Förderkreis der
Angewandten
Informatik
an der
Westfälischen
Wilhelms-Universität Münster e.V.

Working Paper No. 2

Softwaretests

Tim A. Majchrzak,
Herbert Kuchen

Oktober 2010

Westfälische
Wilhelms-Universität
Münster

Working Paper No. 2

IHK-Projekt Softwaretests: Auswertung

Tim A. Majchrzak, Herbert Kuchen

18. Oktober 2010

Prof. Dr. Herbert Kuchen, Tim A. Majchrzak
Institut für Wirtschaftsinformatik
Westfälische Wilhelms-Universität Münster
Leonardo Campus 3
48149 Münster
tima@ercis.de

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http:/dnb.ddb.de abrufbar.

ISSN 1868-0801

Printed in Germany

Vorwort

Computersysteme haben eine hohe Bedeutung für fast alle Arten von unternehmerischen Tätigkeiten. Softwareprobleme können unternehmensinterne Abläufe erheblich stören und zu schwerwiegenden Sicherheitsproblemen führen. Aus diesem Grund ist das Erreichen und Aufrechterhalten einer hohen Softwarequalität unerlässlich.

Zahlreiche Unternehmen im IHK-Bezirk Nord Westfalen produzieren Software für Kunden oder zur Unterstützung ihrer eigenen Prozesse. Alle regionalen Unternehmen profitieren dabei vom Angebot hochwertiger Software. Aber auch *intern* genutzte Software kann durch hohe Qualität den Unternehmen die stets zuverlässige Wahrnehmung ihrer Aufgaben ermöglichen. Im Sinne der Konkurrenzfähigkeit muss Software allerdings nicht nur eine hohe Qualität aufweisen. Vielmehr müssen die Mittel zur Erreichung der Qualität möglichst kosteneffizient ausfallen. Zudem bietet ein Blick auf Softwarequalität im Allgemeinen und Testen im Speziellen die Chance, Prozesse rund um die Entwicklung und Evaluation von Software zu optimieren. Darüber hinaus kann das Anbieten hochqualitativer Software als Alleinstellungsmerkmal Wettbewerbsvorteile eröffnen.

In der vorliegende Broschüre „Softwaretests“ wurde der Status quo des Testens von Unternehmen der Region zusammengetragen. Ferner werden eine Vielzahl von Handlungsempfehlungen vorgestellt. Die Empfehlungen wurden aus erfolgreichen Vorgehensweisen einzelner Unternehmen abgeleitet.

Die Broschüre ist durch den Förderkreis für Angewandte Informatik an der Westfälischen Wilhelms-Universität Münster angeregt und finanziert worden. Der Förderkreis wird von rund 30 Unternehmen aus der Region und der Industrie- und Handelskammer Nord Westfalen getragen. Hauptziel ist die Förderung der praxisorientierten Forschung und Lehre, sowie der schnelle Wissenstransfer. Besonderer Dank gebührt dem Direktor des Instituts für Angewandte Informatik, Herrn Professor Dr. Herbert Kuchen, und seinem Mitarbeiter, Herrn Tim A. Majchrzak, für die außerordentlich engagierte und praxisnahe Umsetzung des Projektes.

Großer Dank gilt auch den Mitgliedsunternehmen, die sich am Projekt beteiligt haben und deren Mitarbeiter für umfangreiche Interviews zur Verfügung standen. Diese Bereitschaft bildet die Basis der Broschüre. Aufgrund der Sensibilität des Status quo werden einzelne Unternehmen in diesem Rahmen nicht benannt.

Martin Kittner
Vorsitzender des Förderkreises
für Angewandte Informatik

Dr. Christoph Asmacher
Industrie- und Handelskammer
Nord Westfalen

Inhaltsverzeichnis

1. Einführung

Für viele Unternehmen in Nord Westfalen besitzt die Entwicklung von Software einen hohen Stellenwert. Neben zahlreichen zumeist mittelständischen Betrieben, die Software sowohl für den Massenmarkt als auch individuell entwickeln, ist dabei vor allem an Finanzdienstleister zu denken. Insbesondere Münster selbst ist ein wichtiger Standort für Banken und Versicherungen. Auch wenn für diese die Softwareentwicklung kein Kerngeschäft ist, so ist die Verfügbarkeit effektiver und effizienter IT-Systeme essentiell. Schließlich sind Finanzdienstleister auf die schnelle Verfügbarkeit von Informationen und die extrem sichere und korrekte Verarbeitung von großen Datenmengen angewiesen.

Mit den Banken und Versicherungen, die sicherlich als „treibende Kraft“ für IT-Entwicklungen verstanden werden können, und dem innovativen Mittelstand ist die Region im Bereich der Softwareentwicklung gut aufgestellt. Gleichzeitig ist es mit der WWU Münster ein Zentrum für Forschung und Lehre. In diesem Kontext agiert das Institut für Angewandte Informatik (IAI) [1]. Zu seiner Arbeit gehört der Austausch von akademischer Forschung und Unternehmenspraxis sowie die anwendungsorientierte Lösung häufiger bzw. ungelöster IT-Probleme der Unternehmen aus der Region. Getragen wird das IAI vom Förderkreis der Angewandten Informatik an der Westfälischen Wilhelms-Universität Münster e. V. [2], in dem sich neben der Industrie- und Handelskammer (IHK) Nord Westfalen [3] IT interessierte Unternehmen zusammengeschlossen haben.

Spätestens mit Beginn der Softwarekrise in der Mitte der 1960er Jahre [Dij72] wurde Softwarequalität eine zunehmend große Aufmerksamkeit geschenkt. Spektakuläre Fehlschläge wie das Ariane-Unglück [4] oder Projekte, die nicht nur erheblich verzögert wurden, sondern auch deutlich höhere Kosten als zunächst geplant verursachten (z. B. Toll Collect [5][6][7] oder die Elektronische Gesundheitskarte [8][9][10]) erreichen eine breite Öffentlichkeit. Viel erschreckender und letztlich inakzeptabel sind allerdings Statistiken zu Projektabbrüchen, Fehlinvestitionen oder der (Un-)Zufriedenheit mit Software. Die Sicherstellung von Softwarequalität ist daher eine immens wichtige Aufgabe. Gleichzeitig ist es eine extrem komplexe Aufgabe, die auch nach über vier Jahrzehnten Forschungsarbeit zahlreiche Fragen offen lässt. Besonders problematisch ist die Tatsache, dass Softwaresysteme bzw. IT-Landschaften im Allgemeinen in ihrer Komplexität ständig wachsen und daher immer bessere Methoden zur Qualitätssicherung erforderlich werden.

Die Wichtigkeit von Softwarequalität auch für Unternehmen aus dem Münsterland und der Emscher-Lippe-Region muss nach obigen Ausführungen nicht mehr betont werden. Aus Gesprächen, die im Rahmen der Arbeit des IAI geführt wurden, war bekannt, dass viele Unternehmen ein grundsätzliches Verbesserungspotential bezüglich ihrer Prozesse für das Testen von Software erkannt haben. Diese Verbesserungen lassen sich allerdings nicht ohne weiteres realisieren. Insbesondere fehlen mitunter Informationen zum Einsatz

von Werkzeugen. Häufig lässt auch der hohe Aufwand des operativen Geschäfts nur gelegentliche und schrittweise Anpassungen der Testprozesse zu. Nichts desto trotz sind die Unternehmen aus Nord Westfalen erfolgreich; Verbesserungsbedarf zu erkennen ist keineswegs mit einer unbefriedigenden Lage oder fehlender Kompetenz gleichzusetzen, ganz im Gegenteil.

Um die Unternehmen der Region zu unterstützen wurde vom Förderkreis der Angewandten Informatik das Projekt „Testen“ initiiert. Der Hauptgedanke setzt sich aus zwei Überlegungen zusammen:

- Sicherlich betrachtet keines der Software entwickelnden Unternehmen seine Testprozesse als *perfekt*. Probleme unterschiedlicher schweregrade bzw. Wünsche, kosteneffektiver zu Testen oder eine höhere Softwarequalität zu erreichen, lassen sich in jedem Unternehmen finden.

- Jedes Unternehmen wird Vorgehensweisen identifizieren können, die in besonderem Maße beherrscht werden und die zur Entwicklung hochwertiger Software beitragen.

Werden demnach einige Unternehmen besucht und Interviews bezüglich ihrer Testprozesse geführt, sollten sich allgemeine Probleme, aber auch besonders herausragende gute Vorgehensweisen (*best practices*) identifizieren lassen. Es ist anzunehmen, dass sich Probleme und Lösungen zumindest zum Teil ergänzen, so dass eine Erfahrungsaustausch zum Vorteil aller teilnehmenden Unternehmen möglich ist.

In der vorliegenden Broschüre sind die Ergebnisse des Projektes zusammengetragen worden. Sie gliedert sich in fünf Kernkapitel, ergänzt durch ein einleitendes und ein Abschlusskapitel. Im ersten Kernkapitel wird in die Thematik eingeführt, indem eine Begriffsbasis geschaffen und das Vorgehen im Projekt vorgestellt wird. Das zweite Kernkapitel stellt den Status Quo des Testens von Software in Unternehmen aus Nord Westfalen dar. Daran anschließend wird ein Ordnungsrahmen für die identifizierten Handlungsempfehlungen entwickelt und motiviert. Erst dann werden im vierten und fünften Kernkapitel konkrete Handlungsempfehlung formuliert. Diese sind aufgeteilt in Strategien, wie das Testen organisatorisch eingebettet werden kann, und in technische Hinweise, etwa bezüglich des Einsatzes von Werkzeugen. Vertiefende Informationen liefert zudem ein umfangreicher Anhang.

Die Autoren der Broschüre danken an dieser Stelle den beteiligten Unternehmen für die freundliche Bereitschaft, (z. T. mehrere) ausführliche Interviews mit für das Testen verantwortlichen Mitarbeitern führen zu können. Ohne diese Gespräche und die dafür notwendige Offenheit der Unternehmen und ihrer Mitarbeiter wäre diese Broschüre nicht denkbar gewesen.

2. Grundlagen

Im Folgenden werden zunächst wichtige Grundbegriffe erläutert. Danach wird das Projekt hinsichtlich des Vorgehens vorgestellt. Schließlich folgen Hinweise zur Verwertung der in dieser Broschüre dargestellten Informationen.

2.1. Wichtige Begriffe

Nicht nur in den Verwendung in Unternehmenspraxis und Wissenschaft sind viele Begriffe der Softwaretechnik unterschiedlich belegt oder werden zumindest aus individuellen Blickrichtungen betrachtet. Vielmehr herrscht auch in der Literatur und im Alltag häufig keine Einigkeit, was dort, wo Trennschärfe gefragt ist, zu Problemen führt. Aus diesem Grund sollen einige der in dieser Broschüre verwendeten Begriffe vorab definiert bzw. erklärt werden. Maßgeblich ist dabei jeweils die Bedeutung im Rahmen dieser Broschüre.

Anforderung Die Aufgaben, die sich mithilfe einer Software bewältigen lassen sollen, sowie die Art und Weise, auf die dies durch die Software unterstützt wird, lassen sich in Form von Anforderungen formulieren.

Anforderungsanalyse (auch: Anforderungserhebung, requirements engineering) Ziel der Anforderungsanalyse ist es, die *Anforderungen* des Kunden an die zu entwickelnde Software zu ermitteln und festzuhalten. Dies wird in der Regel dadurch erschwert, dass es nicht unmittelbar möglich ist, gewünschte Zustände in Worte zu fassen und dass Kunden mitunter selbst von der zu entwickelnden Software noch kein klares Bild haben.

Automatisierter Test Bei automatisierten Tests können einzelne oder alle Funktionen des Testens weitgehend ohne menschlichen Eingriff abgerufen werden. Einfache Automatisierungen umfassen dabei die Ausführung einer Reihe von Testfällen und die anschließende Darstellung geglückter bzw. fehlgeschlagener Fälle. Erweiterte Automatisierung ist etwa die Erstellung von Testdaten durch einen (parametrisierten) Generator sowie „auf Knopfdruck“ ausführbare Regressionstests. Automatisierung meint immer die Automatisierung einzelner Funktionen, nie eine Vollautomatisierung des Testprozesses. Letzteres kann allen Forschungsbestrebungen zum Trotz nur als Fernziel betrachtet werden und ist möglicherweise gänzlich unmöglich.[1]

[1] Aufgrund des so genannten Halteproblems ist die automatische Verifikation von Software nur in einem bestimmten Rahmen möglich. Das Systeme sich im Allgemeinen komplett selbst verifizieren und damit ihre Korrektheit garantieren können ist nach dem heutigen Wissensstand ausgeschlossen.

Black Box-Test Alle Tests, bei denen von den Quelltexten abstrahiert und die eigentliche Implementierung des zu testenden Moduls oder Systems unbedeutend sind, werden als Black Box-Tests bezeichnet.

Build Bei einem Build handelt es sich um einen fertigen Versionsschritt einer Software. Während bei der Entwicklung Komponenten getrennt betrachtet und behandelt werden können, trägt ein Build einen Satz oder gegebenenfalls auch alle Komponenten in kompilierter Form zusammen. Das Ergebnis ist eine ausführbare Version der Software. Vor Veröffentlichung eines *Releases* einer Software gibt es während der Entwicklung in der Regel mehrere Builds.

Capture&Replay (-Werkzeug) Unter Capture&Replay wird eine Technik verstanden, die es erlaubt, bestimmte Vorgänge aufzunehmen und zu einem späteren Zeitpunkt wiederzugeben. Klassisch geht es dabei um die direkte Aufnahme von Nutzerinteraktionen mit der Programmoberfläche (Graphical User Interface, GUI), um etwa wenig personalintensive Regressionstests zu ermöglichen. Moderne Werkzeuge erfassen bestimmte Eingabefolgen und bieten auch die Möglichkeit, diese modifiziert wiederzugeben.

Debugging Mithilfe von Werkzeugen zum Debugging, so genannten *Debuggern*, können Fehler in Software gefunden werden. Häufig kommen Sie zum Einsatz, nachdem das Vorhandensein von Fehlern durch Tests festgestellt wurde. Durch das Debugging wird der Ursprung des Fehlers gesucht und anschließend die problembehaftete Stelle in der Software verbessert.

Fehler (auch: Defekt, defect) Ein Fehler im Sinne dieser Broschüre meint immer eine in der getesteten Software gefundene Eigenschaft, die nicht der Erwartung entspricht. Fehler können auf falsch umgesetzte Anforderungen oder eine fehlerhafte Programmierung zurückgeführt werden. In der Literatur wird häufig eine stärkere Unterscheidung getroffen, etwa in Fehlhandlung, Fehlerzustand und bemerkbare Fehlerwirkung. [SRWL08] Eine solche Untergliederung wird hier nicht benötigt.

Glass Box-Test (auch: White Box-Test) Ein Test, bei dem der Quelltext des zu testenden Moduls oder Systems bekannt ist und dieses Wissen explizit genutzt wird. Beispiele sind Daten- bzw. Kontrollflusstests.

Gray Box-Test Haben Tester, die Black Box-Tests vornehmen, zumindest Teilkenntnisse des zugrunde liegenden Quelltextes und nutzen dieses Wissen, etwa um einige zusätzliche *Testfälle* zu erstellen, so wird von Gray-Box Tests gesprochen.

Kunde Kunden im Sinne dieser Broschüre sind Leistungsempfänger. Es kann sich demnach sowohl um externe Auftraggeber oder Massenmarktkunden handeln, als auch um interne Fachabteilungen, für die eine Anwendung entwickelt wird. Der Kunde muss nicht zwangsläufig mit dem Auftraggeber identisch sein.

Manueller Test Bei manuellen Tests werden alle Schritte des Testens „von Hand“ ausgeführt. Viele Werkzeuge erleichtern zwar manuelle Tests, stellen streng genommen aber keine Automatisierung dar.

Mock-Objekt Siehe *Teststumpf*.

Projektteilnehmer Mit Projektteilnehmern werden in dieser Broschüre die Unternehmen bezeichnet, die an dem ihr zugrunde liegenden Projekt teilgenommen haben und in diesem Rahmen für Interviews besucht wurden. Der Begriff wird in keinem anderen Zusammenhang verwendet, um Missverständnisse zu verhindern.

Regressionstest Bei Regressionstests werden einige oder alle Testfälle erneut ausgeführt. Dadurch soll ausgeschlossen werden, dass durch Änderung an bzw. der Weiterentwicklung von Software Fehler zu bereits getesteten Komponenten hinzugekommen sind.

Release Ein Release stellt die Veröffentlichung einer Version eines Softwareprodukts dar. Im Gegensatz zu einfachen Weiterentwicklungen stellen Releases aller Regel nach in sich abgeschlossene Produkte dar. Software kann als ein einziges Release erscheinen, oder in Form von mehreren Releases veröffentlicht werden. Folgeveröffentlichungen gehen dabei häufig deutlich über den Funktionsumfang der Vorversion hinaus. Die Verteilung von Releases kann kostenlos erfolgen oder eine kostenpflichtige Erweiterung (*Upgrade*) der Software darstellen. Gegebenenfalls wird bei Softwareprodukten im Rahmen von Wartungsverträgen die Bereitstellung von regelmäßigen Aktualisierungen vereinbart.

Software (auch: Applikation, Programm) Als Software wird in dieser Broschüre jede Entwicklung eines auf Computern ausführbaren Programms bezeichnet, die eine ausreichende Komplexität erreicht, um zur eigenständigen Lösung von (betriebswirtschaftlichen) Problemen genutzt werden zu können.

Softwaresystem Als Softwaresysteme wird eine einzelne, sehr komplexe Software bezeichnet, die Schnittstellen zu zahlreichen anderen Systemen unterhält, oder aber ein Verbund funktional zusammengehöriger Software, die gemeinsam zur Erbringung einer Aufgabe eingesetzt werden.

System Wird von Systemen im Allgemeinen gesprochen, so sind damit technische Einrichtungen gemeint, die zur Lösung von (betriebswirtschaftlichen) Problemen zum Einsatz kommen. Ob es sich um reine Softwaresysteme handelt, oder auch die zugrunde liegende Hardwareplattform mit inbegriffen ist, ist in einem Kontext, in dem nur von System gesprochen wird, entweder unbedeutend oder durch den Kontext unmissverständlich vorgegeben.

(Software-)Test Ein Test im Sinne dieser Broschüre ist immer ein Softwaretest. Ziel von Softwaretests ist es, die Erfüllung der an eine Software gestellten Anforderungen durch diese zu überprüfen und durch das Aufdecken von Fehlern ein angestrebtes Qualitätsniveau zu erreichen. Die einzelnen Testarten werden an dieser Stelle

nicht definiert, sondern an der entsprechenden Stelle dieser Broschüre im Detail vorgestellt.

Testen Testen beschreibt die Tätigkeit, bei der eine Software hinsichtlich ihrer Qualität überprüft wird. Es handelt sich demnach um die Ausführung eines oder mehrerer Tests.

Testfall Jeder Testfall stellt eine elementaren Vorgang beim Testen einer Software dar. Testfälle kann es dabei auf unterschiedlichen Abstraktionsstufen geben. Im Sinne eines Komponententests durch den Entwickler etwa handelt es sich um einen Satz von Parametern für den Aufruf einer Funktion bzw. Methode. Bei Oberflächentests hingegen könnte ein Testfall eine festgelegte Folge von Mausklicks und Eingaben beschreiben, die eine bestimmte Aktion im darunter liegenden System auslösen.

Teststumpf (auch: stub) Werden einzelne Module einer Software getestet, so sind andere Module möglicherweise (noch) nicht verfügbar. Auch ist es möglich, dass ein Modul getestet werden soll, ohne das es Einfluss auf andere Module nehmen könnte. In diesen Fällen kommen Teststümpfe zum Einsatz. Sie können Schnittstellen simulieren und kommen vor allem in frühen Testphasen zum Einsatz. Stümpfe im Sinne der Entwicklung von verteilten Anwendungen haben keine Relevanz für diese Broschüre. Mock-Objekt stellen spezielle Objekte dar, die in der objektorientierten Entwicklung genutzt werden. Sie fungieren als Platzhalter für echt Objekte und kommen vor allem in Komponententests zum Einsatz. Teststumpf und Mock-Objekt sind keine Synonyme; letztere kommen vor allem in Rahmen des testgetriebenen Entwicklung zum Einsatz. Während weitere Details für diese Broschüre irrelevant sind, sei der interessierte Leser etwa auf [11] und [12] verwiesen.

Testzentrum (Auch: Testcenter, test center) Ein Testzentrum ist eine zentrale organisatorische Einheit, die Leistungen rund um das Testen von Software erbringt. Der Verantwortungsbereich und das genaue Aufgabenspektrum können von Unternehmen zu Unternehmen stark variieren.

Werkzeug (auch: Tool) Werkzeuge sind Applikationen, die bestimmte Abläufe der computergestützten Arbeit erleichtern bzw. unterstützen. Werkzeuge im Sinne dieser Broschüre sind Werkzeuge für Softwaretests. Darunter fallen kleine Werkzeuge für Programmierer, Programme, die Softwaretests unterstützen, aber auch Software für die Testfallverwaltung.

2.2. Teilnehmer

Aufgrund der Anonymisierung der Ergebnisse ist es nicht möglich, teilnehmende Unternehmen oder gar Ansprechpartner direkt zu benennen. Nichts desto trotz soll ein kurzer Überblick über das Teilnehmerfeld in Bezug auf die Unternehmensgröße, die Branche und die Art der Softwareentwicklung gegeben werden.

Von der sechs Unternehmen, die den Kern der Untersuchung ausmachen, fällt ein Unternehmen in den Bereich von 0-50 Mitarbeiter und zwei in den Bereich von 51-250. Drei Unternehmen schließlich beschäftigen – zum Teil deutlich – über 1.000 Mitarbeiter. Für zwei Unternehmen – beides Finanzdienstleister – ist die Softwareentwicklung Mittel zum Zweck: Sie stützen bzw. ermöglichen durch selbst bzw. in Auftrag entwickelte Software ihr Kerngeschäft. Für die restlichen Unternehmen ist die IT bzw. der Verkauf von IT-Lösungen das Kerngeschäft. Für alle beteiligten Unternehmen sind Softwareentwicklung bzw. zumindest Softwareeinsatz und -erweiterung geschäftskritisch.

Während zwei Unternehmen Software für einen breiten Markt bereitstellen und ihre Hauptprodukte ständig weiterentwickeln, liegt bei den übrigen vier der Fokus auf der individuellen Entwicklung. Ihr Geschäft ist daher vor allem auftragsbezogen. Grundsätzlich agieren alle Unternehmen sowohl bei kleineren als auch bei größeren Projekten. Allerdings ist im Allgemeinen eine Korrelation zwischen Unternehmensgröße und durchschnittlichem Projektumfang festzustellen.

2.3. Vorgehen in den Interviews

In der ersten Phase des Projekts wurden die Unternehmen, die sich zur Teilnahme entschlossen haben, besucht. In den ca. zwei bis dreistündigen Interviews wurde zunächst die grobe Einordnung der Softwaretests in den Unternehmen diskutiert. Dabei galt es abzugrenzen, welche Art von Software in den Unternehmen entwickelt wird, wie das Testen organisatorisch eingebettet ist und wie Testprozesse gestaltet sind. Die Interviews wurden dabei als semi-strukturierte Experteninterviews geführt.

Als nächstes wurde detailliert über die einzelnen Testarten gesprochen. Wichtig dabei war, ein genaues Bild des Vorgehens in den einzelnen Unternehmen zu gewinnen und Stärken und Schwächen – so möglich – direkt zu identifizieren. Schließlich wurde der Werkzeugeinsatz thematisiert. Dabei ging es nicht nur um die Frage, welche Werkzeuge eingesetzt werden, sondern vor allem auch darum, welche Schwierigkeiten es in der Vergangenheit möglicherweise gegeben hat und wie hoch der Anteil der Testautomatisierung ist.

Zum Teil wurden nicht alle Fragen in einem Gespräch geklärt. Mitunter kam es nach dem ersten Termin zu weiteren. Typischerweise wurde das erste Gespräch in diesem Fall mit einem für das Testen zuständigen Leiter geführt, weitere Gespräche erfolgten dann mit Testern bzw. Entwicklern und waren deutlich technischer geprägt.

2.4. Vorgehen bei der Auswertung

Bereits während der Durchführung der Interviews war zu erkennen, dass es einige „gemeinsame“ Probleme gab, aber jeder einzelne Teilnehmer auch erfolgreiche Strategien identifizieren konnte, die auch nur zum Teil deckungsgleich waren. Darüber hinaus war schnell festzustellen, dass die Ergebnisse sehr differenziert zu betrachten sein würden: Aufgrund der Heterogenität der teilnehmenden Unternehmen sind die Voraussetzungen

und auch die Anforderungen an das Testen von Software sehr unterschiedlich. Nichts desto trotz ist ein gegenseitiges Lernen vielversprechend (vgl. mit dem kommenden Kapitel 2.5, das Hinweise zur Verwertung gibt).

Im Rahmen der eigentlichen Auswertung wurden alle Interviews zusammengetragen und analysiert. Als erster Schritt wurde dann der Status quo abgeleitet und in den Kontext der teilnehmenden Unternehmen gesetzt. Im zweiten Schritt wurden die erfolgreichen Strategien extrahiert und verdichtet. Dabei wurde klar, dass diese nicht nur in einzelne Empfehlungen gegliedert werden können, sondern zwei grundsätzliche Arten von Empfehlungen unterschieden werden müssen. Auch wenn erfolgreiche Testprozesse die Interdependenzen zwischen der organisatorischen Einbettung und der technischen Ausgestaltung des Testens berücksichtigen, ist eine Trennung hinsichtlich der Empfehlungen sinnvoll.

Neben der Herausarbeitung einzelner Empfehlungen wurde jeweils auch versucht, Hinweise zu oben angesprochenen Interdependenzen zu geben. Der Anspruch an einzelne Empfehlungen ist zwar, diese „für sich“ nutzen zu können, eine wesentliche Verbesserung der Testprozesse verspricht aber vor allem eine ganzheitliche Sicht auf das Testen von Software, bei der nicht nur einzelne Empfehlungen umgesetzt, sondern kombiniert werden.

Zur besseren Nutzbarkeit der Handlungsempfehlung wird im vierten Kapitel ein Ordnungsrahmen motiviert, in den jede Empfehlung eingeordnet werden kann. Er soll helfen, einzelne Empfehlungen hinsichtlich ihrer Stichhaltigkeit für die unternehmenseigenen Gegebenheiten und Prozesse zu bewerten und die Entscheidung für oder gegen die Umsetzung unterstützen.

2.5. Hinweise zur Verwertung

Wie bereits angesprochen, ist die Entscheidung für eine Aufteilung der Ergebnisse in vier Kapitel gefallen. Die Darstellung des Status quo im folgenden Kapitel kann dazu dienen, das eigene Unternehmen einzuordnen und sollte Ausgangspunkt sein, die eigenen Testprozesse kritisch zu reflektieren. Der dann vorgestellte Ordnungsrahmen dient als Unterstützung bei der Bewertung der Handlungsempfehlungen.

In den zwei darauf folgenden Kapitel werden die konkreten Empfehlungen dargestellt. Für jede Empfehlung wird beschrieben, für welche Rahmenbedingungen sie gelten und wie sie – sofern möglich – an abweichende Rahmenbedingungen angepasst werden können. Hintergrund dieses Vorgehens ist die Vielschichtigkeit der Adressaten dieser Broschüre. Sie soll für sehr kleine Unternehmen ebenso wie für größere mittelständische Unternehmen nutzbar sein. Des Weiteren soll sie sowohl individuelle Softwareentwicklung als auch die Produktion für den Massenmarkt abdecken. Deshalb ist es unabdingbar, die Aussagekraft verschiedener Empfehlungen für entsprechende Rahmenbedingungen einzugrenzen bzw. eine grundsätzliche Empfehlung auszusprechen, aber die Feinjustierung bzw. Parametrisierung dem implementierenden Unternehmen zu überlassen. Sofern möglich werden Hinweise gegeben, wie die Parametrisierung erfolgen kann.

Viele Empfehlungen sind variabel gestaltet. Wenn eben möglich, wurde versucht, zwi-

schen *soll* und *kann* zu unterscheiden und Freiheitsgrade bei der Ausgestaltung der Empfehlung zuzulassen. Nur so kann den variablen Anforderungen der Adressaten und Ihren Möglichkeiten, etwa hinsichtlich Kapital-, Zeit- oder Personaleinsatz, adäquat begegnet werden. Über den veränderlichen Anteil einzelner Empfehlungen hinaus wird für einige Empfehlungen auch angegeben, inwiefern sie in Kombination mit anderen Empfehlungen zum Einsatz kommen sollten bzw. wie sie sich in das Gefüge einzelner Empfehlungen, die zusammen eingesetzt erhebliche Prozessverbesserungen versprechen, einpassen.

3. Status quo der Softwaretests

Die erste wesentliche Aufgabe des dieser Broschüre zugrunde liegenden Projekts ist die Erfassung des Status quo der Softwaretests bei den beteiligten Unternehmen. In den folgenden Unterkapiteln werden daher die Ergebnisse der Interview verdichtet und Gemeinsamkeiten sowie Gegensätze aufgezeigt.

3.1. Allgemeines

Hinsichtlich der Ergebnisse waren die geführten Interviews sehr unterschiedlich. Zwar wurden bei jedem Unternehmen besondere Stärken und Schwächen identifiziert; auch herrscht eine generell hohe Zufriedenheit mit den Softwaretests im eigenen Haus. Aber im Vergleich zeigen sich mitunter erhebliche Unterschiede.

Festzustellen ist vor allem, dass sich Unterschiede offenbar nach der Unternehmensgröße richten. Die Verankerung von Testen als Prozess ist unüblich für kleinere Unternehmen, ebenso wie der Unterhalt eines eigenen Testzentrums. Tests müssen häufig gegenüber dem Kunden motiviert werden, da er sie als Kostenverursacher ohne Nutzen wahrnimmt und dementsprechend einen hohen Testaufwand ablehnt. Gleichzeitig sagten gerade die kleinen Unternehmen, dass die Quote der erfolgreich abgeschlossenen Projekte sehr hoch sei.

Dem gegenüber steht eine viel komplexere Testorganisation bei größeren Unternehmen. Testen ist dort keine Zusatzaufgabe, sondern Kernfunktion. Zusätzlicher Testaufwand muss natürlich auch hier gerechtfertigt werden, aber ein viel höheres Maß wird als grundsätzlich nötig angesehen als in den kleinen Unternehmen. Dennoch scheitern – im Gegensatz zu den kleinen Unternehmen – Projekte in einigen Fällen. Testen wird mit einer zunehmenden Zahl von Mitarbeitern, die mit der Softwareentwicklung betreut sind, zu einer immer fester verankerten Funktion. Auch steigt der betriebene Aufwand mit der Zahl der Mitarbeiter. Gleichzeitig handelt es sich dabei um eine Notwendigkeit. Denn mit zunehmender Projektgröße scheitert die dezentrale und recht unstrukturierte Herangehensweise, wie sie bei kleineren Unternehmen anzutreffen ist. Zudem unterscheidet sich die Kundenstruktur und damit der Qualitätsanspruch. Offenbar ist es bei kleinen Projekten, in denen individuell Software für Kunden entwickelt wird, deutlich weniger problematisch, Fehler erst nach der Auslieferung der Software zu erkennen und zu beheben, als bei größeren Standardanwendungen. Dies gilt umso mehr für sicherheits- bzw. geschäftskritische Software. Bei Unternehmen, die Finanzdienstleistungen erbringen, kommen zudem gesetzliche Vorgaben, etwa bezüglich des Risikomanagements, zum tragen.

