

3.2.3. Management von Softwarefehlern

Der in Abschnitt 2.4.2 dargestellte Standard IEEE 1044 [IEE93] liefert zum einen ein Vorgehensmodell, wie u. a. mit gefundenen Fehlern umgegangen werden soll. Des Weiteren beinhaltet er eine Menge an Attributen, die für einen gefundenen Softwarefehler dokumentiert werden sollen. Das Vorgehensmodell ist universell einsetzbar, auch das Vorgehen im Praxisbeispiel in Kapitel 5 stimmt grob mit ihm überein. Die Menge an Fehlerattributen umfasst ebenfalls einen universell einsetzbaren Basissatz, der eine Klassifizierung von Fehlern ermöglicht.

Fazit für Plattformen

Die Menge an Fehlerattributen stellt lediglich einen Basissatz dar, der zur Klassifizierung von Fehlern und zu ihrer weiteren Bearbeitung benötigt wird. Es ist nicht sinnvoll, eines dieser Attribute wegzulassen. Vielmehr ist es sinnvoll, diese zu erweitern. Insbesondere besteht dabei die Möglichkeit, speziell auf Plattformen einzugehen. Dies wird in Abschnitt 4.2.2 dargelegt. Die Definition der zusätzlichen Fehlerattribute sollte projektspezifisch erfolgen, da auch Plattformprojekte untereinander deutliche Unterschiede aufweisen können, und erfolgt in dieser Arbeit im Rahmen des Praxisbeispiels in Abschnitt 5.2.1.

3.2.4. Auswahl und Auswertung von Metriken

Oft für die Auswahl von Kennzahlen eingesetzt ist die GQM-Methode (vgl. Abschnitt 2.4.2). Einem strikten Top-Down-Ansatz folgend werden Ziele definiert, zu deren Erreichung wichtige Fragen aufgestellt und Kennzahlen entworfen werden. Letztere sollen die Beantwortung der Fragen unterstützen. Dieses Vorgehen ist auch auf die Plattformentwicklung anwendbar und die in Kapitel 4 vorgestellte Vorgehensweise erinnert teilweise durchaus an die GQM-Methodik. Ein grundlegender Unterschied ist, dass GQM die Praktikabilität und Umsetzbarkeit der zu erstellenden Metriken nicht berücksichtigt. Es kann dabei durchaus vorkommen, dass die Kennzahlen zwar in einem zum Teil aufwändigen, dafür nachvollziehbaren und zielorientierten Prozess erarbeitet wurden, letztlich jedoch nie zur Anwendung kommen. Dies liegt in der Umsetzbarkeit begründet, da beispielsweise die Verfügbarkeit von Datenquellen bei der Erstellung der Kennzahlen nicht berücksichtigt

wurde. Die GQM-Methode kann jedoch oft auch innerhalb anderer Verfahren unterstützend angewendet werden.

Andere Ausprägungen des Top-Down-Ansatzes sind das *Quality Function Deployment* [Saa97] und der *Software Quality Metrics* Ansatz [IEE98a]. Wie bereits angedeutet, haben diese drei und alle anderen rein zielorientierten Top-Down Ansätze einen Schwachpunkt: Die entwickelten Kennzahlen können zum Teil nur aufwändig ausgewertet werden, da die benötigte Datengrundlage nicht per se gegeben ist. Auch die Datenqualität spielt eine immer mehr in den Fokus wissenschaftlicher Betrachtung rückende Rolle [Eng99; DJ03] und sollte bereits bei der Auswahl der Kennzahlen berücksichtigt werden.

Das Gegenteil dieses Ansatzes, der strikte Bottom-Up Ansatz, sieht vor, dass zunächst verfügbare Daten gesammelt werden. Darauf aufbauend werden Kennzahlen entwickelt, zunächst ohne ein Ziel zu haben, welche Fragen mit ihnen geklärt werden sollen. Dieser Ansatz ist in der Praxis vor zehn Jahren noch verbreitet gewesen [ED96]. Es wurde jedoch erkannt, dass dadurch ein Datenfriedhof aufgebaut wird, anhand dessen sich nur selten herausragende Ergebnisse erzielen lassen. Erst seit dem Einsatz von Data Mining Tools können diese Datenberge analysiert und sinnvoll und zielgerichtet ausgewertet werden [DJ03].

Das in Abschnitt 2.4.2 dargestellte und weit verbreitete ami-Verfahren stellt den kompletten Prozess der Einführung eines Metriksystems dar. Das zwölfstufige Verfahren beginnt mit der Aufnahme der Ausgangssituation und endet erst mit der Verknüpfung der gesammelten Daten und Kennzahlen mit den zuvor gesetzten Zielen. Das in dieser Arbeit vorgeschlagene Vorgehen zur Auswahl von Testmetriken zur Unterstützung des Qualitätsmanagements von Softwareplattformen lässt sich gut in das ami-Verfahren einbetten, berührt dabei jedoch längst nicht alle zwölf Stufen, da sein Fokus weitaus enger gefasst ist. Der von ami empfohlenen Vorgehensweise wird an manchen Stellen bewusst nicht gefolgt, da die Zielsetzungen sich leicht unterscheiden.

Das von ami empfohlene Vorgehen beinhaltet die Anwendung eines CMM(I) Appraisals. Kennzahlen werden im weiteren Verlauf zu den Punkten erstellt, die im Appraisal schlecht bewertet wurden bzw. bei denen eine Verbesserung notwendig ist, um die nächste CMM(I)-Stufe zu erreichen. Es werden dadurch die Kennzahlen ausgewählt, die am besten zu den schlechtesten Bereichen passen. Das eigene Verfahren hingegen ergibt realisierbare Kenn-

zahlen zu Bereichen, die besonders wichtig (für die Plattformentwicklung) sind. Hierbei ergeben sich zum Teil dieselben Kennzahlen, zum Teil jedoch auch unterschiedliche.

Das in dieser Arbeit vorgeschlagene Verfahren beginnt mit der Priorisierung von Qualitätsattributen. Dieser Schritt kann bereits mit der vierten Stufe bei ami, *from goals to sub-goals*, verglichen werden. Die ersten drei Stufen, die Bestimmung des Ausgangszustands, die Definition der Primärziele und deren Validierung, sind durch den engeren Fokus im eigenen Ansatz nicht explizit nötig, da die Zielsetzung, die Unterstützung des QM, bereits gegeben ist. Eine Verifizierung des Zielbaums (ami-Stufe fünf) ist im eigenen Verfahren nicht vorgesehen, da kein expliziter Zielbaum existiert. Eine Entsprechung könnte dennoch erstellt werden. Auf dieser ami-Stufe werden Widersprüche in den Zielen identifiziert. Dies könnte am ehesten verglichen werden mit der Identifikation von widersprüchlichen Qualitätsattributen. Die Integration eines solchen Schrittes ist jedoch nicht sinnvoll, da die Erstellung und Auswertung von Testmetriken allein noch keine abgeleiteten Handlungen beinhaltet, sondern lediglich stets den Ist-Zustand darstellt.

Die Stufe sechs, *from sub-goals to metrics*, entspricht im eigenen Verfahren der Entwicklung der Kennzahlen. Ein kleiner Unterschied ergibt sich durch die unterschiedliche Intention der Kennzahlen: Bei ami werden Kennzahlen zu Sub-Zielen entwickelt, beim eigenen Ansatz zu Qualitätsattributen.

Eine exakte Übereinstimmung der Stufe sieben, *the measurement plan*, ist nicht vorhanden. Eine grobe Zuordnung zur Auswahl der Kennzahlen kann jedoch erfolgen. Der im *measurement plan* aufzulistende Zielbaum hat seine Entsprechung in der Auflistung der zu betrachtenden (hoch priorisierten) Qualitätsattribute. Eine Auflistung der umzusetzenden Kennzahlen wird ebenfalls erstellt. Das hier vorgestellte Verfahren beinhaltet zusätzlich bei der Auswahl der Kennzahlen als zentralen Punkt die Realisierbarkeit der Kennzahlen.

Die Umsetzung und Auswertung der Kennzahlen ist kein Gegenstand des eigenen Verfahrens. Die entsprechenden ami-Stufen besitzen keine äquivalente Stufe.

Die o. g. Zusammenhänge werden in Tabelle 3.1 grob dargestellt.

Fazit für Plattformen

In dieser Arbeit wird bewusst eine Kombination des Top-Down- und des Bottom-Up-

Eigenes Verfahren	**ami**
Priorisierung von Qualitätsattributen	Stufe 4, *from goals to sub-goals*
– nicht vorgesehen –	Stufe 5, *verifying the goal-tree*
Entwicklung der Kennzahlen	Stufe 6, *from sub-goals to metrics*
Auswahl der Kennzahlen	Stufe 7, *the measurement plan*

Tabelle 3.1.: Abbildung des eigenen Verfahrens auf ami

Ansatzes verfolgt, die zum einen zielorientierte, zum anderen realisierbare Kennzahlen liefert. Ein solches Vorgehen ist in der Praxis wohl nicht neu, in der Literatur jedoch noch nicht explizit dargestellt. Es unterscheidet sich dadurch grundlegend von rein zielorientierten Verfahren.

Das ami-Verfahren ist weit umfangreicher als das eigene Verfahren, welches in ami integriert werden kann. Eine Integration des eigenen Vorgehens unterscheidet sich jedoch an einigen Stellen vom empfohlenen „Standard-ami-Verfahren“, wodurch in der Regel andere Kennzahlen ausgewählt werden.

3.2.5. Anwendunng von Testmetriken

Unter den Begriff Testmetriken fallen grundlegend zwei verschiedene Arten von Kennzahlen, eingeteilt nach ihrer Datengrundlage: Kennzahlen basierend auf statischen und dynamischen Testverfahren. In dieser Arbeit werden ausschließlich Kennzahlen basierend auf dynamischen Testverfahren betrachtet. Eine Ausweitung auf statische Verfahren ist jedoch denkbar, weswegen in diesem Abschnitt auch kurz auf diese Kategorie eingegangen werden soll.

Bewertungen basierend auf statischen Verfahren

Bewertungen von Software-Inspektionen, Reviews und anderen Dokumenten können Kennzahlen hervorbringen [Tha00; Mye99; Sch03b]. Diese sind auf Plattformen in gleicher Weise anwendbar wie auf Anwendungen. Software-Inspektionen und Reviews müssen jedoch, um aussagekräftige, vergleichbare und in Kennzahlen anwendbare Ergebnisse zu liefern, formal einwandfrei und nach einem wohldefinierten Prozess ablaufen.

Inspektionen und Reviews können dafür bereits in der Architektur- und Designphase auf Spezifikationen angewandt werden. Dadurch werden die im weiteren Verlauf nur aufwändig zu behebenden Fehler gefunden, wodurch sich die Planbarkeit des Softwareprojektes erhöht. Es kommt zahlenmäßig zu weniger schwerwiegenden Fehlern in der Testphase. Dadurch muss in späteren Phasen weniger Zeit auf die Behebung von Fehlern verwendet werden [Sch03b]. Auf die Testfälle bzw. deren Spezifikationen angewandt, können Inspektionen und Reviews auch die Qualität des Produkts an sich erhöhen. Dies beruht darauf, dass die Testfälle dadurch besser werden, d. h. mit höherer Wahrscheinlichkeit einen Fehler aufdecken. Generell ist der Einsatz von Inspektionen und Reviews gut mit der Anwendung von (dynamischen) Testmetriken kombinierbar.

Weitere Arten von statischen Verfahren sind Korrektheitsbeweise von Programmen und die statische Code-Analyse. Ersteres kann bislang noch nicht komplett automatisiert durchgeführt werden. Die zentralen Aspekte des Beweises müssen manuell bearbeitet werden, weswegen dieses Verfahren in der Praxis bereits für mittelgroße Programme nicht angewandt werden kann. Für das zweite Verfahren wird der Quellcode auf in der Vergangenheit häufig Fehler enthaltende Codefragmente hin automatisiert geprüft. Das Auftreten eines solchen wird als Fehler gemeldet, muss jedoch bei Programmausführung nicht zwangsläufig zu Fehlfunktionen führen. [Ehr07]

Fazit für Plattformen

Die Ergebnisse beider o. g. Verfahren können als Datengrundlage für die Erstellung von Kennzahlen herangezogen werden. Da in beiden ausschließlich der Quellcode betrachtet wird, sind keine Unterschiede bei der Analyse von Anwendungen und Plattformen zu erwarten.

Bewertungen basierend auf dynamischen Verfahren

Testmetriken basierend auf dynamischen Verfahren verwenden als Datengrundlage Testfälle, Testergebnisse, Testdurchführungen, Testzyklen, Testzeiträume, gefundene Fehler und Ähnliches. Alle haben gemein, dass sie im Zuge der Durchführung von Tests, d. h. der Ausführung der Software mit dem Ziel, Fehler zu finden, entstehen. Grundlegend gibt es

an dieser Stelle keine Unterschiede bei der Anwendung von Testmetriken auf Plattformen und Anwendungen. An einigen Stellen empfiehlt es sich jedoch, auf die in Abschnitt 3.1 dargestellten Unterschiede einzugehen.

Testüberdeckungsmaße

Sehr häufig zum Einsatz kommen sog. Testüberdeckungsmaße. Diese Kennzahlen sollen angeben, in welchem Umfang die zu testende Software getestet wurde oder getestet wird. Hierbei gibt es zwei verschiedene Möglichkeiten.

Zum einen kann bei der Durchführung der Testfälle überprüft werden, welche Teile des Quellcodes durchlaufen werden. Abschließend lässt sich dadurch angeben, wieviel Prozent der Statements, der Codezeilen, der Bedingungen, etc. ausgeführt wurden. Solche Verfahren werden meistens im Rahmen von Unit-Tests oder Komponententests angewandt. Hierfür gibt es eine Reihe spezieller (oft von der Programmiersprache abhängiger) Tools. Da bei der Anwendung solcher Kennzahlen der Quellcode betrachtet wird, ergeben sich keine Unterschiede bei Plattformen und Anwendungen. In höheren Teststufen ist dies in der Regel nicht sinnvoll, da das Testziel dort nicht ist, möglichst viele Teile des Codes zu überprüfen, sondern möglichst viele Funktionalitäten der Software bzw. die Anforderungen zu überprüfen.

Eine Möglichkeit, die Testüberdeckung bei den Systemtests zu bewerten, ist die Angabe der Abdeckung der Anforderungen an die Software durch Tests. Dies erfordert ein durchgängiges Entwicklungs- und Architekturmodell (vgl. Abschnitt 2.2). Auch hier gibt es keine grundlegenden Unterschiede in der Anwendung bei Plattformen und Anwendungen. Bei näherer Betrachtung ist hierfür jedoch eine entsprechende Anpassung des Architekturmodells, wie beispielsweise das in 2.2.2 beschriebene, notwendig.

Darstellungen der Anzahl bestandener und nicht bestandener Tests

Ebenfalls sehr oft angewendet werden grafische Darstellungen der Anzahl bestandener, nicht bestandener und noch nicht durchgeführter Testfälle. Diese werden oft wegen ihrer typischen Verlaufsform S-Kurven genannt. Am Anfang der Testphase sind alle Tests noch nicht durchgeführt. Im weiteren Verlauf nimmt diese Anzahl einem natürlichen Wachstum folgend ab. Die bereits durchgeführten Testfälle beinhalten zu Beginn meist deutlich mehr

nicht bestandene Tests als gegen Ende der Testphase. Das Testende ist im Idealfall erreicht, wenn alle Tests durchgeführt und bestanden sind. Wie bereits erwähnt, wird Software oftmals jedoch mit bekannten Fehlern und damit nicht bestandenen Testfällen ausgeliefert.

Auch diese Auswertungen lassen sich auf Plattformen in gleicher Weise anwenden wie auf Anwendungen. Ein zu erwähnender Punkt ist jedoch, dass, wie oben beschrieben, bei Plattformen die Testfälle, die in Testapplikationen Fehler aufdeckten, im Hinblick auf die Plattform zunächst als nicht durchgeführt zu bewerten sind.

Anzahl Wiederholungen eines Testfalls

Bei den Systemtests ist es oftmals hilfreich, die Anzahl der Ausführungen eines Testfalls zu betrachten. Es ist in den meisten Fällen erstrebenswert, diese gering zu halten. Dies liegt darin begründet, dass ein Testfall nur dann mehrmals ausgeführt wird, wenn entweder ein Fehler behoben wurde oder der Test nicht aussagekräftig war, da beispielsweise zwei Tester gleichzeitig auf derselben Umgebung getestet haben. Im ersten Fall muss die Systemtestumgebung oft aufwändig erneut aufgebaut oder umkonfiguriert werden. Der zweite Fall sollte generell vermieden werden. Auszuschließen sind hierbei Wiederholungen im Rahmen von Analysen, da hierbei oft gezielte Änderungen des Testsystems durchgeführt werden, um weitere Erkenntnisse bei der Fehleranalyse zu erhalten.

Kennzahlen dieser Art sind in gleicher Weise auf Plattformen und Anwendungen anwendbar.

Anzahl und Dauer der Testzyklen und des gesamten Testzeitraums

Unter einem Testzyklus ist hierbei die Durchführung möglichst aller Testfälle auf einer wohldefinierten Umgebung zu verstehen. Meist werden alle Testfälle in einem Testzyklus durchgeführt. Anschließend wird die Testumgebung verändert, beispielsweise werden Fehler behoben oder Konfigurationen angepasst. In einem zweiten Testzyklus werden erneut eine Reihe an Tests ausgeführt, die beispielsweise im ersten Zyklus einen Fehler aufdeckten, der nun behoben sein soll. Hinzu kommt eine Reihe an Regressionstests.

Idealerweise werden zwei Testzyklen durchgeführt, ein erster, der Fehler aufdeckt und ein zweiter, in dem alle Fehler behoben sind. In der Praxis sind es jedoch meist mehr als nur zwei. Ebenfalls lohnend ist die Darstellung bezogen auf verschiedene Ausbaustufen

der Testumgebung. In vielen Fällen gibt es beispielsweise eine geclusterte und eine nicht geclusterte Umgebung. Erstere ist erheblich aufwändiger anzupassen, weswegen es sinnvoll ist, v. a. hier die Anzahl an Testzyklen gering zu halten und beispielsweise mehrere Zyklen auf der nicht geclusterten Umgebung durchzuführen.

Betrachtungen der Länge des Testzeitraums, verglichen mit der Länge des Entwicklungszeitraums, des Vorgängerreleases, etc. können ebenfalls sehr hilfreich sein bei der Identifizierung von Prozessverbesserungsmöglichkeiten.

Auch hier lassen sich keine Unterschiede zwischen Anwendungen und Plattformen erkennen.

Auswertungen gefundener Fehler

Gefundene Fehler können auf sehr vielfältige Weise ausgewertet werden. Es können die Verteilungen bzgl. vieler Fehlerattribute herangezogen werden, beispielsweise eingeteilt nach Fehlerquelle, Schwere, Priorität, Zeitpunkt der Aufdeckung innerhalb des Testzyklus etc. Dadurch lassen sich sehr vielfältige Ergebnisse erzielen, die Rückschlüsse auf den Testprozess und die Testfälle ermöglichen. Falls beispielsweise die meisten schwerwiegenden Fehler erst gegen Ende des Testzeitraums gefunden werden, ist es sinnvoll, die Prioritäten der einzelnen Testfälle (und damit ihre Durchführungsreihenfolge) anzupassen.

Bei diesen Kennzahlen können sich insofern Unterschiede bei Anwendungen und Plattformen ergeben, als sich die Fehlerattribute unterscheiden können (vgl. Abschnitt 3.2.2).

Fazit für Plattformen

Die Anwendung von Testmetriken unterscheidet sich bei Anwendungen und Plattformen nur minimal. Bei beiden lassen sich in gleicher Weise Testüberdeckungsmaße auf Unit-Testebene, Betrachtungen bestandener und nicht bestandener Tests, Auswertungen der Anzahl Wiederholungen von Testfällen, der Dauer der Testzyklen und der Anzahl gefundener Fehler einsetzen. Unterschiede gibt es nur durch den vermehrten Einsatz von Testapplikationen (vgl. Abschnitt 3.1) und ggf. unterschiedliche Fehlerattribute bei Anwendungen und Plattformen.

3.3. Zusammenfassung der Ausgangslage

Wie aus den letzten Abschnitten hervorgeht, liefert der Stand der Technik zur Lösung der in den Abschnitten 1 und 3.1 vorgestellten Ziele und Unterschiede bereits einige Ansätze. Doch in vielen Fällen fehlen einige entscheidende Anpassungen. Dies betrifft insbesondere die Auswahl der in einer Vielzahl bereits vorhandenen Metriken, aber zum Teil auch deren Anwendung. Auch die Interpretation der Messwerte ist in vielen Fällen eine Herausforderung. Doch auch in anderen Bereichen, beispielsweise bei der durchgängigen Verknüpfung von Testfällen und Anforderungen, bestehen noch Lücken.

Eine wichtige Grundlage für die generelle Anwendung von Systemtestmetriken ist die Notwendigkeit, Anforderungen, Features, Plattformen und letztlich auch Testfälle miteinander zu verknüpfen. Speziell für Plattformen entwickelte Verfahren existieren abgesehen vom Plattform-Feature-Modell bislang noch nicht. Dieses bildet einen speziell abgestimmten Rahmen.

Ein weiterer wichtiger und noch offener Punkt ist die differenzierte Entwicklung eines Plattform-Qualitätsverständnisses. Johanssons Untersuchung der Wichtigkeit von Qualitätsattributen [JWBH01] der ISO Norm 25010 bzw. 9126 [ISO05b; ISO01b] stellt hierzu eine gute Grundlage bereit, auf der aufbauend diese Qualitätssicht entwickelt werden kann.

Die Grundlage für den in dieser Arbeit gewählten Ansatz bilden die Softwaretests. Das Ziel von Softwaretests ist stets, Fehler in der Software zu finden und dadurch Qualitätsdefizite aufzudecken. Diese Qualitätsdefizite oder -lücken bilden eine Grundlage, anhand derer sich die Qualität erschließen lässt. Die gefundenen Fehler besitzen stets eine Reihe an Attributen. Wie oben angedeutet ist hierbei eine Anpassung bzw. Erweiterung bezogen auf Plattformen sinnvoll. Auch an dieser Stelle gibt es in der Literatur noch keine speziellen Attribute. Überlegungen zu speziellen Qualitätsattributen von Plattformen sind dabei zu berücksichtigen.

Speziell für Plattformen entwickelte Auswahlverfahren für (Test-)Metriken gibt es bislang nicht. Doch lassen sich die bereits existierenden Verfahren durchaus auch auf Plattformen anwenden. Eine Einbettung speziell angepasster Vorgehen in bestehende Verfahren, wie beispielsweise das eigene Vorgehen in das ami Verfahren, sind möglich und bieten eini-

ge Vorteile: Durch die Verwendung eines bestehenden und in der Praxis weit verbreiteten Verfahrens existieren einige Beschreibungen und Erfahrungsberichte und man verlässt sich auf ein bewährtes Vorgehen. Durch dessen Anpassung bzw. durch die Einbettung eines speziell für Plattformen entwickelten Verfahrens wird zusätzlich eine bessere Anpassung an die Plattformbedürfnisse erreicht.

Spezielle Verfahren für die Auswahl von Testmetriken gibt es ebenfalls noch nicht. Dies ist auch nicht zwingend erforderlich. An dieser Stelle können allgemeine Verfahren eingesetzt werden, da in der Regel das Auswahlverfahren zunächst nicht die Kennzahlen selbst, sondern Qualitätsattribute, zu bewertende Bereiche, zu beantwortende Fragen oder Ähnliches liefert. Erst in einem zweiten Schritt werden dazu Kennzahlen erstellt. Eine Einschränkung auf Testmetriken ist dann problemlos möglich, wobei beachtet werden muss, dass nicht alle Fragen mit Testmetriken beantwortet werden können.

Testmetriken sind jedoch sehr vielseitig einsetzbar, da v. a. die Sytemtests die Software gegen ihre Anforderungen prüfen und damit eine Bewertung des Produkts ermöglichen. Eine Untersuchung des Testprozesses, eingebettet in den gesamten Software-Entwicklungsprozess, bietet darüber hinaus die Möglichkeit, Qualitätsattribute des Prozesses zu betrachten. Mit anderen Worten bilden Testmetriken eine Möglichkeit, basierend auf bereits vorhandenen Datenquellen, die Qualität von Softwareplattformen und ihres Entwicklungsprozesses zu überwachen.

Der Einsatz von Testmetriken bietet jedoch noch weitere positive Aspekte. Zum einen werden, wie bereits erwähnt, ausschließlich bestehende Datenquellen ausgewertet. Zum anderen werden diese Datenquellen selbst näher untersucht und die bei Plattformen aufwändigen Tests ggf. besser ausgenutzt.

Die Bewertung der Qualität der Plattform und die Verbesserung der Planbarkeit dienen in vielen Fällen der Entscheidung, ob die Software für den Produktiveinsatz freigegeben werden kann oder nicht. In Systemtests wird dieser Produktiveinsatz meist simuliert, weswegen weitergehende Auswertungen der Testergebnisse in Form von Testmetriken auch als Unterstützung bei der Freigabeentscheidung ein ideales Mittel bilden.

Die Anwendung von Testmetriken indes unterscheidet sich kaum bei Anwendungen und Plattformen. Nur die Auswertung der Anzahlen bestandener und nicht bestandener

Testfälle und die Auswertungen gefundener Fehler unterscheiden sich leicht. Bei Ersteren muss berücksichtigt werden, dass bei Plattformen vermehrt Testapplikationen eingesetzt werden und dass ein aufgrund einer fehlerhaften Testapplikation nicht bestandener Testfall für die Plattform als nicht durchgeführt zu bewerten ist. Bei Letzteren ist eine ggf. vorgenommene Anpassung der Fehlerattribute zu berücksichtigen.

4. Auswahlprozess von Testmetriken für Softwareplattformen

In diesem Kapitel wird beschrieben, wie die Kennzahlen für Plattformen auszuwählen sind. Die Notwendigkeit, die Kennzahlen gezielt auszuwählen, liegt darin begründet, dass eine große Anzahl an Kennzahlen in den meisten Softwareprojekten oder -entwicklungsabteilungen angewendet werden können. Daraus ergibt sich die Gefahr, mehr Daten anzuhäufen als man auswerten kann oder möchte. Bei einem zu großen „Datenberg“ verliert man leicht den Überblick und kann nur noch schwer sinnvolle Rückschlüsse ziehen. Die für das bestimmte Projekt, Produkt oder Entwicklungsteam wichtigsten Kennzahlen gehen in diesem Fall unter. Das ist zum einen für die Mitarbeiter, die einen Teil ihrer Arbeitszeit mit dem Erfassen und Auswerten von Kennzahlen beschäftigt sind, unbefriedigend und zum anderen teuer. Außerdem wird das Produkt nicht dadurch allein qualitativ besser, dass Kennzahlen definiert und angewandt werden. Vielmehr muss bereits im Vorfeld Zeit für die Analyse der Ergebnisse und die Ausarbeitung und Umsetzung von Verbesserungsmöglichkeiten eingeplant werden. Auch in [Lig02] wird hervorgehoben, dass es zielführender ist, zunächst nur wenige wohlüberlegte Kennzahlen anzuwenden und sinnvoll auszuwerten.

Ein grober Überblick über das Auswahlverfahren in dieser Arbeit wird in Abschnitt 4.1 gegeben. Bevor mit der Auswahl begonnen werden kann, muss jedoch der situative Kontext der Qualitätssicherung überprüft und ggf. angepasst werden. Hierzu wird in Abschnitt 4.2.1 das im Grundlagenabschnitt 2.2.2 dargestellte Plattform-Feature-Modell als Basis für featurebasiertes Testen verwendet. Auch die im Test gefundenen Fehler können auf Plattformen und das Erstellen einer Kennzahlsuite ausgerichtet dokumentiert werden. Hierauf wird in Abschnitt 4.2.2 durch die Beschreibung von Fehlerattributen eingegangen.

Ebenfalls erweitert werden können die Qualitätsattribute. Eine spezielle Anpassung an Plattformen findet in Abschnitt 4.2.3 statt.

Der Abschnitt 4.3 stellt den für den Systemtest und dadurch auch für Testmetriken sehr wichtigen Zusammenhang zwischen Qualitätsattributen, Test- und Qualitätsanforderungen dar. Darauf folgt in Abschnitt 4.4 die Priorisierung der Qualitätsattribute und darauf aufbauend schließlich in Abschnitt 4.5 die Auswahl der Metriken.

4.1. Überblick über den Auswahlprozess

Eine Übersicht über den Auswahlprozess der Kennzahlen, der auch im Praxisbeispiel in Kapitel 5 umgesetzt wurde, ist in Abb. 4.1 dargestellt. Das Auswahlverfahren basiert auf der Kombination eines Top-Down- und eines Bottom-Up-Ansatzes. In [BCR94] stellen Basili, Caldiera und Rombach fest, dass „measurement, in order to be effective must be: 1. Focused on specific goals; 2. Applied to all life-cycle products, processes and resources; 3. Interpreted based on characterization and understanding of the organizational context, environment and goals.“ Daraus schließen sie, dass ein Kennzahlensystem, um effektiv zu sein, durch einen Top-Down-Ansatz vorgegeben werden müsse. Die Grundidee ist hierbei zweifelsohne richtig, das strikte Befolgen dieses Vorgehens jedoch ebenso wenig zielführend wie das Auswerten aller Daten, die man ohne großen zusätzlichen Aufwand zur Verfügung hat. Letzteres wäre das strikte Ausführen eines Bottom-Up-Ansatzes. In diesem Kapitel wird eine Kombination der beiden Ansätze beschrieben, die zunächst Top-Down die Ziele und die zu bewertenden Attribute definiert, bei der Auswahl der Kennzahlen aber darauf achtet, dass v. a. auf bestehende oder leicht zugängliche Daten zurückgegriffen wird.

In einem ersten Schritt werden im Auswahlprozess die Qualitätsattribute plattform-, aber auch projektspezifisch priorisiert. Der zweite Schritt sieht die Entwicklung passender Kennzahlen vor, aus denen im dritten und letzten Schritt die umzusetzenden ausgewählt werden.