3.2. Organisatorische Einbettung und Testverantwortung

Bezüglich der organisatorischen Einbettung des Testens zeigt sich die bereits im vorherigen Kapitel festgestellte Zweiteilung zwischen kleinen und größeren Unternehmen. Erstere testen dezentral, während letztere ein eigenes Testzentrum betreiben.

Kleine Unternehmen unterscheiden in der Regel nicht zwischen Entwicklern und Testern. Häufig testen Entwickler die von ihnen entwickelten Komponenten selbst und nehmen anschließend auch die Integration mit weitere Komponenten der Software vor. Erst bei größeren Projekten werden Rollen festgelegt. Der gängige Ansatz ist hierbei, dass Entwickler wechselseitig auch die Rolle des Testers übernehmen und die Arbeit ihrer Kollegen testen. Das Testen ist dabei niemals anonym, da dies zwar Vorteile bezüglich der Unbefangenheit bieten würde und durch die zusätzliche Einarbeitung eventuell mehr Fehler gefunden würden, die Einarbeitung aber in der Regel zu aufwendig ist. Erst bei größeren Projekten kommen Tester zum Einsatz, die keinen unmittelbaren Kontakt zum Entwickler einer Komponente haben. Allerdings handelt es sich dabei auch weiterhin um ausgebildete Entwickler, nicht speziell geschulte Tester. Zum Teil finden übergeordnete Tests durch einen Teamleiter statt. Dieser achtet dabei weniger auf technische Details, als stärker auf das Einhalten von (Benennungs-) Konventionen und Richtlinien. Die einzige Testphase, die in der Regel nicht allein in der Hand der Entwickler liegt, ist der Abnahmetest, zu dem der Kunde hinzugezogen wird, für den die jeweilige Software entwickelt wird. Die Testverantwortung liegt zunächst beim Entwickler und geht dann mit dem Integrations- oder Systemtest auf den Team- oder Projektleiter über. Allerdings ist diese Verantwortlickeit nur wenig formal geregelt.

Größere Unternehmen unterhalten eine eigenes Testzentrum. Dessen Aufgabe ist in der Regel die Koordination von Tests, die Bereitstellung von Richtlinien und Empfehlungen sowie das Angebot diverser unterstützender Leistungen. Die genaue organisatorische Aufhängung dieser Einheit ist dabei ebenso unterschiedlich geregelt, wie ihr tatsächliches Befugnisspektrum. Ausgestaltungsparameter sind etwa der Verbindlichkeitscharakter von Richtlinien und Empfehlungen oder auch die Frage, ob Freigaben für abgeschlossene Testphasen erteilt werden können. Die Zentralisierung des Testens bzw. zumindest seiner Organisation sowie die Wahrnehmung einer Querschnittsfunktion ist allen Testzentren gleich. Allerdings unterscheidet sich ihre Einbindung von Testphase zu Testphase. So ist es bei größeren Unternehmen üblich, das Testen in mehrere Phasen zu unterteilen. Im Rahmen dieser Phasen liegt die Verantwortung zunächst beim Entwickler und geht dann entweder auf leitende Mitarbeiter oder das Testzentrum über. Entwickler sind daher in der Regel auch nur beim Komponenten- und eventuell beim Integrationstest an den Testaktivitäten beteiligt. Danach übernehmen dedizierte Tester dieser Aufgabe. Insgesamt ist das Testen bei größeren Unternehmen viel stärker als eigener Prozess etabliert und im Softwareentwicklungsprozess verankert. Verantwortlichkeiten sind klar geregelt und alle Phasen in hohem Maße strukturiert und festgelegt.

3.3. Integration in den Softwareentwicklungsprozess

Auch bezüglich der Integration in den Softwareentwicklungsprozess zeigt sich die bereits festgestellte Zweiteilung. Kleinere Unternehmen haben in der Regel keinen definierten Testprozess. Testen ist vielmehr eine Nebenaufgabe der Softwareentwicklung, die mehr oder weniger strukturiert in diesen Prozess eingearbeitet ist. Größere Unternehmen verfügen hingegen über einen mitunter sogar formal definierten Testprozess, der häufig in Form eines Stufenkonzepts (Staging) realisiert ist.

Da in kleineren Unternehmen häufig spontan und wenig formalisiert gearbeitet wird, lassen sich nur wenige generellen Beobachtungen ableiten. Der Testerfolg korreliert häufig mit der Fähigkeit des Entwicklers, seine Software zu testen. Zudem ist er nicht selten von externen Faktoren wie etwa Terminvorgaben und Budgetbeschränkungen abhängig. Das Testen hat sich also eher externen oder mitarbeiterbezogenen Rahmenbedingungen anzupassen, als das es einem festen Prozess folgt. Typisch für kleine Unternehmen ist auch eine an *Extreme Programming* (XP) [Bec00] angelehnte Vorgehensweise. Der Fokus liegt auf Komponententests, die täglich zum Einsatz kommen und zu täglich lauffähigen Builds führen. Der Kontakt zum Kunden ist stets hoch und mündet in einen umfangreichen Abnahmetests. Auch während der Entwicklung wird beim Kunden nachgefragt, falls sich offene Fragen ergeben oder potentielle Probleme offensichtlich werden. Die Testintensität ist dabei relativ gleichmäßig über die Projektdauer verteilt. Neben Tests, die dem selbst gewählten Vorgehen der Entwickler folgen, gibt es zum Teil noch Tests, in denen versucht wird, gegen die Geschäftsprozesse, die das System abbilden bzw. unterstützen soll, zu testen. Dabei sind dann Eingabefolgen mitunter bis ins Detail festgehalten. Aufgrund von Zeitknappheit werden allerdings nur für kritische befundene Geschäftsprozesse getestet. Wenn auch nicht formal geregelt, ist es normal, dass Fehler zum Teil erst nach Produktauslieferung behoben werden. Selbst, wenn Vorgaben für Testabläufe existieren, werden diese nicht zwangsläufig eingehalten. Müssen etwa Sicherheitsaktualisierungen (*Patches*) für bereits ausgelieferte Software erstellt werden, ist es nicht unüblich, Tests sehr zu beschränken. Aufgrund knappen Personals ist ein ausreichend schneller Tests in der zur Verfügung stehenden Zeit nicht möglich. Daher wird auf diesen verzichtet, oder er erfolgt erst nach Auslieferung des Patches. Das Risiko durch eine verspätete Auslieferung des Patches wird demnach für höher erachtet als das Risiko durch neue Fehler, die der Patch möglicherweise enthält.

Zum Teil gibt es bei den kleineren Unternehmen Ansätze, Testen stärker zu formalisieren oder Prozessschritte zu definieren. Typischen Problemen durch die mögliche Willkür der testenden Mitarbeiter wird mit Personalmaßnahmen begegnet. Ein gängiger Ansatz ist es etwa, Mitarbeiter hinsichtlich ihrer Fähigkeiten als Tester einzusetzen. Mit einem durchdachten Konzept können so zahlreiche Probleme durch fehlende Formalisierungen vermieden und gleichzeitig die Flexibilität eines kleinen Unternehmens genutzt werden. Dennoch wurde eine stärkere Prozessorientierung von den kleinen Unternehmen als wünschenswert identifiziert. Wenn möglich, würden diese gerne als erfolgreiche erkannte Vorgehensweisen verbindlich etablieren, aber gleichzeitig zu starre Prozesse vermeiden.

Die Softwareentwicklung bei größeren Unternehmen ist aller Regel nach projektbezogen aufgestellt und erfolgt fachlich getrieben. Sie ist dabei als Vorgehensmodell inklusive aller Schnittstellen abgebildet. Für jeden einzelnen Vorgang ist ein abstrakter Prozess definiert, der auch in der definierten Form zum Einsatz kommt. Ein *Umgehen* von Tests ist somit nicht möglich, zumal Unterprozesse häufig mit formalen Freigaben enden müssen. Das Testen wird zentral durch ein Testzentrum unterstützt und ist als Stufenkonzept abgebildet. Dieses orientiert sich am folgenden, dem V-Modell [DHM98] angelehnten Schema:

1. Komponententest
   Entwickler testen die von ihnen entwickelte Software. Das Vorgehen ist in der Regel inkrementell iterativ. Die Testverantwortung liegt beim Entwickler selbst, allerdings gibt es zum Teil Vorgaben bezüglich der anzuwendenden Verfahren oder Werkzeuge. Das Maß an Dokumentation ist gering. Da der Entwickler seinen Code kennt, sind Tests Glass Box-, bzw. gegebenenfalls Gray Box-Tests.

2. Zentraler Komponententest (optional)
   Komponententests finden nach wie vor weitgehend ohne Integration statt, allerdings nicht mehr auf dem System des Entwicklers. Vielmehr kommt eine Testinfrastruktur zum Einsatz. Der Entwickler wird eventuell von Testern unterstützt.

3. Integrationstest
   Integrationstests finden häufig bereits auf einer Testinfrastruktur statt. In der Regel ist der Entwickler für diesen Tests noch verantwortlich, aber das Testzentrum steht ihm bereits zur Verfügung oder ist sogar direkt an dem Tests beteiligt. Typisch ist auch, dass der Tests mit der Freigabe durch den Entwickler abgeschlossen werden muss. Er bestätigt damit, dass sich das von ihm entwickelte Modul reibungslos in das Gesamtsystem einfügt.

4. Performancetests auf Integrationsebene (optional)
   Mit Leistungstests einzelner Subsysteme soll ein erster Eindruck des Leistungsverhaltens des Systems gewonnen werden. Auch kann festgestellt werden, ob sich bereits Flaschenhälse im Zusammenspiel einzelner Komponenten zeigen oder Algorithmen aufgrund ihres Leistungsverhaltens ungeeignet sind.

5. Systemtest
   Systemtests sind deutlich weniger technisch geprägt als Integrationstests und umfassen zu ihrem Ende hin bereits das Gesamtsystem. Schnittstellen sind vorhanden und werden nicht mehr (durch Teststümpfe o. Ä.) simuliert. In der Regel ist nicht mehr der Entwickler für den Tests zuständig sondern es kommen dedizierte Tester zum Einsatz. Diese kennen den Quellcode nicht mehr, Tests sind also reine Black-Box-Tests. Der Grad der Dokumentation ist hoch. Auch die Menge der Vorgaben durch das Testzentrum ist in der Regel hoch. Mitunter können schon Mitarbeiter des Kunden oder des Vertriebs an dieser Testphase beteiligt sein. Der Systemtest schließt wieder mit einer Freigabe. Diese bestätigt, dass das System ein Qualitätsniveau erreicht hat, mit dem es dem Kunden vorgestellt werden kann.

6. Performancetests auf Systemebene (optional)
   Sobald alle wesentlichen Komponenten des Gesamtsystems integriert sind, können aussagekräftige Leistungsmessungen vorgenommen werden. Diese sind funktional getrieben und liegen ebenfalls nicht mehr in der Verantwortung einzelner Entwickler.

7. Abnahmetest
   Der (gegebenenfalls interne) Kunde wird zum Abnahmetests eingeladen. Ziel ist ein Test gegen die ursprüngliche Beauftragung. Der Test ist dementsprechend gröber als der Systemtest und stellt im Wesentlichen die Erprobung des Systems durch den Kunden dar. Wesentliche bzw. grundsätzliche Fehler sollten in dieser Phase nicht mehr gefunden werden. Auch diese Phase wird mit einer Freigabe abgeschlossen. Freigaben können auch eingeschränkt sein und Auflagen zur Nachbesserung beinhalten.

8. Pilottest (optional)
   Wenn der Kunde eine interne Abteilung ist, die Releases eines Produktes für den Markt koordiniert, so ist ein Pilottest mit ausgewählten Marktkunden denkbar. Ziel dabei ist es, letzte verbliebene Fehler, die sich nur im praktischen Einsatz offenbaren, auszumerzen und den Bedienkomfort überprüfen zu lassen. Bei Individualsoftware ist ein Pilottest als Installation der Software beim Kunden für einen zunächst eingeschränkten Nutzerkreis denkbar. Der Pilottest kann auch wieder eine Phase für Leistungsmessungen erhalten, etwa dann, wenn vorherige Leistungsmessungen mit simulierten Daten nicht realistisch genug waren.

9. Produktiver Einsatz

10. Wartung (optional)
    Gegebenenfalls schließt sich noch die Wartung des Systems an. Die oben skizzierten Testphasen können dabei wiederum Bestandteile des Wartungsprozesses sein. Dies gilt vor allem dann, wenn die Wartung als periodischer Prozess betrachtet wird, der zu neuen Releases führt.

Der skizzierte Prozess kann noch umfangreicher und feingliedriger ausfallen. Das obere Schema gibt aber grob das in der Regel in größeren Unternehmen anzutreffende Vorgehen wieder. Zum Teil sind noch weitere zentrale Einheiten involviert, wie etwa ein Release-Management. Auch können sich Abhängigkeiten deutlich komplexer gestalten.

3.4. Umfang der Tests

Der Umfang der vorgenommenen Tests richtet sich vor allem nach dem Grad seiner Formalisierung. Kommt ein Stufenkonzept zum Einsatz, wie es im vorangegangenen Kapitel gezeigt wurde, ist der Umfang der Tests in der Regel hoch, da auch Mindestanforderungen an das Testen festgelegt wurden und zumindest für spätere Phasen genaue Vorgaben

bezüglich der zu wählenden Vorgehensweisen und der zu benutzenden Werkzeuge existieren. Auch führt die Einführung formaler Freigaben dazu, dass der Umfang der Tests steigt, da nur so das jeweils benötigte Qualitätsniveau erreicht werden kann. Der Umfang der Tests ist dementsprechend als hoch einzustufen, wenn die Unternehmen groß sind.

Bei den kleineren Projektteilnehmern gestaltet sich der Umfang der Tests deutlich variabler. Er hängt sowohl davon ab, welches Qualitätsniveau und damit welcher Testumfang mit dem jeweiligen Kunden vereinbart wurde, als auch von Kapazitäten und Terminen. Ohne, dass sich dies unmittelbar quantifizieren ließe, ist allerdings eindeutig festzustellen, dass der Umfang der durch größere Unternehmen durchgeführten Tests wesentlich größer ist. Zum Teil wird zudem von den kleineren Unternehmen versucht, das Testen durch eine ausgefeilte Anforderungsanalyse zu substituieren. Ziel ist es, Probleme dadurch zu vermeiden, dass fehlerhafte oder ungenaue Anforderungen in Form von Programmfehlern oder Schwächen in der Bedienbarkeit Einzug in die Software finden. Allen Unterschieden zum Trotz berichtete nur eines des teilnehmenden Unternehmen von Kritik wegen nicht ausreichend getesteter Produkte. Das Tests zu knapp ausfallen können, war dem Unternehmen allerdings bekannt; zu knappe Fertigstellungstermine, die durch weisungsbefugte interne Kunden gesetzt wurden, ließen aber kaum eine Alternative zu.

3.5. Testarten

Die folgenden Unterkapitel stellen den Einsatz verschiedener Testarten zusammen. Dabei wird zunächst erörtert, in welchem Ausmaß Tests als Glass Box- bzw. als Black Box-Tests durchgeführt werden. Ebenso wird der Status quo der Testautomatisierung festgehalten. Erst dann wird die Verbreitung und Ausgestaltung einzelner Testarten betrachtet. Sie folgt dabei grob dem oben vorgestellten Stufenkonzept. Auch wenn dieses nur für größere Unternehmen in der statischen Form gilt, so lassen sich auch bei kleineren Unternehmen Testarten anhand ihres Charakters unterscheiden. Als Abschluss des Kapitel wird die Nutzung von Regression beleuchtet.

3.5.1. Glass- und Black-Box

Eine explizite Unterscheidung von Glass Box- und Black Box-Tests findet in der Regel nicht statt; die Begriffe tauchen in diesen Form in internen Prozess- und Methodenbeschreibungen nicht auf. Allerdings erfolgt eine implizite Aufteilung: Komponententests durch Entwickler sind Glass Box-Tests, da die Entwickler die von ihnen erstellte Komponente unter Kenntnis der Quelltexte und mit explizitem Fokus auf denselben testen. Einige Tests können dabei auch Gray Box-Tests sein: Zwar verwendet der Entwickler Testtechniken, die der Kategorie der Black Box-Tests zuzuschreiben sind, allerdings kann er dabei kaum sein Wissen um die Gestalt des einer Funktion zugrunde liegenden Quellcodes ausblenden. In den späteren Phasen sind Tests dann immer reine Black Box-Tests, auch wenn es durch die Beteiligung von Entwicklern zu gewissen „Aufweichungen“ kommen kann, etwa wenn Tester und ein Entwickler mit Quellcode-Kenntnis gemeinsam arbeiten. Gerade bei genau definierten Prozessen kann es aber üblich sein,

reine Black-Box-Tests explizit zu fordern. Gefundene Fehlern werden dokumentiert und dem Entwickler gemeldet. Er hat diese zu beheben; Tester kommen mit dem Quelltext nicht in Berührung.

In einigen Fällen ist es üblich, dass spezialisierte Tester Vorgespräche mit Entwicklern führen, um die Art der zu wählenden Teststrategie und -methode besser Abschätzen zu können. Ob in späteren Phasen zusätzliche Code-Inspektionen durch Tester stattfinden können, ist bei keinem der Projektteilnehmer formal geregelt.

3.5.2. Grad der Automatisierungen

Im Hinblick auf die Automatisierung von Tests zeigt sich ein homogenes Bild: Das manuelle Testen ist in allen Phasen des Testens und bei allen Projektteilnehmern dominant. Testautomatisierungen kommen nur in einigen Fällen zum Einsatz und selbst Teilautomatisierungen sind selten.

Der niedrige Grad der Automatisierung ist für die Autoren dieser Broschüre überraschend. In der Literatur wird die Automatisierung als lohnenswertes Ziel angepriesen und auch Werkzeughersteller werben damit, dass Automatisierung sehr zur Senkung des Testaufwands geeignet ist. Die zum Einsatz kommende Automatisierung bezieht sich in der Regel auf die einfache Wiederholung von Komponententests. Testfälle sind dabei direkt im Quellcode spezifiziert und stellen streng genommen eigene kleine Programme dar. Ein typisches Beispiel hierfür sind Testfälle, die mit dem bekannten JUnit [13] erstellt wurden. Solche Testfälle lassen sich beliebig oft wiederholen. Am Ende eines Testlaufs kann ausgegeben werden, welche Testfälle nicht erfolgreich verliefen. Zum Teil kommen Verfahren zum Einsatz, bei denen per Capture&Replay-Werkzeug manuelle Tests „aufgenommen“ wurden und dann – z. T. parametrisiert – wiedergegeben werden. Dieses Vorgehen fällt allerdings eher unter das Regressionstesten als unter echt Automatisierung.

Für den Verzicht auf Automatisierung wurden verschiedene Gründe genannt:

- Kleine Unternehmen scheuen Ausgaben für umfangreiche Testwerkzeuge. Diese müssten vom Auftraggeber bereitgestellt oder finanziert werden, wozu dieser allerdings nicht bereit ist. Dies deckt sich mit den Angaben größerer Unternehmen, die die Kosten von Werkzeugen auf bis zu siebenstellig bezifferten.

- JUnit-Test können aufgrund technischer Restriktionen nicht ohne weiteres in automatische Build-Prozesse aufgenommen werden.

- Von der zu testenden Software benötigte Systeme können zum Testzeitpunkt nicht eingebunden und auch nicht simuliert werden. Für verschiedene Aufgaben müsste das gesamte Produktivsystem simuliert werden, was für nicht praktikabel erachtet wird.

- Der initiale Aufwand für das Erstellen automatisierter Tests wird für zu hoch angesehen.

- Aufgrund starker Veränderungen von Release zu Release einer Software wird eine Automatisierung nicht für sinnvoll erachtet.[1]
- Es ist nicht bekannt, wie sich „stumpfes Durchklicken“ intelligent automatisieren ließe.
- Regression wird als ausreichend erachtet, die semi-automatische Wiedergabe bereits verwendeter Testfälle sei dabei bereits zielführend.
- Verschiedene Systeme erzeugen als Ausgabe papiergebundene Dokumente. Eine automatisierte Testmöglichkeit ist unbekannt.
- Zum komplett automatisierten Testen müssten genutzte Datenbestände anonymisiert werden, um Risiken auszuschließen. Dies gilt als zu aufwendig.
- Die bezüglich der Kosten der Automatisierung gesammelten Erfahrungen lassen den Schluss zu, dass sie das Ergebnis der Automatisierung nicht rechtfertigen.

Zum Teil wird der geringe Automatisierungsgrad bedauert, zum Teil aber auch als bloße Tatsache aufgefasst. Die meisten Unternehmen wünschen sich allerdings verbesserte Möglichkeiten zur Testautomatisierung. Häufig sind diese Anforderungen den Anforderungen an Regressionstests ähnlich (siehe unten). Die oben aufgeführten Gründe lassen zwei grundsätzliche Schlüsse zu. So scheinen Werkzeuge zur Automatisierung nach wie vor auf erhebliche Grenzen zu stoßen oder wenig intuitiv bzw. wirtschaftlich bedienbar zu sein. Gleichzeitig aber scheint auch das Wissen um Möglichkeiten der Automatisierung knapp zu sein.[2] Die Bereitschaft, neue Werkzeuge auszuprobieren, fällt gering aus, selbst wenn es sich um freie Software handelt, deren Evaluation nur Arbeitsaufwand verursachen würde. Näheres hierzu ist auch im weiter unten folgenden Kapitel 3.6 zu finden.

3.5.3. Komponententests

Kompontententest sind im Allgemeinen die ersten Tests, denen Teile einer zu entwickelnden Software unterzogen werden. Komponenten beschreiben die kleinsten zusammenhängenden und nicht sinnvoll trennbaren Einheiten eines Softwaresystems. Bei objektorientierten Programmiersprachen ist dies etwa die Klasse. Noch feingranularere Tests sind in der Regel weder praktikabel noch sinnvoll. Zudem kann frei festgelegt werden, ob erst komplett fertig gestellte Klassen getestet werden, oder ob Tests iterativ während der Entwicklung derselben erfolgen. Letzterer Ansatz war vor allem bei kleineren Unternehmen anzutreffen, da diese häufig an Verfahren wie Extreme Programming angelehnte iterative Softwareentwicklung betreiben.

[1] Dieser Grund spricht streng genommen nicht gegen die Automatisierung, sondern gegen Regressionstests. Im Rahmen der Analyse in dieser Broschüre wird er dennoch an dieser Stelle aufgenommen.

[2] Diese Aussage ist zunächst wertneutral und impliziert nicht etwa, dass ein Versäumnis aufgedeckt wurde. Die Autoren waren vielmehr verwundert über die Diskrepanz theoretisch verfügbarer Möglichkeiten und deren praktischer Nutzung. Hieraus ergibt sich über diese Broschüre hinaus Forschungsbedarf bezüglich der Eignung von Werkzeugen zur Testautomatisierung.

Bis auf zwei Ausnahmen nutzen alle Projektteilnehmer Komponententests. Bei den größeren Unternehmen sind sie formal vorgeschrieben und schließen zum Teil sogar mit einer Freigabe durch den Entwickler ab. Bei den kleineren Unternehmen fehlt dieses formale Vorgehen. Entwickler testen die von ihnen entwickelten Komponenten im Wesentlich nach eigenem Ermessen und ohne zentrale Kontrolle. Der Fokus der Tests ist technisch; anforderungsbezogenes Testen auf Ebene der Komponenten ist die Ausnahme. Insbesondere sind Tests der Oberfläche selten, es sei denn, die Oberfläche stellt sich selbst als kapselbare Komponente dar. Vereinzelt wurden Probleme mit der Kapselung von Komponenten berichtet. In diesen Fällen fielen Komponententests sehr knapp aus und die Integrationstests wurden umfangreicher.

Der Verzicht auf Komponententests bei zwei Unternehmen hat jeweils unterschiedliche Gründe und ist höchstwahrscheinlich nicht als repräsentativ zu werten. Im ersten Fall wird Software fast grundsätzlich durch Dritte entwickelt und dann in die eigene Systemlandschaft integriert. Zur Zeit des Übergang in das Unternehmens war sie dementsprechend auf Komponentenebene längst getestet. Im zweiten Fall liegt eine technisch verwaltete Entwicklungsumgebung bzw. Programmiersprache vor, die aufgrund von Restriktionen und der tiefen Verankerung in der für den Massenmarkt entwickelten Software nicht ohne weiteres gewechselt werden kann. Einzelne Komponenten lassen sich nur unzureichend abgrenzen, da die entwickelte Software sich grundsätzlich als monolithisch darstellt. Somit lassen sich nur dann Test auf dieser Ebene durchführen, wenn sich Bereiche abgrenzen lassen. Allerdings kommen noch weitere Probleme hinzu, die über das reine Testen hinausgehen. So nannte der Projektteilnehmer einen schlechten Debugger, eine wenig leistungsfähige IDE und die daraus folgende fehlende Syntaxkontrolle als weitere Schwierigkeiten. Als Problemlösung wird derzeit aber nur eine Weiterentwicklung durch den Hersteller der Entwicklungsumgebung angesehen, bzw. die Unterstützung bei der Portierung der bestehenden Software auf ein modernes Programmierparadigma.

3.5.4. Integrationstests

Für die größeren Unternehmen gehören Integrationstests ebenfalls zum üblichen und formal geregelten Vorgehen. Häufig kommt dabei sogar eine spezielle Infrastruktur zum Einsatz, die etwa die Simulation weiterer Systeme ermöglicht und die Dokumentation der Testläufe unterstützt. Bei kleineren Unternehmen herrscht eine noch geringere Verankerung der Integrationstests als dies für die Komponententests gilt. Denn erste gelten – wenn auch nicht verbindlich geregelt – als obligatorisch. Dass ein Entwickler keine Tests durchführt, kann als ausgeschlossen gelten. Explizite Tests zur Integration von Komponenten fehlen aber in der Regel. Gegebenenfalls werden sie ebenfalls von den Entwicklern vorgenommen, wenn diese mehrere Komponenten fertig gestellt haben und deren Zusammenspiel ausprobieren. Bezüglich der Entwicklung von Sicherheitsaktualisierungen (Patches) wird zum Teil auch von den größeren Unternehmen auf Integrationstests verzichtet. Die geänderten Komponenten werden schnellstmöglich isoliert getestet und dann eingespielt bzw. für die Kunden bereitgestellt. Etwaige Probleme zeigen sich dann erst im Betrieb. Gegebenenfalls folgt aber ein nachträglicher Test, der in einen erneuten Patch münden kann, falls integrative Probleme auftreten.

3.5.5. Systemtests

Während Komponenten- und Integrationstests bei kleinen Unternehmen häufig zusammenfallen und vor allem letztere nicht vorgeschrieben sind, sind Systemtests bei den meisten Unternehmen Standard. Nur im Falle einer sehr stark iterativen Entwicklung, die hauptsächlich auf Komponententests setzt und bei der das Gesamtsystem sukzessive wächst, wird auf explizite Tests des Gesamtsystems verzichtet. Bei größeren Unternehmen ist der Formalismus in der Systemtestphase sehr hoch; es gibt zahlreiche Vorgaben und umfangreiche Dokumentationen werden erstellt.

Im Allgemeinen sind Systemtests wenig technisch geprägt. Die zu entwickelnde Software wird hinsichtlich ihrer Aufgabenstellung und der festgehaltenen Anforderungen geprüft. Tests sind demnach häufig Oberflächentests, die aber durchaus noch algorithmische Fehler aufspüren können. Der Aufwand für die Systemtests ist mitunter sehr hoch. So ist es durchaus üblich, Mitarbeiter aus Fachbereichen oder Vertrieb zum Test hinzuzuziehen, um durch unterschiedliche fachliche Sichten die Chance zu erhöhen, Fehler zu finden. Auch finden zum Teil Tests mit anonymisierten Produktivdaten statt (siehe dazu auch die Empfehlungen weiter unten). Insbesondere bei größeren Unternehmen kommt zudem eine spezielle Testinfrastruktur zum Einsatz, die analoge zum System für Integrationstests zu sehen ist, aber eine höhere Komplexität aufweist. Ziel sind möglichst realistische Tests. Dort, wo Software in regelmäßigen Releases entwickelt wird, kann der Systemtests zu einem jährlichen oder halbjährlichen Zyklus gehören und stellt dann die umfangreichste Überprüfung des aktuell zur Veröffentlichung anstehenden Releases dar.

3.5.6. Beta-, Pilot- und Abnahmetests

Abnahmetests sind bei fast allen Projektteilnehmern verbindlich vorgeschrieben und haben in der Regel eine hohe Bedeutung. Ziel des Abnahmetests ist dabei eine letzte funktionale Überprüfung des Softwaresystems, bei dem vor allem auf die Erfüllung der gestellten Anforderungen sowie auf die Bedienbarkeit geachtet wird. Zum Teil sind dem Abnahmetests eigene Mitarbeiter zugewiesen. Häufig sind Kunden in die Tests involviert, um – im wahrsten Wortsinne – die Akzeptanz der entwickelten Software durch diesen überprüfen zu können. Die Einbindung des Kunden ist vor allem auch für kleine Unternehmen wichtig, bei der die Entwicklung ohnehin sehr vom engen Kontakt zum Kunden geprägt ist. Auf Akzeptanztests wird letztlich nur dann verzichtet, wenn Software für den eigenen Bedarf entwickelt oder eingekauft und die Nutzung per Geschäftsanweisung vorgegeben wird. In diesem Fall erfolgen nur Akzeptanztests in der Form von Leistungsmessungen, um etwaige Bedienschwächen durch inakzeptable Reaktionszeiten und Ähnliches auszuschließen.

Im Rahmen von Akzeptanztests benötigen Entwickler bzw. Tester mitunter Branchenwissen, um erfolgreich testen zu können. Wichtig ist auch eine umfassende Dokumentation der Ergebnisse. An den Akzeptanztests schließt bei einigen Projektteilnehmern auch die formale Freigabe durch Vertreter des Kunden oder die beauftragende Fachabteilung an. Von vielen Projektteilnehmern wurde in diesem Zusammenhang auch eine frühe Einbindung des Kunden bzw. der Fachabteilung(en) betont, die der Akzeptanz

sehr zuträglich ist und somit Akzeptanzprobleme weitgehend verhindert.

Pilottests werden zwar von einigen Projektteilnehmern angewendet, sind aber deutlich seltener als Abnahmetests. Bei diesen werden ausgewählte Kunden oder Fachabteilungen mit im Wesentlichen fertig gestellter Software ausgestattet, um diese im produktiven Betrieb zu testen. Dabei können letzte Fehler gefunden und vor allem auf Bedienschwächen reagiert und zusätzliche Wünsche berücksichtigt werden. Pilottest dienen somit der Risikominimierung. Ziel ist ein vollständig „abgerundetes“ Produkt. Zum Teil existieren Kernanwender, die etwa im Rahmen einer regelmäßig aktualisierten Software jeweils als erste mit einem neuen Release arbeiten, welches der allgemeine Anwenderkreis erst zeitverzögert erhält.

Beta-Tests stellen eine Art Pilottest für eine größere Zahl Anwender dar und sind zum Teil sogar öffentlich. Die zu testende Software ist dabei zwar weitgehend benutzbar, die Zahl der noch erhaltenen Fehler aber aller Regel nach deutlich größer als bei einzelnen Pilottests. Dies ist zum Teil durch die hohe Komplexität typischer Software für den Massenmarkt bedingt. In der Vergangenheit sind Beta-Tests allerdings in der Verruf geraten, Hersteller ihrer Pflichten zur Qualitätssicherung zu entbinden und diese Aufgabe auf Kunden zu übertragen. Aufgrund der Zielrichtungen der Softwareentwicklung werden Beta-Tests von den Projektteilnehmern auch nur selten eingesetzt. Sie wären vor allem für Software für den Massenmarkt interessant. Zwar bieten einige der Projektteilnehmer Software auf dem Markt an und zielen auf größere Nutzerzahlen ab. Die Form eines anonymen Verkaufs in sehr großen Stückzahlen spielt allerdings kaum eine Rolle. Daher sind Beta-Tests wenig nahe liegend, was ihre geringe Verbreitung erklären dürfte.