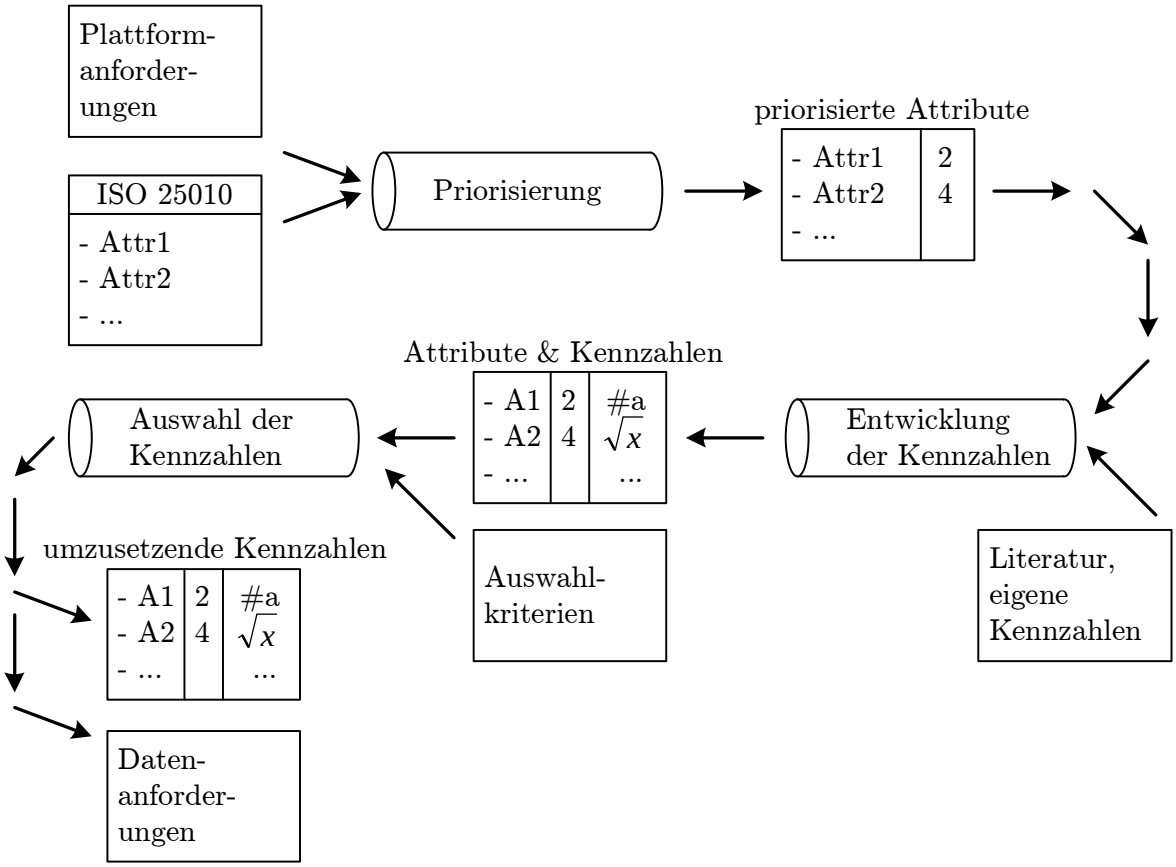

Abbildung 4.1.: Kennzahl-Auswahlprozess.

4.2. Gestaltung des situativen Kontexts

4.2.1. Featurebasiertes Testen

In der Softwareentwicklung werden Tests auf unterschiedlichen Ebenen des Entwicklungsprozesses angewandt. In Unit-Tests werden einzelne Klassen, Funktionen, etc. isoliert durch den Programmierer auf ihr Verhalten hin getestet. Dabei dienen bestehende Unit-Tests zum einen als Absicherung von bereits implementierter Funktionalität bei Änderungen, zum anderen haben sie die Aufgabe, die Neu- oder Weiterentwicklung einer Unit zu prüfen. *Code Coverage*-Analysen sind auf dieser Ebene hilfreich, um die Testabdeckung zu ermitteln und ungetestete Bereiche zu identifizieren. Diese werden auf dieser Ebene mit Erfolg angewandt. Eine Zuordnung zu Features findet auf dieser Ebene noch nicht statt und ist auch nur bedingt sinnvoll, da durch diese Tests ein Feature nur selten vollständig getestet werden kann. Für die Bewertung der Features ist eine Auswertung der Anzahl gefundener

Codefehler dennoch hilfreich.

Die nächste Testebene sind die Komponententests. Hier ist oftmals aus funktionaler Sicht bereits eine eindeutige Zuordnung zwischen Features und Funktionaltests sinnvoll, da neben dem Testen der internen Funktionalität einer Komponente auch die Außensicht anhand der Komponentenschnittstellen zu überprüfen ist. Dies gewährleistet die Überprüfung der möglichen übergeordneten Abläufe. Auf dieser Ebene kann bereits eine featureorientierte Sicht eingeführt werden, da die übergeordneten Abläufe meist im Kontext eines Features entstehen. Auch eine Kombination der beiden Sichtweisen ist denkbar. Der Test eines Features wäre auf dieser Ebene gleichbedeutend mit dem Test derjenigen Komponenten bzw. Komponentenfunktionalitäten, die für das jeweilige Feature notwendig sind. Von einem featureorientierten Standpunkt aus sollten hierbei alle Testfälle durchgeführt werden, die der Featurespezifikation nach die vom jeweiligen Feature vorgegebenen Abläufe überprüfen.

Plattformen bestehen oft aus einer Menge an Komponenten, Basisprodukten und Konfigurationsparametern. Die Features werden durch die Integration dieser Bestandteile realisiert. Das Ziel des Integrationstests ist es, die Interaktionen zwischen diesen Bestandteilen zu testen. Dies kann hauptsächlich aus Featuresicht erfolgen, wodurch im Fokus die Interaktionen liegen, die für die Realisierung der Featurefunktionalitäten notwendig sind.

Wie in Abschnitt 3.1 bereits erläutert, spielen Testapplikationen beim Plattformtest eine wesentliche Rolle. Auch beim Integrationstest ist deren Verwendung sinnvoll, da in vielen Fällen nur unzureichend direkt auf die komponentenübergreifenden Funktionalitäten zugegriffen werden kann. Ein Vorteil, der sich daraus ergibt, ist, dass die Integrationstests bereits aus einer anwendernahen Sicht durchgeführt werden. Allerdings ist durch diese Praxis der Übergang von Integrations- zu Systemtests fließend. Dies wird dadurch noch verstärkt, dass die Testapplikationen meist für funktional orientierte Testfälle (oder Familien von Testfällen) erzeugt werden, welche sich wiederum an den Featurespezifikationen orientieren. Erste Testfälle werden oftmals bereits in der Featurespezifikation selbst festgehalten. Jeder geplante Testfall auf dieser Ebene sollte mindestens einem Feature zuzuordnen sein.

Die höchste Stufe der Plattformtests bilden die Systemtests. Hierbei gibt es eine ganze Reihe an weiteren Testanforderungen. Beispielsweise um die Funktionalität der Plattform

gemäß den Kundenanforderungen auf verschiedenen Betriebssystemen zu testen, muss die entsprechende Infrastruktur zur Verfügung stehen. Hinzu kommen oftmals weitere Infrastrukturkomponenten, wie Proxies, Firewalls, Caches, etc. Das Ziel des Systemtests ist die Überprüfung des Gesamtsystemverhaltens gemäß der Spezifikation der Anforderungen. Aus Anwendungssicht ist die Plattform nur ein Teil des Gesamtsystems und die System- oder -abnahmetests der Plattform sind somit lediglich als Integrationstests anzusehen, da das Gesamtsystem (inklusive der Applikation) darin noch nicht getestet wird. Aus Plattformsicht jedoch ist die Plattform das Gesamtsystem, das durch die Plattformsystemtests geprüft werden soll. Es ergibt sich somit eine Darstellung wie in Abb. 4.2. Die zwei Anforderungsdefinitionen beziehen sich auf Anforderungen der Endanwender an die Anwendung und der Anwendung an die Plattform.

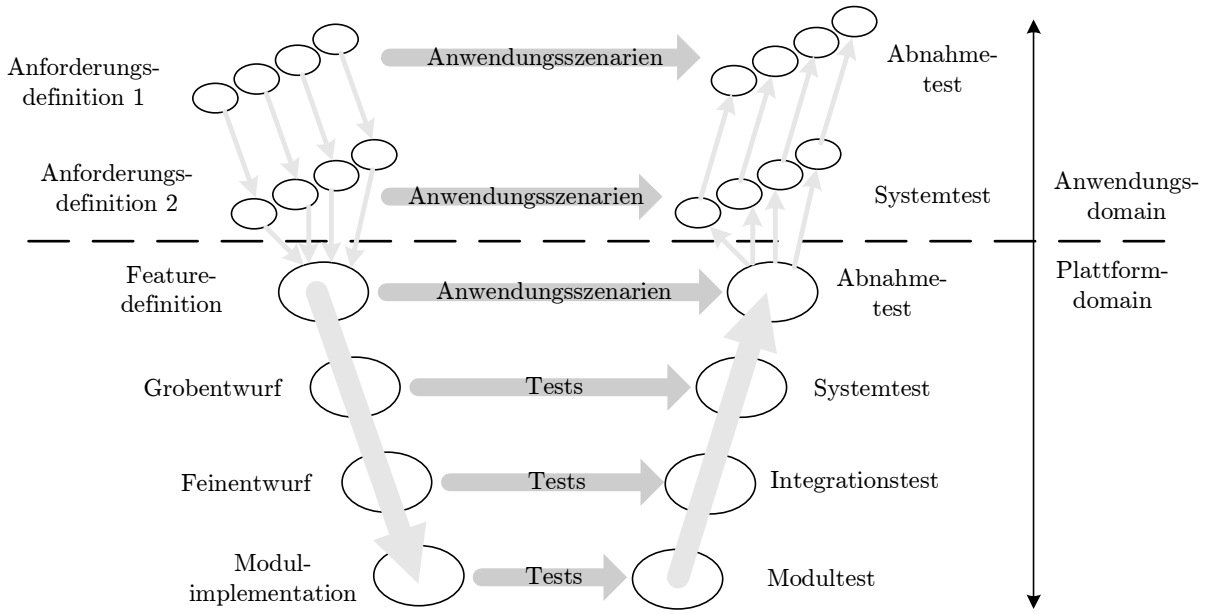

Abbildung 4.2.: Plattformsystemtests im V-Modell.

Wie in Abschnitt 2.2.2 dargestellt, können Features verschiedene Plattformen betreffen. Diese können dadurch nur auf Basis einer Kombination von verschiedenen Plattformen getestet werden. Da immer Plattformreleases getestet werden, wird somit vorausgesetzt, dass die Implementierungen verschiedener Plattformen zum selben Zeitpunkt in der Testumgebung verfügbar sind, was durch die Verfügbarkeit von Hardware und Teammitgliedern begrenzt wird. Dies muss in der Planung und der Architektur der Plattform berücksichtigt

werden.

Die Zuordnung von Testfällen zu je einem Feature ist auf dieser Ebene für viele Funktionaltests ohne Schwierigkeiten darzustellen, da die Testfälle auf den Featurespezifikationen beruhen. Es kommt jedoch vor, dass ein Testfall die korrekte Funktionsweise der Kombination verschiedener Features überprüfen soll. In diesem Fall muss die Darstellung einer *n:m*-Beziehung möglich sein (da ein Feature in der Regel nicht nur durch einen Testfall abgedeckt wird).

Anders verhält es sich bei der Betrachtung der Tests der nichtfunktionalen Anforderungen. Wie in 3.1 dargestellt, basiert deren Auswahl bei Plattformen auf Erfahrungen. Dadurch können viele dieser Testfälle nicht einem oder wenigen Features zugeordnet werden, sondern überprüfen beispielsweise die Stabilität der Gesamtplattform bei einer längeren Laufzeit oder unter Last. Die Zuordnung eines dabei auftretenden Fehlers zu einem bestimmten Feature ist auch im Nachhinein ebenfalls oftmals nicht möglich, sondern betrifft im schlimmsten Fall die gesamte Architektur der Plattform.

4.2.2. Fehlerattribute

Alle bei der Entwicklung herkömmlicher Anwendungen eingesetzten Fehlerattribute, wie Fehlerstatus oder -priorität, können auch beim Plattformtest verwendet werden. Diese können ergänzt werden durch Attribute, die speziell auf Plattformcharakteristiken eingehen. Drei solcher Attribute werden in diesem Abschnitt grob dargestellt. Eine genaue Beschreibung, wie die Fehlerattribute behandelt werden und welche verschiedenen Attributsausprägungen möglich sein sollen, muss jedoch projektspezifisch zusammen mit dem Fehlerlebenszyklus erstellt werden. Dies geschieht im Anwendungsbeispiel in Abschnitt 5.2.1.

Wie bereits erwähnt wird der Releaseumfang bei der Plattformentwicklung oft variiert, um die benötigte zeitliche Planungssicherheit der Auslieferung zu gewährleisten. Es ist oft sinnvoller, die Plattform zeitlich wie geplant auszuliefern, als auf die Behebung aller im Test gefundenen Fehler zu warten. Die gängigen Fehlerattribute von Anwendungen ermöglichen eine Abbildung dieses Sachverhalts nicht. Es ist somit sinnvoll, ein Attribut „Auslieferein-

fluss“ zu definieren, das den Einfluss auf die Auslieferung angibt. Für jeden Fehler kann dadurch modelliert werden, ob und wie eine Plattform auch mit seinem Vorhandensein den Kunden zur Verfügung gestellt werden kann. Mögliche Attributsausprägungen sind:

- „Stopp“: Die Auslieferung kann nicht mit diesem Fehler erfolgen.
- „Workaround“: Den Anwendern der Plattform wird ein vorübergehender Workaround erläutert, der eingesetzt werden kann, bis der Fehler behoben ist.
- „Kein Einfluss“: Der gefundene Fehler hat keine Auswirkung auf die Einsetzbarkeit der Plattform.
- „Release Notes“: Der gefundene Fehler wird in den Release Notes beschrieben.

Im Detail sind die Attributsausprägungen projektspezifisch festzulegen, da beispielsweise nicht jede Plattform mit Release Notes ausgeliefert werden muss.

Als Ergänzung zum Ausliefereinfluss ist die Dokumentation der jeweils nächsten Auslieferung in einem Attribut „nächste Auslieferung“ eines Fehlers sinnvoll, um die offenen Punkte für eine bestimmte Auslieferung im Auge zu behalten. Dies begründet sich dadurch, dass bei jeder Version der Plattform nicht nur die Fehler zu betrachten sind, die während ihrer Entwicklung bzw. den Tests dieser Version auftraten, sondern auch alle noch nicht behobenen Fehler aller Vorgängerversionen können Auswirkungen auf diese Version haben.

Hierbei ist zu beachten, dass dieses Attribut zwar einen guten Überblick über die vorhandenen Fehler der nächsten Plattformversion bietet. Dies ist meist die Plattformversion, die derzeit entwickelt wird. Die für die übernächste Plattformversion relevanten Fehler können dadurch jedoch nicht ohne Weiteres auf einfache Art dargestellt werden. Dies ist insofern ein Nachteil, dass oftmals parallel eine Version entwickelt und die folgende bereits geplant wird.

Plattformen, jedoch auch viele Anwendungen, setzen sich aus diversen Komponenten zusammen, beispielsweise aus eigenentwickeltem Code, aus kommerziellen Basisprodukten, aus Konfigurationseinstellungen, etc. Dadurch bietet sich die Dokumentation der Fehlerquelle in einem entsprechenden Attribut an. Auch hier sind die Attributsausprägungen projektspezifisch. Durch eine solche Darstellung kann meist ein grober Überblick über den

Aufwand zur Behebung der Fehler erlangt werden, da beispielsweise die Änderung eines Konfigurationsparameters einen deutlich geringeren Aufwand darstellt als die Behebung eines Code- oder gar eines Designfehlers.

4.2.3. Erweiterung von Qualitätsattributen für Plattformen

In diesem Abschnitt werden die gängigen Definitionen von einigen Qualitätsattributen erweitert, um die Charakteristiken von Plattformen besser darzustellen. Diese Darstellung baut zum einen auf der ISO/IEC 25010 (siehe Abschnitt 2.4.2) und zum anderen auf der bereits erwähnten Untersuchung von Johansson et. al. [JWBH01] auf.

In der ISO/IEC 25010 (siehe 2.4.2) wird die *Reliability* (Zuverlässigkeit) folgendermaßen definiert: „The capability of the software product to maintain a specified level of performance when used under specified conditions.“ [ISO05b] Die Zuverlässigkeit besteht dort aus den drei Subattributen *Maturity* (Reife), *Fault-tolerance* (Fehlertoleranz) und *Recoverability* (Wiederherstellbarkeit). In dieser Arbeit wird die o. g. Definition nur minimal verändert übernommen:

Definition: Zuverlässigkeit (Reliability)

The capability of the software product to maintain permanently a specified level of performance when used under specified conditions without failures.

Auch die Planbarkeit kann als Qualitätsattribut aufgefasst werden. Sowohl in der gängigen Literatur als auch in Standards und Normen geschieht dies jedoch nicht. Es gibt verschiedene Interpretationsmöglichkeiten der Planbarkeit: Sie kann bezogen auf zeitliche, finanzielle oder qualitative Aspekte aufgefasst werden. Die folgende Definition schließt diese drei Möglichkeiten mit ein:

Definition: Planbarkeit

Die Planbarkeit eines Produkts ist die Eigenschaft, vor und während der Entwicklung fundierte Vorhersagen treffen zu können.

Ein weiteres wichtiges Qualitätsattribut von Plattformen ist die Wiederverwendbarkeit. Johansson ersetzt in [JWBH01] die *Portability* aus der ISO/IEC 25010 durch dieses Attri-

but und versteht darunter: „It shall be possible to reuse the architecture in other applications in the same environment.“ Diese Eigenschaft ist durch die ISO/IEC 25010 in dieser Form noch nicht abgedeckt und könnte am ehesten der *Functionality* oder der *Usability* zugeordnet werden.

Johansson versteht darunter somit die Wiederverwendung einer Plattform in einer bestimmten Umgebung bei mehreren Anwendungen. Diese Sicht kann erweitert werden durch das Weglassen der Bedingung, dass sich die Plattform in derselben Umgebung befinden muss. Dieser Gedanke einer erweiterten Definition der Wiederverwendbarkeit soll in dieser Arbeit gelten, da er einerseits die Eigenschaft der *Portability* aus der ISO/IEC 25010 integriert und andererseits das Einsatzszenario von Plattformen in der Praxis besser widerspiegelt:

Definition: Wiederverwendbarkeit

The capability of the software product to reuse the architecture in other applications.

Die Umgebung ist dabei nicht komplett beliebig, kann aber mitunter stark variieren. Diese Sichtweise ist in Abb. 4.3 der Definition von Johansson gegenübergestellt. Das π-ähnliche Symbol für die Plattformen ist dem *MDA User Guide* [Obj03] entnommen.

4.3. Qualitätsattribute, Test- und Qualitätsanforderungen

Bevor die Qualität eines Produkts verbessert werden kann, muss zunächst das Ziel herausgearbeitet werden. Dazu muss untersucht werden, was die Hauptaufgaben des untersuchten Gegenstands sind. Auf dieser Basis ergeben sich Anforderungen an die Qualität. Diese Anforderungen sollten dann im Rahmen der Qualitätssicherung überprüft werden. Dies trifft auch auf Plattformen zu.

Einer der Hauptzwecke einer Plattform ist, durch eine hohe Wiederverwendungsrate Effizienzsteigerungen zu erzielen. Daraus ergeben sich bereits mehrere Anforderungen an die Qualität: Auf der einen Seite muss die Plattform flexibel einsetzbar sein, da ansonsten die Wiederverwendung entweder nicht in befriedigendem Maße stattfindet oder die Benutzer

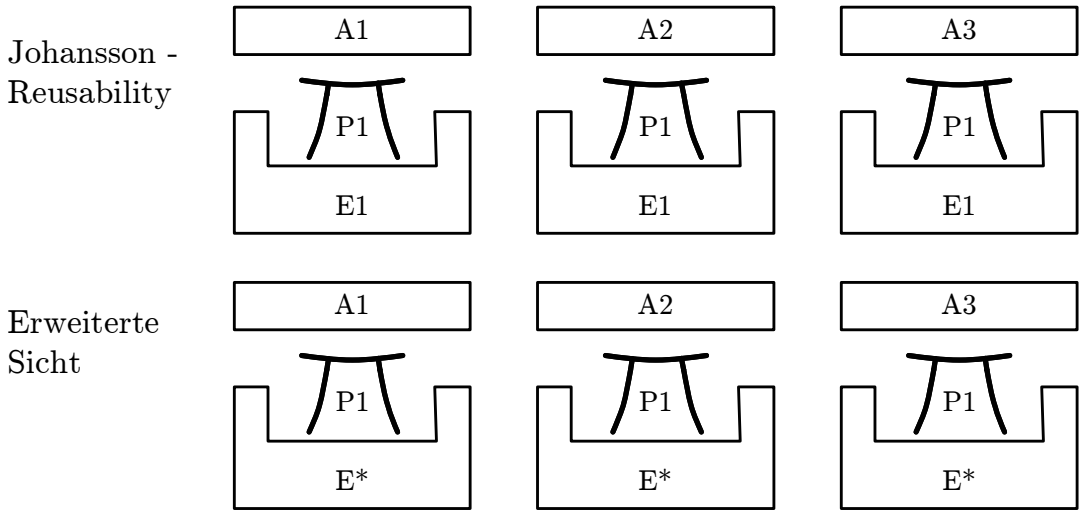

Abbildung 4.3.: Verschiedene Sichten auf Wiederverwendbarkeit.

zu sehr einschränkt. Auf der anderen Seite ist die Stabilität des Codes wichtig, da Änderungen der Plattform viele Anwendungen beeinflussen. In diesem Zusammenhang ist auch die Traceability wichtig, um Auswirkungen von Änderungen gut einschätzen zu können. Auch die Kompatibilität von neuen Plattformversionen zu bestehenden Applikationen ist ein wichtiger Punkt, da andernfalls hohe Migrationsaufwände für viele Anwendungen entstehen.

Diese kleine Untersuchung nur eines wichtigen Punktes bei Plattformen verdeutlicht die Notwendigkeit, an dieses Thema strukturiert heranzugehen. Nicht nur in dieser Arbeit wird sich diesem Problem von anderer Seite aus genähert. Durch die Untersuchung einer möglichst vollständigen Liste der Qualitätsattribute werden ebenfalls alle aus den Aufgaben einer Plattform resultierenden Qualitätsanforderungen erreicht. Die ISO/IEC 25010 [ISO05b] stellt eine fast vollständige Liste von Qualitätsattributen bereit und sieht zumindest ein ähnliches Vorgehen vor (vgl. 2.4.2).

Das Erfüllen der Qualitätsattribute wird als Anforderung an die Software aufgefasst. Diese (Qualitäts-)Anforderungen stellen somit Konkretisierungen der Qualitätsattribute oder -merkmale dar. Das Qualitätsmerkmal „Stabilität“ z. B. kann als Qualitätsanforderung in der Art „ohne Absturz lauffähig für mindestens eine Woche bei mittlerer Beanspruchung“

konkretisiert werden. Die Qualitätsanforderungen können meist objektiv bewertet werden, oftmals nicht nur durch „erfüllt“ oder „nicht erfüllt“, sondern durch Angabe z. B. eines Prozentsatzes[1].

Betrachtet man die Attribute und die abgeleiteten Anforderungen, so benötigt man immer auch eine Interessengruppe, die sich die Erfüllung der Anforderung wünscht. Gemäß den Interessengruppen „Benutzer“, „Unternehmen“ und „Entwicklungsteam“ können alle Anforderungen an Plattformen eingeteilt werden. Hierbei besteht durchaus die Möglichkeit, dass manche der Qualitätsmerkmale mehrfach als Anforderung auftreten, dann jedoch anderer Natur sein können. Die Planbarkeit beispielsweise betrifft alle Interessengruppen – die Kunden bzgl. Liefertermin, die Organisation bzgl. Kosten und das Entwicklungsteam bzgl. der Auswirkungen auf die eigene Urlaubsplanung. Diese Unterscheidung wird in der ISO/IEC 25010 nicht getroffen.

Alle relevanten Qualitätsattribute sollten in mindestens einer Qualitätsanforderung resultieren. Dies ist, wie bereits in 3.1 dargestellt, für nichtfunktionale Anforderungen bei Plattformen nur bedingt möglich. Umgekehrt können alle Qualitätsanforderungen mindestens einem Qualitätsmerkmal zugeordnet werden. Dass jede Qualitätsanforderung Einfluss auf mindestens ein Qualitätsmerkmal hat, ist hierbei offensichtlich. Eine Anforderung kann jedoch Einfluss auf mehrere Merkmale haben: Beispielsweise hat die Anforderung, einen hohen Automatisierungsgrad beim Test zu erreichen, sowohl auf die Reproduzierbarkeit als auch auf die Planbarkeit der Testdurchführung Auswirkungen. Dieser Sachverhalt ist auch in Abb. 4.4 dargestellt.

Daraus ergibt sich für die Auswahl der Kennzahlen (die meist die Qualitätsanforderungen bewerten), dass für die Qualitätsmerkmale nur eine Tendenz abgeleitet werden kann. Anhand einer Kennzahl für eine Anforderung können dafür oftmals Auswirkungen auf eine ganze Reihe von Qualitätsattributen (jeweils zu einem bestimmten Grad) beobachtet werden.

Um die Qualitätsanforderungen an Softwareplattformen in Bezug zu den Qualitätsmerkmalen zu bringen, werden alle in der Norm enthaltenen Qualitätsattribute ohne ihre hier-

[1] Beispielsweise in folgender Form: „70% der von den Kunden gewünschten Funktionen wurden implementiert und getestet.“

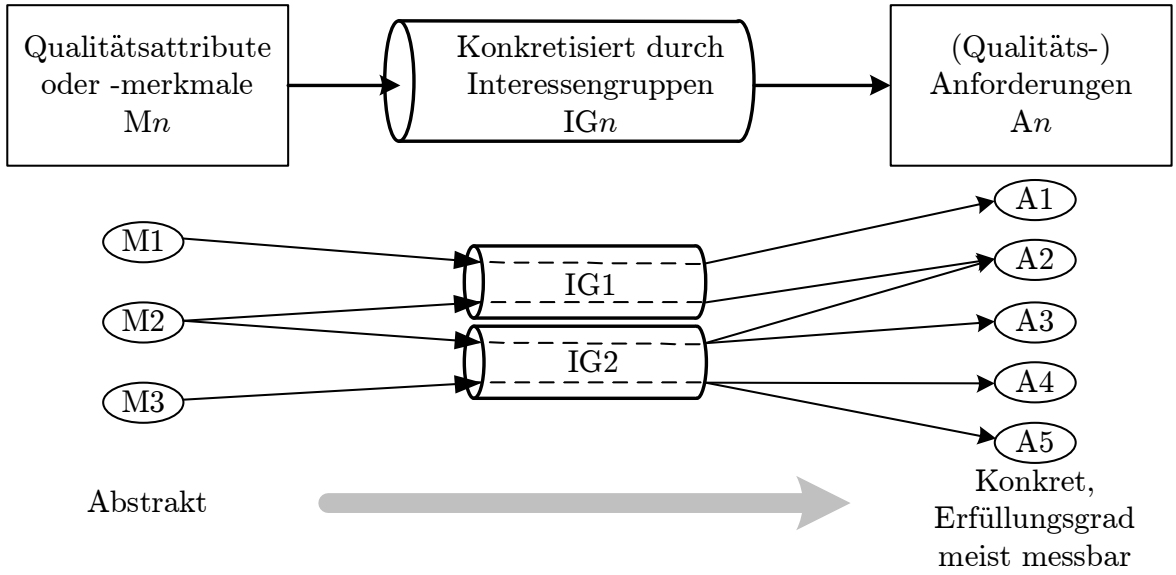

Abbildung 4.4.: Qualitätsattribute und -anforderungen.

archische Struktur[2] betrachtet. Dabei entsteht eine flache Auflistung von Qualitätsmerkmalen, welche dann als Ausgangspunkt der weiteren Analyse zugrunde liegt. Der Vorteil dieser Betrachtungsweise ist, dass auch generell ein Qualitätsattribut als wichtig angesehen und entsprechend priorisiert werden kann, ohne dass genau spezifiziert werden muss, welche der Teilattribute diese Relevanz besitzen.

Die Qualitätsattribute resultieren schließlich in Qualitätsanforderungen, die selten explizit aufgestellt werden und deren Auswahl in den meisten Fällen mehrere, oft nicht nur fachliche Gründe hat. Dies liegt zum einen darin begründet, dass sich aus ihnen meist leicht messbare Aktivitäten ableiten lassen. Zum anderen liegen die Gründe dafür auch in den Auswirkungen dieser Aktivitäten. Wie in Abb 4.4 dargestellt, haben die Qualitätsanforderungen und so auch die damit verknüpften Aktivitäten Auswirkungen auf mehrere Qualitätsmerkmale. Diese können sich auch negativ beeinflussen.[3] Mehrere Beispiele für Qualitätsanforderungen und die damit verbundenen Qualitätsmerkmale befinden sich in Tabelle 4.1.

Die QS sollte die Qualitätsanforderungen überprüfen und deren Erfüllung sicherstellen.

[2]Beispielsweise ist die Analysierbarkeit dort ein Teilattribut der Wartbarkeit.
[3]Eine kurze Gesamtentwicklungszeit (inklusive Architektur, QS, etc.) beispielsweise steht oftmals im Gegensatz zu einem ausgiebigen Test.

Qualitätsmerkmal	**Interessengruppe**	**Qualitätsanforderung**
Effizienz	Unternehmen	Ausschließlich Einsatz wiederverwendbarer Features
Wiederverwendbarkeit	Entwicklungsteam	
Funktionsumfang	Benutzer	

Tabelle 4.1.: Qualitätsanforderungen und deren Auswirkungen

Dies bedeutet, dass die Qualitätsanforderungen in Testfälle umgewandelt und diese dann ausgeführt werden sollten. Die Erfüllung einiger dieser Qualitätsanforderungen kann jedoch nur durch andere Methoden als das Testen sichergestellt werden. Darunter fallen Aktivitäten wie Design-, Dokumenten- oder Code-Reviews.

Viele der Qualitätsanforderungen resultieren in Anforderungen an den Test bzw. an die Testfälle. Zusammen mit den Benutzeranforderungen bilden sie die Testanforderungen, aus denen alle Testfälle gebildet werden. Dies ist in Abb. 4.5 dargestellt.

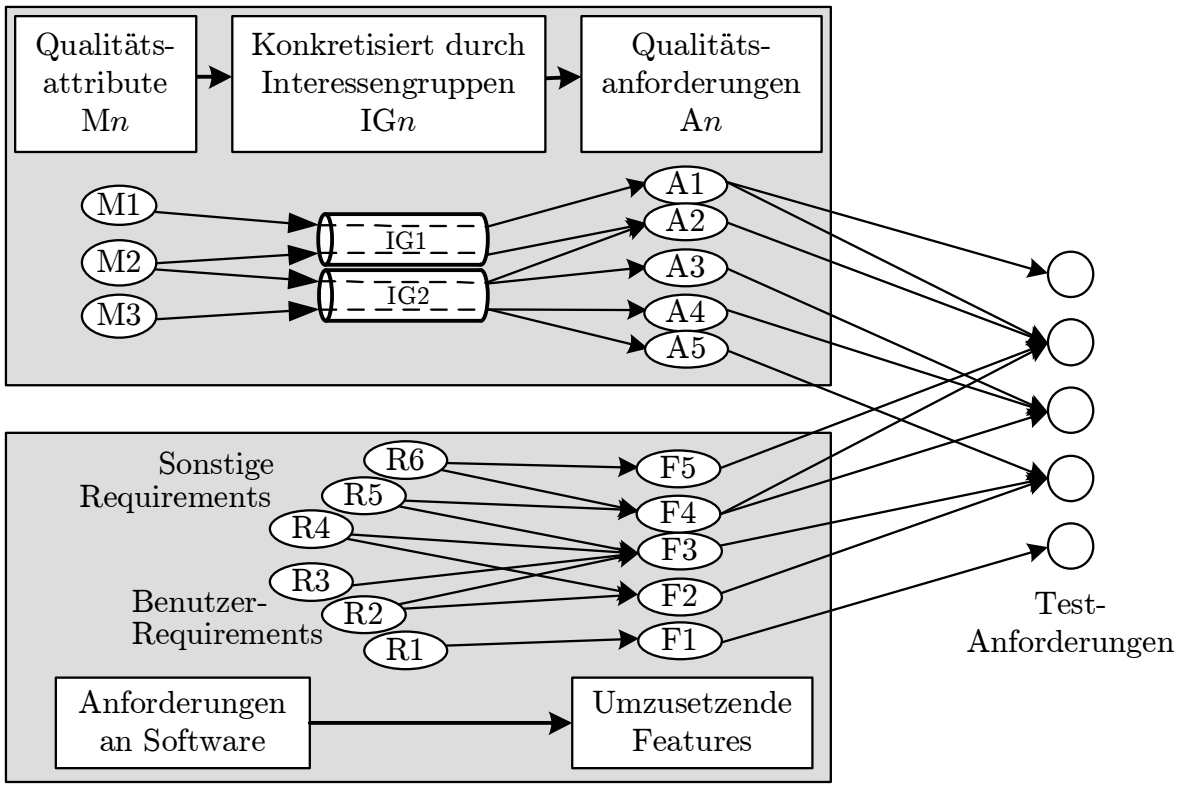

Abbildung 4.5.: Testanforderungen durch Benutzer- und Qualitätsanforderungen.

4.4. Priorisierung von Qualitätsattributen

In diesem Abschnitt soll eine Rangliste erstellt werden, welche Qualitätsattribute am wichtigsten sind. Dazu wird jedem der Qualitätsattribute in der oben beschriebenen Liste in einem ersten Schritt eine Priorität $p \in [1, \ldots, 6]$ zugeordnet. Diese Priorität gibt an, wie wichtig das entsprechende Attribut für eine Softwareplattform ist. Die höchste Priorität ist $p = 1$. Eine Priorität eines Attributs ist jeweils abhängig von einer Interessengruppe. Ein exaktes allgemeingültiges Plattformprofil existiert somit immer nur abhängig vom Blickwinkel. Es ergibt sich eine dreigeteilte Liste bestehend aus der Interessengruppe, dem Qualitätsmerkmal und der Priorität.

Obwohl die Prioritäten hier eher eine grobe Einteilung darstellen, sollten sie nur mit Vorsicht als allgemeingültig angesehen werden. Dies liegt darin begründet, dass erwartungsgemäß bei jedem Softwaresystem die Schwerpunkte anders gesetzt sind. Aus diesem Grund besitzt auch jede Plattform ein eigenes Qualitätsprofil, das teilweise vom durchschnittlichen Plattformprofil abweicht.

Zur Bestimmung der Priorität der Qualitätsattribute sollten mehrere Quellen herangezogen und miteinander verbunden werden. Jedem Attribut oder Subattribut aus der ISO 9126 und dem einen oder anderen weiteren Attribut wird im Praxisbeispiel eine Priorität zugeordnet. Diese ergab sich aus Expertengesprächen im Umfeld des Praxisbeispiels (siehe Kapitel 5). Das Resultat sind Vektoren der Form

$$q_{i,j} \in \{1, ..., 6\} \times ... \times \{1, ..., 6\} \subseteq \mathbb{R}^6,$$

mit Interessengruppe $i \in \{1, 2, 3\}$, Qualitätsattribut $j \in \{1, ..., n\}$ und $n > 6$. Die Attribute $j = 1, ..., 6$ seien die Qualitätsattribute aus der ISO 9126. Es können verschiedene Attribute dieselbe Priorität haben (da $n > 6 = \#\text{Prioritäten}$). Die drei Interessengruppen sind die Organisation bzw. das Unternehmen, das die Plattform entwickelt ($i = 1$), das Entwicklungsteam ($i = 2$) und die Benutzer der Plattform ($i = 3$). Der Anschaulichkeit halber werden auch in diesem allgemeinen Teil dieser Arbeit die projektspezifischen Prioritäten aus dem Anwendungsbeispiel verwendet. Sie sind bei der Anwendung auf andere

Plattformen entsprechend zu ersetzen.

Gleichzeitig wird die Untersuchung von Johansson et al. [JWBH01] (vgl. Abschnitt 3.2.2) herangezogen, um allgemeinere Plattformprioritäten zu erhalten. Die Studie wurde durchgeführt, da Softwareplattformen einen großen Einfluss auf die Kosten, die Entwicklungszeit und die Gesamtqualität von ganzen Generationen von Softwareprodukten haben. Daher ist die Menge an Qualitäten der Plattform wichtiger als die einzelner Anwendungen. Die Untersuchung startete mit einer Pilotstudie, auf die zwei Studien in verschiedenen Unternehmen folgten. Auch Johansson stellt fest, dass in jedem der Unternehmen verschiedene Interessengruppen bei der Bestimmung der gewünschten Qualitätseigenschaften existieren. Er unterscheidet zwischen „Architect“, „System designer“ und „Marketing“ [4]. Dies ist vergleichbar mit der hier getroffenen Einteilung in Organisation, Entwicklungsmannschaft und Benutzer.

Ein Ergebnis der Untersuchung von Johansson sind die Priorisierungen der sechs Qualitätsattribute aus der ISO 9126 durch die verschiedenen Interessengruppen. Untersucht wurden hierbei keine Subattribute und das Attribut „Portability“ aus der Norm wurde durch „Reusability“ ersetzt. Letzteres ist bei der Plattformentwicklung sinnvoll. Das Ergebnis sind somit Vektoren der Form

$$r_{i,j,k} \in \{1, ..., 6\} \times ... \times \{1, ..., 6\} \subseteq \mathbb{R}^6,$$

mit Interessengruppe $i \in \{1, 2, 3\}$, Qualitätsattribut $j \in \{1, ..., 6\}$ und Unternehmen $k \in \{1, 2, 3\}$. Diese $r_{i,j,k}$ enthalten die Reihenfolge der Wichtigkeit der Qualitätsattribute. Durch Aufsummieren der Werte über die verschiedenen Unternehmen und anschließender Zuordnung des Ranges werden Prioritäten $r_{i,j}$ erhalten, die mit den Ergebnissen der eigenen Untersuchung verglichen werden können. Erneutes Summieren von $r_{i,j}$ und $q_{i,j}$ ergibt die Summen

$$S_j = \sum_{i=1}^{3} (r_{i,j} + q_{i,j}), \ j = 1, ..., 6.$$

[4] In einem der beiden untersuchten Unternehmen existieren nur Ergebnisse zu den Interessengruppen „Architect“ und „System designer“.

Qualitätsattribute	$r_{i,j}$			$q_{i,j}$			S_j	p_j
Efficiency	5	6	5	3	6	5	30	9
Functionality	3	4	4	6	6	3	26	7
Reliability	1	1	1	4	1	1	9	1
Usability	6	5	5	5	5	2	28	8
Reusability	2	2	3	2	1	6	16	4
Maintainability	4	3	2	1	1	6	17	5
Ability to plan	-	-	-	2	3	1	12	2
Testability	4	3	2	2	2	6	19	6
Reproducibility	-	-	-	2	3	2	14	3

Tabelle 4.2.: Prioritäten der Qualitätsattribute
Die drei Spalten bei $r_{i,j}$ und $q_{i,j}$ stehen für $r_{1,j}$, $r_{2,j}$, $r_{3,j}$ und $q_{1,j}$, $q_{2,j}$, $q_{3,j}$.

Durch die anschließende Zuordnung des Ranges der S_j werden die Vektoren p_j erhalten, die die Prioritäten der Qualitätsattribute $j = 1, ..., 6$ aus der ISO 9126 darstellen. Dies ist im oberen Teil der Tabelle 4.2 dargestellt.

Da in der eigenen Prioritätsbestimmung weitere Qualitätsattribute enthalten sind, können auch für $6 < j \leq n$ die Vektoren p_j bestimmt werden. Hierbei muss unterschieden werden, ob es sich um Subattribute der Attrbiute $j = 1, ..., 6$ handelt oder um Attribute, die in der ISO 9126 gar nicht vorkommen. In ersterem Fall können die $r_{i,j}$ auf den entsprechenden Wert des Superattributs gesetzt werden (beispielsweise ist die *Testability* ein Subattribut der *Maintainability*). Die übrigen erhalten die doppelte Priorität der eigenen Priorisierung. Da es bei der Auswahl der Kennzahlen unerheblich ist, für welche Interessengruppe das zu bewertende Attribut von großem Interesse ist, können wiederum die Prioritäten der verschiedenen Interessengruppen aufsummiert und gemäß ihres Ranges geordnet werden. Das Ergebnis sind Priorisierungen, wie im unteren Abschnitt von Tabelle 4.2 dargestellt.

4.5. Auswahl von Kennzahlen

An dieser Stelle kann lediglich Allgemeines über die Auswahl von Kennzahlen festgelegt werden. Die konkrete Auswahl muss durch die projektspezifische Bestimmung der relevanten Qualitätsattribute ebenfalls projektspezifisch erfolgen.

Die endgültige Auswahl der Kennzahlen wird in der Praxis nicht nur durch fachliche Themen beeinflusst. Wie in 2.6.1 bereits beschrieben spielen auch „politische“ Faktoren hierbei eine nicht zu unterschätzende Rolle. Entscheidend ist dabei der Umgang mit den Kennzahlen bzw. den Ergebnissen aus deren Auswertung. Beispielsweise ist es für die Entwickler von entscheidender Bedeutung, ob und wie Ergebnisse der Form „Entwickler X ist produktiver als Entwickler Y“ verwertet werden sollen. Aber auch auf anderer Ebene kann die Entscheidung, wie mit den gesammelten Daten umgegangen werden soll, wichtig sein. Wenn sich beispielsweise herausstellt, dass ein Projekt relativ ineffizient, für das Unternehmen insgesamt aber dennoch unverzichtbar ist, sollte es nicht eingestellt, sondern effizienter gestaltet werden. Falls ausschließlich erstere Information vom übergeordneten Management wahrgenommen wird, kann dies zu den falschen Schlüssen führen.

Der Umgang mit den Messergebnissen spielt auch bei rein technischen Kennzahlen eine große Rolle. Da die meisten dieser Zahlen ein breiteres Verständnis des Produkts benötigen und ohne eine wohldurchdachte Interpretation zu den falschen Schlüssen führen können, sollten sie nur vorsichtig weitergegeben werden. In manchen Fällen kann es auch sinnvoller sein, auf die Anwendung mancher Kennzahlen ganz zu verzichten, falls ihre Auswertung ein bzgl. des Nutzens überproportional großes Risiko birgt.

Für die Erstellung der Kennzahlen gibt es einige Quellen, Normen und Standardwerke[5], die herangezogen werden können. Dort erwähnte Kennzahlen können oft fast unverändert etabliert werden. Dies hat den Vorteil, dass bei diesen „gängigen“ Kennzahlen oftmals auch Werte anderer Projekte oder Produkte, zum Teil aus anderen Unternehmen, zugänglich sind und dadurch Vergleichswerte für ein Benchmarking vorliegen. Diese Werte können meist jedoch nicht direkt verglichen werden, sondern lediglich Tendenzen oder um den Mittelwert bereinigte Daten können anhand statistischer Methoden gegenübergestellt werden. Die Vergleiche können dabei nur als Orientierungshilfe gelten, da die meisten Projekte und Produkte sehr verschieden sind. Es ist ratsam, die Kennzahlen nicht ohne Anpassung an Plattformen zu übernehmen. Beispiele dafür, wie dies geschehen kann, werden in Abschnitt 4.6 gegeben.

Aus allen Kennzahlen ergeben sich Bedingungen an Datenquellen, an die Datenqua-

[5]Siehe z. B. [DAS; ISB; ISO03a; ISO03b; ISO04; Fen91; MP93; Tha94] bzw. Kapitel 2.6.

lität und -aktualität, etc. Daraus resultieren meist weitreichende Anforderungen an den Entwicklungsprozess und ggf. auch die Organisationsstruktur. Insbesondere das Thema Datenqualität ist nicht nur im Kontext von Kennzahlen in den letzten Jahren zu einem eigenen wissenschaftlichen Betätigungsfeld herangewachsen und wird oftmals unterschätzt. [Pre04]

Doch auch die anderen der o. g. Bereiche sollten nicht unterschätzt werden. Um beispielsweise eine Auswertung der Fehlerursachen und -status durchführen zu können, müssen diese bekannt sein und sollten protokolliert und in geeigneter Weise archiviert werden. Die Auswertung der Fehlerstatus bedingt das Vorhandensein und die Einhaltung eines Fehlerlebenszyklus. Dies wiederum setzt das Befolgen auch der mit dem Fehlerlebenszyklus in Beziehung stehenden Prozesse voraus. Dies spiegelt sich auch im CMMI wider – dort werden erst ab Stufe 3 („Staged Representation“) Kennzahlen eingeführt [CKS03].

All diese Punkte müssen bei der Auswahl von Kennzahlen beachtet werden. Dies trifft nicht nur auf Plattformen, sondern auf Softwaresysteme im Allgemeinen zu. Es sollten immer stets alle o. g. (und alle relevanten und nicht hier aufgeführten) Auswahlkriterien beachtet werden.

4.6. Testmetriken für Softwareplattformen

In vielen Fällen bietet es sich an, Kennzahlen aus der Literatur vor der Anwendung anzupassen oder komplett neue Kennzahlen zu entwickeln. Der Vorteil liegt dabei auf der Hand – das entstehende Kennzahlsystem ist besser auf das Softwaresystem zugeschnitten.

Die in diesem Abschnitt aufgelisteten Kennzahlen bilden eine exemplarische Zusammenstellung von einigen Beispielen. Sie ist weder vollständig noch als Minimalumfang zu verstehen. In der Literatur gibt es eine Unmenge an verschiedenen Kennzahlen, die in allen erdenklichen Bereichen Anwendung finden können. Die meisten der hier aufgelisteten Kennzahlen sind bereits seit einiger Zeit in der Literatur in ähnlicher Weise veröffentlicht und auch in der Praxis im Einsatz. Sie beinhalten in vielen Fällen eigene Anpassungen an Plattformen und Überlegungen bzgl. ihrer Realisierbarkeit im konkreten Praxisfall (siehe Abschnitt 5). Letztere sind jedoch nicht ohne Einschränkung auf andere Plattformen

übertragbar.

In den nächsten drei Abschnitten werden einige Kennzahlen vorgestellt. Diese werden eingeteilt je nach Interessengruppe, die an der Kennzahl interessiert ist: Benutzer der Plattform, Entwicklungsteam der Plattform und Unternehmen oder Unternehmensbereich, in dem die Plattform entwickelt wird. Diese Unterteilung orientiert sich an den verschiedenen Sichten auf Anforderungen an Plattformen und wird auch in Kapitel 4.3 verwendet. Sie bietet sich an, da dadurch die Sichtweise, aus der die Zahlen zu bewerten sind, verdeutlicht wird. Damit ist in erster Linie keine Gewichtung der Kennzahlen gemeint, sondern vielmehr eine Interpretationshilfe. Einige der Kennzahlen sind für mehrere Interessengruppen interessant. Die Zuordnung wird jeweils im Einzelfall festgelegt.

In manchen Fällen ist es sinnvoll, sich für die Anwendung einzelner Kennzahlen feste Zielgrößen vorzugeben. Diese müssen aber zum einen wohlüberlegt, sinnvoll und realistisch sein, da sie ansonsten schnell kontraproduktiv wirken können (vgl. hierzu Kapitel 2.6.1). Die Tabellen 4.3, 4.4 und 4.5 stellen einige der Kennzahlen und mögliche Zielvorgaben oder Richtwerte dar. Die jeweils angegebene Liste von Qualitätsmerkmalen ist nicht zwangsläufig vollständig. Kennzahlen werden im Allgemeinen von vielen Faktoren beeinflusst und bewerten somit zumindest indirekt eine ganze Reihe von Qualitätsmerkmalen. Die in den Tabellen angegebenen Merkmale haben jedoch den größten Einfluss auf die jeweilige Qualitätsmetrik. Bevor die genannten Kennzahlen angewandt werden können, müssen sie in der Regel zunächst noch eindeutig definiert werden[6].

4.6.1. Kennzahlen aus Benutzersicht

Für Kennzahlen, die Aspekte des Produkts aus Perspektive der Anwender bewerten, bietet sich meist eine Vielzahl an Ansatzpunkten. Zunächst können Testergebnisse ausgewertet und dadurch Kennzahlen wie $\frac{failed}{total}$ erstellt werden, wobei $failed$ für die Anzahl der nicht bestandenen und $total$ für die Anzahl aller Testfälle steht. Diese Kennzahlen können in sel-

[6] Beispielsweise ist der Prozentsatz nicht bestandener Tests in vielen Variationen auslegbar: Ein Test kann entweder ein Testfall sein, der innerhalb eines Releases ggf. mehrmals ausgeführt wird oder ein einzelner Testlauf. Auch die Grundgesamtheit, auf deren Basis die Prozentzahl berechnet werden soll, ist noch nicht festgelegt. Es kann sich hierbei entweder um alle durchgeführten Testläufe handeln, um alle geplanten Tests (die evtl. mehrmalige Ausführung wird dabei nicht gezählt) oder um alle tatsächlich ausgeführten Tests (ohne die evtl. mehrmalige Ausführung).

tenen Fällen ohne Vergleichswerte aussagekräftig sein, meist sollten sie jedoch in Relation zu den Werten des letzten Releases oder Ähnlichem betrachtet werden. Auch die Bewertung der Features und der Vergleich untereinander oder zwischen verschiedenen Releases kann anhand der Anzahl bzw. des Prozentsatzes nicht bestandener Testfälle untersucht werden. Wie bereits erwähnt, ergibt sich bei Plattformen der Unterschied, dass aufgrund fehlerhafter Testapplikationen nicht bestandene Testfälle als nicht durchgeführt zu bewerten sind. Somit können auch nach Abschluss einer kompletten Iteration noch nicht durchgeführte Testfälle bestehen. Eine weitere Anpassungsmöglichkeit an Plattformen besteht in der Einteilung der Testfälle in *Passed*, *Failed* und *Vague*. Letztere bezeichnen Tests, die noch nicht eindeutig als bestanden oder nicht bestanden bewertet werden können. Dies kann bei nichtfunktionalen Tests aufgrund der problematischeren Behandlung von NFRs (vgl. 3.1) vermehrt vorkommen.

Kennzahlen, die sich auf die Anzahl der gefundenen Fehler konzentrieren, können ebenfalls sehr nützlich sein. Einerseits erlaubt die Anzahl der gefundenen Fehler pro Tag einen Ausblick auf die Anzahl der verbleibenden Fehler [Sch03a], wenn sie sich in einer statistisch relevanten Größenordnung bewegen. Letzteres stellt bei Plattformen oft ein nicht zu erfüllendes Kriterium dar, da die Stabilitätstests im Fokus liegen sollten und diese aufwändig (und langwierig) sind, wodurch nur eine geringe Anzahl pro Release ausgeführt werden kann. Doch auch der Vergleich der Anzahl der Fehler in verschiedenen Releases oder Features oder die Verteilung bzgl. der Schwere der Fehler kann aufschlussreich sein.

Speziell bei Plattformen ist eine genauere Untersuchung der Fehlerquelle interessant. Vor dem Hintergrund, dass Fehler in einer Testapplikation oder in Testdaten zwar keinen direkten Einfluss auf die Qualität des Produkts haben, indirekt jedoch bedeuten, dass der entsprechende Testfall nicht ausgeführt wurde, ist dies ein wichtiger Punkt. Diese Überlegung gilt generell auch für normale Anwendungen, ist bei Plattformen jedoch wichtiger, da diese meist zu einem größeren Teil anhand von Testapplikationen getestet werden.

Die explizite Darstellung der Herkunftsversion von noch nicht behobenen Fehlern bietet sich bei Plattformen aufgrund des oft flexibel gestalteten Releaseumfangs an. Dies ist auch bei vielen Anwendungen der Fall, erhält aufgrund der in 3.1 geschilderten Umstände bei Plattformen jedoch eine besondere Relevanz.

Da die Planbarkeit vieler Anwendungen von der Einhaltung des Releasedatums der Plattform abhängt, sind auch Größen wie die Dauer eines evtl. Releaseverzugs wichtig. Dies ist im Zusammenhang mit dem von den auf der Plattform aufbauenden Anwendungen erwarteten Funktionsumfang zu betrachten, da dieser von Anwendung zu Anwendung variiert. Dies liegt darin begründet, dass meist nur wenige Anwendungen alle Features der Plattform einsetzen. An dieser Stelle bietet sich eine explizite Darstellung der Fehler an, die einer sofortigen Auslieferung der Plattform noch im Wege stehen bzw. wie schwerwiegend die Einschränkungen sind, die Anwendungen bei der sofortigen Auslieferung der Plattform haben würden. Dadurch kann bereits frühzeitig abgeschätzt werden, wann das Plattformrelease mit welchem Funktionsumfang ausgeliefert werden könnte.

Dazu werden für jeden gefundenen Fehler F alle Anwendungen A_F ermittelt, die durch diesen Fehler beeinträchtigt werden. Weiterhin wird die Priorität $P(A)$ der jeweiligen Anwendung festgelegt. Dies kann die Priorität der Anwendung für das Unternehmen sein. Eine Möglichkeit der Darstellung der Auswirkung $E(F)$ eines Fehlers F auf das Release ist somit wie folgt:

$$E(F) = \sum_{A_F} P(A)$$

An Stelle der Priorität $P(A)$ kann auch der Grad der Einschränkung einer Anwendung durch einen bestimmten Fehler $G(A_F)$ eingesetzt werden. Dadurch ergibt sich eine gleichgewichtete Betrachtung aller Anwendungen und eine differenzierte Betrachtung des Grads der Einschränkung. Auch eine Kombination der beiden Sichtweisen ist durch die Verwendung des Produkts $P(A) \cdot G(A_F)$ als Summand möglich.

4.6.2. Kennzahlen aus Organisationssicht

Aus Sicht des Unternehmens, das die Plattform(en) entwickelt, bzw. der Entwicklungsleitung ergeben sich ebenfalls Fragen, die anhand von Kennzahlen beantwortet werden können. Diese beziehen sich oft auf allgemeine Dinge zur Effizienz der Arbeit des Projektteams oder der Plattform allgemein. Letzteres ist meist eine berechtigte Frage, da eine Plattform in aller Regel nicht notwendig ist, sondern jedes Anwendungsprojekt auch die

Qualitätsmerkmal[1]	**Kennzahl**	**Zielgröße x**
Funktionalität, Zuverlässigkeit, Performance, Stabilität	Prozentsatz nicht bestandener Tests	$a < x < b$
	Fehler pro Feature	$x < a$
	Fehler nach Schwere S	$x < f(S)$
	Fehler nach Fehlerquelle Q	$x < f(Q)$
Planbarkeit	Releaseverzug in Tagen	$x < c$
	Auswirkung eines Fehlers $E(F)$	$\sum_F E(F) < d$

x: gemessener Wert. a, b, c, d: vorgegebene, kalibrierte Zielgrößen. $f(y)$: von y abhängiger Wert.
[1] Die Unterscheidung auf dieser Ebene ist oft auch von der Art der untersuchten Testfälle bzw. den dazugehörigen Fehlern abhängig.

Tabelle 4.3.: Exemplarische Auswahl von Kennzahlen aus Benutzersicht

von der Plattform benötigten Teile selbst entwickeln könnte. Dies sollte jedoch vor der Entwicklung der Plattform genau untersucht werden. Es können auch Faktoren aufgestellt werden, anhand derer sich im Lauf der Zeit erkennen lässt, ob das Plattformprojekt immer noch lohnend ist. Auch diese Kennzahlen liegen nicht im Fokus dieser Arbeit.

Das Unternehmen ist auch an Informationen interessiert, ob das richtige Produkt entwickelt wird, d. h. ob der Entwurf und die Funktionen der Plattformen die Richtigen sind. Die Projektleitung interessiert zusätzlich die Information, ob die Plattform richtig, d. h. entsprechend der qualitativen Erwartungen, entwickelt wurde. Da dies auch im Kerninteresse der Entwicklungsmannschaft ist, werden Metriken zu diesem Themengebiet im nächsten Abschnitt erörtert.

Die Projektleitung sollte zusätzlich beurteilen, ob das Verhältnis von Qualität und Kosten stimmt oder ob es an mancher Stelle sinnvoll ist, die Bemühungen für eine hohe Qualität zu intensivieren oder Kosten zu reduzieren und eine geringere Qualität zu akzeptieren. Überlegungen dieser Art lassen sich meist auf betriebswirtschaftliche Untersuchungen zurückführen und werden in dieser Arbeit ebenfalls nicht weiter verfolgt.

Sehr nützlich vom Standpunkt der Organisation mit Hinblick auf die Bewertung des Tests ist es, einzelne Features zu betrachten. Ausgehend von der Anzahl der durchgeführten Testfälle pro Feature können mehrere Kennzahlen aufgestellt werden. Es kann anhand eines Vergleichs der Verteilung der Anzahl der Testfälle auf die einzelnen Features mit den Komplexitäten oder der Prioritäten der Features überprüft werden, ob der Testfokus

auf die richtigen Features gesetzt ist[7]. Auch diese Kennzahl kann speziell an Plattformen angepasst werden durch die Bestimmung der Priorität der Features in Abhängigkeit der Anzahl (und / oder Priorität) der Anwendungen, die es einsetzen.

Auch eine Gegenüberstellung der Aufwände für Entwicklung und Test eines Features können sinnvoll sein. Betrachtungen dieser Art müssen jedoch immer mit Vorsicht genossen werden, da nicht notwendigerweise ein Zusammenhang zwischen diesen beiden Faktoren bestehen muss. Sie können somit lediglich als Ausgangspunkt für weitere Untersuchungen dienen.

Auch der Automatisierungsgrad $\frac{\#auto}{\#auto+\#manu}$, $\#auto$ sei hierbei die Anzahl der automatiserten Testfälle, $\#manu$ die der manuellen Testfälle, wird in vielen Fällen berechnet und ausgewertet. Dies ist für Plattformen genauso wichtig wie bei herkömmlichen Anwendungen und bietet sich v. a. bei iterativen Entwicklungen in verschiedenen Versionen an. Es muss an dieser Stelle stets beachtet werden, dass es nicht sinnvoll ist, einfach einen möglichst hohen Automatisierungsgrad anzupeilen. Nur bei sich nicht verändernden Funktionalitäten oder Vorgehensweisen lohnt sich die Automatisierung. Insbesondere bei Plattformen kann der Automatisierungsgrad durch die Verwendung von Testapplikationen relativ hoch gehalten werden. Dadurch, dass ein Anwendungsfall der Plattform jedoch die Entwicklung und der Betrieb einer Anwendung ist, müssen auch einige nicht automatisierbare Testfälle mit einberechnet werden.

Da bei Plattformen oftmals die Behebung von Fehlern auf ein Folgerelease verschoben wird (vgl. Abschnitt 3), ist es für das Releasemanagement erforderlich, einen Überblick über die Anhäufung von Fehlern im Laufe aufeinanderfolgender Releases zu behalten[8]. Dies geschieht oftmals anhand des *Backlog Management Index* und der *Backlog Summe*. In seiner ursprünglichen Form ist der Index der Quotient aus der Anzahl geschlossener und geöffneter Fehler pro Zeiteinheit, oft eine Woche. Aus Normierungsgründen wird diese Zahl meist noch mit 100 multipliziert:

[7] Dies setzt eine einheitliche Granularität der Testfälle voraus.
[8] Wie bereits dargestellt, trifft dies häufig auch auf Anwendungen zu.

$$
\begin{aligned}
B_{Idx} &= 100 \cdot \frac{d_{closed}}{d_{open}} \\
\text{mit } d_{closed},\ d_{open} &: \text{Anzahl geschlossener bzw.} \\
&\quad \text{geöffneter Defects pro Zeiteinheit}
\end{aligned}
$$

Ist der Index gleich 100, so wurden gleich viele Fehler entdeckt wie geschlossen. Er ist größer als 100, wenn mehr geschlossen als geöffnet wurden und entsprechend kleiner, falls das Gegenteil eintritt.

Der Nachteil des Backlog Management Index ist, dass er lediglich einen Trend angeben kann. Es kann nicht festgestellt werden, wieviele Fehler noch bestehen. Auch verschiedene Zeitspannen können nicht miteinander verglichen werden. Dies trifft auch auf den Vergleich verschiedener Releases zu.

Dieser Nachteil des BMI fällt bei der Betrachtung der *Backlog Summe* weg. Hierbei wird die Differenz aller bis zu einem bestimmten Zeitpunkt i geöffneten Fehler $d_{open,i}$ und aller bis i geschlossenen Fehler $d_{closed,i}$ betrachtet und ihre Veränderung in i dargestellt. Das Schaubild zeigt somit stets die Anzahl aller zum Zeitpunkt i offenen Fehler. Diese Betrachtungsweise zeigt den plattformspezifischen, aber releaseunabhängigen „Fehlerberg“.

Auch aus Organisationssicht sind die im letzten Abschnitt bereits angeschnittenen Prozesskennzahlen sinnvoll. Der Entwicklungsprozess von Plattformen lässt sich, im Grunde genauso wie bei der Anwendungsentwicklung, anhand von Prozesskennzahlen bestimmen. Dies liegt jedoch nicht im Fokus dieser Arbeit und erfordert, wie bei der Anwendungsentwicklung auch, die Definition und die Einhaltung eines gut abgestimmten Prozesses und ggf. der Vorgabe von *Quality Gates*. Deren Erfüllung kann dann meist sehr gut auch quantitativ angegeben werden. Auch Metriken, wie die durchschnittliche Zeit für das Beheben von Fehlern, fallen in diese Kategorie.

Qualitätsmerkmal	**Kennzahl**	**Zielgröße x**
Effizienz	Anzahl Tests nach Features und deren Priorität	$x_1 > ... > x_n$
Effizienz, Testbarkeit, ... [1]	Automatisierungsgrad	$a < x < b$
Planbarkeit	Prozentsatz fertiger Artefakte	$x \geq a\%$
	Varianz der Dauer des Handover-Tests	$x < c$
	Installationsdauer	$x < c$

x: gemessener Wert. $x_1, ..., x_n$: Anzahl der Tests für Feature mit Priorität $1, ..., n$. a, b: vorgegebene, kalibrierte Zielgrößen.

[1] Der Automatisierungsgrad bewertet Effizienz, Testbarkeit, Planbarkeit, Reproduzierbarkeit, Wiederverwendbarkeit und zum Teil weitere Attribute.

Tabelle 4.4.: Exemplarische Auswahl von Kennzahlen aus Organisationssicht

4.6.3. Kennzahlen aus Entwicklungssicht

Auch aus Sicht des Entwicklungsteams lassen sich einige wichtige und aussagekräftige Kennzahlen aufstellen. Viele der Themengebiete, die durch sie behandelt werden sollen, sind bereits durch den Bereich der Kennzahlen aus Benutzersicht abgedeckt; den Wunsch, dass das Produkt möglichst gut wird, haben die Benutzer und die Entwickler gemein. Wie oben bereits erwähnt, verfolgt auch das Unternehmen diesbzgl. dieselben Interessen.

Das Entwicklungsteam ist v. a. am Produkt, dessen Güte und Validität interessiert. Kennzahlen in diesem Bereich sollten somit v. a. Fragen bzgl. der Qualität des Produkts beantworten und bei der Einschätzung helfen, ob Beutzeranforderungen richtig interpretiert werden.