3.5.7. Tests der Datenhaltung bzw. -bank

Neben eine Einteilung in vier grundsätzliche Phasen des Testens und der Klärung, welche Bedeutung diesen von den Projektteilnehmern beigemessen wird, sollte auch untersucht werden, ob es explizite Tests der Datenhaltung von Softwaresystemen gibt. Einerseits implizieren Tests datengetriebener Applikationen auch Tests der darunter liegenden Datenhaltungsschicht. Andererseits sind Fehler durch explizite Tests deutlich einfacher zu erkennen.

Nur bei einem der Teilnehmer sind explizite Tests der Datenhaltung obligatorisch. Da bei diesem der Übergang der Daten auf Datenbanken als Schnittstellenfunktion angesehen wird, erfolgen Tests der Datenhaltung zusammen mit Schnittstellentests. Zu diesen Tests gehören ferner Konsistenzüberprüfungen der Übertragung von Daten zu Drittsystemen. Diese finden sogar unternehmensübergreifend statt. Erwartungsgemäß charakterisierte der Projektteilnehmer diese Tests für ihr als „sehr wichtig“. Transaktionsbezogene Fehler sind in seinem Umfeld inakzeptabel.

Alle anderen Projektteilnehmer verzichten auf explizite Tests. Sie erfolgen höchstens im Rahmen von Komponententests, sind dort aber dem Entwickler und seiner Einschätzung der Notwendigkeit überlassen. Die Projektteilnehmer betonten, dass in moderner Software die Datenhaltung oftmals sehr transparent ist; zwar ist eine Schicht anzusprechen, die Daten liefert und etwa auch für Persistierung sorgen kann. Ein Durchgriff auf die zugrunde liegende Datenbank ist aber nicht möglich. Dies gilt sowohl für Stan-

dardsysteme, etwa SAP, wie für Software, die moderne mehr-Schichten-Architekturen nutzt und z. B. auf Rahmenwerke (*Frameworks*) wie EJB, Spring oder ADO.NET zurückgreift.

3.5.8. Leistungsmessungen, Last- und Stresstests

Eine weitere Testart, deren Einsatz explizit erfasst werden sollte, sind Last- und Stresstests bzw. Performancemessungen. Allen ist gemeinsam, dass sie nicht die Funktionen eines Systems testen, sondern dessen Leistungsverhalten. Im Rahmen dieser Tests können verschiedene Schwächen offenbart werden. Dazu zählen *Flaschenhälse*, ein unbefriedigendes Leistungsverhalten, ein inakzeptables Verhalten bei Überlastung sowie Probleme in der Bedienbarkeit. Dementsprechend lassen sich Rückschlüsse auf Programmierfehler, falsch verstandene Anforderungen oder sogar wenig durchdachte bzw. sich als ungeeignet erweisende Anforderungen ziehen.

Die Ergebnisse fallen heterogen aus. Zwar nehmen die meisten Unternehmen Leistungsmessungen vor, allerdings ist die Formalisierung diesbezüglich wenig ausgereift. Explizite Stresstests, deren Ziel es ist, das zu testende System bewusst auf seine Grenzen zu testen und z. B. mit Anfragen zu überlasten, sind bei keinem der Projektteilnehmer obligatorisch. Auch erfolgt in der Regel keine Unterscheidung zwischen Maßnahmen, die der Leistungsmessung dienen und expliziten Lasttests. Auffällig war auch, dass nur selten Werkzeuge zum Einsatz kommen und die empfundene Notwendigkeit von Tests generell offenbar (noch) nicht sehr hoch ist.

Bezüglich der Leistungsmessungen wurden verschiedene Strategien angetroffen:

- Leistungsmessungen nach Auslieferung einer Software an den Kunden: Das grobe Leistungsverhalten wird aufgrund der Anforderungsanalyse abgeschätzt. Aufgrund der typischen Komplexität der vom Kunden im System zu speichernden Daten wird auf Tests bei der Entwicklung verzichtet. Vielmehr müssen Entwickler aufgrund ihrer Erfahrung potentielle Probleme antizipieren. Bei Bedenken bezüglich des Leistungsverhaltens kommt beim Kunden ein Prototyp zum Einsatz, der gegebenenfalls auch einem Stresstests unterzogen wird.

- Leistung wird aufgrund von in die Software eingebauten Messroutinen während des Betriebs überprüft. Dies ist bei der Anpassung großer Standardsysteme wie etwa SAP bereits vorgesehen; für von Grund auf neu erstellte Software existieren häufig geeignete Frameworks und Überwachungsprogramme.[3]

- Funktionale Tests werden so terminiert, dass zahlreiche Tester mehrere Testläufe parallel durchführen. Dies erzeugt eine simulierte Last auf dem System. Die fehlende Isolierung verschiedener Testarten wird dabei in Kauf genommen.

[3] Für in Java geschrieben Programme etwa lassen sich so genannte MBeans erstellen, die bestimmte Leistungsdaten messen oder Verhalten protokollieren können. Diese Daten können dann mit dem im JDK von Sun enthaltenen Werkzeug JConsole dargestellt werden.

- Leistungsmessungen werden nur nach expliziter Anforderung durch den Kunden vorgenommen.

- Es werden isolierte Vorgänge auf der entwickelten Software gemessen. Daraus werden Hochrechnungen erstellt und das Verhalten des Gesamtsystems extrapoliert.

- Leistungsmessungen werden in funktionale Tests mit Vergangenheitsdaten integriert. Dabei werden Testläufe einfach wiederholt und das Verhalten des Systems beachtet. Hierfür sind – verhältnismäßig einfache – Werkzeuge für Batchläufe geeignet. Bei Software, die in regelmäßigen Releases erscheint, wird der Kundenkontakt beibehalten und regelmäßig Evaluationen der Leistung vorgenommen. Daraus werden Rückschlüsse für Anpassungen des aktuellen Releases gewonnen.

Von mehreren Projektteilnehmern wurde darauf hingewiesen, dass für sie weniger technische Leistungsdaten eine Rolle spielen, als vielmehr das „Gefühl“ ihrer Kunden. Diese würden sich auch nicht auf von Werkzeugen gemessene und „garantierte“ Leistungsverhalten verlassen, sondern sich selbst ein Bild machen. Schließlich sei bei der Arbeit am Bildschirm der Eindruck wichtiger als das tatsächliche Leistungsverhalten des Systems. Dementsprechend sind Rückmeldungen der Benutzer sehr wichtig und auch Auslöser für Versuche der Leistungsverbesserung. Von einem Projektteilnehmer kam zudem der Hinweis, dass eine schlechte Leistung der Benutzerschnittstelle aufgrund rechtlicher Rahmenbedingungen problematisch sind, etwa in Bezug auf die Bildschirmarbeitsplatzverordnung (hierzu mehr im Anhang dieser Broschüre). Eine Software, deren Leistungsverhalten eine ergonomische Nutzung nicht zulässt, kann also rechtlich problematisch sein.

3.5.9. Nutzung von Regression

Die Entwicklung von Software ist nicht statisch. Komponenten werden häufig während der Entwicklung angepasst, selbst, wenn sie als fertig gestellt galten. Möglicherweise werden zusätzliche Komponenten eingefügt, und das Zusammenspiel im System ändert sich. Folglich kann es zu jeder Phase der Entwicklung passieren, dass bereits getestete Funktionen nicht mehr korrekt arbeiten. Die Zusammenhänge dabei sind äußerst komplex. Wird etwa eine Funktion angepasst und durchläuft danach alle für sie ausgelegten Tests fehlerfrei, muss das Verhalten des Gesamtsystems nicht gleich bleiben. Vielmehr wäre es möglich, dass andere fehlerhafte Komponenten erst jetzt ihr tatsächliches Verhalten zeigen (etwa, weil die geänderte Komponente grundsätzlich ähnliche Ergebnisse wie vorher liefert, aber eine höhere Präzision aufweist).

Bei den Projektteilnehmern, die Software entwickeln, welche in regelmäßigen Releases erscheint, sind Regressionstests obligatorisch. Ein gängiger Ansatz dabei ist, bei kompletten Releases alle bisherigen Testfälle zu wiederholen. Bei zwischenzeitlichen Aktualisierungen der Software, vor allem bei Patches, erfolgt nur eine teilweise Regression. Der Aufwand für komplette Regressionstests ist sehr hoch und kann bei umfangreichen Softwaresystem einen Zeitraum von einem Monat deutlich übersteigen. Während die

auf Releases bezogenen Regressionstests der Phase des System- oder Abnahmetests zuzuschreiben sind, gibt es zum Teil auch bei Integrationstests Regression, vor allem bei größeren Freigaben. Der hohe Aufwand der Regressionstests erklärt sich dadurch, dass er weitgehend manuell vorgenommen wird.

Auch bei kleineren Projekten kommen Regressionstests zum Einsatz. Diese sind aber häufig nicht formal vorgeschrieben. Es ist dabei unüblich, alle bisherigen Testfälle zu wiederholen, da der Aufwand zu hoch wird. Das dies problematisch sein kann, wurde von den Projektteilnehmern zwar festgestellt, Lösungsansätze existieren aber nur rudimentär. So werden Testfälle z. B. alternierend in aufeinander folgenden Regressionstests ausgeführt, so dass nach einigen Regressionsphasen alle ursprünglichen Tests noch einmal zur Ausführung gekommen sind. Um eine Frustration der Tester durch repetitives Testen zu verhindern, kommen diese rollierend zum Einsatz. Zum Teil wird Regression auch nur dort eingesetzt, wo erwartungsgemäß viele Fehler auftreten bzw. durch wo die Wahrscheinlichkeit des Auftretens neuer Fehler durch Änderungen im System besonders hoch ist. Dies gilt etwa für Schnittstellen.

Erfahrungen mit der Automatisierung von Regressionstests haben die meisten Projektteilnehmer gesammelt, sie fielen aber in der Regel negativ aus. So ist der Aufwand für das Erstellen der Regressionstests häufig hoch. Durch Änderungen von Release zu Release bzw. während der Entwicklung werden zahlreiche Testfälle ungültig, so dass der Gesamtaufwand für manuelles Testen als geringer erachtet wird. Als besonders problematisch werden Oberflächentests beschrieben. Auch mit modernen Werkzeugen bereiten geänderte Dialoge offenbar so erhebliche Schwierigkeiten, dass auf manuelle Tests zurückgegriffen wird. Ein Projektteilnehmer beschrieb einen Ausweg aus diesem Dilemma, zu dem aber keine technische Lösung bekannt ist: Würde zur Gestaltung von Programmoberflächen ein Framework verwendet, das sich automatisch mit Testwerkzeugen synchronisierte, bzw. stünde eine abstrakte Sprache bereit, die eine einheitliche Beschreibung von Programmoberflächen erlaubte, würden Änderungen an der Programmoberflächen die Durchführbarkeit von automatisierten Regressionstests nicht mehr einschränken.[4] Über die technischen Schwierigkeiten hinaus stelle die Beherrschung von Veränderungen im Hinblick auf Regressionstests auch organisatorisch eine Herausforderung dar.

Ein Projekteilnehmer erprobt derzeit gezielt den Aufbau automatisierter Regressionstests. Das dazu zum Einsatz kommende Werkzeug ist *Rational Functional Tester* [15][16] von IBM. Regressionstests werden dabei in eine Skriptsprache beschrieben. Auch wenn die Erfahrungen grundsätzlich positiv sind, erscheint der Gesamtvorgang für den Projektteilnehmer nach wie vor als sehr kostspielig. So musste zunächst Wissen über das Skripting aufgebaut werden. Die damit betreuten Mitarbeiter müssen zudem fachliche Einblicke in die zu testende Software haben. Die dafür nötige Abstimmung mit Entwicklern ist zeitintensiv. Auch müssen automatisierte Testläufe betreut und fachlich abge-

[4] Nach dem Kenntnisstand der Autoren dieser Broschüre existieren Ansätze, wie etwa Windows Forms [14] oder spezielle Modellierungssprachen für Programmoberflächen. Allerdings ist eine Möglichkeit der (idealerweise generellen, herstellerübergreifenden) Koppelung mit Testwerkzeugen unbekannt. Eine solche Koppelung bzw. Strategien, Änderungen der Entwickler automatisiert in Regressionswerkzeuge einfließen zu lassen, stellen einen aktuellen Forschungsbedarf dar.

nommen bzw. freigegeben werden. Zu dem eigentlichen Zeitaufwand für das Skripten kommt eine Auswahlphase, in der ermittelt wird, welche Testfälle sich überhaupt dafür eignen. Ziel dabei ist immer eine generelle Arbeitserleichterung. Als Idealziel wird vom Projekteilnehmer genannt, dass zusätzliche Testzyklen eingeführt werden könnten, bei denen das Testen komplett automatisiert abläuft. Fehler könnten so früher erkannt und einfacher behoben werden. Gleichzeitig stellte er eindeutig fest, dass seiner Einschätzung nach viele Tests stets manuell bleiben würden.

3.6. Einsatz von Werkzeugen

Ein explizites Ziel des dieser Broschüre zugrunde liegenden Projekts ist die Untersuchung des Werkzeugeinsatzes für das Testen. Die Ergebnisse des Interviews mit den Teilnehmern hinsichtlich ihrer Nutzung von Werkzeugen bzw. der Gründe für ein Verzicht auf diese werden in den folgenden Unterkapiteln dargelegt.

3.6.1. Generelles

Die Literatur empfiehlt explizit den Einsatz von Testwerkzeugen. Die Hersteller kommerzieller Werkzeuge preisen ihre Lösungen als umfangreich, einfach zu bedienen und sehr leistungsfähig an. Darüber hinaus existiert eine sehr große Vielzahl frei verfügbarer Werkzeuge für die verschiedenen Einsatzgebiete beim Testen von Software. Die Autoren dieser Broschüre hatten daher erwartet, mit den Projektteilnehmern die Frage des *wie* bzw. *welches* zu diskutieren, und nicht die Frage des *ob*. Die Feststellung eines der Projektteilnehmer, die „Frage, wann sich Tools lohnen können“, sei „ungeklärt“, drückt aus, was für das gesamte Teilnehmerfeld gilt: Der Einsatz von Werkzeugen erfolgt zurückhaltend und zögerlich, obwohl gleichzeitig der Wunsch nach mehr unterstützender Software besteht.

Bezüglich den verfügbaren bzw. bekannten Werkzeugen wird eine Reihe von Problemen genannt. Zum Teil wird auch von im Einsatz erlebten Schwierigkeiten berichtet:

- Trotz umfangreicher Evaluation seien keine wirklich zufriedenstellenden Werkzeuge gefunden worden.
- Der Einsatz von Werkzeugen ziehe viel manuelle Arbeit hinter sich, so dass insgesamt kein Vorteil entstehe, selbst wenn die Werkzeuge akzeptabel seien. Auch der hohe Aufwand der Prüfung der Ergebnisse der Arbeit von Werkzeugen wurde moniert.
- Die individuelle Softwareentwicklung werde in gängigen Testwerkzeugen nicht ausreichend abgebildet.
- Eine Werkzeugunterstützung scheitere an der Komplexität der zu entwickelnden Software. Bei jeglichen Änderungen müssten im Werkzeug hinterlegte Test angepasst werden. Häufig spiele auch die Abgrenzbarkeit einzelner Systeme eine Rolle, da sie sich nur schwierig isoliert testen ließen.

- Werkzeuge haben Schwierigkeiten mit variablen Inhalten wie etwa Datum und Uhrzeit. Während in einfachen Fällen ein Ausblenden solcher Daten ausreicht, sind intelligente Konzepte nötig, wenn die Testergebnisse von der Auswertung solcher Daten abhängen. Ein Beispiel sind Testwerte, die sich in Abhängigkeit von der Zeit ändern (sollen).

- Die eigentliche Qualität der Test sei nicht besser, denn sie hänge von den Testern ab.

- Bei der Entwicklung von Patches sei der Vorgang zu schnell, als dass er geeignet von Werkzeugen unterstützt würde.

- Werkzeuge für Oberflächentests hätten oftmals Schwächen und würden etwa Registerkarten (*Tabs*) in Fenstern nicht unterstützen. Probleme seien ebenfalls zu erwarten, wenn komplexe Änderungen an Oberflächen vorgenommen würden. Auch ältere Software, die keine Standardschnittstellen zur Darstellung von Fenstern und ihrer Inhalte nutze, mache einen Werkzeugeinsatz unmöglich.

- Tools könnten keine Allheilmittel sein. Denn der organisatorische und gedankliche Aufwand beim Testen bleibe immer erhalten.

- Eine Plausibilitätskontrolle durch Software sei schwierig aufgrund des fehlenden Semantikverständnisses der Software.

- Die Einführung automatisierter Werkzeuge sei sehr aufwendig. Die Anfangsinvestition sei demnach so hoch, dass sie die Einführung fraglich erschienen ließe. Eine Investitionsrechnung bzw. die Vorhersage eines Return-On-Investments (ROI) sei kaum möglich.

Als generell schwierig stellt sich auch heraus, dass die ein sehr striktes und diszipliniertes Vorgehen nötig ist, um die initialen Aufwände der Einführung eines Testwerkzeugs auf einen akzeptablen (längeren) Zeitraum zu verteilen. Häufig werden dafür aber Anpassungen des Entwicklungsprozesses nötig, die wenig praktikabel sind.

Von allen Projektteilnehmern wurde gleichzeitig betont, dass sie grundsätzlich einer stärkeren Nutzung von Werkzeugen positiv gegenüber stehen. Erwartet wird dabei eine Software, die mit mäßigem Aufwand erlernbar und bedienbar sein soll, leistungsfähig aber nicht unübersichtlich ist und ein möglichst hohes Verständnis der Semantik der zu testenden Software erreicht. Denn ohne letzteres sei die Arbeit mit Werkzeugen häufig ein „doppeltes Programmieren“, da die Werkzeuge mit Skripten erweitert würden. Als wünschenswert wird ferner eine gute Integration der Werkzeuge untereinander erachtet. Betont wurden zudem, dass die Möglichkeit zur Automatisierung von Tests hilfreich wäre. Werkzeuge sollten dabei die Lücken füllen, die menschliche Tester hinterlassen, etwa weil sie mögliche Testfälle vergessen. Auch die automatische Erkennung von Abhängigkeiten, etwa in Daten oder bezüglich Eingabefeldern auf der Programmoberfläche, sei eine hilfreiche Funktion. Allen Projektteilnehmern war gleichzeitig klar, dass viele der Wünsche an ein Werkzeug zwar einfach zu formulieren, aber nur extrem

schwierig umzusetzen seien. Dies gilt insbesondere für das Verständnis von Semantik sowie die automatische Erkennung von Abhängigkeiten. Daher ist die Erwartung gegenüber künftigen Generation von Testwerkzeugen in dieser Hinsicht sehr nüchtern.

3.6.2. Verwaltung von Testfällen

Bezüglich des Einsatzes von Werkzeugen für die Verwaltung von Testfällen zeigt sich das zu Eingang des dritten Kapitels häufig angetroffene Bild: Bei kleinen Unternehmen findet keine explizite Verwaltung von Testfällen statt. Für größere Unternehmen ist sie obligatorisch. Die kleinen Unternehmen haben dies bisher entweder für nicht notwendig erachtet, oder aber der Aufwand wurde als zu hoch eingeschätzt.

Der Status ist allerdings auch bei den größeren Unternehmen unterschiedlich. Grundsätzlich kann zwischen drei Ausbaustufen unterschieden werden:

1. Die recht informelle und nicht durchgängige Verwendung einer Testfallverwaltung. Aufgrund der niedrigen Strukturierung werden auch Testfälle, die theoretisch wiederverwendbar wären, beim Tests neuer Releases einer Software neu erstellt.

2. Ein durchgängig genutztes Werkzeug befindet sich in der Einführungsphase. Parallel werden weiterhin wenig strukturierte Verfahren genutzt.

3. Eine umfangreiche Testfallverwaltung ist im Einsatz.

Als Problematisch wurde beschrieben, das kommerzielle Lösungen für die Testfallverwaltung mitunter sehr kostspielig sind, insbesondere aufgrund der Lizensierung für zahlreiche Arbeitsplätze. Frei verfügbare Werkzeuge hingegen sind oftmals aufgrund des mangelnden Supports problematisch in der Einführung. Gegebenenfalls eignet sich daher die Erstellung eigener Lösungen. Dieser Ansatz wurde von einem der Projekteilnehmer gewählt. Im Wesentlichen verwendet dieser Tabellen, die mit Tabellenkalkulationsprogrammen erstellt wurden. Dabei kommen verschiedene Vorlagen zum Einsatz, aus denen konkrete Dokumente erstellt werden. So kann ein grober Testplan in einem Übersichtsdokument angelegt werden. Insbesondere kann hier festgelegt werden, welche Testarten in welchen Phasen zum Einsatz kommen sollen. In groben Testdokumenten werden zahlreiche Metadaten festgehalten. Dazu kommt eine grobe Beschreibung des Testablaufs für den Tester, eventuell ergänzt mit Bildschirmfotos (*Screenshots*). Ziel ist eine möglichst gute Unterstützung des Testers sowie die Beschreibung der Testanforderungen. Weitere Dokumente spezifizieren manuelle bzw. automatische Tests im Detail. Listen der einzelnen Testfälle werden in Übersichtsdokumenten zusammengestellt und lassen sich über diese auch verwalten. Aufgrund von Unübersichtlichkeiten und Unflexibilitäten durch die Nutzung von Tabellendokumenten wird derzeit der Übergang zu einem integrierten System geplant.

Zum Teil setzen die Projektteilnehmer weitere Systeme ein, die über die Verwaltung von Testfälle hinausgehen. Beispiele sind Werkzeuge für die Erfassung von Anforderungen und die Verwaltung von Fehlern in Programmen. Die so genannten *Bugtracker* sollten sich idealerweise mit den Systemen zur Testfallverwaltung integrieren und einen

automatischen Datenabgleich bieten. Dieses stellt derzeit aber erst ein Ziel für die Zukunft dar, da die Komplexität einer solchen Integration extrem hoch ist. Der Einsatz von Werkzeugen würde einen hohen Konfigurationsaufwand verursachen, während eigene Lösung sehr umfangreich in der Erstellung wären.

Neben der Verwaltung von Testfällen wurde die Verwaltung von Testdatenbanken von einem Projektteilnehmer als Aufgabe genannt. Testdatenbanken enthalten Standarddaten, die alle realen Datenkonstellationen mit möglichst wenigen Daten abdecken sollen. Sie dienen somit der Vergleichbarkeit und Vereinfachung von Testläufen. Im Falle des Projektteilnehmers wurde die Datenbank komplett manuell erstellt und wird je nach Einsatzzweck teilweise oder komplett genutzt. Für die Zukunft ist eine stärkere anwendungsfallbezogene Untergliederung geplant.

3.6.3. Umfangreiche Lösungen

Mit umfangreichen Lösungen sind im Rahmen dieser Broschüre solche Werkzeuge genannt, die über die Nutzung für einen isolierten Einsatzzweck hinausgehen bzw. äußerst umfangreich konfigurierbar sind. Auch werden Werkzeuge, die sehr stark technisch geprägt sind, nicht zu den umfangreichen Lösungen gezählt. Generell zeigt sich bezüglich des Einsatzes solcher Lösungen die bereits häufig beschriebene Zweiteilung. Trotz der Verfügbarkeit von Demo-Versionen und umfangreichen Materialien im Internet, setzen kleinere Unternehmen keine umfangreichen Werkzeuge ein und ziehen den Einsatz in der Regel auch nicht in Erwägung. Die Gründe hierfür liegen in den immens hohen Lizensierungskosten und der Komplexität der Lösungen, die offenbar die Einarbeitung als nicht lohnenswert erscheinen lassen.

Die größeren Unternehmen im Feld der Projektteilnehmer nutzen umfangreiche Testwerkzeuge regelmäßig. Insbesondere die Produkte der IBM Rational-Reihe [17] sowie HP Mercury (gehören zu *Business Technology Optimization (BTO) Software* [18]) kommen zum Einsatz. Die Nutzungsintensität ist allerdings bei einigen Unternehmen geringer, als zu erwarten wäre. Häufig kommen die Werkzeuge nur für einige, spezielle Aufgaben zum Einsatz. Dies könnte dadurch bedingt sein, dass von verschiedenen Problemen berichtet wird:

- Bei Oberflächentests gibt es Schwierigkeiten. Bei Verwendung spezieller Bibliotheken für das Rendering[5] der Oberfläche erkennen die Werkzeuge etwa nicht, wenn die Oberfläche blockiert. Generell ist vor allem die Unterstützung älterer Anwendungen aus verschiedenen technischen Gründen häufig eingeschränkt.

- Ein Team aus einigen Mitarbeitern ist bei Einführungvorhaben und dem Betrieb einer Lösung nötig. Somit stellt es einen wesentlichen Kostenfaktor dar. Auch der organisatorische Aufwand des Einsatzes darf nicht vernachlässigt werden.

- Die Integration der Werkzeuge, z. B. mit IDEs, ist mitunter sehr aufwendig.

[5] Gemeint sind Programmkomponenten, die dafür Verantwortlich sind, aus der Programmierung der Oberfläche die konkrete Bildschirmausgabe zu erzeugen. Sie werden in der Regel nicht selbst programmiert, sondern als Bibliotheken eingebunden.

- Die semantischen Fähigkeiten der Werkzeuge sind häufig sehr gering. Dies“-be“-züg“-liche Anforderungen müssen aufwendig als Skripte implementiert werden.

- Der Preis für Lizenzen kann derart hoch sein, dass auch größere Unternehmen eine probeweise Einführung bzw. den Kauf weitere Lizenzen intensiv überdenken. Selbst die Unternehmen, die von einer hervorragenden Unterstützung seitens der Hersteller berichten, betonen die dem gegenüber stehenden hohen Kosten.

Auffällig war, dass alle größeren Unternehmen den Einsatz umfangreicher Werkzeuge grundsätzlich für sehr sinnvoll erachten. Gleichzeitig wurden die Produkte vielfach kritisiert. Viele der oben genannten Kritikpunkte widersprechen auch dem, was von Herstellern über ihre Produkte zu lesen ist. Daher schließt sich den Ergebnissen dieser Broschüre die Frage an, ob der Werkzeugeinsatz entweder mit zahlreichen Vorurteilen behaftet ist, oder ob der technische Stand der verfügbaren Werkzeuge niedriger ist, als die Ankündigungen der Hersteller erwarten ließen. Zumindest scheint gesichert, dass die Arbeit mit umfangreichen Lösungen relativ anspruchsvoll ist und das nach wie vor erheblicher Forschungsbedarf bezüglich der Arbeitsweise von Werkzeugen besteht. Ob die Werkzeuge tatsächlich bei vielen Aufgaben versagen, kann im Rahmen dieser Broschüre nicht geklärt werden. Das sie häufig noch an Grenzen stoßen, muss allerdings als Fakt festgehalten werden.

Für die Autoren verwunderlich ist die geringe Bereitschaft zur Evaluation von umfangreichen Werkzeugen. Das gesamte Teilnehmerfeld führt solche Evaluation nur selten durch. Als Gründe wurden der hohe Aufwand genannt, aber auch die Erwartung, dass eine neuere Version eines Werkzeugs keine Verbesserung gegenüber einer bekannten Version bieten würde. Die Zyklen, in denen Werkzeuge erneut geprüft werden, liegen mitunter bei fünf oder mehr Jahren. In dieser Zeit ist allerdings eine erhebliche Weiterentwicklung zu erwarten. Auch die Nutzung von online verfügbaren Materialien zur Evaluation, die Nutzung von Demo-Versionen oder auch Vorführungen durch die Hersteller waren bei den meisten Projektteilnehmern sehr zurückhaltend. Die Gründe hierfür sollten eventuell in einem Anschlussprojekt geklärt und Strategien für effiziente Evaluationen der Werkzeuge entwickelt werden.

3.6.4. Kleinere Werkzeuge

Als kleine Werkzeuge im Sinne dieser Broschüre gelten grundsätzliche alle Programme, die das Testen unterstützen aber nicht als umfangreiche Lösung gelten können. Insbesondere fallen Hilfsprogramme und spezialisierte Testprogramme, etwa für Komponententests, in diese Kategorie.

Generell sind die Auskünfte zum Einsatz kleiner Werkzeuge sehr knapp ausgefallen. Ihr Einsatz ist häufig nicht formal beschrieben. Vielmehr werden sie anwendungsfallbezogen und nach eigenem Ermessen durch Entwickler und Tester eingesetzt. Diese Einsatz ist weder durchgängig, noch einheitlich. Es kommen jeweils die Werkzeuge zum Einsatz, die entsprechenden Mitarbeiter bekannt sind. Eine systematische, unternehmensweite Evaluation findet nicht statt. Auch generelle Empfehlungen sind selten.

Der Einsatz von Werkzeugen für Komponententests wie etwa JUnit, CppUnit oder Ähnlichem kann als Standard angesehen werden. Das Feld weitere Werkzeuge ist viel zu heterogen, um eine sinnvolle Aufzählung oder Beschreibung in dieser Broschüre darzustellen[6]. Die gezielte Evaluation kleiner Werkzeuge scheitert zum einen am Aufwand, zum anderen an der sehr dynamischen Entwicklung in diesem Bereich. Viele Werkzeuge sind sehr neu und die Konzepte unbekannt. Das macht eine Bewertung bzw. schon das bloße Auffinden schwierig. Im Allgemeinen ist offenbar keine übersichtliche Zusammenstellung verfügbar. Die Berücksichtigung von kleinen Werkzeugen durch Fachliteratur fällt ebenfalls spärlich aus.

3.6.5. Relevanz von Eigenentwicklungen

Aufgrund zahlreicher negativer Erfahrungsberichte mit verfügbarer Software, aber dem gleichzeitig anzutreffenden Wunsch einer besseren Unterstützung durch Werkzeuge, lag es nage, die Projektteilnehmer bezüglich der Relevanz von Eigenentwicklungen zu befragen. Zu einen Teil wurde dabei angeführt, dass die allgemeine Arbeitsauslastung so hoch sei, dass keine Zeit zum Entwickeln zusätzlicher Testwerkzeuge bleibe. Wenn Entwickler oder Tester sich einfachste Werkzeuge schrieben, die ihre Arbeitsabläufe vereinfachten, sei dies zudem weitgehend unbekannt. Zum anderen Teil wurde aber durchaus über Eigenentwicklungen berichtet.

Streng genommen können bereits die im Rahmen der Regressionstests beschriebenen Dokumentenvorlagen zur Testfallverwaltung als Eigenentwicklung von unterstützenden Werkzeugen gelten. Denn diese verfügen über Makros, die häufig benötigte Vorgänge beschleunigen. Selbst ausgestaltete Dokumentenvorlagen wie diese kommen häufig zum Einsatz. Von zwei Projektteilnehmern wurde die Programmierung umfangreicher Werkzeuge beschrieben. Diese sollen hier exemplarisch angesprochen werden.