Ein zweiter Bereich, der großen Einfluss auf das Entwicklungsteam hat, ist die Einhaltung von Rahmenbedingungen. Finanzielle und zeitliche Vorgaben beeinflussen beispielsweise die finanziellen Mittel für Weiterbildungen, die Arbeitszeiten, die Urlaubsplanung etc. Mit deren Einhaltung oder Überschreiten steht und fällt oftmals das Projekt. Diese Faktoren werden meist anhand von betriebswirtschaftlichen Kennzahlensystemen (etwa des Controllings) oder Prozesskennzahlen gemessen und nicht in dieser Arbeit behandelt. Aber auch die Bewertung von effizienz- oder qualitätssteigernden Maßnahmen sollte bei Kennzahlen dieser Perspektive erwähnt werden. Diese lassen sich oft anhand der Umsetzung technischer Maßnahmen ermitteln.

Als Beispiel hierfür kann der Prozentsatz an Testfällen angeführt werden, die bereits auf einer Entwicklungsumgebung und nicht erst auf der (System-)Testumgebung ausgeführt werden[9]. Viele der Testfälle sind jedoch erst auf der Systemtestumgebung durchführbar. Aus diesem Grund ist es schwierig, für diesen Aspekt eine valide Kennzahl zu entwickeln. Zusätzlich zur Anzahl der durchgeführten Tests je nach Testebene bzw. Testumgebung ist auch die Anzahl der durchführbaren Tests auf der Entwicklungsumgebung zu ermitteln. Es ist jedoch aufwändig, dies erst im Nachhinein zu erfassen, da dabei alle Testfälle einzeln bewertet werden müssen. Eine weitere Schwierigkeit, die sich bei Anwendung derartiger Kennzahlen stellt, ist der Vergleich sehr unterschiedlicher Testfälle. Bei den Systemtests, nicht nur von Plattformen, sondern auch bei Anwendungen, reicht die Spanne von der automatisierten Installation des gesamten Systems, über einen über das gesamte Wochenende laufenden Stabilitätstest, bis hin zu einer kurzen Überprüfung einer Teilfunktionalität.

Auch für das Entwicklungsteam interessant sind Kennzahlen, die die Tests und die Fehler bewerten. Bereits im Abschnitt über Metriken aus Organisationssicht erwähnt wurden z. B. der Automatisierungsgrad und die Prozesskennzahlen, die auch die tägliche Arbeit der Entwicklungsmannschaft berühren.

Weitere Auswertungen von gefundenen Fehlern sind hilfreich, um die Plan- und die Testbarkeit zu bewerten. So kann beispielsweise untersucht werden, welcher Anteil der Fehler bereits auf der Entwicklungsumgebung entdeckt wurden oder wie lange die mittlere Zeitdauer ist, um einen Fehler zu schließen[10]. Letztere kann ebenfalls eingeteilt nach der Umgebung geschehen, auf der der Fehler entdeckt wurde. Auch die explizite Darstellung der nicht reproduzierbaren Fehler kann von Nutzen sein, da sich zum einen in vielen Fällen darunter durchaus reproduzierbare Fehler verstecken, deren Ursache bislang nur noch nicht gefunden wurde. Zum anderen, im Falle „tatsächlich nicht reproduzierbarer“ Fehler, lassen sich oftmals Schwächen im Testprozess aufdecken. Der Bereich derartiger Fehleruntersuchungen stellt sich bei Plattformen nicht anders dar als bei Anwendungen. Der einzige Unterschied ergibt sich durch die zum Teil unterschiedlichen Fehlerattribute.

[9] Je früher im Entwicklungszyklus ein Testfall ausgeführt und ein Fehler entdeckt wird, umso besser. Ein Fehler, der erst bei den Systemtests auf der Systemtestumgebung gefunden wird, ist selten mit weniger Aufwand zu beheben als ein Fehler, der im Review des ersten Anforderungsdokuments gefunden wurde.

[10] Dazu gehört die Lokalisation, die Behebung und der Re-Test des Fehlers.

Qualitätsmerkmal	**Kennzahl**	**Zielgröße x**
Effizienz, allg. Produktqualität	Tests pro Iteration [1]	$x > a\%$
Effizienz, Testbarkeit	Anteil an Tests auf Entwicklungsumgebung	$x \rightarrow max$
Planbarkeit, Testbarkeit	mittlerer Zeitbedarf, um Fehler zu beheben [2]	$x \rightarrow min$
Testbarkeit, Effizienz	Prozentsatz an Fehlern auf Entwicklungsumgebung	$x \rightarrow max$
Reproduzierbarkeit, Testbarkeit	Anzahl nicht reproduzierbarer Fehler	$x \rightarrow min$

x: gemessener Wert. a: vorgergebene, kalibrierte Zielgröße.

[1] In Relation zu allen für das Release geplanten Testfällen.

[2] Dies kann auch bezogen auf die Schwere der Fehler oder andere Attribute betrachtet werden. Hierbei ist es sinnvoll, auch die Varianz im Blick zu behalten.

Tabelle 4.5.: Exemplarische Auswahl von Kennzahlen aus Sicht des Entwicklungsteams

5. Eine Kennzahlsuite für die proaktive Infrastruktur (PAI)

In diesem Kapitel wird die Realisierung des in Kapitel 4 beschriebenen Ansatzes vorgestellt. Diese praktische Umsetzung erfolgte im Rahmen der Entwicklung der PAI-Plattformen (siehe Abschnitt 5.1) der Daimler AG. Wie in jedem spezifischen Entwicklungsprojekt ergeben sich dadurch einige Anforderungen und Rahmenbedingungen, auf die im Folgenden eingegangen wird. Hierbei können jedoch nur die Wichtigsten dargestellt werden – eine umfassende Betrachtung aller Einflüsse auf das Produkt bzw. dessen Qualitätsbewertung, würde den Rahmen dieser Arbeit sprengen.

5.1. PAI - Proaktive Infrastruktur

5.1.1. Übersicht

Die proaktive Infrastruktur (PAI) ist bereits seit mehreren Jahren eine globale Initiative innerhalb der Daimler AG und adressiert eine große Anzahl sehr unterschiedlicher Anwendungen. Diese besitzen eine Vielzahl an Anforderungen an die Middleware und die Infrastruktur, die durch die von PAI bereitgestellte IT-Infrastruktur vereinheitlicht wird. Viele von ihnen verwenden dieselben kommerziellen Basisprodukte von Herstellern wie IBM, Sun oder CA. Der Ansatz von PAI hat das Ziel, einige dieser konzernweit festgelegten Standardprodukte innerhalb von Plattformen, die fortlaufend weiterentwickelt werden, den Anwendungsprojekten als Produkt zur Verfügung zu stellen. Es ergibt sich durch PAI eine klare Schicht von Plattformen (siehe Abb. 5.1).

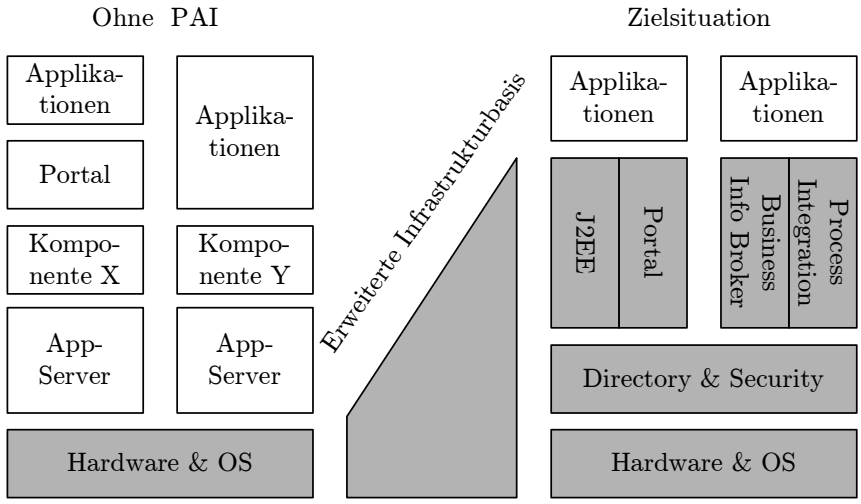

Abbildung 5.1.: PAI-Ansatz zur Infrastrukturkonsolidierung nach [Rei04].

Die Anwender (Projekte, Abteilungen und Bereiche innerhalb der Daimler AG) erhalten dadurch für die Entwicklung nicht nur von großen, unternehmensweiten Softwaresystemen eine infrastrukturelle Basis in Form von Plattformen. Ausgehend von diesen kann die Entwicklung der Anwendungen dadurch effizienter gestaltet werden, dass die Anwendungsprojekte nicht mehr eine Vielzahl von kommerziellen Basisprodukten integrieren müssen. Auf diese Weise wurden und würden weiterhin einige Aufgaben innerhalb des Unternehmens mehrfach erledigt [PS05].

PAI adressiert damit eine ganze Reihe weiterer Wünsche sowohl von Anwendungsseite als auch von Unternehmensseite aus gesehen:

- Anwendungsprojekte erhalten eine getestete und zuverlässige IT-Infrastruktur, die die Unternehmensstandards erfüllt und innerhalb der unternehmenseigenen Datacenter betrieben werden kann.
- Die Anwendungsarchitektur ist in eine umfassendere Unternehmensarchitektur eingebettet.
- Anwendungsprojekte können sich auf die Umsetzung der Geschäftslogik konzentrieren, ohne sich zu sehr mit Fragen bzgl. der Infrastruktur beschäftigen zu müssen.
- Eine proaktive Infrastruktur wird als Basis für zukünftige „On Demand“-Betriebskonzepte und die Erstellung von global, regional und lokal integrierten Applikationen

bereitgestellt.

- Durch das PAI-Patchkonzept werden Patches, Service Updates, Hotfixes, etc. der Basisprodukte und des Glue Codes in einem PAI Patch gesammelt, getestet (auch und v. a. die Verträglichkeit der einzelnen Patches untereinander) und den Anwendern zur Verfügung gestellt.

Die grundlegende Idee von PAI bedeutet zusammengefasst:

> „PAI is to deliver a set of integrated application platforms that hide the infrastructure and middleware complexity from application development and provide a standardized foundation for the application thus reducing middleware variants in the data centers.“ [Qua06]

Die PAI-Plattformen integrieren dazu einige Kombinationen verschiedener Basisprodukte, wie Directories, Datenbanken, Sicherheitssoftware, Application Server und Portal Server, auf verschiedenen Betriebssystemen. Die Auswahl der einzelnen Produkte und insbesondere ihrer Version und des Patchlevels liegt zum Teil im Aufgabenbereich des PAI-Teams, zum Teil kann durch die Verwendung der Plattformen der Einsatz strategischer Standardprodukte konzernweit ohne großen Governance-Aufwand vorgegeben werden.

Die Entwicklung des sog. „Glue-Codes“, der die verschiedenen Produkte verbindet und Schnittstellen optimiert, gehört zu den Kernaufgaben der PAI-Entwicklung. Durch diesen wird in einigen Fällen eine Abstraktion von den verschiedenen herstellerabhängigen Produkten erzielt (siehe Abb. 5.2) und somit eine Substitution eines Anbieters durch einen anderen vereinfacht.

Des Weiteren liefern die PAI-Plattformen zusätzliche Funktionalität, deren Verwirklichung erst durch die Integration der Basisprodukte in einer Plattform ermöglicht wurde. Beispielsweise kann erst nach der Integration aller Komponenten ein Administrations- und Kontrollzentrum für die Plattform erstellt werden.

Entsprechend der Definition einer Plattform (s. o.) gibt es im PAI-Projekt eine Reihe verschiedener Plattformen, d. h. wiederverwendbarer, bereichsspezifischer Konzeptionen, die anhand unterschiedlicher Funktionalitäten getrennt werden können. Zum Zeitpunkt der Erstellung dieser Arbeit sind u. a. die folgenden Plattformen verfügbar (vgl. Abb. 5.1):

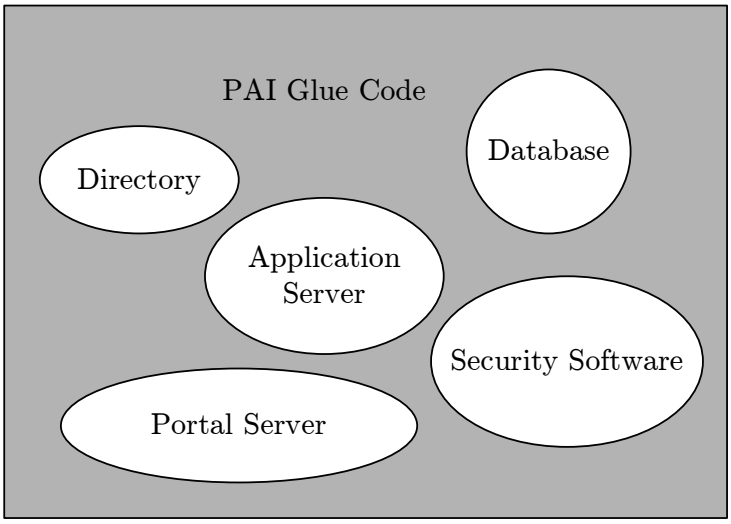

Abbildung 5.2.: PAI Glue-Code als Verbindung zwischen verschiedenen Basisprodukten nach [PSS05].

PAI Directory Plattform, PAI Security Plattform, PAI J2EE Application Plattform, PAI Portal Plattform, PAI Process Integration Plattform, PAI Business Information Broker Plattform (eine Beschreibung der verschiedenen Plattformen s. u.). Zusammen mit diesen werden außerdem einige Test- oder Beispielanwendungen, mit denen die Plattformen getestet werden, und Dokumentation an die Kunden ausgeliefert[1]. Diese erhalten damit eine komplette Grundlage, auf der eine Anwendung, wie etwa ein Internet- oder Intranet-Portal, errichtet werden kann. Bei dessen Entwicklung kann dann die Funktionalität, die Bedienbarkeit, das Design und Ähnliches in den Mittelpunkt gestellt werden.

Bei den Anwendungsprojekten handelt es sich in der Regel um Entwicklungsprojekte verteilter Applikationen, wobei die PAI-Plattformen selbst bereits aus verteilten Funktionseinheiten bestehen. Der von den Plattformen bereitgestellte Funktionsumfang umfasst verallgemeinerbare Services aus den Bereichen Sicherheit, Benutzermanagement, Applikations- und Prozessintegration sowie der Oberflächenintegration auf Basis von Portlet-Technologie. Verschiedene Kombinationen dieser Services ergeben dann für die Kunden sichtbare Features, die zum Teil auch eine Kombination verschiedener Plattformen bedingen. Als Beispiel kann hier die Vorbereitung der J2EE, Directory und Security Plattformen für Single Sign On zwischen verschiedenen Applikationen dienen. [GB05]

[1] Es gibt hierbei jedoch auch „interne“ Testanwendungen, die nicht ausgeliefert werden.

Die folgende Beschreibung der einzelnen PAI-Plattformen entstand in Anlehnung an [PAIb; Qua06; PAIa].

Directory und Security Plattformen

Die Directory Plattform bildet die Grundlage für die Sicherheitsinfrastruktur durch das Bereitstellen des Datenspeichers und des Datenmodells, um das Freigabemodell für alle Benutzer, Organisationen und alle Autorisierungsgruppen abzubilden. Ihre Hauptaufgabe ist es, die benötigten organisatorischen und applikationsspezifischen Benutzerdaten zu liefern. Durch die zentrale Ablage dieser Daten ermöglicht die Directory Plattform ein einheitliches Administrationsmodell unabhängig von der Applikation.

Die Security Plattform besteht aus Komponenten, um globale Sicherheitsrichtlinien zu steuern und umzusetzen. Hierzu werden Basisfunktionen bereitgestellt, um den Zugriff auf Applikationen und Ressourcen basierend auf applikationsspezifischen Benutzergruppen zu regeln. Zusätzlich zur Verwaltung der Autorisierungen bildet die Security Plattform die grundlegende Infrastruktur zur Authentifizierung und für Single Sign On.

J2EE Plattform

Basierend auf der SUN J2EE Technologie[2] stellt die PAI J2EE Plattform alle Mittel bereit, um effizient Unternehmensapplikationen entwickeln, installieren und betreiben zu können. Dabei besteht ein großer Vorteil für die Entwickler darin, dass die Integration der IBM Websphere Produkte, insbesondere des Application Servers, mit den anderen Plattformen bereits besteht. Auf diese Weise können effizient J2EE-konforme Applikationen für die Daimler IT-Infrastruktur entwickelt werden. Durch die Plattform werden auch weitere Standardisierungen innerhalb der Daimler AG weltweit auf flexible Art umgesetzt. Als Beispiel kann hier ein einheitliches Logging genannt werden.

Portal Plattform

Auf Basis der Portal Plattform können viele verschiedene Typen von zu veröffentlichenden Inhalten in einer einheitlichen Weise zusammengefügt werden. Dadurch entstehen Webseiten, auf denen durch die Integration verschiedener Portlets Auszüge aus mehreren

[2]Siehe `http://java.sun.com/javaee/` (zuletzt aufgerufen am 15.09.2008).

Informationsquellen und externen Systemen dargestellt werden können. Die Endbenutzer des Portals können sich diese Seiten dann individualisieren und diverse Portlets hinzufügen oder entfernen. Da die PAI Portal Plattform Schnittstellen zu den PAI Directory und Security Plattformen bereitstellt, kann dies angebunden an zentrale Verzeichnisse anhand der für die Daimler AG einheitlichen Authentifizierungs- und Authorisierungsmechanismen und -richtlinien erfolgen.

Business Information Broker Plattform

Diese Plattform unterstützt die Koordination der Interaktionen verschiedener Business-Anwendungen. Dazu stellt sie umfangreiche Funktionen zur Datentransformation und des Event Controllings nach einem *Publish-Subscribe*-Muster bereit und kann als Service Broker verwendet werden und damit einen Enterprise Service Bus bilden.

Process Integration Plattform

Die Process Integration Plattform stellt eine Laufzeit-Basis für prozessorientierte Anwendungen dar. Sie vereinfacht die Implementierung, die Durchführung und das Monitoring von Geschäftsprozessen und integriert die PAI J2EE Plattform, um die effiziente Implementierung von Prozessmodellen zu gewährleisten. Des Weiteren wird auch bei dieser Plattform eine Schnittstelle zu den Directory, Security und Business Information Broker Plattformen zur Verfügung gestellt.

5.1.2. Aufbau

Die PAI-Plattformen können je nach Sichtweise einer ganzen Reihe an Entwurfsmustern zugeordnet werden. Zunächst sind sie als verteilte oder nebenläufige Systeme aufgebaut, da die Plattformen verschiedene Server (Application Server, Web Server, Process Server) und andere Subsysteme, die meist ebenfalls auf unterschiedlichen Rechnern betrieben werden (Directory, Message Broker, etc.), beinhalten. Die Plattformen selbst sind als Serversysteme ausgelegt, die die Kommunikation mit Clients unterstützen.

Aus Entwicklungssicht ist der Glue Code aufgeteilt in verschiedene Komponenten. Eine oder mehrere Komponenten zusammen bilden ein oder mehrere Features, die die Sicht der Anwender definieren und deren Realisierung auch die Basisproduktfunktionen ver-

wendet. Eine Menge an Features und benötigten Basisprodukten wird dann in einer Plattform zusammengefasst. Manche Features werden erst durch die Kombination verschiedener Plattformen realisiert. Andere verwenden Komponenten, die auch in anderen Features und anderen Plattformen zum Einsatz kommen. Diese Komponenten werden als *Core Assets* bezeichnet. Das Zusammenspiel von Plattformen und Features geschieht analog zum in Abschnitt 2.2.2 erläuterten Modell.

Die Darstellung von Produktlinienplattformen nach [BPL05] besitzt einige Parallelen zu PAI. Die PAI-Plattformen sind mit letzteren jedoch nicht gleichzusetzen, da bei der Entwicklung der Produktlinienplattform die später darauf aufbauenden Anwendungen zum Großteil schon bekannt sind (vgl. Abschnitt 2.1.6).

Auch von einem anderen Standpunkt aus betrachtet gibt es Ähnlichkeiten zwischen PAI und Produktlinien. Die PAI *Core Assets* können ebenfalls als Produktlinienplattform angesehen werden, da sie in einer Reihe von Plattformen Verwendung finden. Doch auch in diesem Fall sind nicht alle Eigenschaften einer Produktlinie vorhanden, da die Aufteilung in Komponenten nicht wie bei Produktlinien üblich in einem Scoping durchgeführt wurde, sondern vielmehr als einfache Wiederverwendung von Code und Funktionalitäten anzusehen ist.

5.2. Anpassung der Rahmenbedingungen

Die Anforderungen an eine Softwareplattform kommen meist aus sehr unterschiedlichen Richtungen. Eine Gruppe bilden dabei die Kunden. Hierbei muss unterschieden werden zwischen direkten Anwendern der Plattform und Endanwendern einer Applikation. Erstere sind in der Regel Softwareentwickler, die aufbauend auf der Plattform Anwendungen zum Gebrauch durch die Endanwender entwickeln. Für die Endanwender sollte es idealerweise nicht transparent sein, ob die von Ihnen benutzte Anwendung mit oder ohne eine Plattform entwickelt wurde. Die verschiedenen direkten Anwender der Plattform haben oftmals sehr unterschiedliche Anforderungen an die Plattform, die zum Teil konträr zueinander stehen. In diesen Fällen muss eine Entscheidung entweder vom Plattformentwicklungsmanagement oder der übergeordneten Organisation getroffen werden. Unter übergeordneter

Organisation ist die Abteilung, der Bereich, das Gesamtunternehmen, etc. zu verstehen, in dem die Plattform entwickelt wird. In vielen Fällen verhält es sich ähnlich wie bei den PAI-Plattformen: Die Plattformen werden in einem Konzernbereich entwickelt, um ausschließlich innerhalb des Unternehmens eingesetzt zu werden. Die übergeordnete Organisation bildet zugleich eine zweite Interessengruppe. Aus diesem Blickwinkel betrachtet ist es oft relevant, dass die Plattform als Mittel zur Verfolgung eines anderen Zwecks entwickelt wird.

Weitere Anforderungen und Randbedingungen entstehen durch das Produkt, die Plattform, selbst. Da die Systemtests über Testapplikationen durchgeführt werden (vgl. Abschnitt 3.1) hängt die Güte der Verifikation des Produkts stark von der Validation der Testapplikationen ab. Auch der Entwicklungsprozess und die verschiedenen Testprozesse als Teile davon müssen betrachtet werden und können weitreichende Auswirkungen haben. Diese Prozesse weisen bei Plattformen im Allgemeinen stets Unterschiede zur Entwicklung von Anwendungen auf. Hauptursache hierfür ist, dass aus Anwendungssicht die System- oder -abnahmetests der Plattform lediglich als Integrationstests anzusehen sind, da das Gesamtsystem (inklusive der Applikation) darin noch nicht getestet wird (vgl. Abb. 4.2).

5.2.1. Umgang mit Defects

Defect Attribute

Das Ziel des Testens ist es, Fehler zu finden. Falls ein Testfall einen solchen Fehler, oder Defect, aufdeckt, muss dieser zunächst protokolliert werden. Hierzu gibt es einige sog. *Bug Tracking Tools*, in denen ein Defect „aufgemacht“ bzw. angelegt werden kann. Mit einigen Attributen versehen wird er dann vom Tester, der ihn entdeckt hat, demjenigen zugewiesen, der dafür verantwortlich ist oder je nach Prozess einer anderen Person. Bei der Entwicklung von PAI hat der Tester mehrere Möglichkeiten. Falls er bereits weiß, wer den Fehler beheben kann (und sollte), wird ihm der Defect zugewiesen. In vielen Fällen ist dies jedoch noch nicht bekannt. Dann kann der Defect entweder dem Teilsystemverantwortlichen oder, falls auch das verursachende Teilsystem nicht bekannt ist, zunächst dem Testmanager zugewiesen werden.

Defect Attribute	**Kurzbeschreibung / mögliche Werte**
Defect ID	Fortlaufende Nummer
Status	New, Open, Reopen, Rejected, Fixed, *Re-Test*, Closed, *Nicht reproduzierbar*
Severity	Show Stopper, High, Medium, Low
Detected in Version	Versionsnummer der getesteten Software
Detected By	Benutzerkennung
Assigned To	Benutzerkennung
Detected on Date	Datum der Fehlerentdeckung
Closing Date	Datum des bestandenen Re-Tests
Closed in Version	Versionsnummer der getesteten Software
Description	Beschreibungsfreitext
Delivery Impact	*Stop, Unknown, Workaround, Release Note, Accepted, Internal*
Next Delivery	Versionsnummer der getesteten Software
Reason	*Unknown, ArchitectureAndDesign, BaseProd, Code, Docu, EnvConfig, ISOCreationAndDistribution, Installation, InternalProcess, TestApp, TestData, TestExec, TestScript, TestSpec, TestTool*

Tabelle 5.1.: PAI-Fehlerattribute im Mercury Quality Center 9.0.
Kursiv gedruckte Attribute und Ausprägungen sind benutzerdefiniert.

Das bei PAI eingesetzte Werkzeug zur Fehlerverwaltung ist ein Modul des Mercury Quality Center 9.0 (siehe S. 146 ff.). Die Standardeinstellung stellt bereits den Großteil der benötigten Attribute eines Defects bereit. Um speziell auf die Bedürfnisse von Plattformen einzugehen, wurden diese durch eigens definierte Attribute und Attributsausprägungen erweitert. Die meisten der zum Teil automatisch ausgefüllten Attribute werden in Tabelle 5.1 zusammen mit ihren möglichen Werten dargestellt. Die benutzerdefinierten Attribute und Attributsausprägungen sind kursiv gedruckt ebenfalls in der Tabelle enthalten.

Im Folgenden werden die wichtigsten und erklärungsbedürftigsten Attribute kurz dargestellt. Eine tiefergehende Beschreibung v. a. der Zusammenhänge befindet sich im Abschnitt Fehlerlebenszyklus ab S. 141.

- **Status**: Der Status gibt den Zustand des Defects an und muss je nach Zustand von verschiedenen Personen (bzw. Rollen) geändert werden.
- **Detected By**: Dieses Feld benennt den verantwortlichen Tester – in der Regel derjenige, der den Fehler gefunden hat. Im Fehlerlebenszyklus ist dieser als *Defect Owner*

bezeichnet.

- **Assigned To**: An dieser Stelle ist derjenige benannt, dem der Fehler zugewiesen wurde. Im Fehlerlebenszyklus entspricht dies dem *Defect Agent*.
- **Delivery Impact**: Der Delivery Impact gibt die Auswirkung auf die Auslieferung der Software an. Manche der gefundenen Fehler müssen vor Release unbedingt beseitigt werden, bei anderen ist es akzeptabel, zunächst eine Übergangslösung zu beschreiben, wie das Fehlverhalten umgangen werden kann[3]. Bei Plattformen kann man durch die Möglichkeit, den Releaseumfang flexibel zu gestalten, manche Defects akzeptieren bzw. das Attribut Delivery Impact auf *Accepted* setzen.
- **Next Delivery**: Da bei Plattformen ein Fehler in einigen Fällen in einem bestimmten Release akzeptiert werden kann, muss protokolliert werden, welches die nächste für diesen Fehler relevante Auslieferung ist.
- **Reason**: Dieses Attribut gibt die Fehlerquelle an. Anhand einer Analyse kann zumindest grob festgestellt werden, in welchem Prozessschritt die größten Verbesserungsmöglichkeiten liegen. Bei einem neu gefundenen Fehler wird dieses Attribut zunächst auf *Unknown* gesetzt.

Die meisten der Attribute werden vom verantwortlichen Tester gesetzt. Manche, beispielsweise der Delivery Impact, bedürfen jedoch einer Managemententscheidung.

Fehlerlebenszyklus

Der Fehlerlebenszyklus beschreibt den Lebenszyklus eines gefundenen Fehlers. Er gibt den Prozess vom Anlegen bis zum Re-Test und dem Schließen eines Defects vor. In diesem Abschnitt wird der bei PAI eingesetzte Prozess beschrieben. Eine Übersicht über den gesamten Lebenszyklus ist in Abb. 5.3 dargestellt. Die Rechtecke stellen Aktionen dar, das Fähnchen rechts oben gibt jeweils an, wer bzw. welche Rolle für die Aktion verantwortlich ist. Dabei steht „O“ für den *Defect Owner*. Dies entspricht in der Regel dem Tester, der den Fehler aufgedeckt hat. „A“ bezeichnet Aktionen des *Defect Agents*, in der Regel derjenige, der den Fehler beheben kann, meist der Entwickler. Und „M“ steht für den *Test* oder

[3]Hierbei ist in der Praxis meist von einem *Workaround* die Rede.

Quality Manager oder das Projektmanagement. Rauten stehen für Entscheidungspunkte; die Beschriftungen an den zugehörigen Achsen spezifizieren die jeweiligen Bedingungen. Rauten, von denen nur eine Abzweigung weg führt, sind keine Entscheidungen, sondern als an eine Bedingung geknüpfte Wartepunkte zu interpretieren. Erst wenn die an der Kante angegebene Bedingung erfüllt ist, wird mit der nächsten Aktion begonnen.

Ein gefundener Fehler unterliegt zunächst noch der Verantwortung des Testers, der ihn gefunden bzw. angelegt hat. Dies ist und wird auch im weiteren Verlauf der Defect Owner sein. Zunächst ist der Zustand des Defects noch *New*. Mit dem Ausfüllen des Feldes *Assigned To* bekommt der Defect Agent den Fehler zugewiesen. Dieser muss ihn anschließend prüfen und entweder ablehnen bzw. auf *Rejected* setzen oder annehmen bzw. auf *Open* setzen.

Im ersten Fall, der Fehler wurde abgelehnt, kann der Defect Owner dies nun akzeptieren und den Defect schließen bzw. auf *Closed* setzen. Falls der Defect Owner die Ablehnung nicht akzeptiert, wird der Status erneut auf *New* gesetzt und dem Defect Agent erneut zugewiesen.

Andernfalls nimmt der Defect Agent den Defect an und setzt ihn auf *Open*. Er ist anschließend für die weitergehende Analyse und die Behebung des Problems verantwortlich. Im Idealfall wird die Fehlerquelle gefunden und beseitigt, woraufhin der Defect Agent den Zustand auf *Fixed* setzt. Nun ist zunächst der Test-Manager dafür verantwortlich, gemeinsam mit der Projekt- und Releaseplanung abzustimmen, wann der geänderte Code auf der Testumgebung eingespielt wird. Ist dies geschehen und die Plattform bereit, einem Re-Test unterzogen zu werden, wird auch der Status vom Test-Manager angepasst und auf *Re-Test* gesetzt. Dies stellt einen Wartepunkt dar.

Nun ist erneut der Defect Owner gefragt, der den Re-Test durchführen soll. Falls dieser Test die Behebung des Defects bestätigt, kann der Fehler geschlossen bzw. auf *Closed* gesetzt werden. Andernfalls wird der Status auf *Reopen* gesetzt und erneut dem Defect Agent zugewiesen, der die Analysephase erneut anstoßen muss.

In manchen Fällen kommt es in der Praxis auch vor, dass ein Defect nicht reproduzierbar ist. In solchen Fällen kann der Defect Owner dem Fehlerstatus den Endzustand *NRep* zuweisen.

Zu bestimmten Zeitpunkten (Quality Gates, etc.) während des Entwicklungsprozesses finden sog. *Defect Adjudge Meetings* statt. In diesen Besprechungen werden alle Defects in oder nach der Analysephase vom Defect Agent, dem Defect Owner, dem Management und allen anderen an den jeweiligen Defects beteiligten Projektmitgliedern betrachtet und bewertet. Dabei wird der *Delivery Impact*, d. h. die Auswirkung des Defects auf die Auslieferung festgelegt. Mögliche Werte sind hierbei:

- *Unknown*: Der Auslieferungseinfluss ist (noch) nicht bekannt. Dies ist der Ausgangszustand jedes Defects.
- *Stop*: Der Defect hält die Auslieferung der Plattform auf.
- *Workaround*: Es gibt eine provisorische Lösung, anhand der das Problem umgangen werden kann. Der Defect hat durch diesen Workaround keinen Einfluss auf die Auslieferung. Dieses Vorgehen sollte in den Release Notes beschrieben werden.
- *Release Note*: In den Release Notes ist eine Beschreibung des Fehlers enthalten. Die Software kann ohne die entsprechende Funktionalität den Kunden zur Verfügung gestellt werden.
- *Accepted*: Der Fehler wird ohne einen Eintrag in den Release Notes akzeptiert und hat keinen Einfluss auf die Auslieferung. Diese Entscheidung muss vom Projektmanagement getroffen werden.
- *Internal*: Bei dem entdeckten Fehler handelt es sich um ein internes Problem (beispielsweise Fehler, die die Testumgebung oder die Testspezifikation betreffen), das keinen direkten Einfluss auf die Anwendung der Plattform oder deren Auslieferung hat. Es sollte beachtet werden, dass z. B. durch eine fehlerhafte Testumgebung die Tests evtl. nicht wie gewünscht ausgeführt werden können und somit nur eine geringere Testabdeckung erreicht werden kann.

Auch dieser Vorgang ist in Abb. 5.3 dargestellt. Am Entscheidungspunkt nach der Analyse und der Behebung des Fehlers führt eine Abzweigung zur Aktion „Execute defect adjudge meeting“. Dies gehört nicht mehr zum eigentlichen Lebenszyklus des Fehlers im engeren Sinne, beeinflusst diesen bzgl. der nächsten Auslieferung jedoch stark. Durch das Festlegen des *Delivery Impact* wird auch die Priorität der Behebung eines Fehlers stark

beeinflusst.

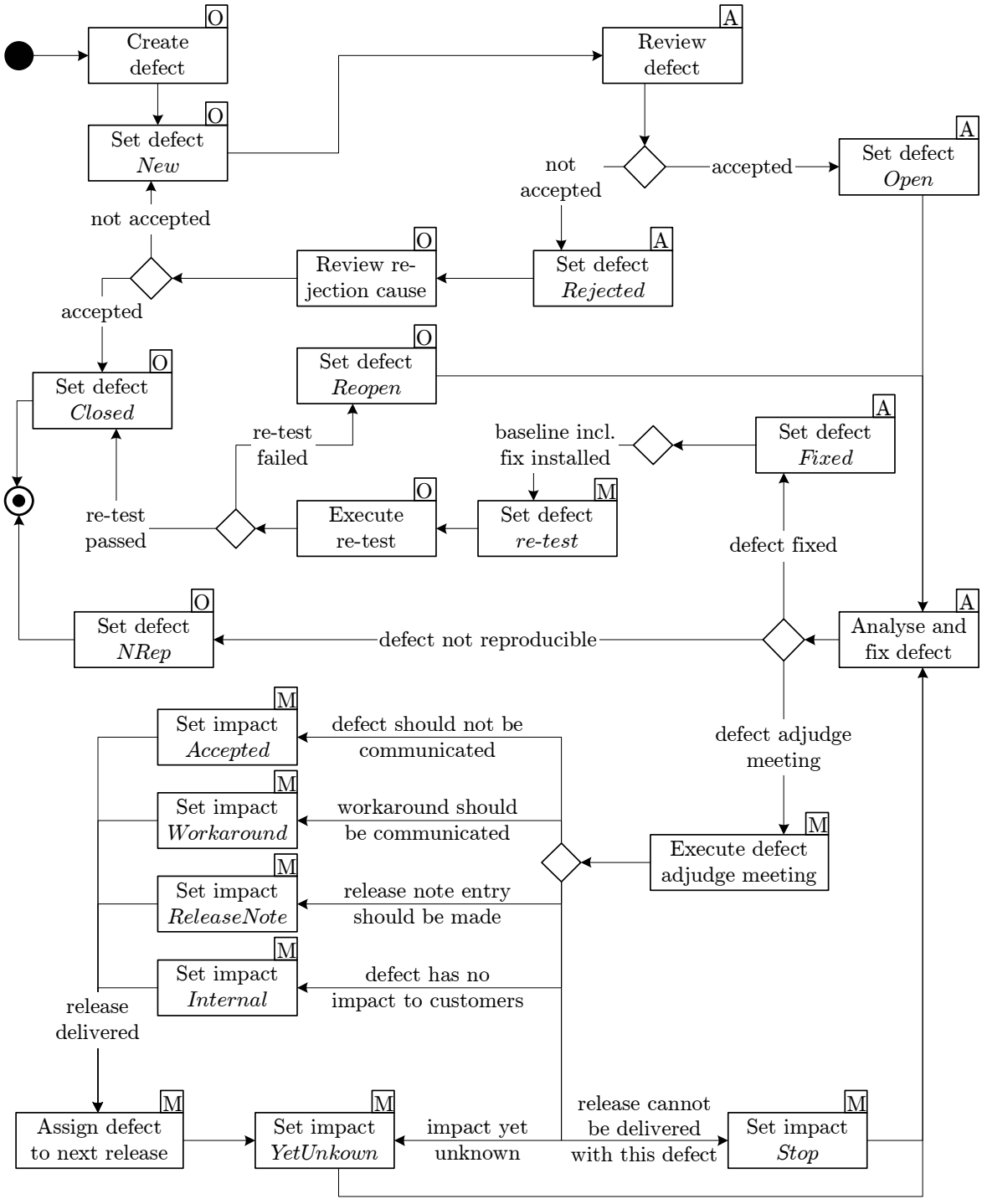

Abbildung 5.3.: Fehlerlebenszyklus.

5.2.2. Testdurchführung und die Quality Center Struktur

Zur Verwaltung der Systemtests von PAI wird das Mercury Quality Center 9.0 eingesetzt. Dieses ermöglicht die Verwaltung des Testprozesses von der Definition der Testanforderungen über die Testdurchführung bis hin zur Fehlerverwaltung in vier Tabs. Hinter einem fünften Reiter verbirgt sich mit dem Mercury Dashboard ein Tool zur Definition und regelmäßigen Auswertung von Kennzahlen. Dieser grobe Aufbau ist auch in Abb. 5.4, die feingranularere Struktur des Quality Center am Ende dieses Abschnitts auf S. 153 in Abb. 5.6 dargestellt. Letztere stellt auch die eigene Entwicklung einer zur Kennzahlanwendung optimierten Testinfrastruktur dar.

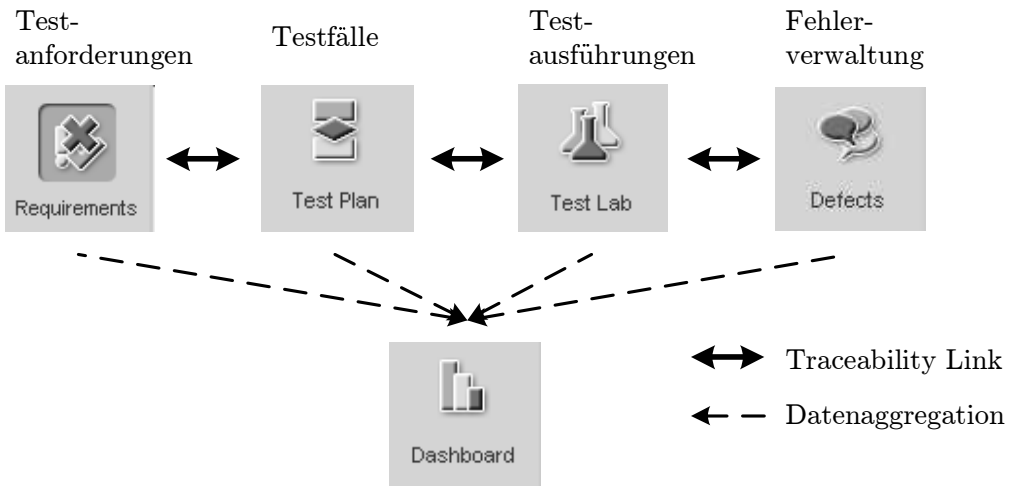

Abbildung 5.4.: Tabs des Mercury Quality Center.

Die ersten vier Reiter, *Requirements, Test Lab, Test Plan* und *Defects*, bilden den Rahmen für die operative Testspezifikation und -durchführung. In ihnen kann auch die Traceability bis zu einem gewissen Grad realisiert werden. Die Dokumentation der Rückverfolgbarkeit hat jedoch ihre Grenzen, da der Requirements-Reiter meist nicht dafür verwendet wird, alle Anforderungen an die Software zu verwalten[4].

Auch bei PAI werden mit dem Quality Center nicht die Anforderungen an die Software verwaltet. Dies geschieht getrennt durch ein anderes Softwaretool[5]. Ein Medienbruch an

[4] In früheren Versionen des Quality Center, damals noch *Test Director*, wurde dies auch durch die Namensgebung deutlich: Der Reiter zur Verwaltung der Anforderungen war mit *Test Requirements* beschriftet.
[5] Telelogic Doors.

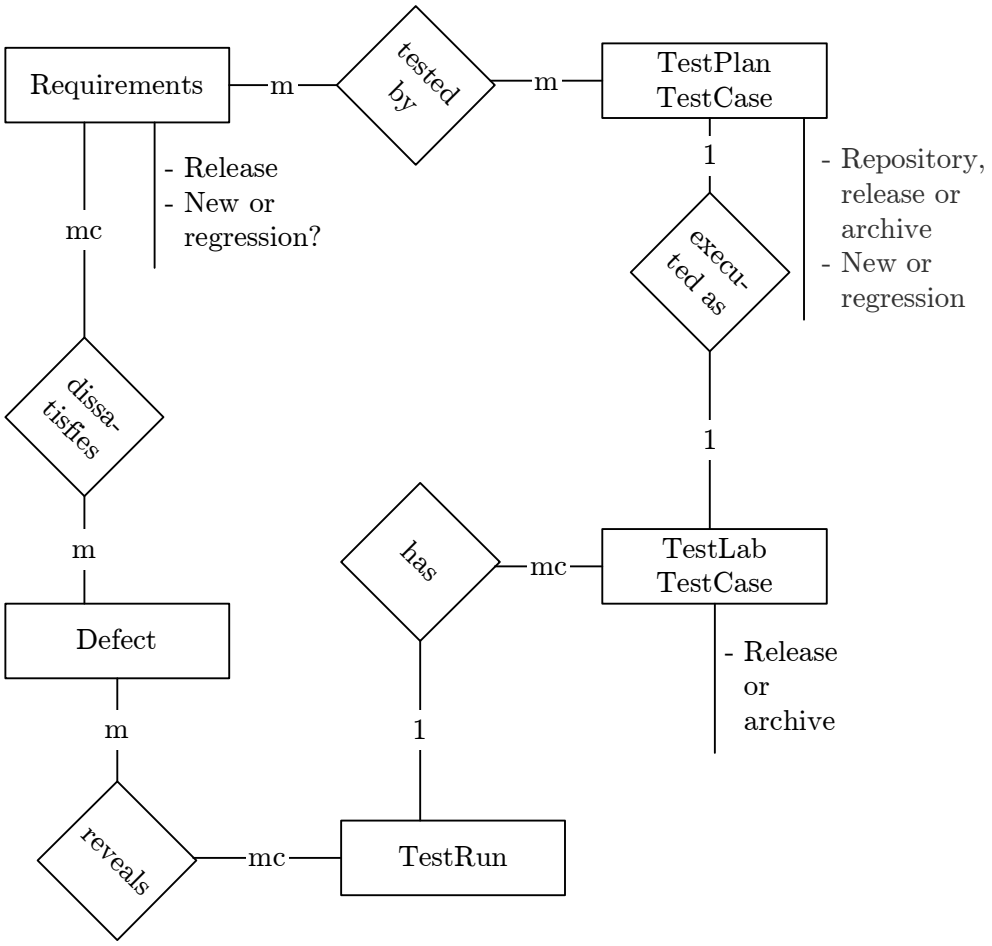

Abbildung 5.5.: Entity Relationship (ER)-Diagramm des Quality Centers.

dieser Stelle ist insbesondere bei höherstufigen Tests unvorteilhaft, da diese direkt auf den Anforderungen aufbauen. Umso erstaunlicher ist die weite Verbreitung der Tool-Trennung von Anforderungen und Tests. Bei der PAI-Entwicklung wird dieser Graben versucht zu überbrücken, indem im Requirements-Reiter explizit Testrequirements verwaltet werden. Unter Testrequirements sind hier Anforderungen an den Test zu verstehen. Diese bilden die Basis für die Entscheidung, welche Tests durchgeführt werden müssen bzw. sollten. Beeinflusst wird die Auswahl zum einen durch Anforderungen an die Software, zum anderen aber auch durch Rahmenbedingungen, wie beispielsweise die Hardwareinfrastruktur, und andere Erfordernisse an den Test. Unter Letzteres fallen Regressions- und Re-Tests und bei Plattformen können aufgrund der fehlenden NFRs auch einige der nichtfunktionalen Tests hierzu gezählt werden.

In den folgenden Abschnitten wird anhand des Tools, des Umgangs damit und eigener Anpassungen dargestellt, wie die Tests durchgeführt und dokumentiert werden und somit die Grundlage für den Aufbau des Kennzahlensystems bilden. Da das Quality Center flexibel und vielseitig einsetzbar ist, muss v. a. bei größeren oder regelmäßigen Testvorhaben die Struktur gut überlegt und schlüssig sein. In Abb. 5.5 sind die Zusammenhänge der verschiedenen Entitäten, die im Quality Center verwaltet werden, in einem *Entity Relationship*-Diagramm dargestellt. Es sind dort nur diejenigen Attribute der Entitäten abgebildet, die durch die Ordnerstruktur im Quality Center ausgedrückt werden.

Requirements

Im Requirements-Tab werden für jedes Release alle Testrequirements in einem Ordner angelegt. Ein großer Teil davon, alle Regressions- und Re-Tests, ergeben sich durch Kopieren aus dem Ordner eines Vorgängerreleases. Im Falle neuer Features werden aber auf Basis der ausformulierten Anforderungen an die Plattform Testanforderungen erstellt. Diese bilden in den meisten Fällen die Plattformanforderungen *1:1* ab und müssen dadurch lediglich auf das entsprechende Anforderungsdokument referenzieren. Nach der Auslieferung eines Releases wird der releasespezifische Ordner in den Archivordner verschoben.

Die Struktur jedes Releases sieht zunächst eine Unterteilung in neue oder bereits bestehende Features vor. Der Ordner für bereits bestehende Features enthält weitere Ordner, je einen für Regressionstest-, Re-Test- und Solutiontestanforderungen[6]. Diese enthalten wie der Ordner für neue Features die Testanforderungen, die sich zum Teil direkt aus den Features ergeben, zum Teil von diesen unabhängig sind. Beispielsweise beziehen sich im Falle der Re-Tests die Testanforderungen auf bekannte und bereits behobene Fehler.

Das Quality Center stellt die Möglichkeit bereit, die Anforderungen mit Testfällen zu verknüpfen. Da jedes Testrequirement einen Bedarf für einen Testfall darstellt, gilt der Grundsatz, dass jedes Testrequirement mit mindestens einem Testfall für das entsprechende Release im Test Plan verknüpft sein muss.

[6]Eine *Solution* ist ein bestimmtes Installationsszenario der Plattform.

Test Plan

Im Test Plan befinden sich alle Testfälle und -szenarien und werden dort für Releases zusammengestellt. Die Darstellung erfolgt an dieser Stelle noch unabhängig von einer späteren Ausführungsumgebung, Testfälle sollten durch entsprechende Parametrisierungsmöglichkeiten flexibel gestaltet sein.

Da das Quality Center keine eigene Versionierungsmöglichkeit von Testfällen bereitstellt, muss auf andere Weise zum einen sichergestellt werden, dass von jedem Testfall die aktuellste Version schnell verfügbar ist. Zum anderen muss garantiert sein, dass auch ältere Versionen von Testfällen nicht verloren gehen.

Dies wird durch einen bestimmten Umgang mit dem Tool erreicht. Im Test Plan gibt es eine Unterteilung in Releasezusammenstellungen und zentralen Testfallspeicher, das *Repository*. Das Repository enthält die aktuellsten Versionen aller Testfälle. Für jedes neue Release wird ein neuer Ordner angelegt und die benötigten Testfälle aus dem Repository kopiert. Nach dem Durchführen der Tests wird der Release-Ordner archiviert. Auf diese Weise werden die o. g. Forderungen realisiert. Dieses Vorgehen ist kritisch, wenn innerhalb eines Releases ein Testfall geändert oder angepasst werden muss. Dies ist hier nur durch eine entsprechende Namensgebung zu realisieren.

Für jedes Release gibt es auch hier eine hierarchische Ordnerstruktur, die auf der obersten Ebene nach der Testart unterscheidet. Mögliche Kategorien sind funktionale, nichtfunktionale und Installationstests. Auf der nächsten Ebene wird zwischen neuen und bereits bestehenden Features unterschieden. Dies entspricht der Einteilung in Regressionstest und Test neuer Funktionalitäten. Die letzte Ebene vor den eigentlichen Testfällen beinhaltet die Einteilung der Features. Diesen werden dann die Testfälle zugeordnet.

Um eine Rückverfolgbarkeit von Testfällen zu Anforderungen zu garantieren, muss jeder Releasetestfall mit mindestens einem Testrequirement verknüpft werden. Hierbei kann es vorkommen, dass ein Testfall mit mehreren Testrequirements verknüpft wird, da manche Testfälle mehr als nur einen Aspekt überprüfen. Die Testfälle im Repository werden nicht mit den (releasespezifischen) Testanforderungen verbunden, da sie unabhängig von einer bestimmten Version sind.

Test Lab

Das Test Lab stellt die Ausführungsumgebung der zur Ausführung vorgesehenen Testfälle dar. Im Idealfall werden alle in den Releasezusammenstellungen im Test Plan vorhandenen Testfälle für ein Release mindestens einmal ohne Fehler durchgeführt. Dafür wird für jedes Release ein Ordner angelegt, der für jede der zu testenden Baselines[7] einen weiteren Ordner enthält. Die zur Ausführung geplanten Testfälle werden in die Baseline-Ordner kopiert und vorbereitet. Diese Vorbereitung beinhaltet beispielsweise die Konfiguration der Testfälle für die spezifischen URLs der Testumgebung. Die Testfälle für die bestimmte Baseline werden daraufhin ausgeführt und falls Fehler gefunden werden, die vor Auslieferung der Software noch behoben werden müssen, wird eine neue Baseline mit den Fehlerkorrekturen erstellt und installiert. Der Testumfang der neuen Baseline beinhaltet die Re-Tests der gefundenen Fehler, die für die vorherige Baseline noch nicht ausgeführten Tests und die Regressionstests für die in der vorherigen Baseline bereits getesteten Bestandteile. Die Inhalte der verschiedenen Ordner sind somit meist unterschiedlich.

Es kann auch vorkommen, dass eine Baseline bereits bei der Installation gravierende Mängel aufweist, die nur in einer neuen Baseline behoben werden können. Beispielsweise wird Baseline *B01* erstellt, bei der Installation jedoch bereits zurückgewiesen, woraufhin *B02* erstellt und installiert wird. Im Test Lab ist somit der erste Ordner der für *B02*. Angenommen im Test werden nun Fehler gefunden, die eine Fortsetzung der Tests verhindern. Dann könnte eine Baseline *B03* erstellt und installiert werden, deren Testfälle dreigeteilt sind: noch nicht durchgeführt, Re-Tests und Regressionstests.

Zusätzlich zu den Baseline-Ordnern wird im Test Lab bei Bedarf ein Analyse-Ordner angelegt, der im Vorfeld ungeplante Testfälle zur Analyse eines Problems beinhaltet. Dies kommt im Performance- und Stabilitätstestumfeld häufig vor und ist aufgrund der schlechten Planbarkeit von den „normalen“ Releasetests zu trennen. Dadurch ermöglicht wird die Analyse des Testfortschritts. Der Prozentsatz geplanter und tatsächlich ausgeführter und bestandener und nicht bestandener Tests kann nun ausgehend von der Grundmenge der im Test Plan vorgesehenen Tests ermittelt werden. Die Voraussetzung hierfür ist die Ver-

[7] Eine Baseline ist ein Entwicklungsstand der gesamten Plattform. Sie wird nur auf der Systemtestumgebung installiert, wenn sie alle geplanten Features in einer finalen Version beinhaltet oder bereits in einer weiteren Baseline die Fertigstellung der noch fehlenden Features geplant ist.

knüpfung der Testausführungen im Test Lab mit den (releasespezifischen) Testfällen im Test Plan. Dies wird bei entsprechender Handhabung bereits durch das Tool durchgeführt. Die weitere Struktur des Test Labs entspricht der des Test Plans.

Im Test Lab werden somit die Ausführungen der Testfälle verwaltet und erstellt. Dadurch ändert sich der Status der Testfälle. Auf der obersten Übersichtsebene einzelner Testfälle wird stets der Status der letzten Ausführung des entsprechenden Testfalls dargestellt. Dieses Attribut kann folgende Werte annehmen:

- *No Run*: Der Testfall wurde noch nicht ausgeführt.
- *Not Completed*: Der Testfall wurde bei seiner letzten Ausführung (noch) nicht komplett ausgeführt[8].
- *Vague*: Der Testfall wurde durchgeführt, die Entscheidung ob er zu *Passed* oder zu *Failed* zu zählen ist, wurde jedoch noch nicht getroffen[9].
- *Passed*: Der Testfall deckte keinen Fehler auf.
- *Failed*: Der Testfall deckte einen Fehler auf.

Defects

Der Defect-Tab ermöglicht durch die Verwaltung der gefundenen Fehler die Dokumentation des in Abschnitt 5.2.1 dargestellten Fehlerlebenszyklus. In tabellarischer Sicht werden alle Fehler mit ihren Attributen dargestellt. Diese Sicht lässt sich beliebig filtern, wodurch beispielsweise alle offenen oder alle in einem bestimmten Release gefundenen Fehler angezeigt werden können. Auch Statusänderungen bestimmter Attribute werden mit Zeitstempel gespeichert.

Um die Fehlerrückverfolgbarkeit zu gewährleisten, muss jeder Fehler mit dem Testfall verknüpft werden, durch den er aufgedeckt wurde. Diese Traceability Links können ebenfalls durch das Quality Center abgebildet werden. Eigene Erweiterungen an dieser Stelle beschränken sich auf die speziell definierten Attribute (siehe Abschnitt 5.2.1).

[8] Bei lang laufenden Stabilitätstests kann dieser Status durchaus über mehrere Tage vorkommen.

[9] V. a. bei Plattformen ist diese Entscheidung im Performancetest-Bereich durch das Fehlen von NFRs oftmals nicht einfach zu treffen. Doch auch bei herkömmlichen Anwendungen muss bei vielen Testfällen zunächst eine Analyse durchgeführt werden.

Dashboard

Der fünfte Tab im Quality Center besteht aus einem Link zum Dashboard, einem modernen Werkzeug zur Unterstützung des Testmanagements. Dieses wird auf einem getrennten Server betrieben, hat eine eigene Datenbank und eigene Web und Application Server. Im Quality Center ist lediglich ein Link hinterlegt.

Das Dashboard greift auf die Quality Center-Datenbank und dessen Logfiles zu, extrahiert und speichert die benötigten Daten. Auf diese Weise wird die Betriebssicherheit der operativen Testumgebung nicht beeinflusst. Durch die Definition von *Key Performance Indicators* (KPIs) wird dem Benutzer ermöglicht, anhand von SQL oder JavaScript Daten des Quality Centers zu aggregieren, in statistischen Auswertungen zu verwenden und in Schaubildern darzustellen. Das Dashboard besitzt eine eigene Benutzerverwaltung, die die gezielte Freigabe oder Sperrung von Schaubildern ermöglicht.

Die KPIs bilden jeweils nur wenige Datenbankaktionen ab. Sie werden in Schaubildern integriert, die wiederum in Modulen und Portlets enthalten sind. Die einzelnen Schaubilder werden regelmäßig (täglich, wöchentlich, etc.) aktualisiert bzw. im Falle von Verlaufsdarstellungen erweitert. Es ist möglich, von einem Schaubild mehrere Instanzen in verschiedenen Portlets zu definieren und so für verschiedene Releases dieselben Schaubilder zu verwenden, ohne sie komplett neu definieren zu müssen.

Ein wesentlicher Nachteil des Dashboards ist, dass historische Daten nicht untersucht werden können. Ein KPI kann stets nur für zukünftige Untersuchungen realisiert werden. Man muss sich somit vor der Anwendung genau überlegen, welche Schaubilder, die auch als pdf-Export zur Verfügung gestellt werden können, erstellt werden sollen.

Der Einsatz des Dashboards in der Praxis und die mit ihm erzielten Ergebnisse werden in Abschnitt 5.3 dargestellt.

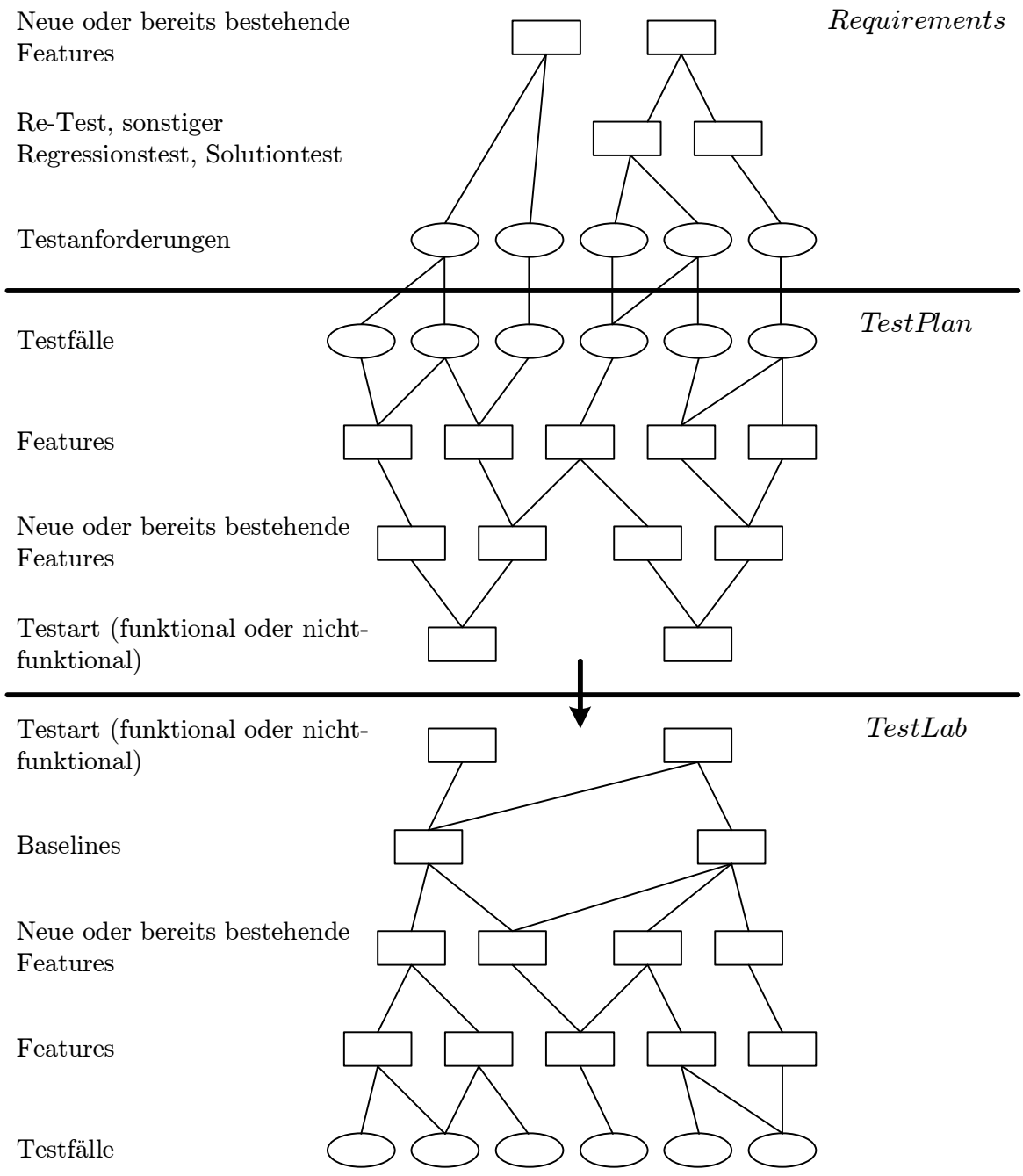

Abbildung 5.6.: Detaillierter Aufbau des Mercury Quality Center bei PAI.

5.3. Darstellung der Kennzahlen

In diesem Abschnitt werden exemplarisch die Umsetzung einiger Kennzahlen u. a. anhand des Mercury Dashboards und einige Ergebnisse dargestellt. Diese entstanden im Rahmen der Entwicklung der „Proaktiven Infrastruktur“ (PAI, siehe 5.1) und bauen auf der Auswahl von Kennzahlen und der Priorisierung der Qualitätsattribute in Kapitel 4 auf. Das Kernstück der Umsetzung in der Praxis besteht aus den sog. *Release Pages*, die einen schnellen Überblick über das Release bieten und in Abschnitt 5.3.1 vorgestellt werden. Sie stellen eine gute Grundlage dar, um das Plattformmanagement bei der Freigabeentscheidung zu unterstützen. In den darauffolgenden Abschnitten werden einige der zum Teil in den Release Pages enthaltenen Schaubilder und Kennzahlen näher betrachtet.

Durch die hier dargestellten Kennzahlen wird v. a. die Entscheidung, ob die Plattform freigegeben werden kann, erleichert. Doch auch eine allgemeine qualitative Einschätzung ergibt sich aus den Kennzahlen. Diese erfolgt jedoch nicht durch eine explizite Aussage, sondern basiert auf der jeweiligen Interpretation durch den Betrachter. Eine Übersicht über den Teststatus wird ebenfalls gegeben durch die grafische Darstellung der Testergebnisse.

5.3.1. Release Page

Die Release Pages (siehe Abb. 5.7) sind Zusammenstellungen der wichtigsten Kennzahlen. Diese bilden sich aus den jeweils tagesaktuellen Daten und werden für jedes Release erstellt. Es gibt bei der PAI-Entwicklung somit in der Regel mehrere aktuelle Release Pages gleichzeitig, da meist mehrere verschiedene Plattformen gleichzeitig entwickelt und getestet werden.