Das erste System dient für Tests auf Ebene der Datenbank. Sein Ziel ist es, konsistente Datenbanken für Tests zur Verfügung zu stellen. Dementsprechend bietet es Duplizierungsfunktionen und ist an Systeme auf Testinfrastrukturen angebunden. Es bietet vor allem die Möglichkeit, Veränderungen an Datenbanken zu überwachen. Somit können Entwickler feststellen, welche Datenbanktabellen von Tests betroffen sind. Dies ist hilfreich, da kein Entwickler alle Tabellen eines umfangreichen Systems kennen kann. Auch ist der Vergleich zwischen Testläufen auf einem alten und dem neuen System möglich. Die Eigenentwicklung des Werkzeugs erfolgte, da keine auf dem Markt verfügbare Lösung bekannt war. Schwächen gibt es hinsichtlich der Bedienung. Daher ist es nur durch wenige Mitarbeiter nutzbar. Neben der Erhöhung des Bedienkomforts wäre auch ein funktionaler Ausbau denkbar. Da das Werkzeug von einem einzelnen Mitarbeiter entwickelt wurde, ist das Verhältnis von Aufwand und Nutzen allerdings sehr gut. Zudem demonstriert es anschaulich, wie aus der von einem einzelnen Mitarbeiter erkannten potentiellen Arbeitserleichterung heraus, die ursprünglich nur für ihn selbst gedacht war, ein nützliches Werkzeug erstellt werden kann.

[6] Einige der von Projektteilnehmern empfohlenen Werkzeuge wurden aber in der Anhang dieser Broschüre aufgenommen.

Das zweite System dient der Anforderungs- und Testfallverwaltung. Es entstand aus der Feststellung heraus, dass Word- bzw. Excel-Dokumente, in denen Anforderungen und Testfälle festgehalten wurden, mit der Zeit zu unübersichtlich werden. Daher wurde eine Datenbank für Anforderungen aufgesetzt. Diese dient auch der Nachverfolgung der Anforderungen, also deren konkreten Umsetzung im System. Danach folgte die Integration der Testunterstützung. Dazu bildet das Werkzeug Testpläne ab. Dank einer guten Dokumentation kann das Werkzeug generell zum Einsatz kommen und wird unternehmensintern weitergegeben. Nach Aussage des Projektteilnehmers besitzt es marktreife, ist aber nicht zum Vertrieb gedacht. Ähnlich wie das erste Beispiel zeigt es, wie aus internen Anstrengungen heraus Werkzeuge entstehen können, die in hohem Maße arbeitserleichternd sind. Von Vorteil ist dabei vor allem die Maßschneiderung der Werkzeuge auf die eigenen Prozesse. Wichtig ist offenbar, auf eine gute Bedienbarkeit und Dokumentation zu achten, damit das Werkzeug wie ein über den Markt erworbenes generell eingesetzt werden kann und nicht nur einem eingeschränkten Expertenkreis vorbehalten bleibt.

3.7. Zwischenfazit

Als Abschluss der Erfassung des Status quo soll ein kurzes Zwischenfazit gezogen werden. Insgesamt fallen die Ergebnisse der mit den Projektteilnehmern geführten Interviews sehr heterogen aus; die grundsätzlichen Vorgehensweisen bezüglich des Testen von Software weisen mitunter erhebliche Unterschiede auf. Im Hinblick auf die uneinheitliche Ausrichtung der Unternehmen entspricht dies aber der Erwartung.

Eine stärkere Homogenität der Ergebnisse zeigt sich, wenn die teilnehmenden Unternehmen in zwei Gruppen aufgeteilt betrachtet werden. So ist es für alle kleineren Unternehmen üblich, wenig formalisiert zu testen. Tests werden je nach Anforderung durchgeführt. Umfangreiche Werkzeuge kommen dabei kaum zum Einsatz, spezialisierte Tester gibt es wenige und eine zentrale Testorganisation ist nicht vorhanden. Allerdings erfolgt die Entwicklung in einigen Projekten testgetrieben mit starker Fokussierung auf Komponenten- und Abnahmetests. Bei den größeren Unternehmen hingegen ist Testen als Prozess innerhalb der Softwareentwicklung etabliert. Er wird zentral geleitet oder zumindest zentral unterstützt, Vorgehensweisen, Methoden und Werkzeuge werden wenigstens in Teilen vorgegeben. Der Formalisierungsgrad ist höher; im Allgemeinen entspricht das Testen eher der Art und Weise, wie sie in der praxisorientierten Fachliteratur beschrieben wird. Dementsprechend folgt er etwa grob der klassischen Phaseneinteilung des V-Modells [DHM98]. Nichts desto trotz stellten beide Gruppen in den Interviews ihre grundsätzliche Zufriedenheit heraus. Als Ausblick auf das fünfte und sechste Kapitel kann dem angeschlossen werden, dass bei jedem Projektteilnehmer erfolgreiche Strategien identifiziert werden konnten, die als Empfehlungen in diese Broschüre eingeflossen sind.

Auffällig ist, dass der Grad der Testautomatisierung sehr niedrig ist. Damit einhergehend werden auch Regressionstests seltener genutzt, als dies im Interesse der Unternehmen ist. Beides kann zum einen durch die extreme Komplexität einiger Projekte sowie

organisatorische Schwierigkeiten, zum anderen durch mangelnden Werkzeugeinsatz erklärt werden. Dieser fällt deutlich geringer aus, als zu erwarten wäre. Alle Unternehmen beklagten Schwierigkeiten mit Werkzeugen und zeigten sich generell unzufrieden. Die Unzufriedenheit reichte dabei von grundsätzlicher Kritik an den zur Verfügung stehenden Werkzeugen bis hin zu Schwierigkeiten mit einigen Funktionen bzw. dem Zusammenspiel mehrerer umfangreicher Werkzeuge.

Der Ausbau von Regressionstests und Testautomatisierung im Allgemeinen, eine stärkere Werkzeugunterstützung dieser Tests, und eine generelle Entlastung der Tester von wenig anspruchsvollen aber zeitaufwendingen Aufgaben durch Werkzeuge können daher als Ziele bzw. Wünsche der Projektteilnehmer festgehalten werden.

4. Ordnungsrahmen der Handlungsempfehlungen

Die in den folgenden zwei Kapiteln 5 und 6 dargelegten Handlungsempfehlungen werden in einen Ordnungsrahmen einsortiert. Dessen Ziel ist es, die Bewertung zu unterstützen, ob eine Handlungsempfehlung für eine konkrete Situation in einem Unternehmen sinnvoll ist. Daher werden die fünf als maßgeblich erachteten Dimensionen erläutert und darauf aufbauend der Ordnungsrahmen vorgestellt.

4.1. Anspruch

Die Umsetzung der einzelnen Handlungsempfehlung ist mit Aufwand verbunden. Während einige Hinweise schnell und einfach in die Tat umgesetzt werden können, sind andere Punkte eher als langfristige Empfehlungen zu werten. Der Aufwand für die Umsetzung kann dabei sehr hoch sein, oder es ist ein lang laufender Änderungsprozess anzustoßen. Dazu kommt, dass die Voraussetzungen, an die in den umsetzenden Unternehmen angeknüpft werden kann, völlig verschieden sein werden. Wir unterscheiden daher hinsichtlich des Anspruchs der Empfehlungen drei Kategorien:

- Grundsätzliches: Einige der Handlungsempfehlungen sind entweder so einfach in der Umsetzung oder aus unserer Sicht so wichtig, dass wir sie als grundsätzlich erachten. Daher empfehlen wir allen Unternehmen, die Empfehlung umzusetzen bzw. zu prüfen, ob die eigenen Prozesse in Anlehnung an die gemachten Vorschläge gestaltet sind.

- Fortgeschrittenes: Viele der Handlungsempfehlungen gehen über grundsätzliche Tipps hinaus. Wenn wir sie dennoch für verhältnismäßig einfach zu implementieren und gleichzeitig effektiv halten, oder festgestellt haben, dass ihre Umsetzung sehr lohnenswert ist, empfehlen wir sie als fortgeschrittene Verbesserung.

- Zielzustand: Zielzustände sind wünschenswerte Ideale, deren Erreichen aber mitunter schwierig sein kann. Nichts desto trotz gelten Empfehlungen, die in diese Kategorie fallen, auf keinen Fall als unerreichbar. Vielmehr sollten sie als Leitlinien betrachtet werden. Ob in die Erreichung der Zielzustandes Mittel investiert werden sollten, hängt von der konkreten Situation im Unternehmen ab. Im Wesentlichen fallen solche Maßnahmen unter diesen Punkt, die zu einer kontinuierlichen Verbesserung der Testprozesse führen, aber deren direkte Auswirkungen eher gering sind.

Handlungsempfehlungen können durchaus in mehrere der Kategorien gleichzeitig fallen. Dies bedeutet, dass bereits in der jeweils niedrigsten Kategorie mit der Umsetzung begonnen werden sollte und die Empfehlungen auch bei teilweiser Umsetzung schon sinnvoll sind, das volle Potential bzw. weitere Verbesserungen aber erst im Rahmen umfassender Maßnahmen realisiert werden können. Dies ermöglicht eine in Stufen aufgeteilte Einführung der Maßnahme und weist zudem daraufhin, dass der Umfang ihrer Implementierung variabel gehalten werden kann.

4.2. Größe des Projekts bzw. des Unternehmens

Einige der vorgestellten Maßnahmen sind nicht unabhängig von der Größe des Projektes bzw. des Unternehmens, in dem Software entwickelt wird. Bestimmte organisatorische Rahmenbedingungen sind etwa in kleinen Unternehmen schon aufgrund der Gesamtzahl der Mitarbeiter unmöglich zu erreichen. Andere Hinweise sind speziell auf kleinere Projekte bzw. Unternehmen zugeschnitten. Daher wird für jede Handlungsempfehlung angegeben, für welchen ungefähren Rahmen sie gedacht ist. Wir unterscheiden dabei zwischen *klein*, *mittel* und *groß*.

In kleine Projekte sind wenige Mitarbeiter involviert, die auch eigenständig für das Testen verantwortlich sind. Mittlere Projekte zeichnen sich dadurch aus, dass Entwickler und Tester unabhängig sind und eine Leitung existiert, die beide Mitarbeitergruppen koordiniert. Große Projekte schließlich zeichnen sich durch einen übergreifenden Ansatz aus. Typischerweise werden große Projekte in größeren Unternehmen mit zumindest einigen hundert Mitarbeitern durchgeführt. Es existieren nicht nur projektbezogene Entwicklungs- und Testteams, vielmehr verfügt das Unternehmen über Querschnittsabteilungen wie etwa eine eigene Testabteilung, die an (fast) allen Projekten beteiligt ist.

Die Unterscheidung bezieht sich grundsätzlich auf Projekte in der oben dargestellten Weise. Allerdings ist diese auch abhängig von der Unternehmensgröße. So ist es ab einer bestimmten Unternehmensgröße fast obligatorisch, dass die Rollen Entwickler und Tester getrennt sind; in kleinen Unternehmen mit weniger als hundert Mitarbeitern kann dagegen fast sicher davon ausgegangen werden, dass es zwar Mitarbeiter gibt, die in mehrere Projekte involviert sind, aber dienstleistende Abteilungen, die auf Anfrage oder verpflichtet Projekten zuarbeiten, nicht anzutreffen sind.

4.3. Individualentwicklung oder Standardsoftware

Nicht alle Handlungsempfehlung besitzen eine über die Art der Softwareentwicklung hinausgehende Gültigkeit. Bei der (einmaligen) Entwicklung von Individualsoftware etwa spielen Regressionstests eine deutlich kleinere Rolle als bei der Entwicklung einer Software für den Massenmarkt, die in regelmäßigen Releases erscheint. Bei letzterer ist die Organisation häufig komplexer. Auch wird Individualsoftware häufig im direkten Kundenkontakt erstellt, Software für den Massenmarkt hat einen mehr oder weniger anonymen potentiellen Abnehmerkreis. Daher wird angegeben, ob es diesbezügliche Ein-

schränkungen gibt. Die meisten Empfehlungen sind allerdings sowohl für die Individualentwicklung als auch für Standardsoftware nützlich. In einigen Fällen liegt der Schwerpunkt bei einer der beiden Entwicklungsformen, auch wenn die Empfehlung grundsätzlich allgemeingültig ist.

4.4. Releasehäufigkeit

Für den Fall, dass Empfehlungen nur dann Gültigkeit haben, wenn zumindest einige Releases einer Software geplant sind, wird darauf hingewiesen. Schließlich sind verschiedene Maßnahmen erst dann sinnvoll, wenn es zumindest einige Releases einer Software gibt. Daher wird zwischen einem einzelnen Release, einigen Releases und regelmäßigen Releases unterschieden. *Regelmäßig* charakterisiert dabei, dass die Software dafür ausgelegt ist, zumindest für einige Jahre, eventuell auch zunächst zeitlich unbeschränkt angeboten zu werden und dass dabei mehrfach aktualisierte Versionen veröffentlicht werden. Analog zur Form der Softwareentwicklung ist es möglich, dass die Gültigkeit der Maßnahme für eine oder zwei der Einstufungen uneingeschränkt gilt, aber auch für die andere(n) Einstufungen grundsätzlich verwendbar ist.

4.5. Phasen des Softwaretests

Viele Handlungsempfehlungen sind für den gesamten Testprozess gültig bzw. von einzelnen Phasen des Softwaretests unabhängig. Allerdings existieren ebenso Empfehlungen, die sich nur auf einige Phasen oder eine Phase beziehen bzw. deren Schwerpunkt auf einigen Phasen oder einer Phase liegt. Im ersten Fall bedeutet dies, dass notwendige Maßnahmen nur für bestimmte Phasen implementiert werden müssen, da sie in anderen Phasen wirkungslos oder sogar unangebracht sind. Im zweiten Fall ist es so, dass die volle Wirkung nur ein einigen Phasen entfaltet werden kann, aber sich durchaus auch Vorteile für die anderen ergeben können. Dem muss durch eine geeignete Prozessintegration Rechnung getragen werden. Sollte nicht implizit klar sein, warum es bei einer Empfehlung einen Phasenbezug gibt bzw. wie dieser aussieht, wird im Rahmen der Empfehlung darauf hingewiesen.

Im Rahmen dieser Broschüre wird grob zwischen folgenden Phasen des Testens unterschieden (vergleiche auch Kapitel 3.3):

Komponententests Entwickler testen den soeben erstellten Code. Ausgangspunkt ist dabei die einzelne Komponente.

Integrationstest Mehrere Module des zu entwickelnden Systems werden integriert und das Zusammenspiel getestet. Es kommen zahlreiche Teststümpfe zum Einsatz.

Systemtest Wesentliche Module des Systems werden im Zusammenspiel getestet. Dabei werden für externe Dienste zum Teil nach wie vor Teststümpfe verwendet. Viele interne Funktionen sind aber verfügbar bzw. Teststümpfe werden während des Tests nach und nach entfernt und durch „echte“ Bindungen ersetzt.

Abnahmetest Bevor das System in den Kundeneinsatz gelangt, erfolgt ein Abnahmetests. Er ist deutlich weniger technisch getrieben als die vorherigen Tests. Das grundsätzlich komplettierte System wird gegen seine Anforderungen getestet.

4.6. Der Ordnungsrahmen

Einerseits ist es wünschenswert, den Ordnungsrahmen durch eine grafische Darstellung zu unterstützen. Andererseits ist es unmöglich, fünf Dimensionen in einer zweidimensionalen Grafik darzustellen. Versuche, dies zu simulieren, führen in der Regel zu Unübersichtlichkeit. Nichts desto trotz ist eine adäquate Darstellung möglich. Dazu werden zwei Dimensionen grafisch dargestellt, die restlichen drei textuell. Der von uns für diese Broschüre entwickelte Ordnungsrahmen ist in Abbildung 4.1 dargestellt.

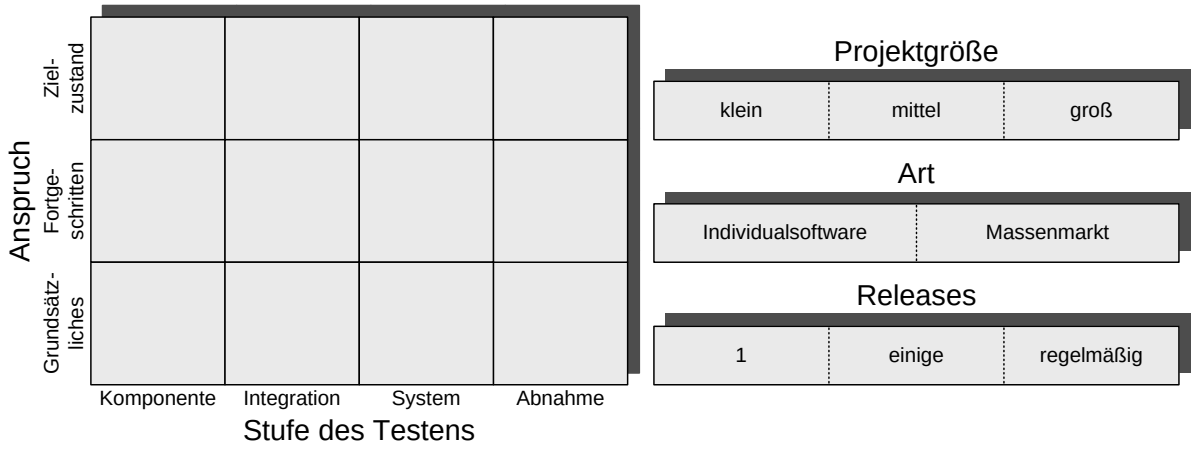

Abbildung 4.1.: Der Ordnungsrahmen

Die Matrix im linken Bereich des Ordnungsrahmens zeigt den Zusammenhang zwischen Anspruch und Phasenbezug der jeweiligen Handlungsempfehlung. Dort, wo ein Zusammenhang besteht, wird dies mit einem grünen Häkchen gekennzeichnet (siehe Beispiel weiter unten). Empfehlungen können mehrere Phasen umfassen und sogar einen gestuften Anspruch haben. Falls einzelne Zuordnungen zwar bestehen, aber als weniger wichtig gelten müssen, als andere, kann das Häkchen auch in Klammern dargestellt werden. Dies bedeutet dann, dass eine Empfehlung zwar für ein bestimmtes Anspruchsniveau in einer bestimmten Phase durchaus relevant hat, aber im Vergleich mit anderen Kombinationen nebensächlich ist. Einzeln dargestellt sind die Projektgröße, die Art der Softwareentwicklung sowie die Anzahl der Releases einer Software. Die jeweils zutreffenden Eigenschaften werden in einem dunkleren grau hinterlegt. Dabei gilt, dass durchaus mehrere Eigenschaften zugleich zutreffend sein können. Ein Übergang zeigt an, dass die Relevanz eingeschränkt ist.

Ein Beispiel für die Verwendung des Ordnungsrahmens ist in Abbildung 4.2 dargestellt. Es ist wie folgt zu lesen:

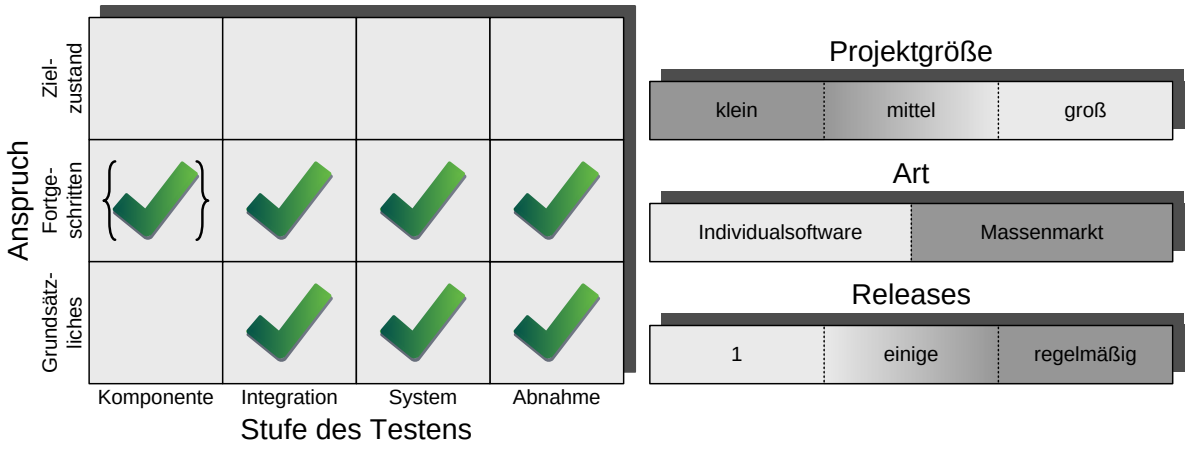

Abbildung 4.2.: Beispielhafte Anwendung des Ordnungsrahmens

- Die Handlungsempfehlung bezieht sich auf den Integrations-, System- und Abnahmetest. Sie hat dabei sowohl einen grundsätzlichen als auch einen fortgeschrittenen Anspruch. Dies bedeutet, dass die konkrete Empfehlung grundsätzlich umgesetzt werden sollte, aber nicht notwendigerweise direkt im vollen Umfang. Vielmehr ergeben sich über grundsätzliche Vorteile hinaus noch weitere Vorteile, die schon als fortgeschritten gelten können.
- Bei einer vollständigen Umsetzung können sich auch Vorteile für den Komponententest ergeben. Diese sind aber im Vergleich zu den Vorteilen für die anderen Phasen eher nebensächlich. Deshalb ist das Häkchen in Klammern dargestellt.
- Die Handlungsempfehlung bezieht sich vor allem auf kleine Projekte und hat für mittlere Projekte ebenfalls Relevanz. Der Übergang der Färbung bedeutet, dass die Relevanz für solche Projekte noch gegeben ist, die Nahe an der Grenze zur Kategorisierung als kleine Projekte liegen, nicht aber für solche, die nahezu als große Projekte kategorisiert werden könnten.
- Ferner bezieht sie sich ausschließlich auf Software, die für den Massenmarkt entwickelt wird. Theoretisch wäre es an dieser Stelle auch möglich, dass eine der beiden Arten zur Hälfte grau hinterlegt ist (wiederum als Übergang). Die Bedeutung wäre dann, dass die Empfehlung sich zwar im Wesentlichen auf eine Art von Softwareentwicklung bezieht, durchaus für die andere aber ebenfalls sinnvoll ist.
- Nur für solche Software, von der zumindest einige Releases erscheinen, sollte die Handlungsempfehlung umgesetzt werden. Sie hat ebenso Gültigkeit, wenn es regelmäßige Releases gibt.

Natürlich können viele Handlungsempfehlungen nicht rein projektbezogen betrachtet werden. Zumindest für größere Unternehmen bzw. größere Softwareentwicklungsabteilungen sollte der Ordnungsrahmen daher so verstanden werden, dass er sich auf typisch durchgeführte Projekte bezieht.

5. Organisatorische Handlungsempfehlungen

In den folgenden Unterkapiteln sind organisatorische Handlungsempfehlungen zusammengestellt. Sie sind nicht nach genereller Wichtigkeit geordnet, aber grob nach potentiellem Adaptorenkreis. Die zuerst genannten Empfehlungen sind also für alle softwareentwickelnden Unternehmen relevant, während die zum Ende hin kommenden Empfehlungen nicht für jedes Unternehmen passend oder praktikabel sind. Eine Ausnahme stellt nur die letzte Empfehlung in Kapitel 5.16 dar, die allgemeine Gültigkeit besitzt und Kapitel 5 abschließt.

5.1. Einführung

Viele der im Folgenden vorgestellten Handlungsempfehlungen sind – zumindest in ihrem Kern – nicht überraschend. Vielfach laufen sie auf ähnliche Grundsätze hinaus: Klare organisatorische Strukturen, definierte Prozesse und eindeutige Zuständigkeiten. Von der Definition und stetigen Optimierung von Testprozessen sind die meisten Unternehmen allerdings weit entfernt.

Gerade für kleinere Unternehmen mag es nicht praktikabel erscheinen, Testprozesse zu definieren, Vorgehensmodelle zu erstellen und Ähnliches. Für ein erfolgreiches und nachhaltiges Testen werden sie dennoch nicht umher kommen, Testen als einen Prozess zu sehen. Im Rahmen der Softwareentwicklung trägt Testen zur Wertschöpfung bei, indem es die Qualität der Produktion erhöht. Testen darf nicht als „notwendiges Übel“ gesehen werden, weder bei der Geschäftsleitung noch bei den Programmierern oder Testern. Testen ist ein integraler Bestandteil der Softwareentwicklung, muss als solcher im Unternehmen gelebt werden und mit dem entsprechenden Stellenwert ausgestattet werden.

Die erste wesentliche Empfehlung dieser Broschüre ist daher, eine Kultur des Testens im Unternehmen aufzubauen. Testen ist viel mehr eine Querschnittsfunktion und ein Bindeglied zwischen technischer und fachlicher Sichtweise, als weithin angenommen wird. Wir empfehlen daher dringend, Testen nicht als reinen Kostentreiber wahrzunehmen, sondern ganzheitlich zu betrachten und in die Wertschöpfung einzubinden. Natürlich ist es gerade für kleine Unternehmen schwierig, zusätzlichen Testaufwand gegenüber dem Kunden zu motivieren. Viel leichter dürfte es fallen, als ein Unternehmen aufzutreten, für das Qualitätssicherung essentieller Bestandteil jeglicher Geschäftsprozesse ist und das qualitativ hochwertige Arbeitsergebnisse nicht als Kür, sondern als Pflicht sieht. So wie viele Unternehmen bereits mit einer Zertifizierung ihrer qualitätssicherenden Maß-

nahmen nach DIN EN ISO 9000 bzw. DIN EN ISO 9001 werben, kann es sich gerade für kleine Unternehmen, die Software entwickeln, anbieten, ihre Fokussierung auf das Testen zu betonen.

Aufgrund der dramatischen Raten von gescheiterten Softwareprojekten vertreten die Autoren dieser Broschüre die Ansicht, dass qualitative Softwareentwicklung nicht ohne adäquates Testen möglich ist. Gleichzeitig ist Testen bei einer ganzheitlichen Betrachtung kein Kostentreiber, sondern Kostensenker. Natürlich bindet Testen Arbeitskraft; Werkzeuge und Systemen müssen angeschafft werden. Die Kosten gescheiterter Projekte sind für den Auftraggeber aber ungleich höher. Wir sehen es daher auch als Aufgabe der entwickelnden Unternehmen an, gerade Auftraggeber mit geringer Affinität für IT von der Notwendigkeit zu überzeugen, zumindest ein gewisses Qualitätsmaß über kurzfristige Kosten zu stellen, da die langfristige Kostenbetrachtung – insbesondere unter Einbeziehung einer Risikoanlayse – völlig anders aussieht als eine auf den Moment bezogene Betrachtung erhöhter Kosten durch qualitätssichernde Maßnahmen. Die Entwicklung qualitativer Software kann zudem als „Aushängeschild“ dienen, wohingegen bei fehlerhafter Software ein Imageschaden droht.

5.2. Eigene Rollen für Entwickler und Tester

Eine einfache, aber grundlegende Empfehlung ist, eigene Rollen für Entwickler und Tester zu definieren. Insbesondere in kleineren Unternehmen ist es üblich, dass Tester und Entwickler in denselben Personen vereint sind. Wann die entsprechenden Mitarbeiter die betroffene Software weiterentwickeln und wann sie Phasen des Testens einlegen, bleibt ihnen im Wesentlichen selbst überlassen. Mitunter erfolgt ein Testen auch relativ unkoordiniert als Tätigkeit, die Lücken im Zeitplan füllt.

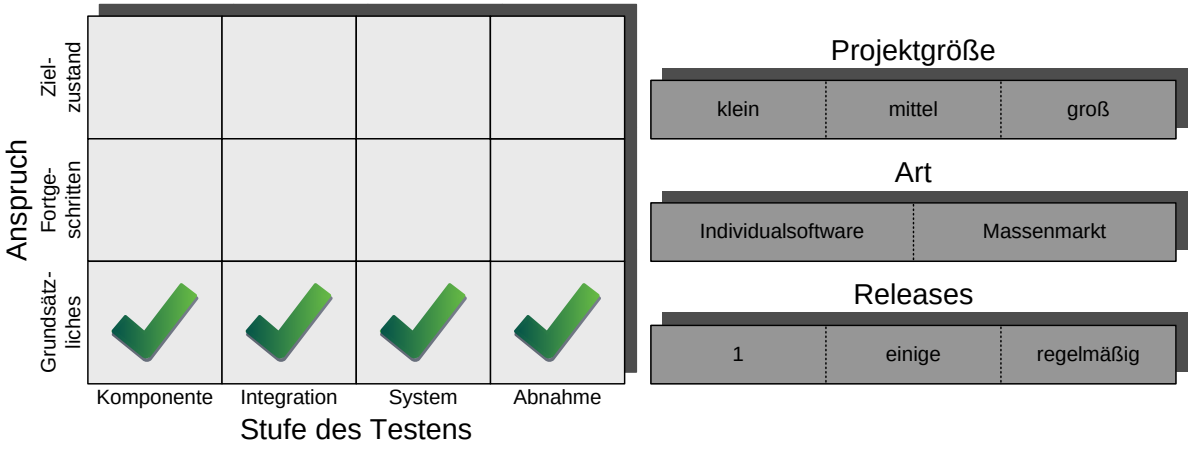

Abbildung 5.1.: Einordnung der Handlungsempfehlung

Selbst bei personeller Knappheit und kleinen Projekten ist es dringend zu empfehlen, das Entwickeln vom Testen zu entkoppeln. Zwar kann das Schreiben von Testfällen etwa bei der testgetriebenen Entwicklung Teil des Entwicklungsprozesses und in diesem

Rahmen auch sehr sinnvoll sein. Spätestens Integrationstests sollten aber als eigene, von der Entwicklung unabhängige Schritte angesehen werden. Die für die Tests eingeteilten Mitarbeiter sollten für die Zeit des Testens die Rolle von Testern einnehmen — Testen darf für sie keine Nebenaufgabe sein. Somit ist es auch für kleine Unternehmen möglich, diese Empfehlung zu befolgen: Tester und Entwickler können demselben Personenkreis entspringen, aber es sollte stets sichergestellt werden, dass in den über einen Komponententest hinausgehenden Testphasen Entwickler nicht mehr ihren eigenen Code testen.

Die Fokussierung des Testers auf seine Aufgabe sollte durch eine entsprechende Arbeitszeitplanung abgedeckt sein. Das Testen sollte dabei als eine wichtige Aufgabe im Unternehmen motiviert werden; gilt es als notwendige, aber lästige Aufgabe, die Zeit, welche ansonsten für die Entwicklung zur Verfügung stünde, vernichtet, werden die Tester nicht so effektiv arbeiten, wie sie könnten. Das Testen sollte – wiederum zumindest dem Komponententest folgend – konsequent ausgeführt und abgeschlossen werden. Gefundene Fehler müssen dokumentiert werden. Sie direkt dem Entwickler zu melden oder gar unmittelbar durch den Tester beheben zu lassen ist sehr wenig empfehlenswert. Auf diese Weise können neue Fehler in den Code gelangen, während der eigentliche Tests unstrukturiert abläuft und viele Fehler übersehen werden können. Auch wird verhindert, dass Entwickler und Tester aus typischen Fehlern lernen und diese in der Zukunft vermeiden bzw. schneller erkennen und beheben können.

Wir empfehlen weiterhin dringend, Testen als ebenso anspruchsvolle Aufgabe wie die Entwicklung anzusehen. Entwickler mit unterdurchschnittlichen Fähigkeiten oder wenig Erfahrung entwickeln mit hoher Wahrscheinlichkeit eine Software, die grundsätzlich funktioniert. Ob sie aber eine Software entwickeln, die *gut* ist, ist sehr zweifelhaft. Analog verhält es sich bei Testern: Natürlich kann Software von Testern mit unterdurchschnittlichen Fähigkeiten oder wenig Erfahrung getestet werden. Natürlich werden diese Tester einige, vielleicht sogar viele Fehler finden und damit zur Erhöhung der Qualität der Software beitragen. Ob die Zahl und vor allem die Art der gefundenen Fehler dazu ausreicht, dass die fertige Software vom Kunden als *gut* betrachtet wird, ist aber ebenfalls äußerst zweifelhaft. Leider wird Testen häufig als die soeben skizzierte sekundäre Aufgabe wahrgenommen, für die nicht die „besten Leute“ herangezogen werden sollte. Diese Betrachtungsweise ist abzulehnen. Im Idealfall sind Tester und Entwickler auf demselben Niveau. Insbesondere sollten weniger erfahrenen Mitarbeiter auch erfahrene und besonders qualifizierte Mitarbeiter als Ansprechpartner zur Seite stehen.

Zu beobachten ist auch, dass Entwickler mit dem Testen von Software fachlich überfordert sein können, ohne dies zu bemerken. Je diversifizierter die Anwendungsentwicklung ist, desto geringer wird die durch einen einzelnen Entwickler erbrachte Fertigungstiefe. Eine Modularisierung der Systeme und eine Spezialisierung der Entwickler auf bestimmte Bereich ist durchaus erwünscht, aber in der Regel nicht mit fundamentiertem Testwissen kombiniert. Konzentrieren sich Entwickler in der Rolle als Tester darauf, festzustellen, ob „ihr“ Modul korrekt arbeitet, werden schwerwiegende Fehler wahrscheinlich übersehen. Das System scheint korrekt zu funktionieren, weil es alle intuitiven Testläufe fehlerfrei durchlaufen hat. Viel sinnvoller und schließlich effektiver ist ein strukturierter Ansatz, der methodisch begleitet ist. Beispiele sind die Überdeckungsanalyse aus Code-naher Sicht sowie die Äquivalenzklassenbildung aus funktionaler Sicht. Tiefere Kenntnisse die-

ser Ansätze sind allerdings in der Regel nicht zu erwarten und müssen daher explizit vermittelt werden, wenn Entwickler als Tester agieren sollen.

Welche Mitarbeiter als Tester geeignet sind, muss unternehmensindividuell entschieden werden. Dabei ist sicherlich zu beachten, dass kleinere Unternehmen keine dedizierten Tester beschäftigen können. Um so wichtiger ist für sie eine klare Rollendefinition und eindeutige Zuständigkeiten in Projekten. Zudem sollte, möglicherweise sogar formal, festgehalten und evaluiert werden, welche Mitarbeiter sich für welche Arten von Tests besonders eignen. Eventuell lassen sich sogar Kombinationen bzw. Teams aus Entwicklern und Testern finden, die im Zusammenspiel besonders hochwertige Software erstellen. Kann aus einen größeren Pool von Mitarbeitern gewählt werden bzw. werden Stellen für dedizierte Tester geschaffen, sollte ein durchdachtes Auswahlverfahren zum Einsatz kommen. Dabei sollte vor allem die Erfahrung der Tester eine Rolle spielen, aber durchaus auch ihre Mentalität. Von einem der teilnehmenden Unternehmen erhielten wir den Hinweis, dass ihre Tester eher wenig affin für Technik seien, sondern eine betriebswirtschaftliche Ausbildung genossen haben. Sie stünden neuer Software viel kritischer gegenüber. Würden zudem Mitarbeiter ausgewählt, die scherzhaft auch als „Wadenbeisser“ oder „Erbsenzähler“ bezeichnet würden, wären gute Ergebnisse fast sicher. Führungskräfte hingegen, die wenig Kenntnisse der zugrunde liegenden Vorgänge hätten, seien als Tester weniger geeignet. Sie testeten – schon aus Zeitgründen – zu oberflächlich. Damit sei sicher, dass sie kaum zusätzliche Fehler finden (und als alleinige Tester ungeeignet sind).

Für die späteren Testphasen kann es sinnvoll sein, das Programm durch Mitarbeiter mit Kundenkontakt testen zu lassen, insbesondere, wenn es sich um die Entwicklung von individueller Software handelt. Die Mitarbeiter erhalten eine kurze Einführung und versuchen dann, produktiv mit der Software zu arbeiten. Durch ihren Kundenbezug wissen sie, was die Software leisten soll und finden eventuell zusätzliche Fehler und können überdies schwächen in der Bedienung identifizieren. Durch die späte Testphase ist aber sichergestellt, dass sich nur noch wenig an der Software ändert. Somit lernen die Mitarbeiter ohne umfangreiche Schulung direkt das System kennen, das sie dem Kunden vorstellen müssen. Sowohl die Entwickler und Tester als auch für die Vertriebsmitarbeiter profitieren somit von einem Wissenstransfer.

5.3. Einbeziehung des Kunden und Anforderungsanalyse

Softwarequalität beschränkt sich nicht auf die Forderung nach Software, die möglichst fehlerarm funktioniert. Vielmehr soll sie im Rahmen ihres Einsatzzwecks möglichst dienlich sein, die Arbeit erleichtern und Geschäftsprozesse unterstützen bzw. ermöglichen. Dies impliziert, dass bereits die Analyse der Anforderungen extrem wichtig ist. Fehler, die in dieser Phase begründet liegen, lassen sich durch Testen nicht mehr beheben. Ungenauigkeiten können das Testen erheblich verkomplizieren.

Software wird letztlich immer für Kunden entwickelt. Ob es sich dabei um auftragsbezogene Entwicklung für externe Kunden handelt, oder auftragsbezogene oder initiative Entwicklung für interne Kunden bzw. für Marktkunden, ist streng genommen unbe-

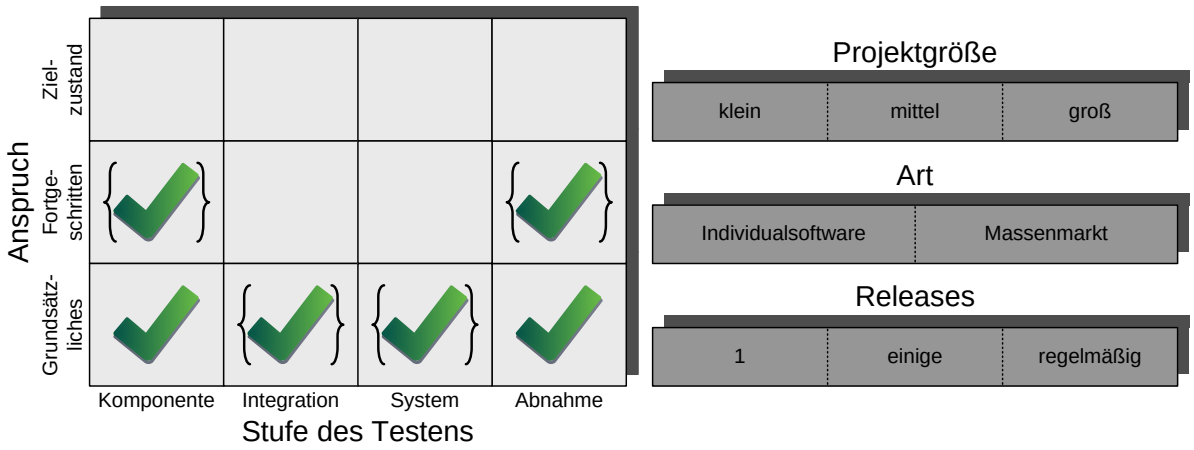

Abbildung 5.2.: Einordnung der Handlungsempfehlung

deutend. Wichtig ist, die (möglicherweise noch unbekannten) Wünsche der Kunden zu treffen. Dies ist durch eine Einbindung der Kunden möglich.

Insbesondere bei der auftragsbezogenen Entwicklung hat es sich als erfolgreiche Strategie erwiesen, die Anforderungen mit dem Kunden gemeinsam zu diskutieren, anstatt streng anhand eines Lastenhefts zu entwickeln. Idealerweise hat der Kunde dabei nicht nur Kontakt zum Vertriebspersonal, sondern direkt zu Entwicklern. Diese sollten derart geschult werden, dass sie über die reine Entwicklungstätigkeit hinaus zu „Anforderungsanalysten“ werden. Je stärker sich ein Entwickler in *der Welt des Kunden* zurechtfindet, desto eher wird er Software entwickelt, die den Wünschen des Kunden entspricht. Sinnvoll kann es auch sein, gerade Tester in das Team aufzunehmen, das die Anforderungen ermittelt [Toc08]. Die Erfassung von Anforderungen sollten sich an gängigen Vorgehensweisen orientieren, die in der einschlägigen Literatur beschrieben sind. Gegebenenfalls kann auch auf Standard wie [IEE98a] zurückgegriffen werden.

Die Einbeziehung des Kunden sollte sich aber nicht nur auf die Erfassung der Anforderungen beschränken. Eine ständige Einbindung des Kunden (bzw., im unternehmensinternen Kontext, der Fachabteilung) ist der Softwarequalität ebenfalls sehr zuträglich. Auf diese Weise wird nicht nur sichergestellt, dass das in den Entwicklung befindliche Produkt den Wünschen des Kunden entspricht und bei Abweichungen direkt Maßnahmen ergriffen werden können. Vielmehr kennt der Kunde auch die Gegebenheiten, unter denen das Produkt zum Einsatz kommen soll. Auch wird er ein Bild davon haben, welche Arten von Daten das Produkt verarbeiten und in welchen Kontext es eingebunden werden wird. Damit kann er Tests bereits in frühen Phasen unterstützen und somit helfen, systematische Schwächen zu einem frühen Zeitpunkt zu erkennen und zu beheben.

Eine starke Einbeziehung ist für viele Kunden sicherlich ungewohnt und wird von klassischen Softwareentwicklungsprozessen nicht vorgesehen. Nichts desto trotz kann als Ergebnis festgehalten werden, dass sich eine Ausrichtung auf den Kunden und die gegebenenfalls nötige Überzeugungsarbeit lohnt. Vor allem auch die Arbeit mit Pilotkunden hat sich als erfolgreiche Strategie herausgestellt. Im internen Kontext kann die Einbindung

der Fachabteilung, für die eine Software entwickelt wird, nicht nur die Qualität steigern, sondern auch Akzeptanzprobleme minimieren. Der Aufbau einer engen Beziehung zu einigen Pilotkunden ist auch im Falle der Entwicklung von Software für den Massenmarkt sinnvoll. Idealerweise können neue Releases von Produkten so vor der Veröffentlichung in einer produktiven Umgebung getestet werden. Falls es möglich ist, dies in Ansprache mit den Kunden auf wenig kritischen Systemen oder für wenig kritische Vorgänge zu beschränken, ist auch das Risiko für die Kunden gering; die Tatsache, dass sie neue Releases frühzeitig kennen lernen und sogar hinsichtlich ihrer Gestaltung Einfluss nehmen können, ist sogar ein Anreiz für sie. Ist die Bindung zu einem Kund sehr eng, kann sogar neue entwickelte Software in Zusammenarbeit mit diesen Kunden getestet werden. Natürlich ist die Voraussetzung dafür, dass die Software dem entsprechenden Kunden einen Zusatznutzen verspricht. Im Idealfall entsteht so mittelfristig ein Verhältnis zum Kunden, bei dem beide Seiten von der engen Zusammenarbeit profitieren.

5.4. Definierte Prozesse, Projektbezug und Anreizsysteme

Eine stark projektbezogene Tätigkeit kann helfen, die Qualität der entwickelten Software zu erhöhen. Wichtig ist, dass Prozesse klar definiert sind und der Ablauf des Projektes im Voraus festgelegt wird. Somit sind Strukturen und Zuständigkeiten ebenso festgelegt, wie Vorgehensweisen und Rahmenbedingungen. Dabei muss die Festlegung keineswegs zu starren Prozessen führen; eine Flexibilisierung kann nach wie vor erhalten bleiben, um etwa auf zusätzliche Anforderungen des Kunden reagieren zu können. Auch diese Möglichkeit sollte dann allerdings im Projektplan vorgesehen sein. Empfehlenswert ist auch, keine Prozesse „von der Stange“ zu implementieren. Beratungsunternehmen empfehlen typischerweise (eventuell geringfügig anpassbare) Standardprozesse und auch Werkzeuge geben häufig bestimmte Arbeitsschritte vor. Auch wenn es natürlich grobe Vorgehensweisen gibt, die allgemein sinnvoll sind, sollte stets nach einem auf die Unternehmensituation angepassten Prozess gestrebt werden. Erweist sich ein Werkzeug als zu unflexibel dazu und ist es nicht anzupassen, sollte eher der Einsatz desselben überdacht, als der ungünstige Prozesse implementiert werden.

Durch ein projektbezogenes Vorgehen identifizieren sich Mitarbeiter stärker mit der zu entwickelnden Applikation. Da das Testen direkt in das Projekt und den zugrunde liegenden Gesamtprozess integriert wird, etabliert es sich als selbstverständliche Maßnahme, die nicht in Frage gestellt wird. Auch die Zusammenarbeit bzw. die Kompromissfindung kann verbessert werden, etwa im Zusammenspiel von Analysten, Entwicklern und Testern.

Ohnehin muss die Integration des Testens in den Entwicklungsprozess als erfolgreiche Strategie festgehalten werden. Insbesondere, wenn es dedizierte Tester, möglicherweise sogar eine eigene Testabteilung gibt, entsteht für Entwickler schnell der Eindruck, dass es sich bei Testern um „Gegenspieler“ handelt. Diesem Eindruck muss entgegengewirkt werden. Tester müssen mit den Entwicklern zusammenarbeiten. Durch die Einrichtung

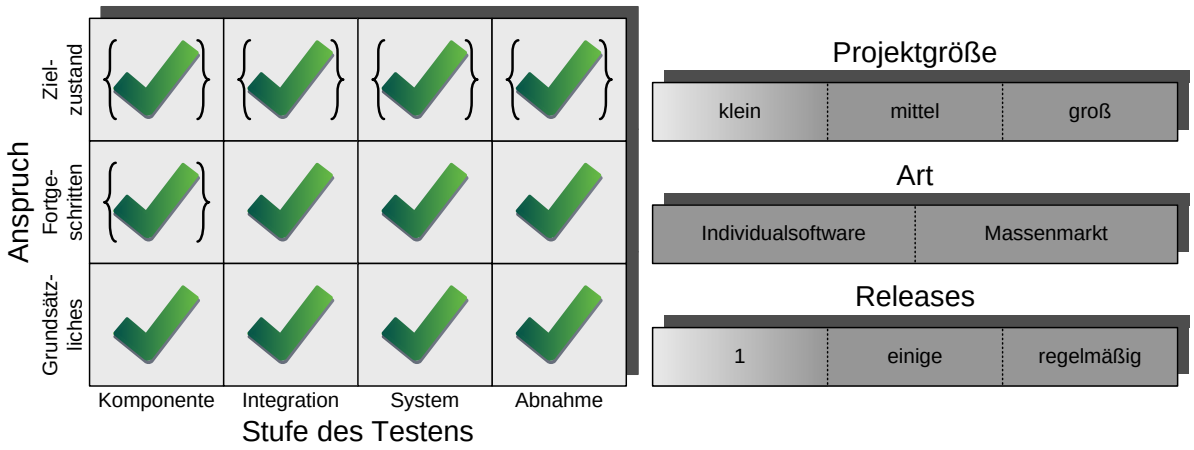

Abbildung 5.3.: Einordnung der Handlungsempfehlung

entsprechender Anreizsysteme muss das Testen als Selbstverständliche Aufgabe etabliert werden, die auch von Entwicklern in allen Phasen voll akzeptiert und unterstützt wird. Keineswegs sollte es – wie vielfach anzutreffen – als potentielle „Bedrohung“ gesehen werden.

Insbesondere bei kurzen Projektlaufzeiten und kleineren Entwicklungsgruppen steht der Entwickler häufig vor einem Dilemma. Wenn Kosten und Zeitaufwand vorgegeben sind, ist die in der Entwicklung zu erreichende Qualität ebenfalls fix. Ist die Zeit zu knapp kalkuliert, muss der Entwickler entweder eine Entscheidung gegen die Qualität treffen, oder den angesetzten Termin überschreiten. In beiden Fällen hat er in der Regel mit negativen Konsequenzen zu rechnen. Im ersten Fall ist es wahrscheinlich, dass er zwar den gewünschten Funktionsumfang implementiert hat, aber „keine Zeit mehr“ für die nötigen Komponententests hatte. Im zweiten Fall hat er wahrscheinlich ausreichend getestet, aber aufgrund der Zeitüberschreitung bleibt keine Möglichkeit mehr für einen Integrations- und Systemtest. Beide Fälle können die Qualität der Software nachhaltig schmälern.

Die Lösung dieses Dilemmas muss organisatorisch erfolgen: Entwickler dürfen für Zeitüberschreitungen durch notwendige Tests nicht „abgestraft“ werden. Vielmehr sollte eine ganzheitliche Testorganisation und ein durchgängiger Softwareentwicklungsprozess, der Testprozesse für unterschiedliche Testarten integriert, zum Standard werden. Am einfachsten ist dieses mit dem oben motivierten Prozessbezug möglich. Wenn Mitarbeiter nicht in Projekte eingebunden, sondern „frei“ sind, ist es deutlich schwieriger, das Einhalten der Prozesse und den adäquaten Einsatz der Mitarbeiter für die ihnen zugeordneten Projekte sicherzustellen.

Die oben angesprochenen, groben Ziele lassen sich im Detail durch zahlreiche in dieser Broschüre vorgeschlagene Maßnahme erreichen, bzw., die Erreichbarkeit der Ziele lässt sich durch die Maßnahmen erhöhen. Weitere Maßnahmen können dazu beitragen, mögliche Probleme im Vorfeld zu erkennen. So kann etwa die Erfassung und Auswertung von Kennzahlen (siehe Kapitel 5.13) dazu beitragen, zeitliche Probleme recht-

zeitig zu erkennen und ihnen entgegen zu steuern, bevor sie ernsthafte Konsequenzen nach sich ziehen. Um Maßnahmen zur Strukturierung und auch den Anreizsystemen den entsprechenden Nachdruck zu verleihen, kann darüber nachgedacht werden, sie per Geschäftsanweisung vorzugeben. Bestimmte Bestandteile der Testprozesse und auch die zu verwenden Werkzeuge als Richtlinie, zumindest aber als Empfehlung vorzugeben, hilft, diese durchzusetzen. Die Richtlinien sollten dabei so gestaltet werden, dass sie ein einheitliches Vorgehen unterstützen, aber Innovationen und Prozessoptimierungen nicht verhindern. Dies wäre etwa möglich, indem die optionale Verwendung zusätzliches Tools erlaubt und ein Vorschlagswesen installiert wird.

Wir empfehlen allen Unternehmen, die in diesem obigen Abschnitten diskutierten Maßnahmen zu bedenken. Alle von ihnen bieten die Möglichkeit, ohne allzu großen Aufwand implementiert zu werden. Gleichzeitig haben sie das Potential, erheblich ausgebaut zu werden.

5.5. Klare Verantwortlichkeiten

Es scheint außer Frage zu stehen, dass die Verantwortung für bestimmte Schritte der Softwareentwicklung, insbesondere einzelne Testprozesse, klar festgelegt werden. Tatsächlich sind Verantwortlichkeiten allerdings häufig nicht klar geregelt. Eventuell wechseln sie von Projekt zu Projekt oder sind gar willkürlich. Dies ist äußerst kontraproduktiv und kann schnell zu Konflikten führen. Schlimmer noch, es besteht die Gefahr, dass bei Problemen kein klarer Ansprechpartner zu benennen ist und eine Eskalation sehr zeitaufwendig wird.

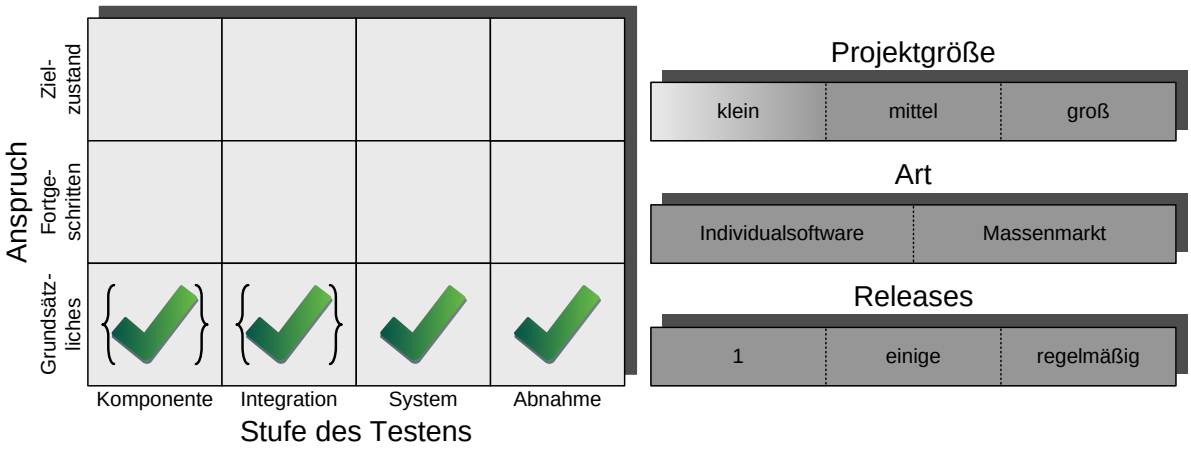

Abbildung 5.4.: Einordnung der Handlungsempfehlung

Werden Verantwortlichkeiten festgelegt, sollte dabei nicht nur an die mögliche *Haftung* des Verantwortlichen gedacht werden, also das Einstehen für die Erreichung der in seinem Verantwortungsbereich festgelegten Ziele (vergleiche auch mit der Diskussion von Anreizsystemen in Kapitel 5.4). Vielmehr benötigt er auch die entsprechende *Macht* zur

Durchsetzung notwendiger Maßnahmen. Selbiges gilt für die erforderlichen Ressourcen. Im Zuge der Festlegung von Verantwortlichkeiten sollte demnach auch geprüft werden, ob die Verantwortungsbereiche gut strukturiert sind und in sinnvoller Weise Entwicklungsschritte und Ähnliches kapseln. Als Schnittstelle zwischen Verantwortungsbereichen sollte dabei immer eine formal geregelte Freigabe (siehe auch unten) dienen. Nur so kann sichergestellt werden, dass die klare Strukturierung sich in der betrieblichen Praxis widerspiegelt.

Die Empfehlung, auch Verantwortlichkeiten festzulegen, schließt sich an den Tipp an, Rollen festzulegen und Prozesse zu definieren. Die dadurch geschaffene Strukturierung macht die Testprozesse umso effektiver, je klarer die Verantwortlichkeiten festgelegt sind. Ziel ist dabei keine starre Struktur; vielmehr sollte erreicht werden, dass Mitarbeiter mit den für sie geeignetsten Aufgaben betreut werden und in Problem- und Sonderfällen stets Ansprechpartner zur Verfügung stehen. Konfliktäre Situation können dann häufig direkt und ohne aufwendigen Eskalationsprozess gelöst werden. Klare Verantwortlichkeiten haben dabei grundsätzlich Vorteile für alle Phasen des Testens, aber in den frühen Phasen sind die Auswirkungen unklarer Verantwortlichkeiten geringer.

5.6. Aufbau eines Testzentrums

Ab einer deutlich zweistelligen Anzahl von Mitarbeitern in der Softwareentwicklung ist es für ein Unternehmen ohne Frage sinnvoll, Stellen für dedizierte Tester zu schaffen. Darüber hinaus ist es für sie lohnenswert, über die Schaffung einer zentralen Testorganisation (z. B. als Organisationseinheit Testzentrum) nachzudenken. Für Unternehmen, die eine dreistellige Anzahl von Mitarbeitern in der Softwareentwicklung beschäftigen, sollte die Einrichtung eines Testzentrums als obligatorisch gelten. Dabei geht es nicht darum, das Testen an sich zu zentralisieren. Vielmehr soll eine unternehmensweite Instanz geschaffen werden, die das Testen koordiniert, Empfehlungen und Richtlinien erstellt, eventuell Ressourcen und Mitarbeiter zur Verfügung stellt und die Testprozesse transparenter machen kann.

Die vordergründigste Aufgabe des Testzentrums ist die Koordination. Es bildet eine Querschnittsfunktion. Von den Abteilungen, die seine Leistungen in Anspruch nehmen, darf es als „ordnende Hand“ wahrgenommen werden, solle aber stets eine Ausrichtung stärker als Unterstützer denn als Regulierer haben. In dieser Funktion sollte es einheitliche Verfahren und Methoden für das Testen festlegen und kommunizieren. Bei der Implementierung sollte es beratend zur Verfügung stehen. Des Weiteren kann es Empfehlungen und Richtlinien erstellen. Die Richtlinien kommen überall dort zum Tragen, wo ein einheitliches Vorgehen zwingend erforderlich ist oder sich besonders sinnvolle Vorgehensweisen herauskristallisiert haben. Die Empfehlungen haben für die Abteilungen einen optionalen Charakter, sollten aber idealerweise als zuverlässige Hinweise auf gute Strategien wahrgenommen werden. Sinnvoll ist etwa, dass das Testzentrum zusammen mit einigen Abteilungen oder im Rahmen von einigen Projekten Werkzeuge evaluiert und die Ergebnisse dieser Evaluation dann zur Verfügung stellt. Als weitere Aufgabe kann das Testzentrum auch als Vermittler zwischen Entwicklern und Fachabteilungen

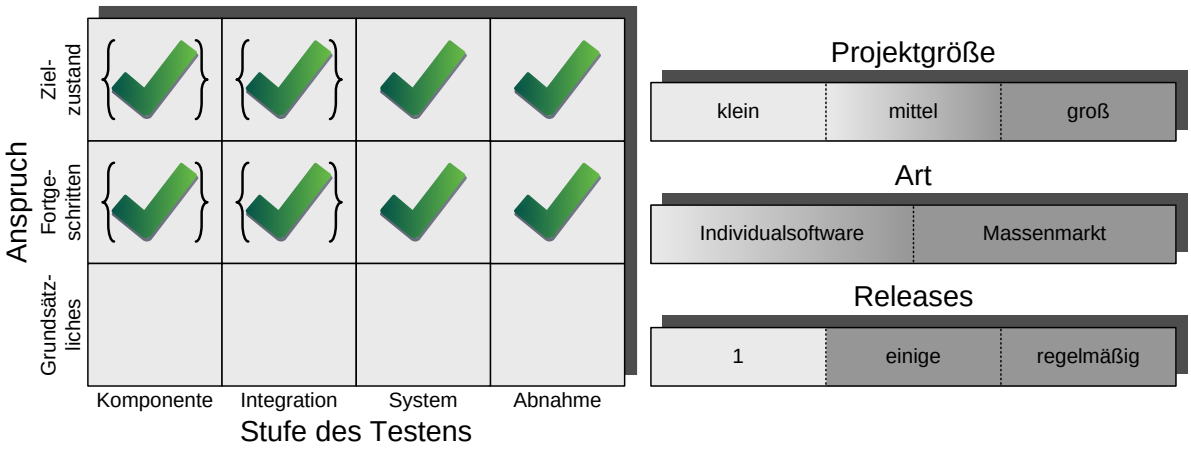

Abbildung 5.5.: Einordnung der Handlungsempfehlung

bzw. Vertrieb auftreten. Seine Mitarbeiter sollten stets über den reinen Entwicklungshorizont hinaus blicken können; ihr Fokus liegt auf der Softwarequalität, nicht etwa auf technischen Details.

Falls es Fachabteilungen gibt, die keine Tester beschäftigen aber Software entwickeln, oder falls projektbezogen Tester fehlen, könnte das Testzentrum einen Pool von Testern bereitstellen, um auszuhelfen. Auch wäre es denkbar, dass die späteren Testphasen stets von Mitarbeitern des Testzentrums begleitet werden. Diese könnten ihre Erfahrung einbringen, um die zu erstellende Software qualitativ „abzurunden“. Detaillierte Empfehlungen können an dieser Stelle nicht gegeben werden, aber die Möglichkeit, Ressourcen und Mitarbeiter durch das Testzentrum bereitzustellen, sollte insbesondere durch größere Unternehmen in Erwägung gezogen werden.

Ein Testzentrum kann auch für eine erhöhte Transparenz sorgen. So macht es Sinn, durch dieses die Dokumentation von Tests zusammen mit den Entwicklern und Fachabteilungen koordinieren zu lassen. Dabei sollte es etwa einheitliche Benennungskonventionen und Ähnliches vorgeben. Gegebenenfalls verwaltet es standardisierte Dokumentenvorlage oder Verwaltungssysteme. Als erweiterte Maßnahme bildet es auch den richtigen Rahmen, ein Testfall-Controlling aufzusetzen (siehe Kapitel 5.13).

Insgesamt ist zu sagen, dass ein Testzentrum Wissen und Erfahrung bündeln sollte. In dieser Rolle kann es das Wissen zurück in das Unternehmen und in einzelne Projekte tragen. Ein im Rahmen des Projektes interviewter Mitarbeiter sprach in dieser Hinsicht von einem „Bauchgefühl“ der erfahrenen Mitarbeiter des Testzentrums. Auch kann es Aufgaben bündeln, die häufiger anfallen, aber mitunter projektbezogen nicht wirtschaftlich sind, wie die umfangreiche Evaluation von Testwerkzeugen. Ein großer Vorteil ist, dass das Testzentrum losgelöst von der Entwicklung arbeitet. Es kann seine Meinung offen kundtun, ohne das es dadurch Konflikte gibt. Uneinheitlichkeit herrscht allerdings bezüglich der Macht, mit der das Testzentrum ausgestattet werden soll. So muss das Verhältnis von Empfehlungen zu Richtlinien, die durch das Testzentrum erstellt werden, behutsam auf die Unternehmenssituation angepasst werden. Möglicherweise sollte

ein Testzentrum zunächst als rein beratende Instanz starten und die Empfehlungen mit zunehmender Erfahrung verbindlich werden. Festzulegen ist auch, ob das Testzentrum einen starken Einfluss auf Projekte nehmen soll. Denkbar ist zum einen der Ansatz, dass es zwar durch Richtlinien bestimmte Vorgehensweisen vorgibt, ansonsten aber passiv ist und auf Anfrage tätig wird. Zum anderen wäre ein (pro-)aktives Testzentrum möglich, das Einfluss auf – aus seiner Sicht – problematische Projekte nimmt oder dieser gar stoppen kann. Diesbezügliche Empfehlungen gehen über das Ziel dieser Broschüre hinaus; die Implementierung eines Testzentrums wird allerdings als starke Empfehlung für größere Unternehmen festgehalten.

5.7. Abbau systematischer Komplexität

Umfangreiche Projekte können schnell eine sehr hohe Komplexität erreichen. In noch stärkerem Maße gilt dies für stetig gepflegte Produkte, von denen regelmäßig neue Releases erscheinen. Wachsen solche Systeme über die Zeit, ergeben sich komplexe Abhängigkeiten. Und auch die Einbettung neuer Systeme sowie die Pflege bestehender Komponenten umfangreicher Unternehmensarchitekturen kann eine sehr hohe Komplexität erreichen.