Die Release Pages setzen sich zunächst aus jeweils fünf Schaubildern zusammen, die alle den tagesaktuellen Datenstand darstellen. Auf diese Weise lassen sich schnell der Testfortschritt und die Auslieferbarkeit, der Release-Status, erkennen. In vielen Fällen kann auf einen Blick entschieden werden, ob die Plattform für den Produktiveinsatz freigegeben werden kann. Zeitliche Verläufe lassen sich durch Klicken auf das jeweilige Schaubild für drei der fünf Darstellungen anzeigen. Dies ist über sog. *Drill-to Portlets* realisiert, welche im Mercury Dashboard ohne großen Aufwand zu erstellen sind. Auch die Fehlerherkunft

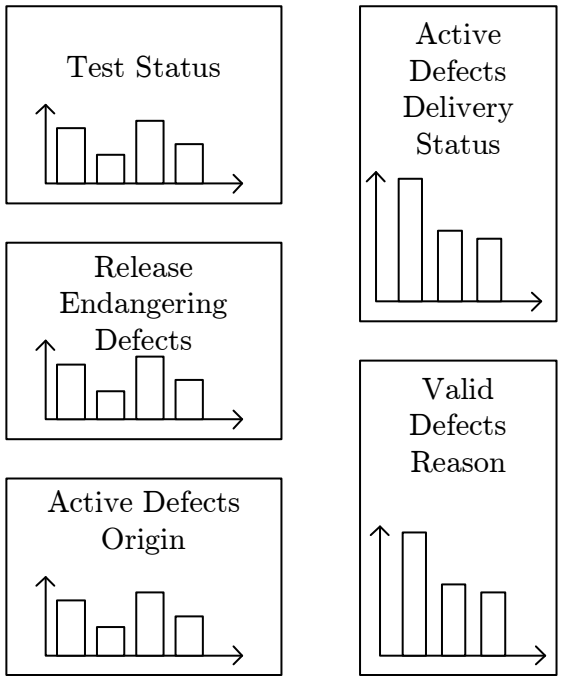

Abbildung 5.7.: Schematische Darstellung der *Release Page*.

und die für dieses Release akzeptierten Fehler werden deutlich gemacht. Letzteres hebt die aufgeschobenen Fehler hervor, die in einem der nächsten Releases angegangen werden müssen und verdeutlicht dadurch die Gefahr des Aufbaus eines großen Fehlerbergs. Im Folgenden werden die fünf Schaubilder kurz erläutert.

Test Status

Zunächst enthalten die Release Pages die jeweiligen Test Status, dargestellt als Balkendiagramme (siehe Abb. 5.8). Die fünf Balken repräsentieren die Testergebnisse aller Tests – der Regressionstests, der Tests neuer Features, der Funktionaltests und der nichtfunktionalen Tests. Die einzelnen Balken sind jeweils unterteilt nach den verschiedenen Status der Testfälle. Hierbei wird der Zustand des aktuellsten Testlaufs jedes für das Release geplanten Testfalls im Test Plan verwendet. Dieser Zusammenhang wird auch im ER-Diagramm in Abb. 5.5 deutlich. Mögliche Status eines Tests sind *Failed*, *Not_completed*, *No_Run*, *Vague* und *Passed*. Die Status *Vague* und *Not_completed* treten nur selten bzw. immer nur für kurze Zeit auf, wodurch sie in den tagesgenauen Schaubildern nur selten vorkommen.

Zu Beginn der Testphase, wenn der Test Plan komplett, im Test Lab jedoch noch keine Testausführung vorhanden ist, stehen in der Regel alle Balken komplett auf *No_Run*. Die Höhe der Balken, d. h. die Anzahl der zur Ausführung geplanten Testfälle, sollte über den Testzeitraum hinweg nahezu stabil bleiben, sofern keine über den geplanten Testumfang hinausgehenden Analyse-Tests nötig sind.

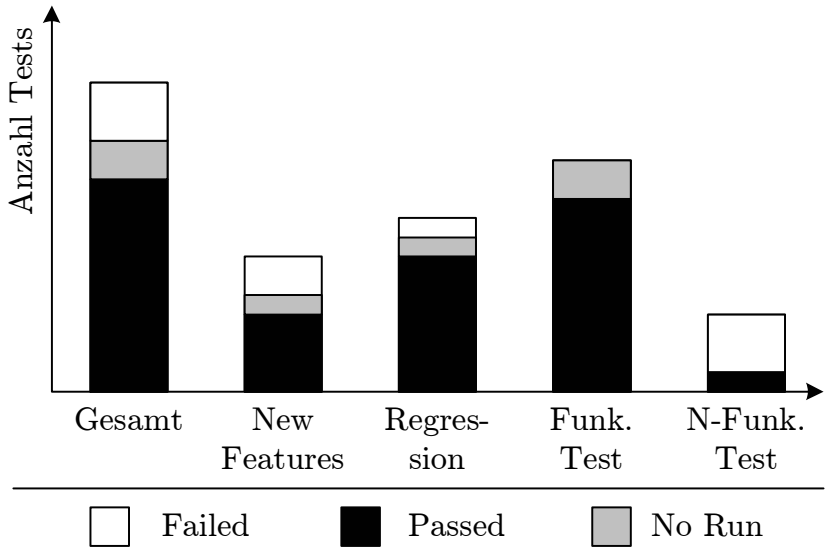

Abbildung 5.8.: Aktueller Teststatus nach Testtyp bzw. -bereich.

Das SQL-Statement, das die Datengrundlage für diese Kennzahl liefert, ist denkbar einfach:

```
SELECT Tests.Execution_Status,count(*) as value
FROM Tests
GROUP BY Tests.Execution_Status;
```

Die Datenqualität ist im Falle eines gut gepflegten und bereits zu Beginn der Testausführungsphase vollständigen Test Plans sehr gut. Die Schaubilder ermöglichen es, auf einen Blick abzuschätzen, wieviele der Tests durchgeführt wurden, welcher Anteil Fehler aufdeckte und wieviele der Tests welchen Typs noch ausstehen.

Die zeitlichen Verläufe dieser Schaubilder sind ebenfalls von großem Interesse, da sie Unregelmäßigkeiten im Testverlauf aufzeigen können und eine grobe zeitliche Einschätzung

der restlichen Testphase ermöglichen. Auch in der Literatur sind derartige Darstellungen verbreitet und werden, benannt nach ihrer typischen Form, oftmals S-Kurven genannt [Kan03].

Eine spezielle Anpassung an Plattformen bildet hierbei zum einen der Teststatus „Vague“, zum anderen die Einteilung von aufgrund fehlerhafter Testapplikationen nicht bestandener Testfälle als nicht ausgeführt (vgl. S. 123).

Release Endangering Defects

Ein gutes Instrument zur Bewertung, ob ein Release ausgeliefert werden kann, ist die Darstellung der Anzahl der releasegefährdenden Fehler in einem Balkendiagramm. Hierzu zählen alle Fehler, deren *Next Delivery* das entsprechende Release ist, deren Status weder *Closed* noch *NRep* und deren *Delivery Impact* weder *Stop* noch *Unknown* ist. Anders ausgedrückt sind dies alle Fehler, die potentiell einer sofortigen Auslieferung des Releases noch im Wege stehen. Die Höhe der einzelnen Balken stellt jeweils die absolute Anzahl an Fehlern dar, die Unterteilung bezieht sich auf die verschiedenen Status.

Abgesehen vom Auslieferzustand ermöglicht die zeitliche Darstellung des Verlaufs die Einschätzung weiterer Aspekte. Diese wird durch Klicken in das Schaubild dargestellt und ist ebenfalls als Balkendiagramm aufgebaut (siehe Abb. 5.9). Jeder Balken stellt einen Tag dar und ist unterteilt nach den verschiedenen Fehlerstatus. Als Datengrundlage dient dasselbe SQL-Statement wie bei der Darstellung der tagesaktuellen Werte:

```
SELECT Status, count(*) as value
FROM Defects
WHERE DeliveryImpact = 'Stop'
   OR DeliveryImpact = 'Yet Unknown'
GROUP BY Status;
```

Die Datenqualität ist hierbei abhängig von der manuellen Pflege des Fehlerstatus, des *Delivery Impact* und der Einhaltung des Fehlerlebenszyklus. Die Pflege des *Delivery Impact* liegt meist im Entscheidungsbereich des Managements und bedarf oftmals der Abstimmung

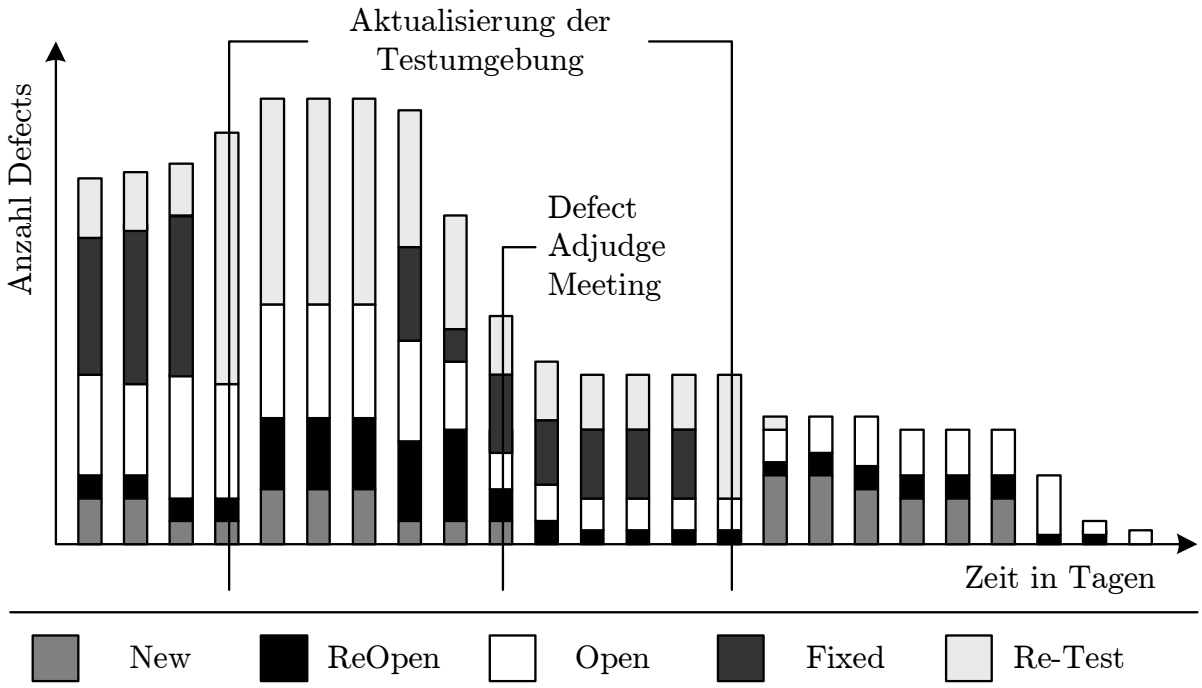

Abbildung 5.9.: Releasegefährdende Fehler.

mit verschiedenen internen Stellen (der Qualitätssicherung, den Entwicklern, etc.) und auch der Kunden. Dadurch ist eine zeitnahe, korrekte und endgültige Einordnung nicht möglich. Des Weiteren wird dadurch implizit die in Abschnitt 4.6.1 beschriebene Gewichtung von Fehlern nach den von ihnen betroffenen Features und Anwendungen realisiert.

Eine Untersuchung des dargestellten Schaubilds liefert mehrere Ergebnisse, die Verbesserungsmöglichkeiten aufdecken. Zunächst ist ein starker Rückgang der absoluten Anzahl an Fehlern kurz vor und am Tag des *Defect Adjudge Meetings* (siehe S. 143) zu erkennen. Dies ist damit zu begründen, dass einerseits im Vorfeld der Besprechung die Tester und die Entwickler verstärkt an den Fehlern gearbeitet haben, um sich entsprechend vorzubereiten. Dabei wurden einige Fehler auf *Fixed* gesetzt, einige nach einem erfolgreichen Re-Test geschlossen. Letzteres ergibt sich jedoch nicht aus diesem Schaubild. Andererseits werden in den *Defect Adjudge Meetings* die Defects gemeinsam analysiert und deren Auswirkung auf das Release bewertet. Dabei wird das Attribut *Delivery Impact* oftmals auf *Release Note*, *Workaround* oder *Accepted* gesetzt, wodurch der Fehler in dieser Auswertung nicht länger auftaucht.

Des Weiteren sind zwei Stellen zu erkennen, an denen alle auf *Fixed* stehenden Fehler

den Status *Re-Test* erhalten. An diesen beiden Tagen wurde die Testumgebung mit einer neuen Baseline aktualisiert, wodurch bereits von den Entwicklern beseitigte Fehler im Code für einen Re-Test bereitgestellt wurden.

Auffällig ist, dass fast immer ein großer Anteil der Fehler den Status *Re-Test* haben. Durch eine Erhöhung der Testressourcen während der Testphase kann somit die Planungssicherheit verbessert werden bzw. bereits früher konkrete Aussagen über den aktuellen Stand gemacht werden, da die Re-Tests dadurch in vielen Fällen zeitnaher ausgeführt werden können.

Valid Defects Reason

Ein weiteres Schaubild stellt die Fehlerquellen aller nicht zurückgewiesener, reproduzierbarer Fehler dar. Dieses beruht auf der Auswertung des Fehlerattributs *Reason* (vgl. Abschnitt 5.2.1). Beim Anlegen eines Fehlers wird dieses Attribut in der Regel zunächst auf *Unknown* gesetzt, da sich erst nach einer weiteren Analyse eine Einschätzung geben lässt, wo der Fehler lag. Mit endgültiger Sicherheit kann man dies erst nach der Fehlerbehebung und einem erfolgreichen Re-Test, also beim Schließen des Defects, sagen.

Diese Darstellung ermöglicht, Rückschlüsse auf die Qualität des Produkts und die Qualität der Tests zu ziehen, da festgestellt wird, ob die gefundenen Fehler im eigenen Code, in Fremdprodukten, in fehlerhaften Tests, etc. stecken. Dadurch ergeben sich Erkenntnisgewinne für zukünftige Entwicklungen. Eine Auflistung derzeit bei PAI eingesetzter Attributsausprägungen befindet sich in Tabelle 5.1. Eine mögliche Erweiterung dieser Liste ist denkbar, dem entgegen stehen jedoch zwei Hindernisse. Zum einen ist eine einmal eingeführte und verwendete Fehlerquelle nur schwer wieder zu beseitigen, da dann alle Fehler, deren Quelle diese Attributsausprägung zugeordnet haben, manuell geändert werden müssten. Zum anderen wird die Pflegbarkeit dieses Attributs durch eine zu lange Liste möglicher Fehlerquellen eingeschränkt.

Die Datenqualität beruht hier komplett auf der Güte der manuellen Pflege des Fehlerattributs. Diese ist relativ hoch, da beim Setzen des Fehlerstatus auf *Fixed* auch dieses Feld mit ausgefüllt werden kann und beim Schließen des Defects der Wert durch den Defect

Owner erneut durchgesehen wird. Ersteres basiert darauf, dass der Defect Agent nach dem Beheben des Fehlers weiß, was er geändert hat.

Die Auswertung dieser Kennzahl ist, wie auf S. 124 beschrieben, bei Plattformen von besonderer Relevanz, da Fehler in Testapplikationen differenziert zu betrachten sind.

Active Defects Delivery Status

Die ausschließliche Darstellung der releasegefährdenden Defects im o. g. Schaubild birgt die Gefahr, mehr Fehler zu akzeptieren und „auf die lange Bank zu schieben“ als für die mittel- bis langfristige Plattformentwicklung gut ist. Aus diesem Grund ist die Darstellung aller noch nicht geschlossener Fehler (*Active Defects*) eingeteilt nach ihren *Delivery Status* als Ergänzung zu den releasegefährdenden Fehlern wichtig. Die Darstellung der tagesaktuellen Verteilung der *Delivery Status* zeigt auf, wieviele der Fehler auf eines der nächsten Releases verschoben wurden.

Zusätzlich gibt es auch hier die Möglichkeit durch einen Klick ins Schaubild den zeitlichen Verlauf darzustellen und dadurch tiefere Einblicke in den Umgang mit gefundenen Fehlern zu erhalten.

Das Schaubild ist analog zum zeitlichen Verlauf der releasegefährdenden Fehler aufgebaut als Balkendiagramm. Jeder Balken stellt einen Tag dar und ist unterteilt nach dem zu betrachtenden Attribut, in diesem Falle dem *Delivery Impact*. Die technische Umsetzung und die Datenqualität sind ebenso mit denen der Verlaufsdarstellung der releasegefährdenden Fehler vergleichbar. Die Datenbasis liefert dasselbe SQL-Statement wie die tagesaktuelle Variante und die Datenqualität hängt von denselben Attributen ab wie die Betrachtung der releasegefährdenden Fehler.

Die analoge Betrachtung des Schaubildes derselben Plattform wie oben zeigt zunächst im Gegensatz zu obigem Ergebnis nur eine geringe Reduktion der Gesamtfehlerzahl. Lediglich am Tag des *Defect Adjudge Meetings* und nach der zweiten Aktualisierung sind kleine Sprünge zu betrachten. Dies bedeutet, dass nur ein geringer Teil der Fehler geschlossen bzw. beseitigt wurden. Die meisten wurden für dieses Release als nicht entscheidend eingestuft und auf *Accepted* gesetzt.

Active Defects Origin

Im fünften Schaubild (siehe Abb. 5.10) der Release Page wird die Herkunftsversion der für das bestimmte Release aktiven Fehler dargestellt. Dies zeigt, wie lange manche Fehler bereits „mitgeschleppt“ werden. Des Weiteren verdeutlicht ein Vergleich der Schaubilder mehrerer Releases zu festgelegten Zeitpunkten, am besten direkt vor dem Testbeginn, ob die absolute Anzahl an „Fehler-Altlasten“ und die Streuung der Fehlerherkunft zu- oder abgenommen hat.

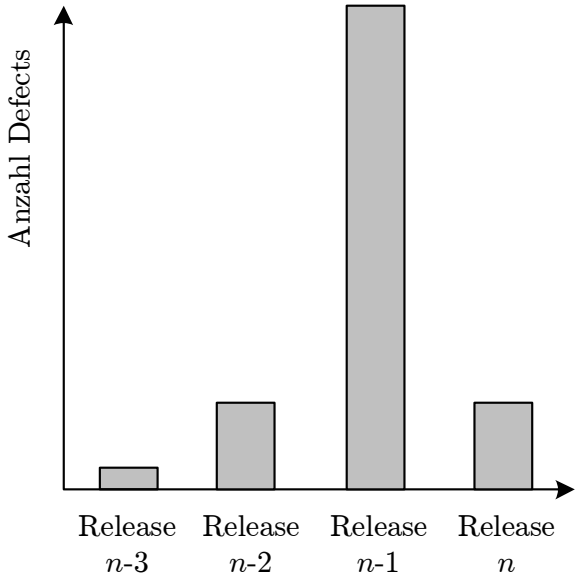

Abbildung 5.10.: Herkunftsversionen aktiver Fehler.

Auch hier hängt die Datenqualität an der manuellen Pflege des entsprechenden Fehlerattributs. Da beim Anlegen des Fehlers im Quality Center jedoch stets bekannt ist, in welcher Version der Fehler auftauchte bzw. erstmals entdeckt wurde, kann eine fehlerhafte Eintragung hier nur aufgrund eines Versehens des Testers geschehen. Ein nicht ausgefülltes Feld ist an dieser Stelle nicht möglich, da es sich um ein Pflichtfeld handelt und das Quality Center hierbei auf der Eintragung eines Wertes besteht.

Auch diese Auswertung ist bei Plattformen durch die häufig variierenden Releaseumfänge von hoher Wichtigkeit (vgl. S. 124).

5.3.2. Backlog Management Index (BMI)

Wie bereits erwähnt, birgt die Möglichkeit, die Releaseumfänge flexibel zu gestalten, die Gefahr des „Vergessens“ von auf das nächste Release verschobenen Defects und damit den Aufbau eines immer größer werdenden Fehlerberges. Da die Fehler oftmals nicht nur für ein Release gelten, sondern an das jeweils nächste vererbt werden können, ist eine releaseübergreifende Darstellung der Veränderung der Gesamtzahl an Fehlern über einen längeren Zeitraum hinweg sinnvoll. Der sog. *Backlog Management Index* bietet hierzu eine gute Möglichkeit. Bei der Plattformentwicklung ist, wie bereits erläutert, hierauf ein besonderes Augenmerk zu richten.

In seiner ursprünglichen Form ist der Index der Quotient aus der Anzahl geschlossener und geöffneter Fehler pro Zeiteinheit, oft eine Woche. Aus Normierungsgründen wird diese Zahl meist noch mit 100 multipliziert:

$$
\begin{aligned}
B_{Idx} &= 100 \cdot \frac{d_{closed}}{d_{open}} \\
\text{mit } d_{closed},\ d_{open} &: \text{Anzahl geschlossener bzw.} \\
&\quad \text{geöffneter Defects pro Zeiteinheit}
\end{aligned}
$$

Ist der Index gleich 100, so wurden gleich viele Fehler entdeckt wie geschlossen. Er ist größer als 100, wenn mehr geschlossen als geöffnet wurden und entsprechend kleiner, falls das Gegenteil eintritt.

Bei der PAI-Entwicklung werden durch eine Aktualisierung der Testumgebung oftmals viele Fehler am selben Tag nach einem bestandenen Re-Test geschlossen, wodurch hier eine zweiwöchige Darstellung von Vorteil ist. Es gibt hierbei dennoch einige Situationen, in denen der Index Ausreißer nach oben hat, da es vorkommen kann, dass nach Abschluss einer Testphase zunächst keine weiteren Tests durchgeführt, sondern spezifiziert werden. Dadurch werden auch keine weiteren Fehler entdeckt. Um eine gut lesbare Darstellung des Index zu erhalten, wird bei PAI der zur Basis 10 logarithmierte Index betrachtet. Die

Werte sind dadurch nicht mit 100, sondern mit $\log_{10} 100 = 2$ zu vergleichen.

Da die Darstellung zweiwöchentlich erfolgt, wird die Datenqualität an dieser Stelle auch durch eine nicht zeitnahe Pflege der Fehlerattribute nicht beeinträchtigt. Dennoch kann es vorkommen, dass $d_{open} = 0$ ist, wodurch eine Division durch 0 erfolgen würde. In diesem Fall wird weiter unterschieden, ob auch $d_{closed} = 0$ ist („$\frac{0}{0}$“). Falls ja, wird der BMI auf $2 := \log_{10} 100$ gesetzt, also von Konstanz ausgegangen. Falls $d_{closed} \neq 0$ wird $d_{open} := 0,5$ definiert.

Um den verschiedenen Einstufungen der Schwere von Fehlern gerecht zu werden und dies auch im BMI widerzuspiegeln, werden Fehler verschiedener Schwere folgendermaßen unterschiedlich gewichtet:

- *Show Stopper*: 2, 5-fach
- *High*: 2-fach
- *Medium*: 1, 5-fach
- *Low*: einfach

Die Umsetzung in der Praxis konnte ebenfalls mit dem Mercury Dashboard in einem SQL-Statement erfolgen, wobei das Werkzeug nicht in der Lage ist, historische Werte zu analysieren (vgl. Abschnitt 5.2.2). Die Darstellung in Abb. 5.11 wurde deshalb anhand Microsoft Excel manuell erstellt. Ein weiterer Nachteil des Mercury Dashboard ist, dass bei der Kombination verschiedener SQL-Statements (was nur durch Einbinden verschiedener beliebiger KPIs in einen JavaScript KPI möglich ist) jeweils dieselbe Grundmenge verwandt werden muss. Aus diesem Grund wurde die Möglichkeit bevorzugt, den Backlog Management Index in einem einzigen, dafür etwas größeren SQL-Statement umzusetzen:

```
SELECT (
    CASE WHEN (opened <> 0 AND closed <> 0)
            THEN ROUND(LOG(100*closed/opened, 10), 2)
        WHEN (closed <> 0 AND opened = 0)
            THEN ROUND(LOG(100*closed/0.5, 10), 2)
        WHEN (closed = 0 AND opened <> 0)
            THEN ROUND(LOG(100*0.5/opened, 10), 2)*100
```

```
        ELSE 2
    END
    ) as value
FROM
    (SELECT SUM(closed_sev) as closed
    FROM
        (SELECT (
            CASE Defects.SEVERITY
                WHEN 'Show Stopper' THEN 2.5*count (*)
                WHEN 'High' THEN 2*count (*)
                WHEN 'Medium' THEN 1.5*count (*)
                WHEN 'Low' THEN count (*)
           END
            ) as closed_sev
        FROM Defects D, Log L, Properties P
        WHERE L.ENTITY_TYPE = 'BUG' and
            L.ACTION_ID = P.ACTION_ID and
            P.FIELD_NAME = 'BG_STATUS'  and
            P.OLD_VALUE <> 'Closed' and
            P.NEW_VALUE = 'Closed' and
            L.TIME >= date_today-14 and
            L.TIME < date_tomorrow and
            L.ENTITY_ID = D.BUG_ID
        GROUP BY Defects.SEVERITY
        )
    ) A,
    (SELECT SUM(opened_sev) as opened
    FROM
        (SELECT (CASE Defects.SEVERITY
                WHEN 'Show Stopper' THEN 2.5*count (*)
                WHEN 'High' THEN 2*count (*)
```

```
            WHEN 'Medium' THEN 1.5*count (*)
            WHEN 'Low' THEN count (*)
        END
        ) as opened_sev
    FROM Defects
    WHERE Defects.DETECTION_DATE >= date_today-14 and
        Defects.DETECTION_DATE < date_tomorrow
    GROUP BY Defects.SEVERITY
    )
) B
```

Der `WHERE`-Ausdruck im Sub-Statement `A` lässt sich nur durch den Zugriff auf die interne Struktur des Quality Center, auf Log- und Properties-Tabellen, realisieren. Dies ist notwendig, da vom jeweiligen Ausführungstag an gerechnet immer nur die Aktionen der letzten zwei Wochen betrachtet werden sollen (`L.TIME >= date_today-14 and L.TIME < date_tomorrow`).

Anhand der Darstellung des BMI für ca. ein halbes Jahr (siehe Abb. 5.11) lassen sich einige Erkenntnisse gewinnen, die in dieser Form aufgrund der feineren Granularität in anderen Schaubildern nicht erkenntlich sind. Zunächst kann beobachtet werden, dass v. a. am Anfang und in der Mitte der beobachteten Periode meist direkt nach der Aktualisierung der Testumgebung mehr Fehler gefunden werden. Dies ist insofern erstaunlich, als in den beiden bereits dargestellten Verlaufskennzahlen ein gegenteiliger Effekt zu beobachten ist – dort jedoch in einem viel kürzeren Zeitraum.

Ebenfalls zu erkennen ist, dass das Release in KW 37 relativ früh ausgeliefert wurde, da die darin gefundenen Fehler zum damaligen Zeitpunkt noch nicht behoben wurden. Aufgrund der Möglichkeit der flexiblen Releaseumfänge muss dies jedoch nicht zu früh gewesen sein. Es müssen aber, um dem Anhäufen von unüberschaubar vielen Fehlern entgegenzuwirken, stets auch Phasen eingeplant werden, in denen Fehler aktiv analysiert und auch wieder geschlossen werden können. Ab ca. KW 1 wurde dies umgesetzt und bis zur Auslieferung des Folgereleases wurden, wie zu erwarten, in einigen Phasen mehr Fehler geschlossen als geöffnet.

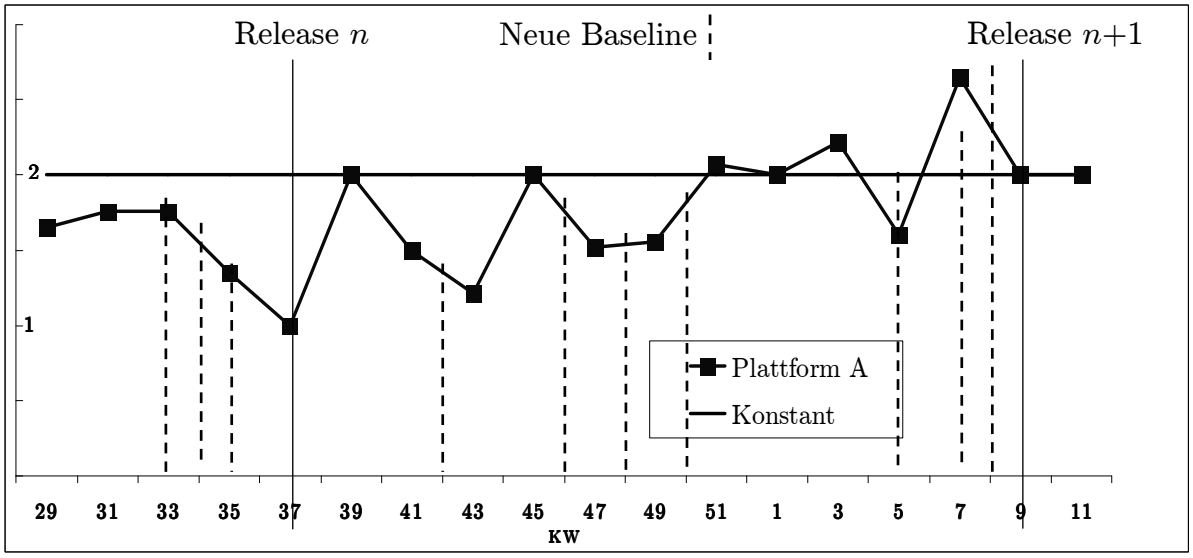

Abbildung 5.11.: Backlog Management Index.

Wie bereits erwähnt, kann der Backlog Management Index lediglich einen Trend angeben. Beispielsweise kann anhand des Index nicht ohne Weiteres festgestellt werden, ob in einer Zeitspanne von KW 45 bis KW 49 genauso viele Fehler gefunden wurden, wie in der Zeitspanne von KW 4 bis KW 9 anschließend geschlossen wurden (siehe Abb. 5.11).

5.3.3. Backlog Summe

Die Backlog Summe betrachtet die Differenz aller bis zu einem bestimmten Zeitpunkt i geöffneten Fehler $d_{open,i}$ und aller bis i geschlossenen Fehler $d_{closed,i}$. Im Anwendungsbeispiel sind $d_{open,i}$ alle Fehler, die zum Zeitpunkt i weder den Status *Closed* noch den Status *NRep* haben. Unter $d_{closed,i}$ fallen alle Fehler, die zum Zeitpunkt i entweder den Status *Closed* oder den Status *NRep* haben.

Die verschiedenen Fehlerschweren fließen hierbei nicht in die Betrachtung mit ein. Eine Gewichtung verschiedener Fehlerklassen analog zum BMI würde die absolute Interpretationsfähigkeit der Zahlenwerte verhindern. Die Fokussierung auf die Veränderungsgeschwindigkeit, wie sie der BMI bietet, tritt bei dieser Darstellung in Anbetracht der absoluten Fehlerzahlen in den Hintergrund.