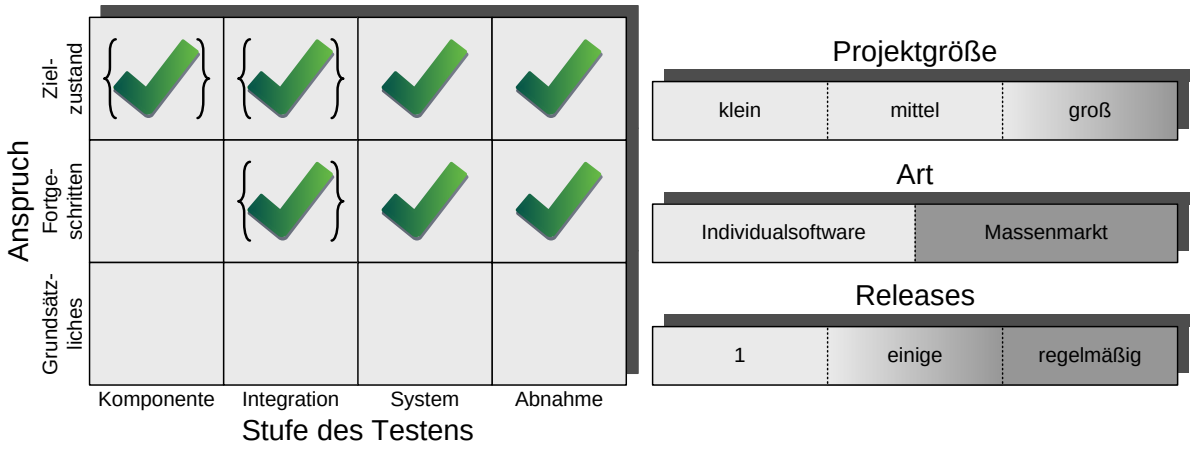

Abbildung 5.6.: Einordnung der Handlungsempfehlung

Komplexität ergibt sich dabei in verschiedener Hinsicht. Besteht ein Softwaresystem aus zahlreichen Komponenten, wird es zunehmend schwierig, den Überblick über alle Komponenten und Ihre Schnittstellen zu behalten. Erscheinen regelmäßig Releases, müssen Abhängigkeiten zu anderen Systemen beachtet, etwaige Sonderbedingungen beim Einsatz bei einigen Kunden im Hinterkopf behalten und die Migration von der letzten Version geplant und vorgenommen werden. Bei komplexen, gewachsenen Unternehmensstrukturen kann es mitunter schwierig sein, einfachste Systeme auszutauschen, wenn unbekannt ist, welche Systeme in welcher Form von ihnen abhängen und wie Schnittstellen spezifiziert sind. Hinzu kommen technische Hürden wie inkompatible Hardware oder

Software (z. B. Betriebssysteme oder Applikationsserver) oder eine heterogene Programmiersprachenverwendung. Mithin ist für alte Systeme kein Quelltext mehr vorhanden, so dass ihre Analyse oder Weiterentwicklung äußerst schwierig und zeitaufwendig ist. Auch mangelt es häufig an einer ausreichenden und aktuellen Dokumentation der Systeme.

Ohne geeignete Maßnahmen wächst der Aufwand, der von der Bewältigung der Komplexität verursacht wird, nicht linear mit dem Umfang der Systeme. Vielmehr wächst er (über-)exponentiell. Daher sollte – auch schon bei mittleren Projekten mit Wachstumsaussichten – darauf geachtet werden, systematische Komplexität von Beginn an unter Kontrolle zu halten. Neben der klaren Definition von Prozessen, Zuständigkeiten usw. (siehe Kapitel 5.4) ist dafür vor allem eine klare Strukturierung nötig. In diesem Rahmen sollten geeignete Abstraktionsstufen gebildet werden, die aber über Schnittstellen zu der jeweils niedrigeren Stufe verfügen. Um ein einfaches Beispiel zu konstruieren: Es existieren drei Softwaresysteme, die untereinander Daten austauschen. Sie werden unabhängig durch Teams von Experten entwickelt. Zur Verwaltung sollte es nicht nur Gespräche zwischen den Leitern der drei Teams geben; vielmehr wäre eine Rolle einer Verwalters sinnvoll, der alle drei Systeme und ihre Schnittstellen grob kennt. Darüber hinaus muss er den jeweils geeignetsten Experten für die den jeweiligen Schnittstellen zugrunde liegenden Programmmodule kennen. So können die Experten sich auf ihren Kernbereich beschränken und werden, wenn nötig, direkt mit konkreten Fragestellungen angesprochen. Der Verwalter behält den Überblick, ohne zu tiefe technische Details zu kennen.

In der Praxis lassen sich für größere Projekte oder umfangreiche Systemlandschaften mehrere Abstraktionsebenen finden. So ließen sich etwa Gruppen von Systemen, die sehr eng miteinander arbeiten, zu einer Verwaltungseinheit höherer Ordnung zusammenfassen. Einerseits führt diese Form von Abstraktion zu einem gewissen Verwaltungsaufwand, da Anfragen gegebenenfalls mehrere Stufen durchlaufen, bevor sie den eigentlich Ansprechpartner erreichen. Andererseits wird der richtige Ansprechpartner mit hoher Wahrscheinlichkeit erreicht. Zudem stehen die Verwalter als Mittler zur Verfügung, etwa für Gespräche zwischen Technikern und Betriebswirten. Die Strukturierung und Modularisierung in organisatorischer Hinsicht führt darüber hinaus zu einer exponentiellen Verringerung der wahrgenommenen Komplexität des einzelnen Mitarbeiters — denn ab einem gewissen Komplexitätsniveau ist es unmöglich, einen komplett detaillierten Gesamtüberblick zu behalten. An dessen Stelle tritt ein Überblick über einen Teilbereich (oder das Projekt bzw. die Systemlandschaft) mit dem Wissen um die richtigen Ansprechpartner und groben Zusammenhänge. Die technische Strukturierung bzw. Modularisierung wird auch in Kapitel 6 thematisiert.

Auf einer niedrigeren technischen Ebene lässt sich der Komplexität innerhalb und zwischen Systemen damit begegnen, dass Entwicklungsparadigmen entsprechend angepasst werden. Empfehlenswert ist vor allem eine lose Kopplung und recht einfache Schnittstellen. Dabei ist insbesondere auf die inneren Adhäsion zu achten [YC79]. Sinnvoll ist auch die Orientierung an fachlich übergreifenden Referenzarchitekturen.

Bei umfangreichen Softwareprojekten, die regelmäßig zu neuen Releases der Software führen, ist unbedingt zu empfehlen, alle entstandenen Abhängigkeiten zu dokumentieren. Idealerweise sollten neue Releases automatisch auf die Verletzung von Schnittstellenkon-

trakten geprüft werden. Für umfangreiche Systemlandschaften, in die Software integriert werden soll, empfiehlt sich die Erstellung so genannter *Systemlandkarten*. Diese stellen die verfügbaren Systeme und ihre Abhängigkeiten dar. Wichtig ist, dass die Landkarten in unterschiedlichen Granularitäten vorliegen und situationsadäquate Informationen liefern. So kann es etwa für die Verwaltung von Systemen eine Rolle spielen, auf welcher Hardware dieser laufen. Für die Kopplung derselben kann von dieser Schicht aber komplett abstrahiert werden.[1]

Wichtig ist, dass möglichst frühzeitig mit der Dokumentation entsprechender Informationen begonnen wird. Ist ein Projekt erst einmal gewachsen oder eine umfangreiche Systemlandschaft vorhanden, ist es sehr schwer, diese korrekt zu dokumentieren. Beginnt die Dokumentation hingegen zu einem Zeitpunkt, bei dem die Überschaubarkeit noch gegeben ist, fällt sie viel leichter. Wichtig ist ebenfalls, die Daten stets aktuell zu halten und nicht mehr relevante Daten zu entfernen. Idealerweise ist auch ein Risikomanagement auf Basis der Daten möglich. Es könnte z. B. Programmmodule oder Systeme identifizieren, die in Zukunft Flaschenhälse darstellen könnten, oder warnen, wenn nur noch wenige Mitarbeiter Kenntnisse einer Funktion haben (Gefahr des Wissensabflusses).

Für größere Unternehmen bietet sich darüber hinausgehend die Option, ein *Betriebliches Kontinuitätsmanagement* (*Business Continuity Management*) einzuführen. Systemlandkarten und ähnliche Dokumente können hierfür die Grundlage bieten. Ein Kontinuitätsmanagement hat zum Ziel, Konzepte zu erstellen und Maßnahmen vorzusehen, die der Aufrechterhaltung des Betriebs auch in problematischen Situationen ermöglicht. Während die angesprochenen Dokumente bereits pro-aktiv verwendet werden können, um Probleme zu vermeiden, helfen sie auch im Rahmen eines Kontinuitätsmanagements akute Probleme schneller in den Griff zu bekommen und in ihren Folgen zu mindern. Das Thema *Betriebliches Kontinuitätsmanagement* geht eindeutig über den Rahmen dieser Broschüre hinaus. Die Empfehlung für die Beschäftigung mit der Thematik sei dennoch ausgesprochen. Auch für Unternehmen, die zumindest eine deutlich zweistellige Mitarbeiterzahl haben, empfiehlt sich ein Blick auf Möglichkeiten zur Risikoprävention und Krisenbewältigung.

5.8. Enge Zusammenarbeit der Entwickler / Vier-Augen-Prinzip

Manche Prinzipien sind naheliegend, aber dennoch in der Praxis nicht weit verbreitet bzw. nicht konsequent umgesetzt. Dazu zählt vor allem eine enge Zusammenarbeit unter den Entwicklern einer Software. In den Gesprächen mit dem am Projekt beteiligten Unternehmen kristallisierte sich heraus, dass es nicht mehr zeitgemäß ist, einzelne Programmierer vorgegebene Teile eines Systems entwickeln zu lassen und diese erst im Rahmen einer deutlich später angesetzten Integrationsphase zusammenzuführen. Vielmehr wurde empfohlen, eine enge Zusammenarbeit der Entwickler anzustreben.

[1] Vergleiche hierzu auch die Ausführungen zur Modularisierung und Dienstorientierung in Kapitel 6.5.

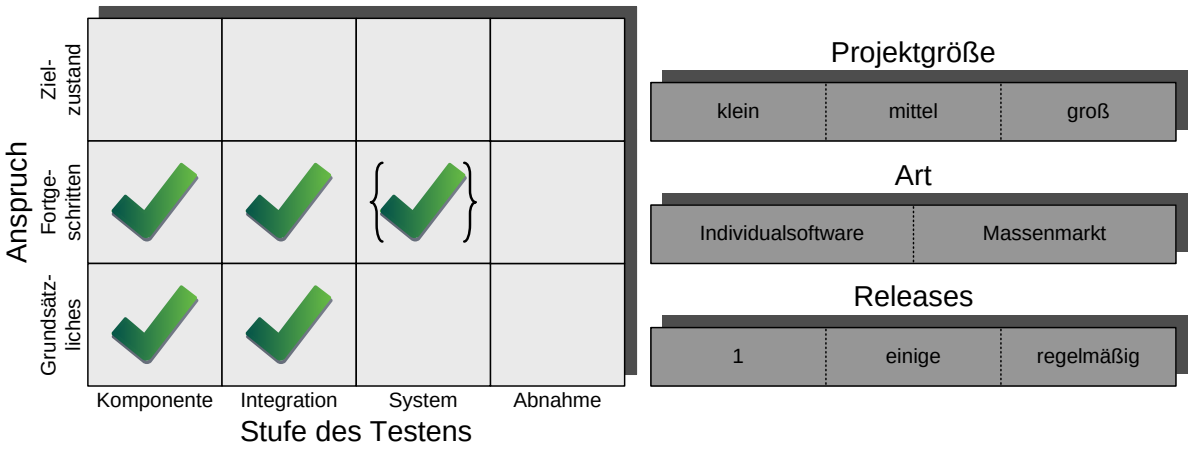

Abbildung 5.7.: Einordnung der Handlungsempfehlung

Die Zusammenarbeit lässt sich sowohl formell als auch informell regeln. Ein typischer Ansatz ist die aus dem Extreme Programming [Bec00] bekannte Paarprogrammierung. Dabei arbeiten Entwickler nicht allein, sondern in Paaren. Die scheinbar niedrigere Effizienz durch die höhere Personalbindung wird durch eine höhere Code-Qualität und höhere Produktivität (über den Tage verteilte Konzentrationsschwächen des einzelnen Entwicklers werden ausgeglichen) wieder aufgewogen. Paarprogrammierung als generelles Konzept ist sicherlich nicht unumstritten, als Maßnahme für besondere Fälle aber unbedingt zu empfehlen. Beispiele sind die Programmierung von Modulen, die ohnehin durch mehrere Entwickler genutzte werden sollen (es entfällt die Einarbeitung) oder besonders komplizierter Algorithmen.

In eine ähnliche Richtung wie die Paarprogrammierung geht die gegenseitige Begutachtung der Entwickler. Dabei werden Module zwar in Einzelarbeit entwickelt, bevor sie als fertig gestellt gelten erfolgt allerdings eine Begutachtung durch einen zweiten, gegebenenfalls auch einen dritten Entwickler. In den am Projekt beteiligten Unternehmen wurde dieses Vorgehen auch als *Vier-* bzw. *Sechs-Augen-Prinzip* der Entwicklung bezeichnet. Wichtig dabei ist, dass die Begutachtung verbindlich geregelt ist. Sie muss zwar nicht notwendigerweise für alle Quelltexte erfolgen, sondern kann etwa auf besonders wichtige Module oder sehr komplizierte Algorithmen beschränkt werden. Wird sie aber eingesetzt, so sollte an ihrem Ende eine formelle Freigabe durch den oder die Begutachter stehen. Wie Fehler behoben werden, ist Auslegungssache. Möglich ist, das Modul zur Überarbeitung zurück an den Entwickler zu geben, eine Behebung möglicher Fehler bzw. Probleme durch einen Begutachter, oder eine gemeinsame Bearbeitung durch Entwickler und Begutachter gemeinsam. Besonders empfehlenswert ist die letzte Variante. Schließlich sollte der Begutachtungsprozess von den Entwicklern nicht als Möglichkeit für Kollegen wahrgenommen werden, unsachliche Kritik zu üben. Durch die gemeinsame Behebung von Problemen kann die Kommunikation unter den Entwicklern verbessert werden. Idealerweise lernen sie voneinander. Demnach sollte die Begutachtung von den Mitarbeitern nicht gefürchtet werden; Entwickler sollten weder mit Sanktionen aufgrund

gefundener Fehler rechnen müssen, noch in ihrer Rolle als Begutachter solche für ihre Kollegen befürchten und dadurch unbewusst weniger genau arbeiten. Vielmehr sollte die Begutachtung als Prozess zur ständigen Optimierung der Qualität implementiert und gelebt werden.

Als Nebeneffekt der Begutachtung wächst der Kreis der Entwickler, die sich mit einer bestimmten Stelle des Quelltextes auskennen. Dies ist äußerst nützlich, wenn Mitarbeiter urlaubs- oder krankheitsbedingt nicht zu erreichen sind oder aus dem Unternehmen ausscheiden. Eine aufwendige Übergabe oder die fehlerträchtige Einarbeitung entfällt, da in der Regel mindestens ein weiterer Mitarbeiter zur Verfügung steht, der sich mit dem betroffenen Modul auskennt.

Wichtig für ein erfolgreiches Vier-Augen-Prinzip ist, dass die Verantwortlichkeiten klar geregelt sind (siehe auch oben). Die Zuordnung von Entwicklern und Begutachtern muss mit Bedacht gewählt werden, denn es lassen sich sowohl Argumente für häufige Wechsel der Zuordnung wie für die Etablierung eingespielter Teams finden. Ebenso festgelegt werden muss, für welche Entwicklungen die Begutachtung zum Einsatz kommt. Um Willkür zu verhindern und um die Begutachtung als verlässliches Instrument zu implementieren, sollten alle Parameter bereits im Vorfeld der zu begutachtenden Entwicklung festgelegt und in der Regel nach nicht mehr angepasst werden.

5.9. Stufenkonzept und Einbettung in ein Gesamtkonzept

Das Testen von Software ist eine sehr vielschichtige Aufgabe, die verschiedenste Phasen umfasst, die sich in einzelne Schritt untergliedern lassen, und zu der mannigfaltige Aktivitäten gehören. Zu den wesentlichen Erkenntnissen, die in den Interviews mit den am Projekt beteiligten Unternehmen gewonnen werden konnten, zählt, dass es sehr erstrebenswert ist, das Testen in ein Gesamtkonzept einzubetten und ein Stufenkonzept zu erarbeiten.

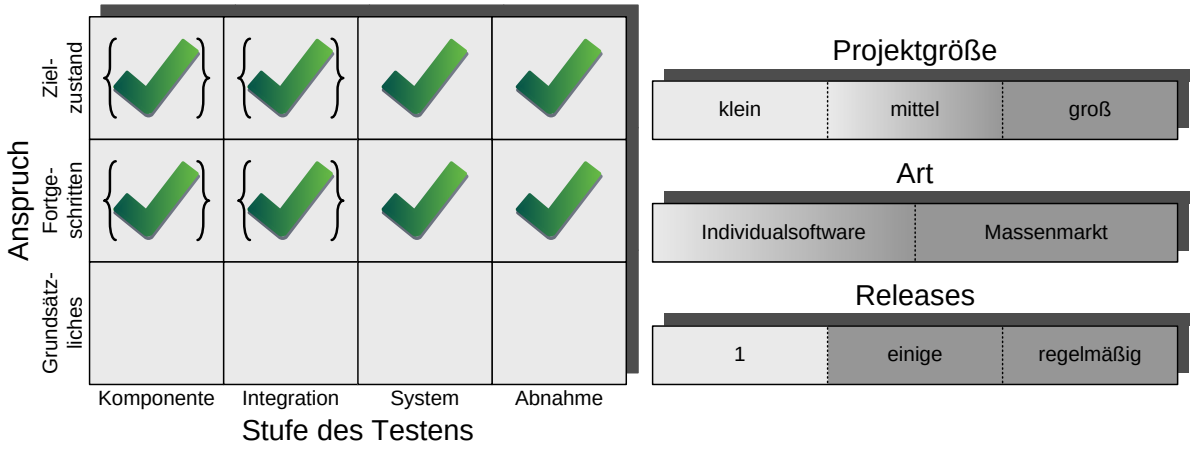

Abbildung 5.8.: Einordnung der Handlungsempfehlung

Ein Stufenkonzept, in der Praxis z. T. auch als *Staging-Konzept*, *Staging-Plan* oder schlicht *Staging* bezeichnet, untergliedert den Testprozess ist unterschiedliche Phasen. Typischerweise wird dabei zwischen den groben Testphasen unterschieden, die schon in Kapitel 4.5 motiviert wurden:

- Komponententest: Entwickler testen den Code, den die selbst erstellt haben. Getestet werden isolierte Module. Die Formalisierung des Tests ist in der Regel recht gering.

- Integrationstest: Mehrere Module des zu entwickelnden Systems werden integriert und das Zusammenspiel getestet. Verantwortlich sind häufig nicht mehr die Entwickler selbst, sondern Tester bzw. speziell mit der Integration beauftragte Mitarbeiter.

- Systemtest: Wesentliche Module des Systems werden im Zusammenspiel getestet. Dabei steht das System größtenteils, zumindest zum Ende des Systemtests hin, in der Form zur Verfügung, in der es auch produktiv laufen soll.

- Abnahmetest: Bevor das System in den Kundeneinsatz gelangt, erfolgt ein Abnahmetests. Dabei wird das System gegen seine Anforderungen getestet. Mitunter sind Pilotkunden an diesem Test bereits beteiligt.

Je nach Situation im Unternehmen ist es auch möglich, eine weitere Untergliederung vorzunehmen und etwa eine Beta-Test-Phase einzuführen, bei der ein größerer Kreis ausgewählter Kunden eine Vorab-Version der zu entwickelnden Software testet. Ein ebenfalls im Rahmen des Projekts gesehener Ansatz war, zwei Integrationstest-Phasen vorzusehen. In der ersten integrieren Entwickler neu geschriebene Module in das Gesamtsystem und prüfen, ob das Zusammenspiel erfolgreich ist. Die zweite Phase entspricht dann dem oben beschriebenen Integrationstest, bei dem Entwickler und Tester nicht mehr identisch sind (vergleiche auch Kapitel 3.3).

Besonders wichtig ist, die Phasen aneinander anzuschließen. Die Ergebnisse jeder Phase sollten formell dokumentiert werden. So kann für zukünftige Tests gelernt werden und die Mitarbeiter, die mit der jeweils nachfolgenden Phase betreut sind, erhalten die für ihre Arbeit benötigten Informationen. Am Ende jeder Phase sollte eine Freigabe stehen. Mit dieser wird bescheinigt, dass die Software allen Anforderungen, die sie am Ende der jeweiligen Phase genügen soll und muss, tatsächlich genügt und somit ein Niveau erreicht hat, bei der die nächste Phase angeschlossen werden kann. Die Freigabe entspricht somit der formellen Zusicherung einer bestimmten Qualität. Wird diese Qualität nicht erreicht, kann die Zusicherung erst dann gegeben werden, wenn entsprechende Änderungen an der Software vorgenommen wurden. Im Rahmen des Stufenkonzepts sollte dabei genauestens geregelt werden, *welche* Mitarbeiter bzw. *welche* Rollen in *welcher* Phase *welche* Aufgabe übernehmen, durch *wen* die Freigabe erteilt wird, *welche* Art von Informationen dokumentiert und weitergegeben wird, und *wie* bei etwaigen Probleme zu verfahren ist. Auch wenn die Verantwortung des Entwicklers etwa mit der Freigabe für den Integrationstests grundsätzlich enden könnte, muss auf ihn bei gefundenen Fehlern in den

folgenden Phasen weiterhin zurückgegriffen werden. Die genaue Konzeption des Stufenkonzepts und die strikte Einhaltung der Freigaben (keine Phase darf ohne Freigabe in der vorherigen Phase starten) kann als sehr erfolgreiche Strategie festgehalten und sehr empfohlen werden.

Denkbar sind weitere Ausbaumöglichkeiten des Stufenkonzepts. So wäre es etwa möglich, die Freigabe in der Systemtestphase in mehrere Hände zu legen. Wird das System von Testern untersucht, garantiert die Freigabe mit hoher Wahrscheinlichkeit die Erreichung des angestrebten Qualitätsniveaus in technischer Hinsicht. Ist allerdings zusätzlich noch die Freigabe durch den projektverantwortlichen Fachbereich oder in Kundenkontakt stehender Mitarbeiter nötig, kann viel eher auch die fachliche Qualität garantiert werden. Als weitere Möglichkeit könnten ferner die jeweils verwendeten Testsysteme skizziert werden, die mit jeder Phase aufwendiger werden. Auch der Einbau von Last- und Stresstests ist denkbar. Zur Veranschaulichung des Stufenkonzepts ist ein einfaches Beispiel in Abbildung 5.9 dargestellt.

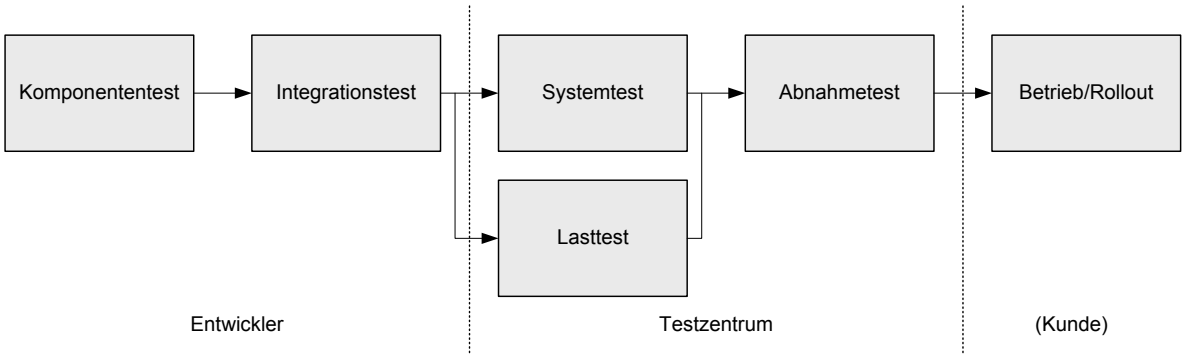

Abbildung 5.9.: Einordnung der Handlungsempfehlung

Die Diskussion weiterer Möglichkeiten übersteigt an dieser Stelle sicherlich den Rahmen der Broschüre; festgehalten werden kann, dass es zahlreiche Möglichkeiten des Ausbaus gibt. Ein Stufenkonzept muss durch seinen bloßen Umfang und die innewohnenden Komplexität schrittweise aufgebaut und an die Realität im Unternehmen angepasst werden. Dort sollte es auf ein Umfeld stetiger Optimierungen treffen, so dass es Teil des gelebten Kultur und integraler Bestandteil der Softwareentwicklung wird.´

Ganz in diesem Sinne ist es weiterhin sehr empfehlenswert, die Softwareentwicklung und insbesondere das Testen in ein Gesamtkonzept einzubetten. Auch wenn das Testen im Unternehmen nicht stiefmütterlich betrachtet, sondern als wichtiger Prozess angesehen wird, mangelt es sehr häufig an einem umgebenden Rahmen. Noch über ein Stufenkonzept hinaus sollte das Testen als Bestandteil der Softwareentwicklung betrachtet und gelebt werden. Je selbstverständlicher es wird, desto höher wird der Beitrag zur Erreichung der Qualitätsziele ausfallen. In diesem Zusammenhang sollten vor allem auch willkürliche Vorgehensweisen ausgeschlossen werden. Natürlich sollten Entwickler und Tester in einem dynamischen Umfeld arbeiten können und nicht auf starre Arbeitsmethoden beschränkt sein. Nichts desto trotz ist eine Formalisierung des Testen, eine Einigung auf gemeinsame Methoden und Werkzeuge und vor allem auf eine gemeinsa-

me Sprache unerlässlich. Nur so kann das Testen wirklich effektiv gestaltet und ständig optimiert werden. Die Erarbeitung des zugrunde liegenden Gesamtkonzept sollte dabei einer von höchster Ebene getragene Aufgabe sein, die das Testen begleitet. Einzelne Empfehlungen können dazu kaum gegeben werden, allerdings lassen sich alle in dieser Broschüre angesprochenen Maßnahmen in einem Gesamtkonzept unterbringen. Die konkrete Ausgestaltung allerdings ist stark vom Unternehmen abhängig und wird immer einem Wandel unterliegen. Wichtig ist, sich die Notwendigkeit eines Gesamtkonzept und der Formalisierung bestimmter Vorgänge und Anforderungen bewusst zu werden. Eine Kultur, in der Veränderungen gescheut werden, weil der Status quo *akzeptabel* ist, gilt es zu vermeiden.

5.10. Dokumentation

Ein häufig unterschätzter aber wichtiger Bestandteil von Softwaretest ist die Dokumentation. Erst die Dokumentation stellt sicher, dass Tests aufeinander aufbauen können, aus Tests gelernt werden kann und eine kontinuierliche Verbesserung des Testprozesses möglich wird.

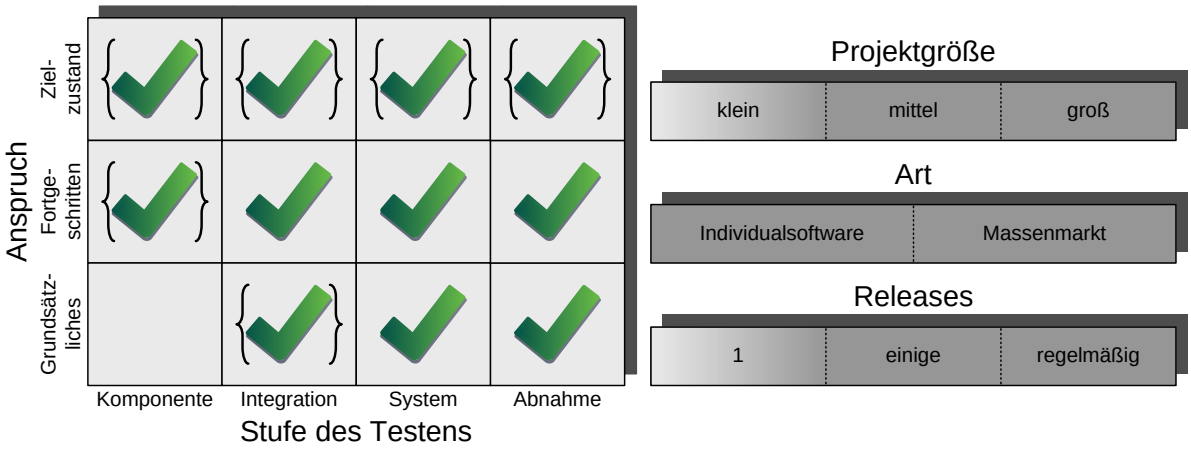

Abbildung 5.10.: Einordnung der Handlungsempfehlung

In den unterschiedlich Testphasen kommen verschiedene Arten von Tests zum Einsatz. Auch liegt der Schwerpunkt jeweils anders; er ist bei Komponententests sehr technisch getrieben und zielt in späteren Testphasen stärker auf das Zusammenspiel von Komponenten und die Funktionsweise des Gesamtsystems ab. Damit die Tests aufeinander abgestimmt werden können, empfiehlt sich die durchgängige Dokumentation. Dabei sollten sowohl organisatorische Daten festgehalten werden, wie etwa die zuständigen Mitarbeiter und zeitliche Angaben, als auch technische Daten, wie der genaue Testablauf, verwendete Werkzeuge und Parametrisierungen. Und natürlich muss festgehalten werden, ob Tests erfolgreich verlaufen sind oder ob Fehler gefunden wurden, die eine Behebung nötig machen. Idealerweise können die Daten direkt mit entsprechenden Testklassen oder -skripten verbunden abgelegt werden (siehe auch Kapitel 6 *Testfallverwaltung*). Somit ist

die Dokumentation auch für den Aufbau von Testfalldatenbanken sowie für die Erstellung von Regressionstests sinnvoll. Die Erfassung, dass gefundene Fehler gelöst wurden, erleichtert das erneute Testen entsprechender Funktionen. Auch kann mithilfe dieser Daten festgestellt werden, ob derselbe Fehler später erneut auftritt. Denn die Behebung des Fehlers muss nicht notwendigerweise direkt perfekt sein, vielmehr könnte sie den Fehler nur kaschiert haben, so dass er zwar im System verblieb, beim konkreten Testfall aber nicht mehr auftrat.

Nur durch die durchgängige Dokumentation ist eine Nachprüfbarkeit und die Möglichkeit des Lernens gegeben. Im Hinblick auf die Nachprüfbarkeit geht es weniger darum, einzelne Mitarbeiter als Verantwortliche für Programmfehler zu identifizieren. Vielmehr hilft die Dokumentation, die korrekten Ansprechpartner zu finden. Vor allem ist sie sehr nützlich, wenn bei vergleichbaren Softwareprojekten oder im Rahmen von späteren Releases ähnliche Fehler bei Tests gefunden werden, wie sie bereits in der Vergangenheit gefunden wurden. Die Dokumentation kann dann nicht nur helfen, Mitarbeiter mit einer ausreichenden Lösungskompetenz zu finden, sondern enthält idealerweise direkt Daten zur Lösung. Sie kann somit helfen, die Zeiten zur Behebung des gefundenen Fehlers drastisch zu reduzieren.

Bereits im Vorfeld kann eine umfangreiche Dokumentation genutzt werden, um typische Probleme zu identifizieren, die bei Software auftreten können. Dies gilt insbesondere für solche Projekte, die Projekten aus der Vergangenheit ähnlich sind. Darüber hinaus sind noch weitere Lerneffekte denkbar: Die Dokumentation stellt bezüglich der Programmanalyse, Testfallgenerierung, Testdurchführung, Fehlerauffindung und -behebung eine Wissensdatenbank dar. Werden Informationen strukturiert eingepflegt und konsequent eine Datenbank aufgebaut, ist der Wert der enthaltenen Daten unschätzbar. Auch für neue Mitarbeiter ist die Datenbank nützlich, um sich in Projekte oder in die Art und Weise, wie im Unternehmen getestet wird, einzuarbeiten. Gegebenenfalls kann die Datenbank auch als Wiki gestaltet werden, um einen stetigen Ausbau der Inhalte zu erreichen. Des Weiteren können die Daten auch dafür genutzt werden, statistische Auswertungen zu erstellen. Weitere Informationen hierzu finden sich im weiter unten folgenden Kapitel 5.13 zu Test-Controlling und Kennzahlensystemen.

Das Dokumentationsformat sollte möglichst strukturiert sein. Zudem sollte es Erweiterungsmöglichkeiten für die Zukunft offen halten. Sehr empfehlenswert ist es überdies, ein zentrales System für die Dokumentation zu verwenden. Häufig gewählte Formate wie etwa Excel-Tabellen stoßen schnell an ihre Grenzen, wenn die Informationen von diversen Personen parallel oder quasi-parallel genutzt werden. Auch drohen System- bzw. Medienbrüche bei der Erstellung von Statistiken. Erweiterte Möglichkeiten des Datenaustauschs werden ebenso erschwert. Detaillierte Informationen hierzu finden sich vor allem in der Diskussion technischer Maßnahmen in Kapitel 6. Denn während die organisatorischen Vorteile sicherlich motiviert wurden, ist die konkrete technische Umsetzung anspruchsvoll und muss daher im Detail dargelegt werden. Als Idealzustand ist eine vollständige Integration der Systeme anzustreben, bei der nicht nur eine unternehmensweite Wissensdatenbank entsteht, sondern ein Informationssystem den effizienten Ablauf des Testens unterstützt. Wird ein solches System geschaffen, kann es die kontinuierliche Verbesserung des Testprozesses sehr fördern, indem es die Datengrundlage für zahlreiche

Folgemaßnahmen bietet und auch helfen kann, den Erfolg von Verbesserungsmaßnahmen zu bewerten.

Als Anregung für das Format der Dokumentation kann der Standard IEEE 829 [IEE98b] in Betracht gezogen werden. Es enthält sehr detaillierte Empfehlungen für den Aufbau entsprechender Dokumente. Letztlich müssen Format und Struktur aber vor auf die eigenen Bedürfnisse abgestimmt sein. Auch sollten Erweiterungsmöglichkeiten offen gehalten werden. So kann mit einer groben Dokumentation gestartet wird, die mit zunehmender Erfahrung verfeinert wird.

Wir empfehlen dringend, Tests strukturiert und detailliert zu dokumentieren. Als Ausgangspunkt, vor allem für kleine Unternehmen, können dabei durchaus einfache Tabellen dienen. Insbesondere für größere Entwicklungsteams und Projekte, die zu häufigen Releases führen, sollte darüber hinaus darüber nachgedacht werden, ein unternehmensweites System zur Dokumentation von Tests einzuführen und dieses nicht als Archiv, sondern als Wissensdatenbank zu betrachten.

Im Rahmen der Auswertung der Ergebnisse wurden wir auch darauf hingewiesen, dass Dokumentationspflichten und das Einhalten der Ordnungsmäßigkeit von Tests zunehmend in Prüfstandards wie *CobiT* (Control Objectives for Information and Related Technology) verankert würden. Hierbei geht es nicht nur um die Dokumentation einzelner Tests, sondern der zugrundeliegenden Prozesse, getroffener Entscheidungen und Ähnlichem. Dokumentiert werden sollten also sowohl technische Vorgänge, als auch organisatorische Rahmenbedingungen. Auf eine ordnungsgemäße Dokumentation würde mittlerweile auch durch Wirtschaftsprüfer verstärkt geachtet.

5.11. Schulungen und Zertifizierungen

Im Umfeld des Testens existieren zahlreiche Zertifizierungsmöglichkeiten. Bekannt sind insbesondere die des *International Software Testing Qualifications Board* (ISTQB) [19], in Deutschland vertreten durch das *German Testing Board* [20].[2] Angeboten werden grundsätzliche, aber auch fortgeschrittene Zertifikate. Letztere Bescheinigen herausragende Kenntnisse in Teilbereichen, etwa bezüglich des Testmanagements oder des technischen Testens.

Zertifizierungen bringen zwar Kosten mit sich, da Gebühren für Seminare und Prüfungen anfallen und Mitarbeiter für die Teilnahme an diesen freigestellt werden müssen. Sie bieten allerdings erhebliche Vorteile. Sobald Softwaretests zumindest zum Teil durch dedizierte Tester vorgenommen werden, muss diesen Vertrauen entgegengebracht werden. Dieses kann über Zertifizierungen erreicht werden.

Nach den Erfahrungen der für diese Broschüre interviewten Unternehmen ist es die Ausnahme, dass neue Mitarbeiter umfangreiche Testerfahrungen mitbringen. Mitarbeiter, die für Tests eingesetzt werden, kommen in der Regel entweder aus der Entwicklung

[2] Es gibt darüber hinaus noch zahlreiche weitere Zertifizierungen im Bereich der Softwareentwicklung, auch speziell für das Testen. Aufgrund des eher allgemeinen Charakters der Thematisierung in dieser Broschüre kann und soll an dieser Stelle weder eine Empfehlung ausgesprochen, noch sollen einzelne Zertifikate verglichen werden.

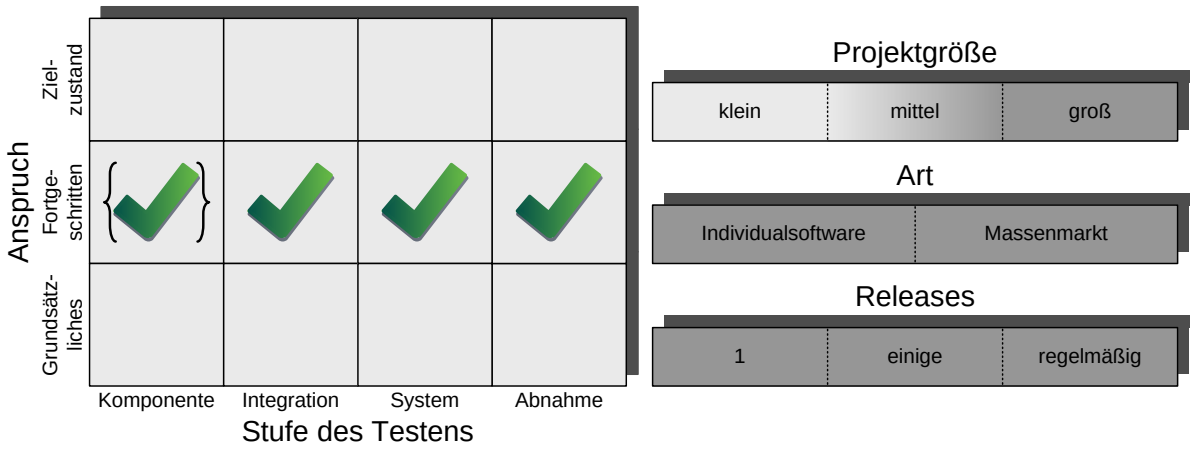

Abbildung 5.11.: Einordnung der Handlungsempfehlung

oder aus den Fachbereichen. Erstere haben ein gutes Verständnis der Programmlogik und können aufgrund ihres Wissens etwa von Elementen der Oberfläche auf Funktionen schließen. Letztere haben vor allem fachliche Ansprüche an die zu testende Software. Insbesondere in Kombination sind beide Arten von Testern wichtig; allerdings fehlt es ihnen in den meisten Fällen an speziellem Testwissen. Dies führt dazu, dass möglicherweise notwendige Tests ausgelassen werden und das Testen weniger effizient gestaltet wird, als es möglich wäre. Zudem kann es zu begrifflichen Problemen kommen. Der Gedanke hinter Äquivalenzklassentests etwa ist relativ intuitiv[3], der Begriff an sich aber vielen Entwicklern und vermutlich fast allen aus Fachbereichen stammenden Testern unbekannt.

Schon durch einfache Weiterbildungsmaßnahmen ist eine erhebliche Steigerung der Testeffizienz zu erwarten. Alle Tester erhalten dadurch eine gemeinsame Basis, so dass begriffliche Probleme vermieden werden. Das Wissen um Testmethoden, Testformalisierung und gängige Teststrategien, eventuell verbunden mit einer anschließenden Werkzeugschulung, kann in hohem Maße dazu beitragen, das Testen zu professionalisieren. Die Testqualität steigt, da „korrekt“ getestet wird und die Tester gleichzeitig ihr entwicklungs- bzw. fachspezfisches Wissen noch gezielter einbringen können. Auch die korrekte Dokumentation von Tests (siehe Kapitel 5.10) muss erlernt werden.

Eine der Fortbildung folgende Zertifizierung zeigt den Kenntnisstand direkt. Nicht nur gegenüber dem Kunden, sondern auch projektintern signalisiert die Zertifizierung, dass den Testern das nötige Vertrauen entgegengebracht werden kann. Dieses legt auch Verantwortung und die Macht zur Durchsetzung notwendiger Interessen in die Hände der Tester, verhindert gleichzeitig aber auch Kompetenzstreitigkeiten unter Entwicklern und Testern.

[3]Bei Äquivalenzklassentests wird versucht, mit wenigen Testfällen, die aus Äquivalenzklassen abgeleitet werden, eine möglichst hohe Fehlerentdeckungsrate zu erreichen. Für ein Formularfeld, das Prozentangaben erfasst, ließen sich etwa die Werte 0 (Minimum), 100 (Maximum) und 50 (Durchschnitt) testen, dazu noch -1 und 101 (ungültige Werte durch Unter- bzw. Überschreitung der zulässigen Grenzen). Ein solches Vorgehen entspricht dem intuitiven Ansatz vieler Tester, auch wenn es systematisiert werden sollte.

Sobald Entwicklungsabteilungen Größen erreichen, in denen einzelne Entwickler keinen Überblick über alle Kollegen mehr haben und dementsprechend Kompetenzen nicht mehr ohne weiteres selbst abzuschätzen vermögen, können sich Zertifizierungen lohnen. Bei Mitarbeiterzahlen im drei oder sogar vierstelligen Bereich sind sie dringend empfehlenswert.

Einfache Zertifizierungen gestalten sich recht kostengünstig und liegen deutlich unter 2.000 EUR für ein mehrtägiges Seminar, das Grundkenntnisse vermittelt (vgl. etwa [21]). Dazu kommen Kosten von einigen hundert EUR für die Prüfung, bei deren Bestehen dass Zertifikat ausgehändigt wird. Weitere Seminare, etwa die Advanced Level des ISTQB [22], sind nicht notwendigerweise viel kostspieliger. Durch *in-house*- oder Gruppenseminare lassen sich die Kosten eventuell reduzieren.

Wenn Zertifizierungen in der eigenen Unternehmenskultur als nicht wichtig angesehen werden und auch keine positive Wirkung auf den Kunden zu erwarten ist, empfehlen sich dennoch zumindest unternehmensinterne Schulungen. Ziel sollte es sein, ähnliche Inhalte wie für die Zertifizierung-Grundprüfungen zu vermitteln (wie etwa für den ISTQB Certified Tester- Foundation Level [23]) und eine interne Bezeichnung für Mitarbeiter einzuführen, die diese Schulung durchlaufen haben. Alternativ ist es möglich, nur noch solche Mitarbeiter als Tester einzusetzen, die vorher geschult wurden. Das Schaffen einer gemeinsamen Begriffs- und Wissensbasis ist mit relativ geringem zeitlichen und personellen Aufwand möglich und selbst für kleine Entwicklungsteams sehr ratsam. Die verfügbare Literatur kann auch genutzt werden, das Vorgehen im eigenen Unternehmen zu reflektieren. Hierfür eignen sich insbesondere Werke zu fortgeschrittenen Themen, etwa zum Testmanagement wie [SRWL08].

5.12. Etablierung von Fach- und Branchenwissen

Unabhängig von möglichen Schulungen und Zertifizierungen stellten die meisten der Teilnehmer des zugrunde liegenden Projekts heraus, dass die Förderung ihrer Entwickler und Tester im Hinblick auf deren Wissensstand sehr von Vorteil ist. Darüber hinaus wurde empfohlen, möglichst auch fachlich qualifizierte Mitarbeiter in Projekte einzubringen.

Zahlreiche Probleme, die in Software auftreten können, lassen sich bereits in den sehr frühen Phasen der Softwareentwicklung vermeiden. Auch wird hier der Grundstein dafür gelegt, dass die entstehende Software den Anforderungen des Kunden entspricht. Neben einer sensiblen Anforderungsanalyse (siehe Kapitel 5.3) ist dafür vor allem Fachwissen und Erfahrung auf Seiten der Entwickler und Tester wichtig. Wenn erstere konzeptionell durchdacht und zukunftsorientiert arbeiten und zweitere aufgrund ihrer Erfahrung und ihres Wissens besonders viele und vor allem besonders schwere bzw. lästige Fehler finden, erhöht dies die Qualität der entstehenden Software wesentlich. Dementsprechend wird der Aufbau von Fachwissen bei den Entwicklern und Tester empfohlen. Von einem der teilnehmenden Unternehmen wurde sogar empfohlen, den Wissenserwerb in die Fachkarriere von Entwicklern einzubetten. Dabei ginge es vor allem darum, dass Entwickler den Überblick über Gesamtprojekte behalten und Konzeptwissen erwerben können. Letzteres sei Voraussetzung für den Aufstieg.

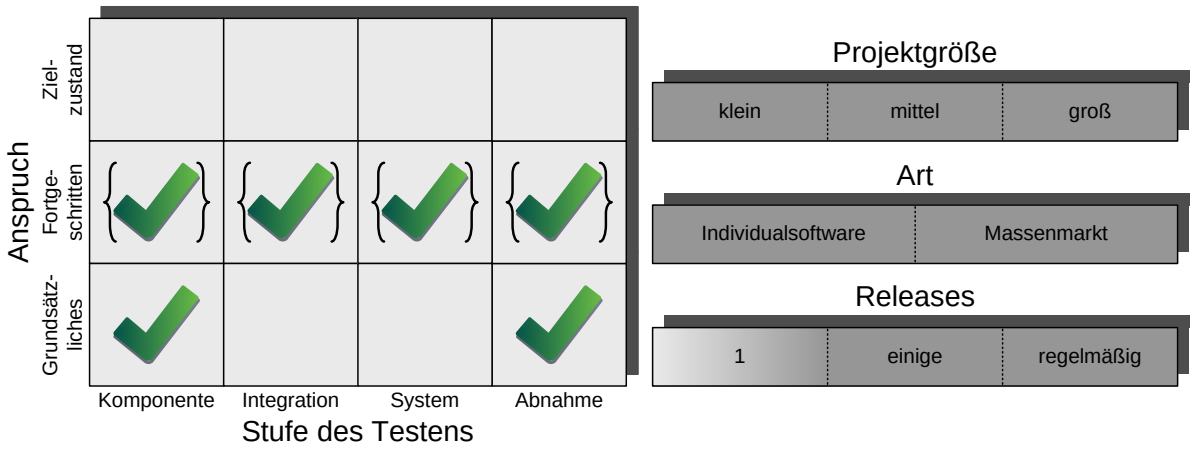

Abbildung 5.12.: Einordnung der Handlungsempfehlung

Fachwissen kann in die Entwicklung und das Testen auch durch explizite Einbeziehung von Fachabteilungen bzw. Kundenbetreuern hineingetragen werden. Insbesondere dann, wenn die Aufgabenstellung sehr fachlich getrieben und auch fachlich kompliziert ist, verhindert dies Fehlentwicklungen. Die Einbeziehung solle daher möglichst schon für die Integration einzelner Komponenten bzw. den Integrationstest erfolgen. Mitarbeiter der Fachabteilungen haben häufig eine skeptische bzw. kritische Haltung gegenüber neuer Software; Techniker sind durch die Gewöhnung an die Technik weniger kritisch. Zudem liegt den Technikern häufig nur eine Beschreibung der Anforderungen an die Software vor. Nicht alle angestrebten Ziele lassen sich allerdings formalisieren und auch nicht alle formalisierbaren Ziele werden formal festgehalten. Demnach können Mitarbeiter der Fachabteilungen bzw. Kundenbetreuer sehr wertvolle Hinweise liefern. Während von ihnen vorgenommene Tests sicherlich keine zusätzlichen Hinweise auf inkorrekte Algorithmen, Schnittstellenfehler und ähnliches liefern, erkennen sie fachliche Verstöße (etwa falsche Berechnungen aufgrund falsch verstandener oder inkorrekt beschriebener Zusammenhänge) und nicht-funktionale Schwächen (schlechte Bedienbarkeit, Verletzung von Design-Schemata) sehr schnell. Die Einbeziehung von nicht-Technikern mit sehr spezifischem Projektwissen ist also eine weitere Empfehlung.

Nicht zuletzt kann es ebenfalls hilfreich sein, Branchenwissen bei den Entwicklern und Technikern zu verankern. Dabei kann es schon ausreichend sein, wenn einige Mitarbeiter über solches Wissen verfügen, dann aber sehr verstärkt dieses Wissen für ihre Arbeit nutzen. Schon bei der Anforderungsanalyse gilt es, technische Schwierigkeiten zu erkennen. Ein Teilnehmer des Projektes formulierte, Kunden hätten mitunter „abstruse“ Ideen. Tatsächlich ist es für Menschen ohne technisches Detailwissen schwierig, die Folgen von Anforderungen abzuschätzen. Beispielsweise können geringe Modifikationen von Anforderungen ausreichen, um von einem Algorithmus mit quadratischer Laufzeit auf eine Alternative mit logarithmischer Laufzeit ausweichen zu können. Je früher derartige Zusammenhänge erkannt werden, desto eher können sie mit dem Kunden geklärt werden. Somit können Entwickler und Tester gerade in den frühen Enwicklungsphasen,

in denen Änderungen der Anforderungen noch vertretbar sind, dazu beitragen, dass ein qualitativ höherwertiges Softwareprodukt entsteht. Streng genommen tragen sie damit sogar dazu bei, dass der Kunde ein in seinem Sinne *besseres* Produkt erhält, selbst wenn einige seiner ursprünglichen Anforderungen modifiziert wurden. Darüber hinaus schärft ein zumindest grundsätzliches Verständnis der Grundlagen einer Branche die Sinne des Entwicklers für die Dokumente, anhand derer er die Software entwickelt, bzw. die Sinne des Testers, der die Software gegen die Dokumente überprüft. Das Risiko von Missverständnissen lässt sich so stark senken.

5.13. Test-Controlling und Kennzahlensysteme

Ein ebenso anspruchsvolles und nur mit Aufwand zu erreichendes wie sehr erstrebenswertes Ziel ist der Aufbau eines Test-Controllings. Idealerweise geht dies einher mit einem Kennzahlensystem, das eine präzise statistische Auswertung von Tests erlaubt. Somit wird es möglich, die Entwicklung des Testens im Zeitablauf zu betrachten. Mögliche Probleme können rechtzeitig erkannt und Maßnahmen ergriffen werden.

Abbildung 5.13.: Einordnung der Handlungsempfehlung

Voraussetzung für den Aufbau des Controlling ist die umfangreiche, vollständige und zuverlässige Dokumentation von Tests. Wie in Kapitel 5.10 beschrieben, sollte die Dokumentation obligatorisch werden; dabei ist eine umfangreiche Pflege des Datenbestandes unbedingtes Ziel.[4] Erste Aufgabe des Test-Controllings ist, die dabei anfallen Daten zu überprüfen. Überprüfungen beginnen bei dem simplen Nachsehen, ob etwa alle zu einem Testfall oder Testlauf gehörigen (Pflicht-) Datenfelder ausgefüllt wurden. Darüber hinaus sollte auch die Sinnhaftigkeit der eingetragenen Werte und deren Konsistenz überprüft werden. So kann ein Testfall, der im Mai erstellt wurde, naturgemäß nicht in einem

[4]Die Erreichung von Qualitätszielen hinsichtlich der Dokumentation kann sehr gut technisch unterstützt werden. Näheres dazu findet sich in Kapitel 6.

Testlauf aus dem März verwendet worden sein. Entsprechende Probleme und Abweichungen sollten mit Entwicklern und Testen besprochen werden, damit die betroffenen Daten nachgepflegt werden. Des weiteren sollte bereits in diesem Schritt auf mögliche Probleme geachtet werden, die sich ergeben können. Geht aus den Daten etwa hervor, dass eine Entwicklungsphase bereits seit einiger Zeit abgeschlossen ist, Testläufe aber erst deutlich später stattgefunden haben und möglicherweise noch nicht abgeschlossen sind, kann dies ein Zeichen von Problemen im Projekt sein. Die wesentliche Leistung des Test-Controllings bzw. seiner Mitarbeiter ist folglich nicht die statische Überprüfung der Daten (die auch z. T. automatisiert oder zumindest werkzeugunterstützt erfolgen kann), sondern das Erkennen sich ergebender Probleme. Controller müssen daher den Kontakt zu den Mitarbeitern suchen, die entsprechende Daten eingepflegt haben bzw. weiterhelfen können.

Eine hochqualitative Dokumentation eröffnet weitere Möglichkeiten. So fallen Fehler und Inkonsistenzen in den bestehenden Testfällen viel eher auf. Auch wird es leichter, offensichtlich nicht abgedeckte Module bzw. fehlender Testfälle zu identifizieren. Demnach kann auch die Qualität des Tests an sich gesteigert werden. Darüber hinaus steigen die Möglichkeiten für Regressionstests bzw. die Wiederverwendung bestehender Testfälle und Lösungsansätze. Das Controlling sollte sich auch zu einem Teil eines möglichen Stufenkonzepts entwickeln: Werden Probleme in den dokumentierten Daten gefunden, sollte eine Freigabe erst dann möglich werden, wenn diese behoben wurden.

In der Praxis fallen bereits bei mittleren Softwareprojekten hunderte, bei größeren Projekten auch schnell tausende von Testfällen für jede Entwicklungsphase an. Wenn hochqualitative Daten für jeden einzelnen Testfall und Testlauf vorliegen, bietet es sich an, diese Daten zu verdichten, um Kennzahlen zu ermitteln. Beispielsweise können die im Durchschnitt gefundenen Fehler pro Testfall bzw. pro Testlauf ermittelt werden. Interessant sind auch Werte wie die gefundenen Fehler pro Testphase. Nimmt die Zahl der gefundenen Fehler hierbei nicht von Phase zu Phase ab, kann dies auf Probleme hindeuten. Wird die schwere der gefundenen Fehler zusätzlich kategorisiert, sind noch aussagekräftigere Auswertungen möglich. Auch kann es sinnvoll sein, Daten zu Testfällen, Fehlern etc. weiter zu kategorisieren, etwa nach Programmbestandteilen oder -funktionen. Betreffen z. B. viele Fehler die Oberfläche der zu entwickelnden Software oder erweisen sich Tests der Schnittstellen als sehr aufwendig, können aus diesen Informationen geeignete Schlüsse gezogen werden. Mithilfe der Verknüpfung von Daten und Mitarbeitern lassen sich solche Entwickler und Tester identifizieren, die für bestimmte Aufgaben ganz besonders qualifiziert sind.[5] Darüber hinaus kann auch ausgewertet werden, wie schnell und wie gut Fehler behoben werden. Treten etwa Fehler trotz ihrer Feststellung und eines Behebungsversuchs in vielen Fällen erneut auf, kann auch dies auf Probleme hindeuten.

Einen besonderen Wert erreicht ein Kennzahlensystem, wenn es im Zeitablauf betrachtet wird. So ist es etwa bei Software, die in regelmäßigen Releases erscheint, sehr interessant, die Daten unterschiedlicher Releases zu vergleichen. Idealerweise sollte da-

[5] Da es sich hierbei um die Auswertung personenbezogener Daten handelt, ist unbedingt zu überprüfen, welche Maßnahmen ergriffen werden müssen, um rechtlichen Anforderungen zu genügen. Eine Erörterung entsprechender Gesetzte und Ähnlichem geht über diese Broschüre hinaus und wird daher nicht thematisiert.

bei – bei etwa gleicher „Größe“ bzw. gleichem Umfang eines Releases – die Zahl der gefundenen Fehler sinken. Ähnliches gilt für die benötigte Dauer, Fehler zu beseitigen. Somit lässt sich die Effizient des Entwickelns und Testens überprüfen. Auch lässt sich mit den entsprechenden Zahlen nachweisen, ob Verbesserunsmaßnahmen anschlagen. Wird eine Maßnahme implementiert, z. B. wie in dieser Broschüre vorgeschlagen, lassen sich die Folgen gut abschätzen. Kommen zusätzlich etwa statistische Signifikanztests zum Einsatz und werden weitere Einflüsse eliminiert, lässt sich die Wirksamkeit einzelner Maßnahmen gar formal feststellen. Dies ist auch sehr hilfreich, wenn der Aufwand für Maßnahmen beträchtlich ist oder ein höherer Mitteleinsatz erforderlich wird. Generell lassen sich umfangreiche statistische Auswertungen vornehmen.

Ein messbarer Erfolg bzw. überhaupt die Möglichkeit, Auswirkungen formell festzustellen, ist für Geschäftsleitungen sehr attraktiv. Ein im Rahmen des Projektes interviewter Mitarbeiter gab an, dass Zeitreihen, bzw. daraus abgeleitete Trendanalysen hervorragend für „Management-Präsentationen“ geeignet seien. Denn im Gegensatz zu Mutmaßungen explizieren sie Vorgänge und Zusammenhänge.

Über die Kontrolle und Bewertung von Veränderungen und Entwicklungen hinaus kann ein sehr feingranular arbeitendes und täglich, idealerweise sogar stündlich oder in Echtzeit verfügbares Kennzahlensystem auch als Indikator bzw. *Frühwarnsystem* gelten. Denn aufbauend auf den Daten der Vergangenheit lassen sich Prognosen für die Zukunft vornehmen bzw. Toleranzbereiche definieren. Somit können schon vor Projekt- oder Phasenbeginn Mitarbeiter und Ressourcen allokiert werden. Kurzzeitigen oder generellen Engpässen, etwa der Mangel an qualifizierten Testern, kann so sehr frühzeitig begegnet werden. Mögliche Probleme entstehen erst gar nicht bzw. können durch ausgeklügelte Planungen beseitigt werden, ohne, dass die betroffenen Mitarbeiter dies überhaupt mitbekommen und in ihrer Arbeit gestört werden. Die Folge sind deutlich reibungslosere und effizientere Prozesse.

Doch nicht nur die Planung kann wesentlich verbessert werden. Auch während laufender Prozesse kann eingegriffen werden, wenn sich Fehlentwicklungen abzeichnen. Auch dies soll Anhand eines Beispiels motiviert werden: Umfangreiche Tests lassen sich nicht ad hoc durchführen. Vielmehr arbeiten eine Reihe von Testern einige Tage, bei umfangreichen Produkten sogar bis zu einige Wochen an den Tests bzw. deren Ausführung. Werden in einem Kennzahlensystem die geplanten und abgeschlossenen Testfälle erfasst, so werden Fehlentwicklungen schnell deutlich. Sind etwa nach 5 von 10 geplanten Testtagen erst 100 von 450 geplanten Testfällen als abgeschlossen eingestellt und keine Hinweise erkennbar, dass die ersten hundert Testfälle besonders aufwendig waren, ist von einer Verzögerung auszugehen. Deren Ursache sollte geklärt werden. Möglicherweise muss mehr Zeit eingeplant oder es müssen weitere Mitarbeiter zum Testen eingeteilt werden. Ähnliches gilt für den Fall, dass in Tests viel mehr Fehler gefunden werden, als zu erwarten wäre. In diesem Fall könnte es zum Einhalten von Fristen und Termin nötig werden, zusätzliche Mitarbeiter mit der Behebung derselben zu betreuen. Mit der Zeit können Controller eine umfassende Erfahrung aufbauen, die ihnen hilft, aufgrund der vorliegenden Daten potentielle Probleme sehr frühzeitig zu antizipieren.

Zusammenfassend kann durch die Einrichtung eines Test-Controlling die Qualität des Testens deutlich erhöht werden. Die Controller sind dabei nicht nur im Hintergrund tätig,

sondern arbeiten mit Entwicklern und Testern eng zusammen bzw. koordinieren das Handeln bei Abweichungen von Soll-Zuständen. Ein Test-Controlling kann dementsprechend hervorragend in ein zentrales Testzentrum (siehe Kapitel 5.6) eingebettet werden. Der zusätzliche Aufbau eines Kennzahlensystems bietet die große Chance, tatsächlich die Kenntnis darüber zu erlangen, wie effizient Entwicklung und Test und ob Verbesserungsmaßnahmen effektiv sind. Der Aufbau stellt sich als schwierige Aufgabe dar, die nur nach und nach bewältigt werden kann und selbst bei fortgeschrittenem Umsetzungsgrad stetige Verbesserungspotentiale bietet. Nichts desto trotz ist ein wichtiges Ergebnis dieser Broschüre, dass der Aufbau eines Kennzahlensystems sehr zu empfehlen ist. Die technischen Details einen solchen Systems werden hier nicht behandelt. Sie richten sich nach den zum Einsatz kommenden Werkzeugen und zahlreichen anderen Rahmenbedingungen. In Kapitel 6 werden allerdings zahlreiche Maßnahmen beschrieben, die den Aufbau sehr begünstigen, wie etwa die Integration der Werkzeuge.

Ein auf dem Markt verfügbaren System, das Auswertungen in den oben beschriebenen Granularität erlaubt, ist den Autoren dieser Broschüre nicht bekannt. Bei der Eigenentwicklung eines Systems ist vor allem darauf zu achten, die Flexibilität zu wahren und eine möglichst große Automatisierung zur Verfügung zu stellen. So ist es etwa sinnvoll, mit einem System zu beginnen, das tägliche Auswertungen liefern kann. Idealerweise könnte es nach und nach auf kürzere Auswertungszyklen umgestellt werden. Die Überprüfung von eingepflegten Daten sollte – wo möglich – nicht von Hand erfolgen müssen. Eine (Halb-)Automatisierung, etwa durch *reguläre Ausdrücke*, ist empfehlenswert. Auch ist es sehr hilfreich, verschiedene Rollen im System zu definieren. Daten sollten in unterschiedlichen Verdichtungsstufen zur Verfügung stehen, da Mitarbeiter in unterschiedlichen Rollen stark verschiedene Detaillierungsstufen benötigen. Idealerweise entsteht nach und nach ein zentraler Anlaufpunkt für alle Entwickler, Tester und Controller. Über das System können dann Aufgaben automatisch priorisiert und allokiert werden. Mitarbeiter würden automatisch über wichtige Vorgänge informiert. Die Möglichkeiten können als sehr weitgehend angesehen werden. Die Vision ist ein integriertes System, das Testfallverwaltung, Entwicklung- und Testplanung, Mitarbeiterplanung, Zeiterfassung, Aufgabenverwaltung, Controlling und *Management-Cockpit* vereint.

5.14. Externe Einschätzungen und Reifegradmessungen

Qualitätssicherung im Allgemeinen ist ein Prozess, der nie zu einem definierten Ende führt, sondern sich gerade dadurch auszeichnet, fortlaufend zu sein. Insbesondere, wenn die Testprozesse bereits in einer Art und Weise gestaltet sind, die zu einer hohen Zufriedenheit mit diesen führen, wird es zunehmend schwierig, weitere Verbesserungspotentiale aufzudecken. Darüber hinaus kann eine komplexe Prozesslandschaft auch zu einer gewissen „Betriebsblindheit“ führen. Zu guter Letzt ist auch noch denkbar, dass von den im Unternehmen für die Softwaretests verantwortlichen Mitarbeitern bestimmte Maßnahmen als Notwendig erachtet werden, sie mit etwaigen Vorschlägen aber kein Gehör beim (Top-)Management finden. Aus diesen Gründen kann es sich lohnen, eine externe Einschätzung der Testprozesse einzuholen.