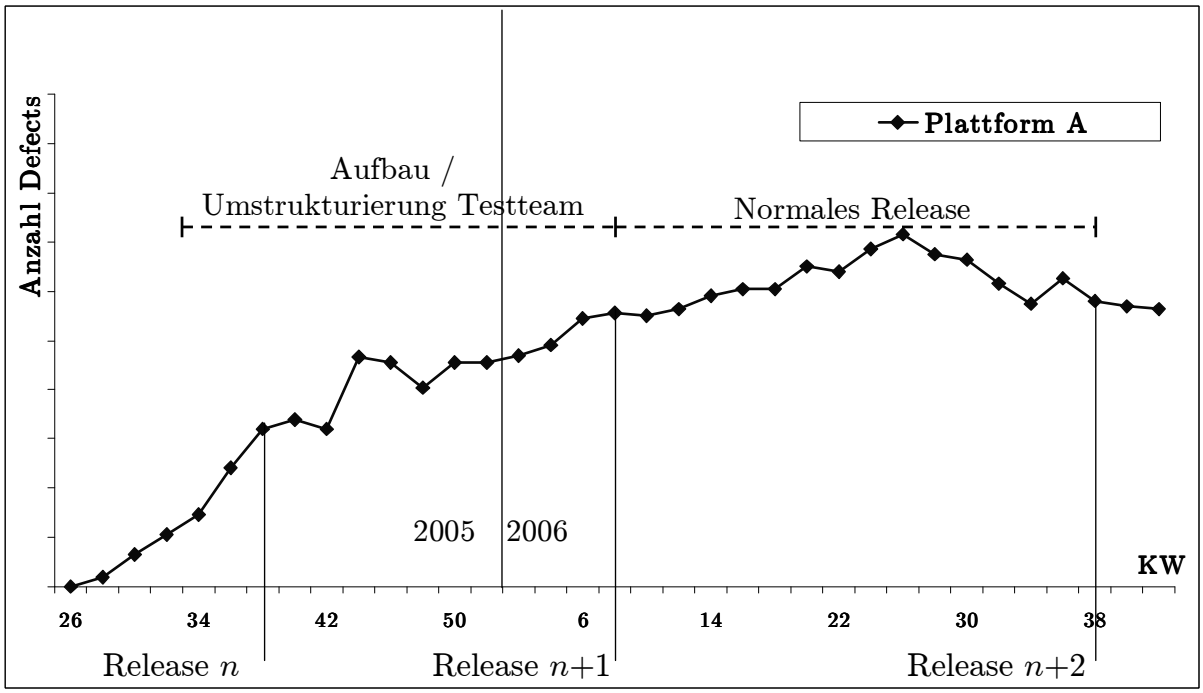

Abbildung 5.12.: Backlog Summe (Anzahl geöffneter Fehler).

In Abb. 5.12 wird die Backlog Summe über zwei Releases und damit auch zwei Testzeiträume betrachtet. Beim ersten Release ist auffällig, dass am Ende deutlich mehr Fehler offen waren als am Anfang. Die Auslieferung der Software enthielt somit eine ganze Reihe an (bekannten) Fehlern. Am Ende des zweiten Releases waren in etwa gleich viele Fehler enthalten wie zu Beginn der Testphase. Dies lässt auf einen normalen Releaseablauf schließen.

Bei den anschließenden Überlegungen, welche Gründe für den starken Anstieg an Fehlern während des ersten dargestellten Releases zu finden sind, denkt man zunächst an ein nicht ausreichend großes Entwicklungsteam, das in der kurzen Zeit bis zur Auslieferung der Plattform der Fehlerbearbeitung nicht gewachsen war. Auch eine enge Planung bzgl. des zweiten Releases und die daraus resultierende parallele Entwicklung des zweiten und Test des ersten Releases kann ein möglicher Grund sein. Eine dritte Möglichkeit könnte ein unterschiedlich langer Testzeitraum sein. Bei genauerer Betrachtung stellt man keine dieser Gegebenheiten fest – das Entwicklungsteam ist bei beiden Releases gleich strukturiert, besteht aus denselben Personen und hat eine vergleichbar hohe Arbeitslast. Der Testzeitraum war bei beiden Releases ebenfalls gleich lang. Während des ersten Releases

hat jedoch eine Umstrukturierung des Testteams stattgefunden. Dadurch konnten die Tester zum einen nicht, wie im zweiten Release, auf die Behebung der von ihnen gefundenen Fehler achten. Zum anderen entstanden vermehrt Fehler in Testspezifikationen und Testskripten. Im zweiten dargestellten Release war die Umstrukturierung vollzogen und der Testbetrieb konnte wieder normal ablaufen.

Diese fast paradoxe Auswirkung, dass ein nicht eingespieltes Testteam zu einem Anhäufen von Fehlern führen kann, ist hierbei als erstes Ergebnis festzuhalten. Aus der dargestellten Grafik lässt sich jedoch noch ein weiterer Punkt herauslesen: Aufgrund des normal abgelaufenen zweiten Releases kann festgestellt werden, dass ohne einen erhöhten Aufwand die im ersten Release angehäuften Fehler nicht zu reduzieren sind. Dies wird deutlich, da die Anzahl der enthaltenen Fehler zu Beginn des zweiten Testzeitraums in etwa gleich hoch ist wie an dessen Ende und somit auch zu Beginn eines dritten Releases. Es ist somit ratsam, bei einem der nächsten Releases einen Zeitraum zur Behebung der angehäuften Fehler einzuplanen.

5.4. Zusammenfassung

In diesem Abschnitt wurde die praktische Umsetzung an einem spezifischen Projekt vorgestellt. Auch auf einige Ergebnisse des Ansatzes konnte bereits eingegangen werden.

Zunächst wurde der Rahmen dargestellt und die verschiedenen Interessengruppen, die sich aus dem Entwicklungsteam, den Benutzern und der Organisation, in der die Plattform hergestellt wird, gegeneinander abgegrenzt. Auch auf den technischen Rahmen der Umsetzung der Qualitätssicherung wurde eingegangen. Dabei standen die Systemtests, die anhand von Testapplikationen durchgeführt werden, und der Umgang mit den dabei gefundenen Fehlern im Mittelpunkt. Es wurden die entsprechend angepassten und zum Teil eigens für diese Plattform entwickelten Fehlerattribute, beispielsweise der *Delivery Impact*, erläutert und der Fehlerlebenszyklus vorgestellt. Das bei Plattformen sinnvolle Konzept des featurebasierten Testens, das auf dem bereits vorgestellten featurebasierten Plattformkonzept aufbaut, wurde ebenso erläutert. Abschließend wurde auf den Testprozess anhand des Aufbaus des Mercury Quality Center eingegangen.

Im Rahmen der Vorstellung der Implementierung der *Release Page* und der *Backlog*-Darstellungen konnten auch bereits erste Ergebnisse vorgestellt werden. Die Release Page als Ganzes gesehen ermöglicht eine schnelle Überprüfung des Release Status. Hierbei steht zunächst der Test Status im Mittelpunkt, dessen explizite Darstellung in einem Schaubild beispielsweise erkennen ließ, dass auch nach Testbeginn noch einige Tests geplant wurden. Daraus konnte geschlossen werden, dass entweder die Testspezifikationsphase zu kurz war oder dass zu diesem Zeitpunkt noch für die Spezifikation der Tests benötigte Informationen gefehlt haben. Auch die Darstellungen der releasegefährdenden Fehler und deren Zeitverlauf, die Auswertung der Herkunftsversionen gefundener Fehler und die *Backlog*-Darstellungen zeigten einige Verbesserungspunkte der Qualitätssicherung auf.

Die in den Schaubildern dargestellten Kennzahlen waren an die Entwicklung der PAI Plattformen angepasst, lassen sich jedoch auch verallgemeinern. Bei der Anwendung bei anderen Plattformen sind lediglich leichte Anpassungen, etwa an der Granularität des BMI nötig, der generelle Aufbau der Kennzahlsuite ist übertragbar.

6. Fazit

In diesem Kapitel sollen eine Zusammenfassung und Hinweise auf weitere Forschungsfelder und Ausbaumöglichkeiten des bestehenden Testmetriksystems gegeben werden.

6.1. Zusammenfassung

Der Kernpunkt der vorliegenden Arbeit ist die Definition eines Prozesses, um ein Testmetriksystem zu erstellen. Dieses Testmetriksystem soll die Auslieferentscheidung und die Bewertung der Qualität während des Testprozesses erleichtern. Damit wird ein speziell auf Softwareplattformen angepasstes Verfahren etabliert, um den Besonderheiten dieser Softwarekategorie gerecht zu werden. Im Folgenden werden die Erkenntnisse der Arbeit anhand des Testprozesses aufgeführt.

Zunächst war die Anpassung des Rahmens der Entwicklung und des Tests notwendig. So wurde in Zusammenarbeit mit R. Brcina ein Plattform-Feature-Modell erweitert, um featurebasiertes Testen zu ermöglichen. Dadurch konnte die Rückverfolgbarkeit von im Systemtest gefundenen Fehlern auf entsprechende Features sichergestellt werden.

Die bereits in der Einleitung thematisierte flexible Handhabung des Lieferumfangs und die Abwägung, ob die Qualität zu Gunsten der Einhaltung des Liefertermins herabgesetzt werden kann bzw. soll, sollte ebenfalls durch diese Arbeit fundiert behandelt werden. Eine Basis hierfür lieferte die Erweiterung der Qualitätsattribute der ISO 9126 bzw. der ISO 25010 um Attribute wie beispielsweise die Planbarkeit. Diese Anpassungen wurden konsequent weitergeführt und resultierten auch in der Anpassung der Fehlerattribute. Durch die Einführung der Attribute *Next Delivery* und *Delivery Impact* wurde die Grundlage geschaffen, um den flexiblen Auslieferumfang zu visualisieren.

Eine Besonderheit beim Systemtest von Softwareplattformen lag in der Behandlung der NFRs und der daraus resultierenden Einteilung von Last- und Stabilitätstests in „bestanden“ und „nicht bestanden“. Um eine genauere Analyse im Einzelfall zu ermöglichen, wurde der Teststatus *Vague* eingeführt. Dadurch konnte anschließend auch in den Testmetriken die Realität besser widergespiegelt werden.

Einen zentralen Teil des Auswahlprozesses der Testmetriken stellte die Priorisierung der Qualitätsattribute dar. Durch diese Herangehensweise konnte bedarfsorientiert vorgegangen werden, da alle Kennzahlen letztlich eingesetzt werden, um ein oder mehrere Qualitätsattribute zu bewerten. Hierzu wurde die allgemeine Studie über Qualitätsattribute bei Softwareplattformen von Johansson [JWBH01] als Basis verwendet. Bei jeder Anwendung sollten die Daten der Studie mit Daten aus dem jeweiligen Plattformentwicklungsprojekt zusammengeführt werden. Ist dies aufgrund fehlender Daten aus dem speziellen Projekt nicht möglich, so kann durch dieses Vorgehen dennoch die allgemeine Anpassung an Plattformen basierend auf der Studie verwendet werden. Die Priorisierung der Qualitätsattribute fand weiterhin eingeteilt nach Interessengruppen statt. Die Interessengruppen Benutzer, Organisation und Entwicklungsmannschaft bildeten alle Anspruchsberechtigten einer Softwareplattform. Dadurch wurde die Bedarfsorientierung des Ansatzes unterstrichen, da sich die Priorisierung für alle Interessengruppen unterscheidet.

Der nächste Schritt im Prozess sah die Erstellung und Anpassung von Kennzahlen vor. So konnte etwa durch die Parametrisierung von Kennzahlen mit der Anzahl und der Relevanz von Kundenprojekten eine besser an die Realität angepasste Darstellung von gefundenen Fehlern erstellt werden. Auch die Anpassung der Qualitätsattribute ermöglichte eine detailliertere Darstellung verschiedener Aspekte in Kennzahlen durch die Integration der „neuen“ Attribute in die Testmetriken. Auch hier ist eine Betrachtung eingeteilt in zuvor bereits verwendete Interessengruppen möglich. Bei der Auswahl der Kennzahlen wurde jedoch bewusst darauf verzichtet, da es unerheblich sein sollte, von wem die Bewertung eines bestimmten Attributs gewünscht wurde.

Der erarbeitete Prozess wurde anschließend in einem Fallbeispiel realisiert. Den Kernpunkt bildete dabei die *Release Page*. Sie besteht aus einer Übersichtsseite, auf der fünf Kennzahlen in Schaubildern dargestellt werden. Dargestellt wurde auch die exemplarische

Implementierung im Mercury Dashboard, einem Werkzeug zur Anwendung von Kennzahlen im Umfeld der Softwarequalitätssicherung, das Teil eines der am weitesten verbreiteten Softwaresysteme zur Qualitätssicherung ist. Der *Backlog Management Index* und die *Backlog Summe* rundeten die Kennzahldarstellungen, ebenfalls inklusive ihrer Implementierungen im Mercury Dashboard, ab.

Insgesamt betrachtet zeigte die Fallstudie, dass das hier vorgestellte Vorgehen eine Möglichkeit ist, ein pragmatisches, umsetzbares und auch zielgerichtetes Kennzahlsystem zur Unterstützung des Qualitätsmanagements von Softwareplattformen aufzubauen. Die Planbarkeit konnte z. B. durch die explizite Darstellung der releasegefährdenden Fehler deutlich verbessert werden. Die Zuverlässigkeit der Plattform kann durch ein System wie das vorliegende zwar nicht beeinflusst werden, doch die Eingruppierung und Darstellung der Zuverlässigkeitstests bzw. deren Ergebnisse verdeutlichen die Bemühungen, die zu ihrer Überprüfung betrieben werden. Auf Basis der erstellten Kennzahlsuite lässt sich die Freigabeentscheidung nicht nur durch quantitative Werte untermauern. Auch durch die Darstellung der fünf Schaubilder auf einer Seite kann ein umfassender Überblick in Form eines Managementcockpits über den qualitativen Zustand der Plattform auch während des Testprozesses gewonnen werden.

Das entstandene Vorgehensmodell kann in das in der Praxis weit verbreitete ami-Modell eingebettet werden, weicht an entscheidenden Punkten jedoch vom im ami-Modell vorgeschlagenen Standard-Vorgehen ab. Durch die Verwendung der für Plattformen angepassten und priorisierten Qualitätsattribute wird das ami-Modell ebenfalls für den Einsatz in der Plattformentwicklung angepasst. Generell kann das beschriebene Vorgehen auch für andere Arten von Software verwendet werden. Hierzu werden jedoch andere Eingangsdaten (aus der Studie) und entsprechend angepasste Rahmenbedingungen, wie beispielsweise die veränderten Qualitätsattribute, benötigt.

6.2. Ausblick

Mögliche zukünftige Arbeiten lassen sich grob in zwei verschiedene Bereiche einteilen: Operative Erweiterungen des bestehenden Modells und der im Praxisbeispiel vorgestellten

Umsetzung und weitere Forschungen im Umfeld des Qualitätsmanagements von Softwareplattformen.

Erweiterungen des bestehenden Systems können auf vielfältige Art und Weise erstellt werden. Zunächst können zur Untersuchung der Zuverlässigkeit absolute Zahlen aus Tests verwendet werden. Ein Zuverlässigkeitsindex könnte möglicherweise erstellt werden und die Zuverlässigkeit quantifizieren. Ein erster Schritt hierbei wäre, wenige Zuverlässigkeitskennzahlen (z. B. $\frac{\#Test-Stunden}{\#gefundeneFehler}$) verschiedener Releases miteinander zu vergleichen, um dadurch die geeignetsten Kennzahlen zu identifizieren. Auch bzgl. anderer Aspekte kann die gezielte Gegenüberstellung verschiedener Releases erfolgen und aufzeigen, ob die Plattform besser oder schlechter geworden ist. An dieser Stelle steht die operative Umsetzung im Vordergrund, obgleich die wissenschaftliche Literatur hierfür bislang nur sehr elementare Kennzahlen liefert [Cop04; MIO87; EDBS04].

Bei der Erweiterung um indirekte Kennzahlen, beispielsweise in Form eines Zuverlässigkeitsindex, ist eine gezielte Validierung der Kennzahlen nötig. Diese war bisher noch nicht zwingend erforderlich, da es sich bei den hier umgesetzten Kennzahlen ausschließlich um direkte Kennzahlen handelt. Bei einer Erweiterung, die indirekte Kennzahlen umfasst, ist dieser Schritt jedoch unerlässlich. In der Literatur gibt es an dieser Stelle bereits einige Ansätze [KPF95].

Die bisher im Fokus des Modells liegende Datenquelle ist der Systemtest. Diese kann durch weitere ergänzt werden, beispielsweise durch die Ergebnisse von Dokumentenreviews oder Tests auf anderen Ebenen, beispielsweise Unit-Tests. Die Standardliteratur liefert hierzu bereits eine breite Basis. [Fen91; EDBS04; Kan03; Mel95]

Weiterer Forschungsbedarf kann etwa auf dem Gebiet der Performance-Vorhersageverfahren für Plattformen identifiziert werden. Dadurch ließen sich die Bewertungen von Stabilitätstests direkt erleichtern und präzisierten die Aussagen der Kennzahlen. [FKB+05] Entsprechend valide Vorhersagen ließen sich evtl. in den erarbeiteten Prozess einbauen und erleichterten dadurch den Umgang mit Stabilitätstests, die im Praxisbeispiel den Zustand „Vague“ haben.

In den letzten Jahren etabliert sich immer mehr der Gedanke, Software als Service inklusive Betrieb zur Verfügung zu stellen. Diese Entwicklung macht auch vor Softwa-

replattformen nicht halt und mittlerweile gibt es in der Praxis einige Services, die den Betrieb einer Softwareplattform anbieten. Die Wissenschaft hat jedoch noch Aufholbedarf was den systematischen Test dieser Plattformen anbelangt. Auch muss geprüft werden, ob sich Kennzahlensysteme aus dem Bereich der Rechenzentren ohne Weiteres auf diese Betriebsszenarien übertragen lassen oder ob Anpassungen an dieser Stelle nötig sind.

A. Qualitätsattribute der ISO/IEC 25010 bzw. der ISO/IEC 9126

Die in Tabelle A.1 dargestellten Attribute und Sub-Attribute bilden die externen und internen Qualitätsattribute der ISO/IEC 9126 und bleiben auch im Nachfolgestandard ISO/UEC 25010 erhalten.

An Stelle des Begriffes „Zuverlässigkeit“ wird oft auch der Begriff „Ausfallsicherheit“ verwendet, der klarer zum Ausdruck bringt, was mit diesem Qualitätsattribut ausgedrückt werden soll.

Attribute	**Sub-Attribute**
Funktionalität	Angemessenheit Richtigkeit Interoperabilität Sicherheit Konformität
Zuverlässigkeit	Reife Fehlertoleranz Wiederherstellbarkeit Konformität
Benutzbarkeit	Verständlichkeit Erlernbarkeit Bedienbarkeit Attraktivität Konformität
Effizienz	Zeitverhalten Ressourcenverbrauch Konformität
Wartbarkeit	Analysierbarkeit Modifizierbarkeit Stabilität Testbarkeit Konformität
Portabilität	Anpassbarkeit Installierbarkeit Koexistenz Austauschbarkeit Konformität

Tabelle A.1.: Externe und interne Qualitätsattribute und deren Sub-Attribute aus der Norm ISO/IEC 25010 [ISO01b]

B. Skalentypen

Insgesamt werden fünf Skalentypen unterschieden, deren zulässige Operationen von einer zur nächsten erweitert werden. Im Folgenden werden die fünf Typen nach [Lig02; Fen91] beschrieben:

- Nominalskala: Die einzige zulässige Operation ist die Überprüfung auf Gleichheit oder Ungleichheit. Ein Beispiel sind Namen oder Bezeichnungen.
- Ordinalskala: Verschiedene Objekte können durch eine Ordinalskala geordnet werden bzw. ihnen kann eine Reihenfolge zugewiesen werden. Als Beispiel kann die Priorisierung von Aufgaben oder Ähnlichem dienen.
- Intervallskala: Zusätzlich zur Reihenfolge ist auch der Abstand festgelegt. Als Beispiel kann hier die Temperaturskala in $°C$ dienen ($20°C$ ist $30°C = 20°C - (-10°C)$ wärmer als $-10°C$).
- Ratioskala: Die Bildung von Verhältnissen verschiedener Merkmalswerte ist hier erlaubt. Dadurch können Aussagen wie „Auto A ist doppelt so lang wie Auto B“ gemacht werden. Es existiert ein absoluter Nullpunkt. Als Beispiele können der Blutdruck oder die Länge von einem Brett oder das Lebensalter herangezogen werden.
- Absolutskala: Absolutskalen stellen immer die einzige Möglichkeit dar, das entsprechende Objekt zu messen. Es handelt sich hierbei oftmals um Anzahlen (Anzahl der Einwohner einer Stadt oder der gefundenen Fehler in einem Softwaresystem) oder Wahrscheinlichkeiten.

Abkürzungsverzeichnis

ami	**A**pplication of **M**etrics in **I**ndustry
BMI	**B**acklog **M**anagement **I**ndex
CAFÉ	From **C**oncept to **A**pplication in System-**F**amily **E**ngineering
CBS	**C**omponent oder **C**OTS **B**ased **S**ystem
CMMI	**C**apability **M**aturitiy **M**odel **I**ntegration
COTS	**C**ommercial oder **C**omponent **O**ff **T**he **S**helf
EDV	**E**lektronische **D**aten**v**erarbeitung
ESAPS	**E**ngineering **S**oftware **A**rchitectures, **P**rocesse**s**
FAMILIES	**Fa**ct-based **M**aturity through **I**nstitutionalisation **L**essons-learned and **I**nvolved **E**xploration of **S**ystem-family engineering
FODA	**F**eature **O**riented **D**omain **A**nalysis
GQM	**G**oal **Q**uestion **M**etric
IT	**I**nformations**t**echnologie
ITEA	**I**nformation **T**echnology for **E**uropean **A**dvancement
KPI	**K**ey **P**erformance **I**ndicator
KW	**K**alender**w**oche
MDA	**M**odel **D**riven **A**rchitecture

OMG ..... **O**bject **M**anagement **G**roup

PC ..... **P**ersonal **C**omputer, Heimcomputer

PDA ..... **P**ersonal **D**igital **A**ssistant

PIM ..... **P**latform **I**ndependent **M**odel

PSM ..... **P**latform **S**pecific **M**odel

QS ..... **Q**ualitäts**s**icherung

SEI ..... **S**oftware **E**ngineering **I**nstitute der Carnegie Mellon University

SPICE ..... **S**oftware **P**rocess **I**mprovement and **C**apability D**e**termination

SQuaRE ..... **S**oftware product **Qua**lity **R**equirements and **E**valuation

TMAP ..... **T**est **M**anagement **Ap**proach

TPI ..... **T**est **P**rocess **I**mprovement

TQM ..... **T**otal **Q**uality **M**anagement

UML ..... **U**nified **M**odeling **L**anguage

Literaturverzeichnis

[ABMD04a] *Kapitel* Software Quality. In: Abran, Alain (Hrsg.) ; Bourque, Pierre (Hrsg.) ; Moore, James W. (Hrsg.) ; Dupuis, Robert (Hrsg.): *Guide to the Software Engineering Body of Knowledge SWEBOK*. IEEE Computer Society, 2004, S. 158–171 35

[ABMD04b] *Kapitel* Software Testing. In: Abran, Alain (Hrsg.) ; Bourque, Pierre (Hrsg.) ; Moore, James W. (Hrsg.) ; Dupuis, Robert (Hrsg.): *Guide to the Software Engineering Body of Knowledge SWEBOK*. IEEE Computer Society, 2004, S. 74–89 63

[AKCK05] Al-Kilidar, Hiyam ; Cox, Karl ; Kitchenham, Barbara: The Use and Usefulness of the ISO/IEC 9126 Quality Standard, 2005, S. 126–132. – 4th IEEE/ACM International Symposium on Empirical Software Engineering (ISESE 2005), Noosa, Australia 35, 39

[AQDH05] Abran, Alain ; Qutaish, Rafa A. ; Desharnais, Jean-Marc ; Habra, Naji: An Information Model for Software Quality Measurement with ISO Standards. In: *Proceedings of the International Conference on Software Development (SWDC-REK)*. Reykjavik, Iceland, Mai 2005, S. 104–116 37, 41

[BA99] Boehm, Barry ; Abts, Chris: COTS Integration: Plug and Pray? In: *Computer* 32 (1999), Januar, Nr. 1. `http://dx.doi.org/http://dx.doi.org/10.1109/2.738311`. – DOI http://dx.doi.org/10.1109/2.738311 18

[Bal98] Balzert, Helmut: *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Heidelberg, Berlin : Spektrum Akademischer Verlag, 1998 24, 33, 36, 62

[Bal00] Balzert, Helmut: *Lehrbuch der Software-Technik: Software-Entwicklung, 2.Auflage.* Heidelberg, Berlin : Spektrum Akademischer Verlag, 2000 15

[BB01] Basili, Victor R. ; Boehm, Barry: COTS-Based Systems Top 10 List. In: *Computer* 34 (2001), Mai, Nr. 5, S. 91–93 19

[BBL76] Boehm, Barry W. ; Brown, J. R. ; Lipow, M.: Quantitative evaluation of software quality. In: *ICSE '76: Proceedings of the 2nd international conference on Software engineering.* Los Alamitos, CA, USA : IEEE Computer Society Press, 1976, S. 592–605 40

[BCR94] Basili, Victor R. ; Caldiera, Gianluigi ; Rombach, H. D.: The goal question metric approach. In: *Encyclopedia of software engineering.* Jon Wiley & Sons, 1994 42, 106

[Bei90] Beizer, Boris: *Software Testing Techniques, second edition.* Van Nostrand Reinhold International, 1990 61, 62, 63, 66

[BG03] Bertolino, Antonia ; Gnesi, Stefania: PLUTO: A Test Methodology for Product Families. In: [Lin04], S. 181–197 21, 68

[BGBW05] Bernard, Tom ; Gallagher, Brian ; Bate, Roger ; Wilson, Hay: CMMI Aquisition Module (CMMI-AM), Version 1.1 / Carnegie Mellon University, Software Engineering Institute. Version: Mai 2005. `http://www.sei.cmu.edu/cmmi/models/`. Pittsburgh, PA, Mai 2005. – Forschungsbericht. – CMU/SEI-2005-TR-011, ESC-TR-2005-011 43

[Bin00] Binder, Robert V.: *Testing Object-Oriented Systems - Models, Patterns, and Tools.* Addison Wesley Longman, 2000 16, 17, 53, 62, 66, 79

[BKP04] Böckle, Günter ; Knauber, Peter ; Pohl, Klaus ; Schmid, Klaus (Hrsg.): *Software-Produktlinien.* Heidelberg : dpunkt.verlag, 2004 22, 68

[BM04] Bertolino, Antonia ; Marchetti, Eda: A Brief Essay on Software Testing / Instituto di Scienzy e Tecnologie „A Faedo“ Area della ricera CRD di Pisa. 2004. – Forschungsbericht 64

[BMB96] Briand, Lionel C. ; Morasca, Sandro ; Basili, Victor S.: Property-Based

Software Engineering Measurement. In: *IEEE Transactions On Software Engineering* 22 (1996), Januar, Nr. 1, S. 68–86 18

[BOS00] Brownsword, Lisa ; Oberndorf, Tricia ; Sledge, Carol A.: Developing New Processes for COTS-based Systems. In: *IEEE Softw.* 17 (2000), Juli / August, Nr. 4, S. 48–55. http://dx.doi.org/http://dx.doi.org/10.1109/52.854068. – DOI http://dx.doi.org/10.1109/52.854068 18

[BP06a] Beschorner, Dieter ; Peemöller, Volker H.: *Allgemeine Betriebswirtschaftslere*. Berlin : Verlag neue Wirtschaftsbriefe, Herne, 2006. – 2. Auflage 2

[BP06b] Brcina, Robert ; Prechtel, Markus: Feature-orientierte Plattformentwicklung und Verfolgbarkeit. In: *Softwaretechnik-Trends* 26 (2006), November, Nr. 4 25, 26, 27, 29, 90, 91, 205

[BPL05] Böckle, Günther ; Pohl, Klaus ; Linden, Frank van d.: *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin, Heidelberg : Springer-Verlag, 2005 138

[BR87] Basili, Victor R. ; Rombach, H. D.: Tailoring the software process to project goals and environments. In: *ICSE '87: Proceedings of the 9th international conference on Software Engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1987, S. 345–357 41

[Bro05] Brockhaus: *Computer und Informationstechnologie*. Bibliographisches Institut & F. A. Brockhaus AG, Mannheim, 2005 19

[Bro06] Brockhaus: *Der Brockhaus: in 15 Bänden. Permanent aktualisierte Online-Auflage*. Bibliographisches Institut & F. A. Brockhaus AG, Mannheim, Leipzig, 2006 32

[BS798a] *British Standard 7925: Software testing*. British Standards Institution (BSI), 1998. – Part 1: Vocabulary 8

[BS798b] *British Standard 7925: Software testing*. British Standards Institution (BSI), 1998. – Part 2: Software component testing 56, 58, 60, 69

[BSMM99] Bronstein, I.N. ; Semendjajev, K.A. ; Musiol, G. ; Mühlig, H.: *Taschenbuch der Mathematik, 4. überarbeitete Auflage*. Verlag Harri Deutsch, 1999 72

[Bun] Bundesamt für Sicherheit in der Informationstechnik: *Biometrie*. – `http://www.bsi.de/fachthem/biometrie/` , zuletzt aufgerufen im Januar 2008 72

[BV02] Bertoa, Manuel F. ; Vallecillo, Antonio: Quality attributes for cots components. In: *Proceedings of the 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2002)*, 2002 40

[CKS03] Chrissis, Mary B. ; Konrad, Mike ; Shrum, Sandy: *CMMI : Guidelines for Process Integration and Product Improvement*. Addison-Wesley Professional, 2003 43, 44, 122

[CMM06] Carnegie Mellon University, Software Engineering Institute: CMMI for Development, Version 1.2, Improving processes for better products. Version: August 2006. `http://www.sei.cmu.edu/cmmi/models/`. Pittsburgh, PA, August 2006. – Forschungsbericht. – CMU/SEI-2006-TR-008, ESC-TR-2006-008 43

[Cop04] Copeland, Lee: Quality Metrics For Testers: Evaluating Our Products – Evaluating Ourselves. Anaheim, CA, USA, November 2004. – STARWEST Conference 173

[CPL04] Cysneiros, Luiz M. ; Prado Leite, Julio Cesar S.: Nonfunctional Requirements: From Elicitation to Conceptual Models. In: *IEEE Transactions on Software Engineering* 30 (2004), Mai, Nr. 5, S. 328–350 31

[CPR04] Chen, Yanping ; Probert, Robert L. ; Robeson, Kyle: Effective test metrics for test strategy evolution. In: *CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, IBM Press, Oktober 2004, S. 111–123 75

[CT04] Chen, DeJiu ; Törngren, Martin: A metrics system for quantifying ope-

rational coupling in embedded computer control systems. In: *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*. New York, NY, USA : ACM, 2004. – ISBN 1–58113–860–1, S. 184–192 75

[DAS] *Homepage der DASMA, Deutschsprachige Anwendergruppe für Software-Metrik und Aufwandschätzung e.V.*. – `http://www.dasma.org/` , zuletzt aufgerufen im März 2008 121

[Dem82] Deming, W. E.: *Out of the crisis*. MIT Press, 1982 44

[Dij70] Dijkstra, Edsger W.: Notes on structured programming / Technological University Eindhoven, Department of Mathematics. 1970 (T.H.-Report 70-WSK-03). – Forschungsbericht. – 2nd edition 53

[DIN95a] *DIN 55350-11: Begriffe zu Qualitätsmanagement und Statistik. Teil 11: Begriffe des Qualitätsmanagements*. August 1995. – DIN Deutsches Institut für Normung e.V. 32

[DIN95b] *DIN EN ISO 8402: Qualitätsmanagement – Begriffe*. August 1995. – DIN Deutsches Institut für Normung e.V. 32, 54

[DJ03] Dasu, T. ; Johnson, T.: *Exploratory Data Mining and Data Cleaning*. New York, NY, USA : John Wiley & Sons, Inc., 2003 (Probability and statistics) 95

[DP03] Dahlstedt, Åsa G. ; Persson, Anne: *Requirements interdependencies – moulding the state of research into a research agenda*. Requirements Engineering Forum on Software Quality (REFSQ), Klagenfurt/Velden, Austria, Juni 2003. – S. 71–80 25

[Dud07] Duden: *Fremdwörterbuch : auf der Grundlage der neuen amtlichen Rechtschreibregeln*. 9. Mannheim : Dudenverlag, 2007. – ISBN 3–411–04059–9 71

[Dum03] Dumke, Reiner: *Software Engineering*. Vieweg Verlag, 2003 72

[ED96] Ebert, Christof ; Dumke, Reiner: *Software-Metriken in der Praxis*. Springer Verlag, 1996 71, 80, 82, 95

[EDBS04] Ebert, Christof ; Dumke, Reiner ; Bundschuh, Manfred ; Schmietendorf, Andreas: *Best Practices in Software Measurement*. Berlin, Heidelberg : Springer Verlag, 2004 173

[Ehr07] Ehrhardt, Christian: *Static code analysis in multi-threaded environments*, Universität Ulm, Fakultät für Mathematik und Wirtschaftswissenschaften, Diss., November 2007. http://vts.uni-ulm.de/query/longview.meta.asp?document_id=6082 98

[Els99] Elstrodt, Jürgen: *Maß- und Integrationstheorie*. Springer-Verlag, 1999 72

[Eng99] English, Larry P.: *Improving Data Warehouse and Business Information Quality*. John Wiley & Sons, Inc., 1999 95

[FAM] *FAMILIES – FAct-based Maturity through Institutionalisation Lessons-learned and Involved Exploration of System-family engineering, Projekt-Homepage*. – http://www.esi.es/en/Projects/Families/ , zuletzt aufgerufen am 16.05.2006 21

[Feh05] Fehlmann, Thomas M. ; Bischoff, Rainer (Hrsg.): *Six Sigma in der SW-Entwicklung*. Friedrich Vieweg & Sohn Verlag, 2005 79

[Fen91] Fenton, Norman E.: *Software Metrics - A rigorous approach*. London : Chapman & Hall, 1991 75, 76, 79, 121, 173, 177

[FKB+05] Firus, Viktoria ; Koziolek, Heiko ; Becker, Steffen ; Reussner, Ralf ; Hasselbring, Wilhelm: Empirische Bewertung von Performanz-Vorhersageverfahren für Software-Architekturen. In: Liggesmeyer, Peter (Hrsg.) ; Pohl, Klaus (Hrsg.) ; Goedicke, Michael (Hrsg.): *Software Engineering 2005 Proceedings - Fachtagung des GI-Fachbereichs Softwaretechnik* Bd. 64, Bonner Köllen Verlag, März 2005 (GI-Edition of Lecture Notes in Informatics), S. 55–66 173

[FN99] Fenton, Norman E. ; Neil, Martin: A Critique of Software Defect Prediction Models. In: *IEEE Transactions on Software Engineering* 25 (1999), September / Oktober, Nr. 5, S. 675–689 4

[FO00] Fenton, Norman E. ; Ohlsson, Niclas: Quantitative Analy-

sis of Faults and Failures in a Complex Software System. In: *IEEE Trans. Softw. Eng.* 26 (2000), August, Nr. 8, S. 797–814. http://dx.doi.org/http://dx.doi.org/10.1109/32.879815. – DOI http://dx.doi.org/10.1109/32.879815. – ISSN 0098–5589 74

[GB05] Grochtmann, Jürgen ; Baresel, André: Test verteilter Systeme – Stand der Technik und Wissenschaft / DaimlerChrysler AG. 2005. – interner Bericht 19, 135

[GC87] Grady, Robert B. ; Caswell, Deborah L.: *Software metrics: establishing a company-wide program.* Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1987 79

[GHK+01] Graves, Todd L. ; Harrold, Mary J. ; Kim, Jung-Min ; Porter, Adam ; Rothermel, Gregg: An empirical study of regression test selection techniques. In: *ACM Trans. Softw. Eng. Methodol.* 10 (2001), April, Nr. 2, S. 184–208. http://dx.doi.org/http://doi.acm.org/10.1145/367008.367020. – DOI http://doi.acm.org/10.1145/367008.367020 65

[GKMS00] Graves, Todd L. ; Karr, Alan F. ; Marron, J. S. ; Siy, Harvey: Predicting Fault Incidence Using Software Change History. In: *IEEE Trans. Softw. Eng.* 26 (2000), Nr. 7, S. 653–661. http://dx.doi.org/http://dx.doi.org/10.1109/32.859533. – DOI http://dx.doi.org/10.1109/32.859533. – ISSN 0098–5589 74

[Got95] Gotel, Orlena Cara Z.: *Contribution Structures for Requirements Traceability.* Bd. Ph.D. Thesis, University of London. 1995 29

[GPHG04] Grabert, Matthias ; Prechtel, Markus ; Hrycej, Tomas ; Günther, Winfried: An Early Warning System for Vehicle Related Quality Data. In: *Lecture Notes in Computer Science* 3275 (2004), November, S. 88– 95 205

[HN01] Hansen, H. R. ; Neumann, G.: *Wirtschaftsinformatik 1.* Stuttgart : 8. Auflage, Lucius & Lucius, 2001 15, 17, 18, 20

[Hor06] Horváth, Andreas: *Erstellung eines prozessbasierten Modells zur Messung*

von Integrationsaufwänden bei der Entwicklung von IT-Systemen in Großunternehmen, Hochschule Heilbronn, Diplomarbeit, 2006 73

[HP07] Horváth, Péter ; Partners: *Balanced Scorecard Umsetzen*. Schäffer-Poeschel, 2007 2

[IEE90] *IEEE 610.12-1990: Glossary of Software Engineering Terminology*. Software Engineering Standards Committee of the IEEE Computer Society, 1990 32, 54, 73

[IEE93] *IEEE 1044: Standard Classification for Software Anomalies*. Software Engineering Standards Committee of the IEEE Computer Society, 1993 8, 48, 94

[IEE98a] *IEEE 1061-1998: Revision of IEEE Std 1061-1992, Standard for a Software Quality Metrics Methodology*. Software Engineering Standards Committee of the IEEE Computer Society, 1998 73, 95

[IEE98b] *IEEE 730: Standard for Software Quality Assurance Plans*. Software Engineering Standards Committee of the IEEE Computer Society, 1998 56

[IEE98c] *IEEE 829: Standard for Software Test Documentation*. Software Engineering Standards Committee of the IEEE Computer Society, 1998 57

[ISB] `http://www.isbsg.org/` , zuletzt aufgerufen im August 2006 121

[ISO98] *International Standard ISO/IEC 14598-5: Information technology – Software product evaluation*. International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 1998. – Part 5: Process for evaluators 40

[ISO99a] *International Standard ISO/IEC 14598-1: Information technology – Software product evaluation*. International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 1999. – Part 1: General overview 40

[ISO99b] *International Standard ISO/IEC 14598-4: Information technology – Software product evaluation*. International Organization for Standardization (ISO) and

International Electrotechnical Commission (IEC), 1999. – Part 4: Process for acquirers 40

[ISO00a] *International Standard ISO/IEC 14598-2: Information technology – Software product evaluation*. International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2000. – Part 2: Planning and management 40, 41

[ISO00b] *International Standard ISO/IEC 14598-3: Information technology – Software product evaluation*. International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2000. – Part 3: Process for developers 40

[ISO01a] *International Standard ISO/IEC 14598-6: Information technology – Software product evaluation*. International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2001. – Part 6: Documentation of evaluation modules 40, 41

[ISO01b] *International Standard ISO/IEC 9126-1: Software Engineering – Product Quality*. International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2001. – Part 1: Quality model 8, 34, 39, 92, 102, 176

[ISO02] *International Standard ISO/IEC 15939: Software Engineering*. International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2002. – Software measurement process 41

[ISO03a] *International Standard ISO/IEC 9126-2: Software Engineering – Product Quality*. International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2003. – Part 2: External metrics 35, 121

[ISO03b] *International Standard ISO/IEC 9126-3: Software Engineering – Product Quality*. International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2003. – Part 3: Internal metrics 35, 121

[ISO04] *International Standard ISO/IEC 9126-4: Software Engineering – Product Quality*. International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2004. – Part 4: Quality in use metrics 35, 121

[ISO05a] *International Standard ISO/IEC 25000: Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE)*. International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2005. – Guide to SQuaRE 37, 38

[ISO05b] *International Standard ISO/IEC 25010: Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Quality Model*. International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 2005. – Guide to SQuaRE 34, 38, 39, 92, 102, 112, 114

[ISO06] *International Standard ISO/IEC 25051: Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE)*. International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), April 2006. – Requirements for quality of Commercial Off-The-Shelf (COTS) software product and instructions for testing 38

[JA97] Jacquet, Jean-Philippe ; Abran, Alain: From Software Metrics to Software Measurement Methods: A Process Model. In: *ISESS '97: Proceedings of the 3rd International Software Engineering Standards Symposium (ISESS '97)*. Washington, DC, USA : IEEE Computer Society, 1997, S. 128 74

[Joh05] Johansson, Enrico: *Quality Improvement with Focus on Performance in Software Platform Development*, Lund University, Diss., Juni 2005 1

[JWBH01] Johansson, Enrico ; Wesslén, Anders ; Bratthall, Lars ; Höst, Martin: The Importance of Quality Requirements in Software Platform Development - A Survey. In: *34th Hawaii International Conference on Systems Science (HICSS)*, 2001 3, 8, 13, 92, 102, 112, 119, 171

[Kam03] Kamiske, Gerd F. ; Umbreit, Gunnar (Hrsg.): *Qualitätsmanagement*. Leipzig : Fachbuchverlag Leipzig im Carl Hanser Verlag, 2003 36

[Kan03] Kan, Stephen H.: *Metrics and Models in Software Quality Engineering (2nd Edition)*. Addison-Wesley Professional, 2003 80, 81, 157, 173

[KCH+90] Kang, C. K. ; Cohen, S. ; Hess, J. ; Novak, W. ; Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, SEI Institute, Carnegie Mellon University (1990) 5, 28

[Kea97] Kean, Liz: *Domain engineering and domain analysis*. Januar 1997. – `http://www.sei.cmu.edu/activities/str/descriptions/deda_body.html`, aufgerufen im Oktober 2005 16

[Kin03] King, Margaret: Living up to standards. In: *Workshop on Evaluation Initiatives in Natural Language Processing: are evaluation methods, metrics and resources reusable? EACL*, 2003 41

[KLPN97] Kitchenham, Barbara ; Linkman, Steve ; Pasquini, Alberto ; Nanni, Vincenzo: The SQUID approach to defining a quality model. In: *Software Quality Control* 6 (1997), September, Nr. 3, S. 211–233. `http://dx.doi.org/http://dx.doi.org/10.1023/A:1018516103435`. – DOI http://dx.doi.org/10.1023/A:1018516103435 35

[Kos76] Kosiol, Erich: *Organisation der Unternehmung*. Wiesbaden : Betriebswirtschaftl. Verl. Gabler, 1976. – 2. Auflage 2

[KP03] Koomen, Tim ; Pol, Martin: *Test Process Improvement – A practical step-by-step guide to structured testing*. Addison Wesley, ACM Press, 2003 48

[KPF95] Kitchenham, Barbara ; Pfleeger, Shari L. ; Fenton, Norman E.: Towards a Framework for Software Measurement Validation. In: *IEEE Transactions on Software Engineering* 21 (1995), Dezember, Nr. 12, S. 929–943 173

[Lig02] Liggesmeyer, Peter: *Software-Qualität – Testen, Analysieren und Verifizieren von Software*. Heidelberg, Berlin : Spektrum Akademischer Verlag, 2002 34, 35, 36, 41, 54, 63, 72, 74, 77, 105, 177

[Lin02] Linden, Frank van d.: Software Product Families in Europe: The Esaps &

Café Projects. In: *IEEE Software* 19 (2002), Juli / August, Nr. 4, S. 41–49 21

[Lin04] Linden, Frank van d. (Hrsg.): *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers*. Bd. *3014*. Springer, 2004 (Lecture Notes in Computer Science) 181, 191

[MCM+99] McGregor, John D. ; Cho, Il-Hyung ; Malloy, Brian A. ; Curry, E. L. ; Hobatr, Chanika: Collecting Metrics for CORBA-Based Distributed Systems. In: *Empirical Software Engineering* 4 (1999), September, Nr. 3, S. 217–240. http://dx.doi.org/http://dx.doi.org/10.1023/A:1009878626684. – DOI http://dx.doi.org/10.1023/A:1009878626684. – ISSN 1382–3256 75

[MCN92] Mylopoulos, John ; Chung, Lawrence ; Nixon, Brian A.: Representing and Using Non-functional Requirements: A Process Oriented Approach. In: *IEEE Transactions on Software Engineering* 18 (1992), Juni, Nr. 6, S. 483–497 31

[MDR05] Muccini, Henry ; Dias, Marcio S. ; Richardson, Debra J.: Towards software architecture-based regression testing. In: *ICSE, WADS '05: Proceedings of the 2005 workshop on Architecting dependable systems*. New York, NY, USA : ACM Press, Mai 2005, S. 1–7 65

[Mel95] Melton, Austin: *Software Measurement*. International Thomson Computer Press, 1995 75, 173

[MIO87] Musa, John D. ; Iannino, Anthony ; Okumoto, Kazuhir: *Software Reliability – Measurement, Prediction, Application*. McGraw-Hill Book Company, 1987 173

[MP93] Möller, Karl H. ; Paulisch, Daniel J.: *Software-Metriken in der Praxis*. München : Oldenbourg Verlag, 1993 (Handbuch der Informatik) 121

[MSBT04] Myers, Glenford J. ; Sandler, Corey ; Badgett, Tom ; Thomas, Todd M.: *The Art of Software Testing, Second Edition*. John Wiley & Sons, New York, 2004 53

[Mye79] Myers, Glenford J.: *The Art of Software Testing.* John Wiley & Sons, New York, 1979 53, 60, 61

[Mye99] Myers, Glenford J.: *Methodisches Testen von Programmen (6. Auflage).* München, Wien : R. Oldenbourg Verlag, 1999 53, 97

[Neu05] Neun, Daniel: *Codemetriken zur Bewertung und Prognose der Fehlerhäufigkeit*, Universität Stuttgart, Diplomarbeit, Juli 2005 74

[NFLTJ03] Nebut, Clémentine ; Fleurey, Franck ; Le Traon, Yves ; Jézéquel, Jean-Marc: A Requirement-Based Approach to Test Product Families. In: [Lin04], S. 198–210 22, 69

[Obj03] Object Management Group (OMG): *Model Driven Architecture (MDA) Guide.* Juni 2003. – `http://www.omg.org/docs/omg/03-06-01.pdf` , zuletzt aufgerufen im März 2007 13, 113

[OHE99] Orfali, Robert ; Harkey, Dan ; Edwards, Jeri: *The essential client/server survival guide (3rd ed.).* New York, NY, USA : John Wiley & Sons, Inc., 1999 20

[OMG] *Unified Modeling Language – UML Resource Page.* – Object Management Group, `http://www.omg.org/uml`, zuletzt aufgerufen im Juni 2007 24

[PAIa] *Proactive Infrastructure (PAI) Info Center for the PAI J2EE Application Platform 3.0.1.* DaimlerChrysler AG, . – erhältlich über [PAIb] 136

[PAIb] *Website der Proactive Infrastructure (PAI).* Intranet der DaimlerChrysler AG, . – `http://intra-frameworks.daimlerchrysler.com`, zuletzt aufgerufen im September 2008 136, 192, 193

[Pal02] Palmer, Stephen R.: *Feature-Driven Development and Extreme Programming.* März 2002. – Pearson Education, article is provided courtesy of Prentice Hall. `http://www.informit.com`, zuletzt aufgerufen im Juni 2007 5, 26

[Par98] Partsch, Helmuth: *Requirements-Engineering systematisch.* Springer-Verlag, 1998 24, 30, 31

[Per95] Perry, William: *Effective Methods for Software Testing.* John Wiley & Sons, 1995 75

[Pfi06] Pfitzmann, Andreas: Biometrie – wie einsetzen und wie keinesfalls? In: *Informatik Spektrum* 29 (2006), Oktober, Nr. 5, S. 353–356 72

[PKCS95] Pulford, Kevin ; Kuntzmann-Combelles, Annie ; Shirlaw, Stephen: *A quantitative approach to Software management – The ami Handbook*. Addison-Wesley Publishing, 1995 44

[PKS00] Pol, Martin ; Koomen, Tim ; Spillner, Andreas: *Management und Optimierung des Testprozesses - Ein praktischer Leitfaden für erfolgreiches Testen von Software mit TPI und TMap*. dpunkt.verlag, 2000 37, 64

[PP04] Pomberger, Gustav ; Pree, Wolfgang: *Software Engineering – Architektur-Design und Prozessorientierung*. München, Wien : Carl Hanser Verlag, 2004 16

[Pre04] Prechtel, Markus: *Entwurf und Implementierung eines Frühwarnsystems zur Qualitätssicherung in der Fahrzeugproduktion*, Universität Ulm, Diplomarbeit, 2004 122

[PRP04] Pashov, Ilian ; Riebisch, Matthias ; Philippow, Ilka: Supporting Architectural Restructuring by Analyzing Feature Models. In: *CSMR '04: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering*. Washington, DC, USA : IEEE Computer Society, 2004, S. 25 29

[PS05] Prechtel, Markus ; Sellentin, Jürgen: Die Problematik beim Testen und Messen von Software-Plattformen. In: *Kolloquium „Testen im System- und Software-Life-Cycle“*, Technische Akademie Esslingen (TAE), November 2005 133, 205

[PSS05] Prechtel, Markus ; Sellentin, Jürgen ; Schweiggert, Franz: Test and Software Measures for Software Platforms / Frameworks. In: *Proceedings of the 3rd World Congress for Software Quality (3WCSQ)*, 2005 135, 205

[Qua06] Quast, Gerrit: *PAI in a Nutshell*. Intranet der DaimlerChrysler AG, 2006. – Dokumentation der PAI J2EE Application Platform 3.0.1 P1 + P2, erhältlich über [PAIb] 134, 136

[RB87] Rombach, H. D. ; Basili, Victor R.: Quantitative Software-Qualitätssicherung: Eine Methode zur Definition und Nutzung geeigneter Maße. In: *Informatik Spektrum* 10 (1987), Nr. 3, S. 145–158 40

[Rei04] Reimann, Wilfried: Building Enterprise Applications with an Integrated Application Platform. Erfurt, Germany, September 2004. – NET.ObjectDays Conference 133

[Rie97] Riedemann, Eike H.: *Testmethoden für sequentielle und nebenläufige Software-Systeme*. Stuttgart : B.G. Teubner Verlag, 1997 53

[Rie04] Riebisch, Matthias: Supporting Evolutionary Development by Feature Models and Traceability Links. In: *ECBS '04: Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-based Systems*. Washington, DC, USA : IEEE Computer Society, 2004, S. 370 29

[RKPR05] Reuys, Andreas ; Kamsties, Erik ; Pohl, Klaus ; Reis, Sascha: Szenariobasierter Systemtest von Software-Produktfamilien. In: *Inform., Forsch. Entwickl.* 20 (2005), Oktober, Nr. 1-2, S. 33–44 69

[RMB01] Ruh, William A. ; Maginnis, Francis X. ; Brown, William J.: *Enterprise Application Integration*. New York, Weinheim : Wiley Computer Publishing, 2001 19

[RPS03] Reussner, Ralf H. ; Poernomo, Iman H. ; Schmidt, Heinz W.: Contracts and Quality Attributes for Software Components. In: Weck, Wolfgang (Hrsg.) ; Bosch, Jan (Hrsg.) ; Szyperski, Clemens (Hrsg.): *Proceedings of the Eigth International Workshop on Component-Oriented Programming (WCOP'03)*, 2003 18

[RR06] Recknagel, Matthias ; Rupp, Chris: Messbare Qualität in Anforderungsdokumenten. In: *Java Magazin* 1 (2006), Februar, S. 12–17 78

[Rup04] Rupp, Chris: *Requirements-Engineering und -Management*. Carl Hanser Verlag , SOPHIST GROUP, 2004 23, 24, 31

[Saa97] Saatweber, Jutta: *Kundenorientierung durch Quality Function Deploy-*

ment: Systematisches Entwickeln von Produkten und Dienstleistungen. München, Wien : Carl Hanser Verlag, 1997 95

[SB97] Solingen, Rini van ; Berghout, Egon: Assessing Feedback Of Measurement Data: Relating Schlumberger Rps Practice To Learning Theory. In: *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics*. Washington, DC, USA : IEEE Computer Society, 1997. – ISBN 0–8186–8093–8, S. 152 42

[SB01] Solingen, Rini van ; Berghout, Egon: Integrating Goal-Oriented Measurement in Industrial Software Engineering: Industrial Experiences with and Additions to the Goal/Question/Metric Method (GQM). In: *METRICS '01: Proceedings of the 7th International Symposium on Software Metrics*. Washington, DC, USA : IEEE Computer Society, 2001, S. 246 42, 74

[SBS06] Sneed, Harry M. ; Baumgartner, Manfred ; Seidl, Richard: *Der Systemtest*. Hanser Verlag, 2006 64

[SCA06] SCAMPI Upgrade Team: Standard CMMI® Appraisal Method for Process Improvement (SCAMPISM) A, Version 1.2: Method Definition Document / Carnegie Mellon University, Software Engineering Institute. Version: August 2006. `http://www.sei.cmu.edu/cmmi/appraisals/`. Pittsburgh, PA, August 2006. – Forschungsbericht. – HANDBOOK, CMU/SEI-2006-HB-002 44

[Sch84] Schmidt, M.: Software-Metrik – Das aktuelle Schlagwort. In: *Informatik Spektrum* 7 (1984), Nr. 1, S. 41–42 71

[Sch03a] Schmid, Hermann: *Konzeption einer pragmatischen Testmethodik für den Test von eingebetteten Systemen*, Universität Ulm, Diss., 2003 124

[Sch03b] Schwinn, Thilo: *Effiziente Software-Inspektion durch ein Rahmenwerk zur antizipativen Berücksichtigung des Return-On-Investment*. Aachen, Diss., 2003. `http://bibserv21.bib.uni-mannheim.de:8080/openedu/library/opac/library.opac.ht` – XII, 188 S. : graph. Darst. 52, 78, 97, 98

[Sel00] Sellentin, Jürgen: *Datenversorgung komponentenbasierter Informationssysteme*. Berlin, Heidelberg : Springer-Verlag, 2000 (Informationstechnologien für die Praxis) 20

[SL04] Spillner, Andreas ; Linz, Tilo: *Basiswissen Softwaretest*. Heidelberg : dpunkt.verlag, 2004 53, 56, 58, 60, 63, 80

[Sne03] Sneed, Harry: *TestMetriken - Maßzahlen zur Bewertung des Testbetriebes*. http://www.gm.fh-koeln.de/~winter/tav/html/tav20/P3Sneed1TAV20.pdf,aufgerufena Version: Oktober 2003. – Gesellschaft für Informatik e.V., Treffen der Fachgruppe TAV im FB Softwaretechnik, Test, Analyse und Verifikation von Software 5

[Sog06] Sogeti Deutschland GmbH: Testoptimierung mit TPI. 2006. – Forschungsbericht. – erhältlich über http://www.sogeti.de , zuletzt aufgerufen am 15.02.2007 47, 48

[Som04] Sommerville, Ian: *Software Engineering (8th Edition)*. Pearson Addison Wesley, 2004 35, 43, 61, 63, 66, 78

[SR02] Sametinger, Johannes ; Riebisch, Matthias: Evolution Support by Homogeneously Documenting Patterns, Aspects and Traces. In: *Proceedings of the 6th European Conference on Software Maintenance and Reengineering* (2002) 27, 91

[SRWL06] Spillner, Andreas ; Rossner, Thomas ; Winter, Mario ; Linz, Tilo: *Praxiswissen Softwaretest – Testmanagement*. dpunkt.verlag, 2006 2, 3, 4, 49

[Ste78] Steinle, Claus: *Führung*. Stuttgart : Schäffer-Poeschel, 1978 2

[SW00] Schneider, Uwe ; Werner, Dieter: *Taschenbuch der Informatik*. Leipzig : Fachbuchverlag Leipzig im Carl Hanser Verlag, 2000 (3. Auflage) 13

[Szy03] Szyperski, Clemens: Component Technology - What, Where, and How? In: *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, IEEE Computer Society Press, 2003 18

[TAB03] Tyson, Barbara ; Albert, Cecilia ; Brownsword, Lisa: Interpreting Capability Maturity Model Integration (CMMI) for COTS-Based Systems /

Carnegie Mellon University, Software Engineering Institute. 2003. – Forschungsbericht 18

[Tha94] Thaller, Georg E.: *Software-Metriken einsetzen, bewerten, messen.* Hannover : Heinz Heise Verlag, 1994 121

[Tha00] Thaller, Georg E.: *Software-Qualität.* Berlin, Offenbach : VDE Verlag, 2000 (EDV-Praxis) 97

[TRK05] Tang, Johan van d. ; Rumpt, Harm van ; Kasperkovitz, Dieter: HW/SW Co-Design for SoC on Mobile Platforms. In: *Fifth International Workshop on System-on-Chip for Real-Time Applications (IWSOC'05)*, 2005, S. 19–23 13

[VXT08] *V-Modell XT.* Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung im Bundesministerium des Innern (KBSt), 2008. – Version 1.2.1.1, `http://www.v-modell-xt.de/`, zuletzt aufgerufen im Januar 2009 24, 61

[Wal90] Wallmüller, Ernest: *Software-Qualitätssicherung.* München, Wien : Carl Hanser Verlag, 1990 52, 61, 62, 63, 64, 77

[WN94] Watkins, Robert ; Neal, Mark: Why and How of Requirements Tracing. IEEE, ICSE'01, 23rd International Conference on Software Engineering (1994) 27

[Woh06] Wohlfarth, Sven: Entscheidungsprozess für rationale Architekturentscheidungen. In: *Softwaretechnik-Trends* 26 (2006), November, Nr. 4 33

Stichwortverzeichnis

Danksagung

Mein Dank richtet sich natürlich in erster Linie an Prof. Dr. Franz Schweiggert, der trotz eines immer vollen Terminkalenders stets ein offenes Ohr hatte und mir durch viele Gespräche neue Sichtweisen eröffnete. Er konnte mir durch seine unermessliche Erfahrung viele mögliche Problemfelder und unvereinbare Zielstellungen, die sich bei der Anfertigung einer Dissertation in einem Unternehmen ergeben, frühzeitig aufzeigen. Seine unkomplizierte Art und die zeitintensive Betreuung haben maßgeblich zum Gelingen dieser Arbeit beigetragen.

Ebenfalls herzlich danken möchte ich Oliver Fritz, meinem ehemaligen Vorgesetzten bei der DaimlerChrysler AG und später der Daimler AG. Seine überaus zielgerichtete Einstellung gaben mir viele Anregungen nicht nur bei der Stukturierung meines Vorhabens.

Eine nicht minder wichtige Rolle gespielt hat Dr. Jürgen Sellentin, der die Idee für eine wissenschaftliche Untersuchung dieses Bereichs hatte und mir v. a. in der Anfangsphase viele Hilfestellungen nicht nur informationstechnischer Natur gab.

Meinen Arbeitskollegen, derer ich viele aufzählen könnte, danke ich für alle Gespräche, die wir bei Kaffeepausen, beim Essen, aber auch sonst immer wieder zwischendurch führten. Hervorzuheben ist an dieser Stelle Robert Brcina, dem ich nicht nur für das Korrekturlesen meiner Arbeit zu Dank verpflichtet bin, sondern der mir auch die Bedeutung einer durchgängigen Modellierung der Anforderungen und der Dokumentation der Traceability aufzeigte.

Bedanken möchte ich mich auch bei Wilfried Reimann für die Möglichkeit, im Rahmen des PAI Projekts an einem Plattformprojekt mitzuarbeiten und meine Ideen dort einbringen zu können.

Daniel Schreiber danke ich nicht nur für das Korrekturlesen, sondern auch für seinen unerschütterlichen Optimismus, der mir stets neuen Ansporn gab, diese Arbeit fertigzustellen.

Sehr dankbar bin ich auch den anderen Doktoranden der Daimler AG, insbesondere Caro, Frauke, Markus und Saskia, die immer ein offenes Ohr für alle erdenklichen Belange hatten. Auch dankbar bin ich der Doktorandengruppe der Daimler AG, nicht nur für die Ermöglichung des Austauschs z. B. mit obigen Doktoranden bei den Stammtischen, Werksbesichtigungen, Kolloquien, Hüttenseminaren, etc.

Dankbar bin ich auch Claudia für das Korrekturlesen.

Meinen Freunden, die ich nicht alle aufzähle und die sich nun hoffentlich alle angesprochen fühlen, danke ich für genügend Ablenkung, schöne und interessante Feierabende, Skiurlaube, etc. ohne die mir vermutlich oftmals die Freude an der Arbeit gefehlt hätte.

Zu guter letzt möchte ich meinen Eltern und meiner ganzen Familie (meinen Geschwistern, deren Ehepartnern und v. a. auch meiner Nichte und meinen Neffen) herzlich dafür danken, dass sie mich stets unterstützt haben und mir immer einen sicheren Hafen boten.

Curriculum Vitae

Markus Prechtel

Studium und Schulbildung

16.11.1979	Geboren in Heidenheim an der Brenz
09/1986 – 08/1990	Grundschule Silcherschule, Heidenheim
07/1990 – 09/1999	Hellensteingymnasium, Heidenheim
10/1999 – 05/2004	Universität Ulm, Studium der Mathematik (Abschluss Diplom Mathematiker)
seit 07/2004	Universität Ulm, Promotion am Institut für angewandte Informationsverarbeitung betreut durch Prof. Dr. F. Schweiggert

Praktika, Berufserfahrung

07/1999 – 09/1999	Softcon IT Service GmbH, München: Praktikant
10/2000 – 08/2003	Universität Ulm, Abteilungen Analysis und Angewandte Informationsverarbeitung: studentische Hilfskraft
07/2003 – 12/2003	DaimlerChrysler AG, Research and Technology, Information Mining (RIC/AM), Ulm: Diplomand
07/2004 – 06/2007	DaimlerChrysler AG, IT Management Infrastructure, Shared Services (ITI/SP), Stuttgart: Doktorand
seit 08/2007	BeOne GmbH, Stuttgart: Consultant

Veröffentlichungen, Vorträge

07/2004	Veröffentlichung als Co-Autor mit Dr. M. Grabert et. al.: “An Early Warning System for Vehicle Related Quality Data“ [GPHG04]
09/2005	Vortrag beim 3rd World Congress for Software Quality, München: „Test and software measures for software platforms and frameworks“ [PSS05]
11/2005	Vortrag an der Technischen Akademie Esslingen im Rahmen eines Kolloquiums, Esslingen: „Die Problematik beim Testen und Messen von Software-Plattformen“, [PS05]
10/2006	Veröffentlichung als Co-Autor mit R. Brcina bei GI-Workshop, Ladenburg: „Feature-orientierte Plattformentwicklung und Verfolgbarkeit“, [BP06b]