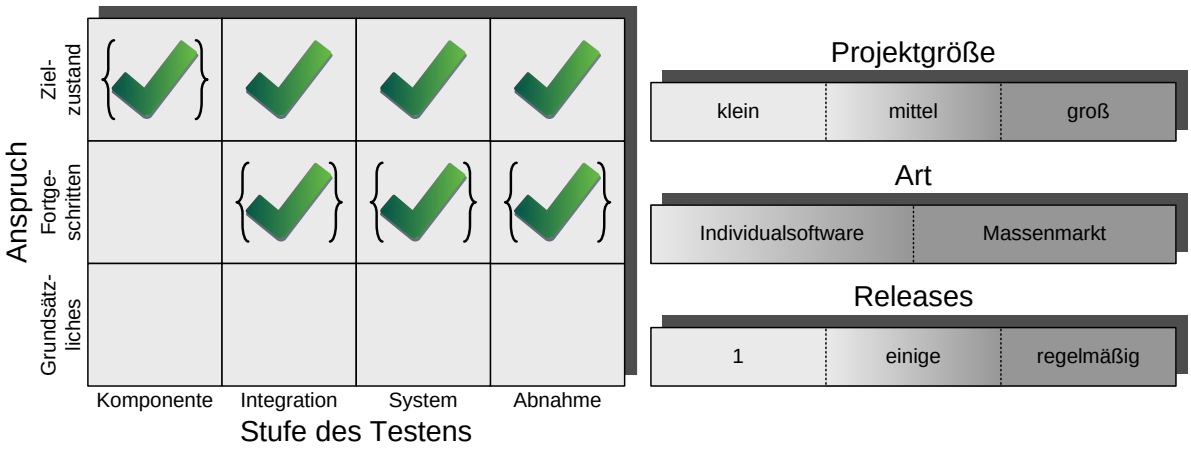

Abbildung 5.14.: Einordnung der Handlungsempfehlung

Sicherlich ist eine externe Meinung auf die Zahl der Entwickler und die organisatorische Einbettung des Testens abzustimmen. Grundsätzlich erscheint es aber in jedem Fall als sinnvolle Ergänzung. Insbesondere der Effekt als „Türöffner“ mit Blick auf das interne Marketing sollte nicht unterschätzt werden. Auch wenn die notwendige Begutachtung der Prozesse grundsätzlich auch durch eigenes Personal möglich wäre, sind externe Experten, also etwa auf das Thema Testen spezialisierte Unternehmensberater, wahrscheinlich kostengünstiger, da sie hochgradig fokussiert arbeiten und von einem großen Erfahrungsschatz profitieren. Auch ist die Bindung von eigenem Personal geringer.

Für größere Entwicklungsabteilungen ist zudem über die Einführung einer Reifegradmessung nachzudenken. Ziel dabei ist es, den Status quo der eigenen Testprozesse in verschiedenen Kategorien zu erfassen und hinsichtlich der Erfüllung der gesteckten bzw. möglichen Ziele zu bewerten. Somit entsteht eine Matrix, aus der sich die Reifegrade einzelner Prozesskomponenten ablesen lassen. Neben dem Nutzen durch die grundsätzliche Evaluation der Testprozesse bietet die Reifegradmessung auch die Möglichkeit, die kontinuierliche Weiterentwicklung kritisch zu beobachten. Somit lassen sich Erfolge identifizieren, aber auch Problemfelder offenlegen. Insbesondere bei identifizierten „Rückschritten“ lassen sich schnell Gegenmaßnahmen ergreifen.

Als Beispiel für ein Reifegradmodell sei das *Test Process Improvement*-Modell (TPI) [24] von Sogeti [25] genannt. Bekannt ist auch das *Test Maturity Model Integrated* (TMMi) [26]. Es ist als Ergänzung zu CMMI (Capability Maturity Model Integration) mit speziellem Fokus auf das Testen gedacht. Die Messung von Reifegraden ist empfehlenswert, sobald ein Unternehmen eine deutlich zweistellige Mitarbeiterzahl in der Entwicklung erreichen oder das Testen zentral steuert, etwa über eine Testorganisation. Eine Vertiefung der Thematik geht über dieser Broschüre hinaus. Eine Reifegradmessung kann definitiv als lohnenswerter Schritt auf dem Weg zur kontinuierlichen Optimierung der Testprozesse festgehalten werden. Ein umfassender Überblick über Prozess- und Reifegradmodelle ist in [SRWL08] dargestellt.

5.15. Nachbildung produktiver Systeme

Bei Entwicklungen für den Massenmarkt ist es üblich, Software auf unterschiedlichen als Zielsysteme bestimmten Konfigurationen von Hardware und Software zu testen. Bei der individuellen Entwicklung ist das Zielsystem in der Regel bekannt und die zu entwickelnde Software bzw. neue Versionen derselben können auf diesem System getestet werden.

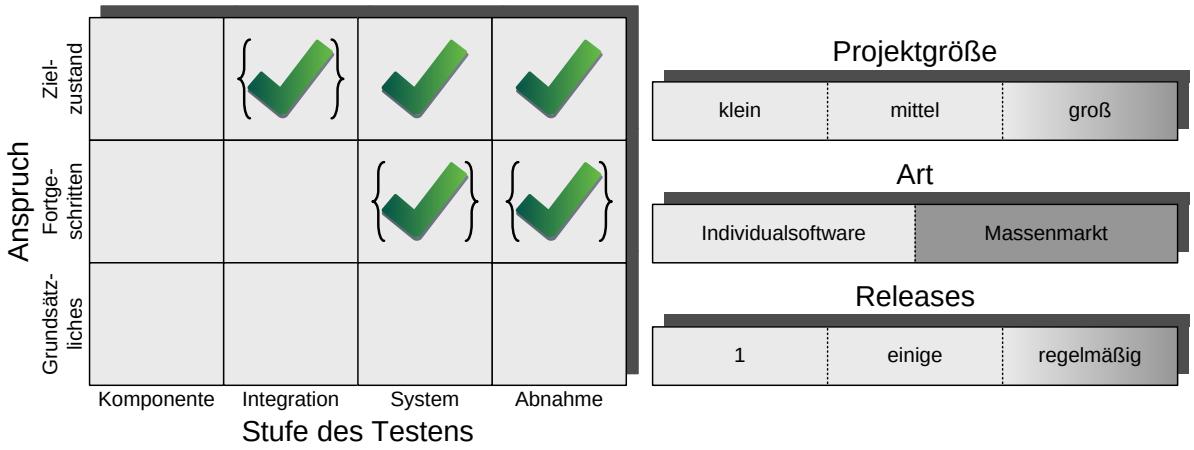

Abbildung 5.15.: Einordnung der Handlungsempfehlung

Normalerweise wird in dem Moment auf Tests auf der Zielumgebung verzichtet, in dem mindestens eine der vier folgenden Voraussetzungen erfüllt sind:

- Das System verfügt über zahlreiche Schnittstellen zu weiteren Systemen oder zu Subsystemen. Der Datenaustausch kann mitunter komplex sein. Um eine Testumgebung aufzusetzen, müssen diese Systeme ebenfalls in der Testumgebung vorhanden sein oder simuliert werden.
- Zahlreiche komplexe Hardware-Komponenten werden für den Betrieb des Systems benötigt, etwa Großrechner oder Cluster-Systeme, bei denen nur ein Test auf dem Cluster, nicht etwa auf einem einzelnen (Blade-)Server Sinn macht.
- Die Systeme müssen mit umfangreichen Datenmengen arbeiten, welche sich schlecht simulieren lassen. Benötigt werden anonymisierte Produktivdaten.
- Um das System sinnvoll zu testen, müssen verschiedenste Teilnehmer (Nutzer) abgebildet und parallel getestet werden, etwa für ein Transaktionssystem.

In den meisten Fällen erklärt bereits eine der Voraussetzungen das Fehlen einer solchen Testumgebung. Nicht zuletzt kommen aber bei Systemen entsprechender Komplexität (etwa Branchenlösungen oder andere Produkte, die Standardprozesse einer bestimmten Zielgruppe in einer großen Breite abdecken) drei oder alle vier Voraussetzungen zusammen, insbesondere wenn die Lösung als Application Service Providing (ASP) angeboten

wird oder zumindest einige Komponenten zentral bereitgestellt werden. Die Erfahrungen aus den Interviews haben gezeigt, dass die Einführung einer Nachbildung der Produktivumgebung für Testzwecke nicht ausgeschlossen werden sollte.

Gerade dann, wenn das System mehrere der genannten Voraussetzungen erfüllt, können klassische Tests sehr aufwendig werden und schlimmstenfalls nicht mehr zur gewünschten Qualität führen. Schließlich werden mit zunehmender Komplexität des Systems eine immer größere Zahl so genannter Teststümpfe (*Stubs*) bzw. Mock-Objekte zum Einsatz. Die Nutzung solcher Komponenten soll nicht in Frage gestellt werden; allerdings kann es für komplexe Systeme überaus aufwendig werden, sie zu entwickeln und aktuell zu halten. Zudem besteht die Gefahr, dass potentiell kritische Zustände des Systems nicht mehr abgebildet werden können, da die simulierten Komponenten nicht das Verhalten des Produktivsystems aufweisen. Fehler treten dann möglicherweise erst im produktiven Betrieb auf, wenn alle Systeme angebunden sind. Gleichzeitig kann auf Integrations- und Systemtests unmöglich verzichtet werden, ohne das Erreichen der Qualitätsziele zu gefährden.

Bei den teilnehmenden Unternehmen herrscht Einigkeit, dass der Test auf Produktivsystemen eine absolute Ausnahme darstellen muss. Dies gilt vor allem dann, wenn kritische Daten verarbeitet werden. Dazu zählen neben allen Arten von monetären bzw. buchhalterisch relevanten Daten auch solche, die datenschutzrechtlich kritisch sind oder engen Bezug zu Kunden haben. Würden durch Tests, die auf einem Produktivsystem ausgeführt werden, etwa Buchungen durchgeführt, in einem buchhalterischen System hinterlegt und nicht rückgängig gemacht, könnte dies ernsthafter Konsequenzen haben. Ähnliches gilt für den Test mit produktiven Daten: Sind diese nicht anonymisiert und die Systeme, auf denen getestet wird, nicht von der Außenwelt abgeschottet, könnten diese im Rahmen der Tests Kontakt mit letzterer aufnehmen. Ein Kunde, der einige (oder schlimmer: 500) Bestellbestätigungen für Produkte erhält, die er nie bestellt hat, dürfte nicht nur verstimmt sein. Vielmehr droht ein Vertrauensverlust, da der Eindruck entstehen kann, Daten seien nicht „in sicheren Händen“. (Zur technischen Tragweite vergleiche auch mit Kapitel 6.8.)

Aus obigen Überlegungen ergibt sich, dass ein Test von hochgradig komplexen Systemen, die ein Zusammenspiel verschiedener Komponenten erfordern, nur auf einer dedizierten Testumgebung möglich ist, die die Produktivumgebung möglichst genau nachbildet. Ein solches System ist von unschätzbarem Wert. Es kann – auch im Zusammenspiel mit den Kunden – besonders umfangreiche und genaue Tests erlauben und somit dazu beitragen, Software bereitzustellen, die höchsten Qualitätsansprüchen genügt.

Die Einführung eines solchen Systems ist sowohl aufgrund der immensen Kosten als auch des organisatorischen Aufwands nicht in einem Schritt realisierbar. Es kann aber problemlos aus einer ständig erweiterten und verfeinerten Umgebung wachsen. Die Entscheidung ist aufgrund des Investitionsvolumens und der Langfristigkeit sicherlich nicht einfach zu treffen. Als Ergebnis dieser Broschüre kann aber empfohlen werden, über den Schritt der Einführung eines solchen Testsystems nachzudenken, wenn entsprechende Voraussetzungen gegeben sind. Ein klares Zeichen dafür, dass über die Einführung nachgedacht werden soll, ist eine sinkende Qualität der Testergebnisse, da die verfügbaren Methoden an ihre Grenzen stoßen. Im Rahmen der Entwicklung umfangreicher Bran-

chenlösungen kann die Einrichtung eines umfangreichen Testsystems ein entscheidender Wettbewerbsvorteil sein.

5.16. Etablierung des Testens als eigene Kompetenz

Die letzte Empfehlung im Rahmen des fünften Kapitels ist *selbst dann* obligatorisch, wenn auch nur ein Teil der vorgestellten Maßnahmen für die Umsetzung im eigenen Unternehmen in Betracht gezogen werden oder sogar im Einsatz sind. Testen sollte für jedes Unternehmen, dass ernsthaft Software entwickelt, eine eigene Kompetenz werden, bei der die Abgrenzung von der Konkurrenz angestrebt wird. Denn Testen ist ein zu integraler Bestandteil der Entwicklung, als dass er das es als Nebentätigkeit gelten kann. Auch ist es zu komplex und zu vielschichtig, als dass es sich für das Outsourcing anbieten würde.

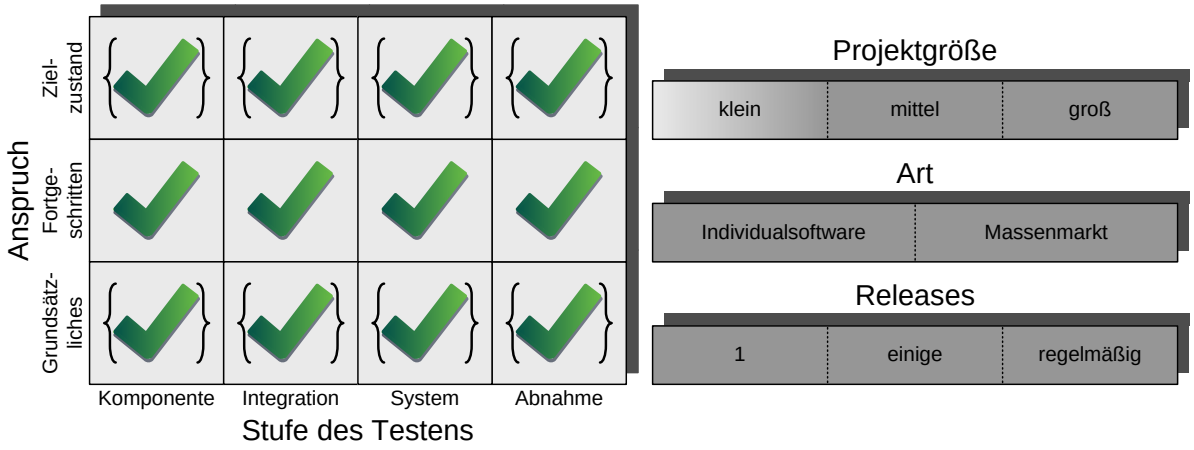

Abbildung 5.16.: Einordnung der Handlungsempfehlung

Einzig für Unternehmen, die Software beschaffen und in ihre eigene Systemlandschaft einfügen, kann es überlegenswert sein, die Evaluation und damit verbundene Tests an Dritte zu vergeben. Aus Sicht der Autoren dieser Broschüre bietet sich dieser Schritt für Software entwickelnde Unternehmen nicht an. Gestützt wird diese Feststellung durch die Erfahrungen, die uns ein Mitarbeiter eines der für dieser Broschüre interviewten Unternehmen schilderte.

Dabei handelte es sich um das Angebot eines Offshoring-Dienstleisters. Er bot die Möglichkeit, Tests unter massivem Personaleinsatz durchzuführen. Die im Verhältnis zum Personalaufwand fällig werdenden Entgelte wären dennoch vergleichsweise niedrig gewesen, da im Wesentlichen Mitarbeiter in Indien beschäftigt worden wären, deren Lohnkosten dramatisch unter dem Deutschen Niveau liegen. Nichts desto trotz kam der Vertrag nicht zustande. Denn für ein Testen, dass über das bloße Sicherstellen technischer Funktionalität hinausgeht, ist ein umfangreiches Fachwissen bei den Testern nötig (vergleiche auch mit den obigen Kapiteln, vor allem 5.2 und 5.12). Die Weitergabe dieses

Wissens ist äußerst schwierig. Übersetzungen und Formalisierungen von implizitem Wissen sind mühsam und aufwendig, die Weitergabe personalintensiv und fehleranfällig. Die nötig werdenden extrem genauen Beschreibungen führten zu dem Schluss, dass bei einem Erstellen derselben die komplette Entwicklung hätte abgegeben werden können. Dies war aber nicht gewünscht. Auch war aufgrund des hohen zu erwartenden Aufwand fraglich, ob dieser nicht schlicht den sonst für die Tests nötigen Aufwand substituieren würde. Schlimmstenfalls wäre der Aufwand sogar gestiegen. Dazu kommt, dass sich Aufwände schlecht abschätzen lassen und der Erfolg nur ungenügend kontrolliert werden kann. Zwar ist die Arbeit mit Kennzahlen nötig; verbleiben mehr Fehler, als allgemein akzeptabel nach dem Testprozess in der Software, stellt sich dennoch die Frage, ob dies an einer ungenauen Beschreibung, ungenügendem Testen oder einer ungenügenden Fehlerbehebung lag. Aufgrund dieser Überlegungen schließen sich die Autoren dieser Broschüre der eher konservativen Sicht an, die ein generelles Outsourcing bzw. Offshoring des Testprozesses ablehnt.

Gerade in einem Bereich, in dem nicht nur harte Fakten eine Rolle spielen, sondern vor allem auch Erfahrungen, erscheint die Abgabe des Testprozesses fragwürdig. Jedes der von uns besuchten Unternehmen betonte mehr oder weniger deutlich, dass gute Tester über die formalisierte Abarbeitung von Testfällen hinausgehen. Begriffe wie „Bauchgefühl“ sollten in einem solchen Kontext nicht gescheut werden. Trotz aller in Kapitel 5 vorgeschlagenen organisatorischen Maßnahmen bleibt Testen ein in Teilen intuitiver Akt, der persönliche Kompetenz erfordert. Dies macht es erklärlich, dass ein Outsourcing des Testens auch bei geklärten Rahmenbedingungen das Gefühl hinterließe, einen wichtigen Teil des Prozesses aus der Hand zu geben. Dieses Gefühl basiert auf der Einsicht, dass Entwicklung und Testen zwei untrennbare Bestandteile der Entwicklung von Software sind; streng genommen wäre es sogar sinnvoll, von Entwicklung i. w. S. zu sprechen, und das Testen mit einzubeziehen, als zweite Komponente neben der Entwicklung i. e. S., dem Programmieren (sowie dem Systementwurf). Wir hoffen, dass dieser Zusammenhang in den zurückliegenden Kapiteln ausreichend motiviert wurde und empfehlen dringend, Testen als Kernkompetenz in die Unternehmenskultur aufzunehmen und als integralen Bestandteil der Entwicklung von Software zu betrachten.

Je nach Aufgabenstellung kann möglicherweise darüber nachgedacht werden, den Aufbau einer eigenen Kompetenz mit den Leistungen externer Tester zu verknüpfen. Auch wenn sich dies im Rahmen des zugrundeliegenden Projekts nicht als empfehlenswerte Strategie dargestellt hat, möchten wir an dieser Stelle auf weitere Literatur verweisen. Wann und wie Testen in Form eines Outsourcings betrieben werden kann und was dabei zu beachten ist, wird etwa in [Jos09] diskutiert.

6. Technische Handlungsempfehlungen

6.1. Einführung

Testen i. w. S. umfasst natürlich zahlreiche organisatorische Aspekte. Es hat sogar strategische Bedeutung. Empfehlungen für eine erfolgreiche Testorganisation wurden im vorangegangenen Abschnitt umfangreich dargelegt. Testen i. e. S. ist allerdings eine rein technische Aktivität. In den folgenden Empfehlungen wird daher herausgestellt, durch welche technischen bzw. technisch getriebenen Maßnahmen die Qualität der entwickelten Software verbessert werden kann. Hierzu zählt insbesondere auch der Werkzeugeinsatz.

Im Gegensatz zu dem vorangegangenen Kapitel spielt die Größe des Unternehmens und auch die Art der Softwareentwicklung für die technischen Aspekte eine untergeordnete Rolle. Vielmehr muss hier das anzustrebende Qualitätsniveau im Vordergrund stehen und daraus abgeleitet der investierbare Aufwand. Natürlich sind auch rein finanzielle Aspekte zu bedenken, gerade im Hinblick auf die Anschaffung von Werkzeugen. Allerdings – das zeigt die Erfahrung durch das Projekt – sind für viele Arten von Werkzeugen auch freie Alternativen verfügbar, so das keine Lizenzkosten mehr anfallen. Natürlich verbleiben gegebenenfalls Beratungs- und Wartungskosten (freie Software ist zwar kostenlos, im Betrieb ist eine externe Unterstützung aber häufig sinnvoll), dennoch dürften sich in allen Bereichen auch passende Lösungen für kleinere Unternehmen bzw. Projekte mit sehr begrenztem Budget finden lassen.

Wie in Kapitel 5 auch sind die Empfehlungen geordnet von solchen, die die Autoren der Broschüre als absolute Notwendigkeit ansehen, hin zu solchen, die als alternative oder zusätzliche Strategien gelten müssen. Wenn möglich werden auch wieder Hinweise gegeben, unter welchen Rahmenbedingungen einzelne Empfehlungen geeignet sind bzw., wie sie sich auf konkrete Konstellationen bezogen umsetzen lassen.

An dieser Stelle sei auch auf Anhang B verwiesen, in dem frei verfügbare Werkzeuge für das Testen zusammengetragen sind. Die Werkzeuge sind in Kategorien eingeteilt und werden jeweils kurz beschrieben.

6.2. Nutzung der Möglichkeiten moderner IDEs

Moderne Integrierte Entwicklungsumgebungen (IDE) bieten reichhaltige Möglichkeiten, zu einer Erhöhung der Qualität der zu erstellenden Software beizutragen. Sowohl im Bereich der Überprüfung des Quelltextes direkt bei der Eingabe, der Hervorhebung potentieller Probleme, der Strukturierung, des Debugging als auch der Einbindung zusätzliche Werkzeuge können sie hervorragende Dienste leisten. Da IDEs aller Regel nach ohnehin

genutzt werden oder kostenlos verfügbar sind, stellt der Gebrauch ihrer Möglichkeiten einen sehr kosteneffektiven Vorteil dar. Zudem sind die im folgenden vorgestellten Maßnahmen leicht umzusetzen.

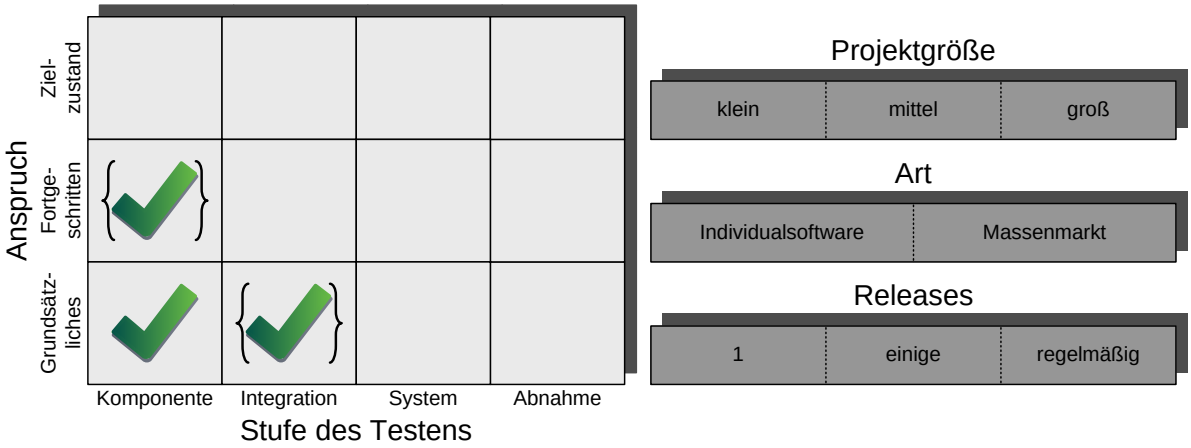

Abbildung 6.1.: Einordnung der Handlungsempfehlung

Falls IDEs zu Einsatz kommen, die einige der Funktionen nicht bieten und sich auch nicht – etwa über Plugins – erweitern lassen, sollte der Umstieg auf eine neuere Version des IDEs oder ein anderes IDE in Erwägung gezogen werden. Eclipse [27], das möglicherweise bekannteste und umfassendste IDE, bietet direkte Unterstützung von Java (speziell auch Java EE) und C bzw. C++. Über Erweiterungen lässt sich die Unterstützung auf zahlreiche weitere Sprachen ausdehnen, etwa PHP [28]. Neben zahlreicher „eingebauter“ Funktionen (siehe unten) lässt sich die IDE über Plugins erweitern. Die Zahl der Plugins liegt mittlerweile in einem vierstelligen Bereich [29][30][31]. Soll die Leistungsfähigkeit der eigenen IDE gemessen werden, ist ein Vergleich mit einer marktführenden IDE sicherlich sinnvoll. Die Erfahrungen in den Interviews haben gezeigt, dass mitunter IDEs zum Einsatz kommen (beziehungsweise sogar ohne „echte“ IDE entwickelt wird), die vom Leistungsumfang von Produkten wie Eclipse oder auch Microsoft Visual Studio, um den kommerziellen Marktführer im Bereich der Entwicklung in .NET zu benennen, weit entfernt sind. Im Vergleich bieten sie noch nicht einmal den Funktionsumfang, den Eclipse oder Visual Studio schon vor mehreren Jahren erreichten.

Die Hervorhebung der Syntax des Quelltextes (*Syntax Highlighting*) und automatische Vorschlagsfunktionen (*Code Completion*). gehören mittlerweile zum Standard, den IDEs bieten. Die Vorschlagsfunktionen können dabei auch direkt die Dokumentation von Schnittstellen anzeigen, so dass Entwickler etwa vor der Verwendung ungeeigneter (als *deprecated* markierter) Aufrufe gewarnt werden. Auf die Dokumentation müssen sie nur in Einzelfällen zurückgreifen, da diese zusammengefasst eingeblendet wird. Somit beschleunigt sich die Entwicklung.

Viele IDEs gehen über obige Funktionen allerdings hinaus und bieten eine unmittelbare Überprüfung des Quelltextes an. Dazu werden die Quelltexte direkt auf mögliche Programmierfehler überprüft. Zum Teil ist auch eine partielle Kompilierung möglich.

Der Programmierer erhält so Meldungen des Compilers, ohne die Kompilierung explizit starten zu müssen. Diese Funktionen sorgen dafür, dass Quelltexte erst gar nicht kompiliert werden, ohne nicht syntaktisch korrekt zu sein.

Die semantische Korrektheit lässt sich zwar nicht automatisiert garantieren, wohl aber viele typische Fehler verhindern. So bieten IDEs es an, bestimmte Warnstufen festzulegen. Hier ist es dringend empfehlenswert, möglichst viele Warnstufen einzuschalten. Am Beispiel von Java und Eclipse würde etwa durch eine gelbe Unterschlängelung hervorgehoben werden, wenn Variablen potentiell den Wert `null` haben könnten, dies aber nicht abgefragt wird. Solcher Code provoziert so genannte `NullPointerException`s. Auch viele weitere, potentiell unsicherer oder in Hinblick auf die Wartbarkeit und Verständlichkeit „unsaubere“ Vorgehensweisen können so hervorgehoben werden. Für Entwickler ist dies zunächst gewöhnungsbedürftig, allerdings lassen sich sehr viele Probleme bereits im Vorfeld verhindern. Typischerweise existiert zudem die Möglichkeit, solche Warnungen, die der Programmierer für überflüssig erachtet, zu unterdrücken. Um beim obigen Beispiel zu bleiben: Weiß ein Programmierer, dass in einer bestimmten Methode eine Variable den Wert `null` niemals annehmen kann, so kann er für die Methode mithilfe der `@SuppressWarnings`-Annotation die Warnung unterdrücken. Dennoch wird bei anderen Methoden auch weiterhin vor derartigen Problemen gewarnt.

Einen Schritt weiter gehen Plugins für IDEs, die das Einhaltung so genannter *Code Rules* (also Programmierregeln) überprüfen. Diese können individuell festgelegt werden. Möglich sind Überprüfungen ähnlich der obigen. Der Unterschied ist im Wesentlichen die Erweiter- und Konfigurierbarkeit. Oben beschriebene Warnungen werden vom Kompilierer erzeugt und durch die IDE dargestellt. Plugins für Code Rules können eine eigene Logik besitzen, welche die Quelltexte überprüft. Auch lassen sich bestimmte Standards erzwingen, etwa einheitliche Benennungsschemata für Variablen. Auch wenn eine derart strikte Handhabung nur bei wenigen der teilnehmenden Unternehmen zu finden war, gilt ein solches Vorgehen als grundsätzlich empfehlenswert. Es ist zwar einerseits wichtig, den Entwicklern nicht das Gefühl zu geben, durch eine – möglicherweise selbst nicht fehlerfreie – Logik „bevormundet“ zu werden. Das Erzwingen bestimmten Mindeststandards ist allerdings dringend empfehlenswert. Wenn alle Entwickler ähnlich aussehenden und sich an ähnlichen Standards orientierten Quelltext erzeugen, so hat dies überdies unschätzbare Vorteile für die Wartbarkeit. Viele Probleme entstehen dadurch, dass mehrere Programmierer an demselben Code arbeiten oder gegenseitig auf erstellte Module zurückgreifen, diese aber nicht im letzten Detail verstehen oder – deutlich schlimmer – anders deuten. Gemeinsame Schemata, Benennungskonventionen etc. können derartige Probleme deutlich verringern.

Hingewiesen sei auch noch auf die Debugging-Fähigkeiten moderner IDEs: Werden Fehler beim Testen festgestellt, ist es mitunter sehr schwierig, die Ursache für Fehler festzumachen. Moderne Debugger können dabei helfen. In vielen Systemen ist es allerdings offenbar noch üblich, ein „print()-Debugging“ zu betrieben: Variablen werden zu bestimmten Punkten im Programmfluss ausgegeben. Dies ist extrem unflexibel und umständlich. Die gekonnte Nutzung eines *Trace-Debuggers*, der idealerweise den kompletten Zustand eines Programms zu – auch während der Programmausführung veränderlichen – Stoppmarken (*Breakpoints*) visualisieren kann, ist sehr hilfreich und

dringend empfehlenswert. Mithilfe ausgeklügelter Plugins ist darüber hinaus auch eine Visualisierung von Kontroll- und Datenflussgraphen möglich.

Zusammenfassend muss die Nutzung moderner IDE dringend empfohlen werden. Gegebenenfalls ist dafür sogar der Umstieg auf eine moderne Programmierarchitektur zu überprüfen. Ferner sollten Standards für die Formatierung von Programmcode und die Benennung von Variablen, Methoden etc. geschaffen werden. Häufig ist dabei eine Orientierung an gängigen Schemata möglich. Diese existieren etwa für .NET-Programmiersprachen [32] oder für Java [33]. Bei der Einführung ist darauf zu achten, die Entwickler nicht an einer flexiblen Arbeitsweise zu hindern, sondern die Regeln vielmehr als die Zusammenarbeit fördernde Standards zu motivieren. Gleichzeitig kann es nötig sein, bestimmte Rahmenbedingungen vorzugeben und damit die Einführung moderner Werkzeuge und Techniken voranzutreiben.

6.3. Einsatz einer Testfallverwaltung

In kleineren Projekten wird Testen häufig zustandslos betrachtet. Sobald ein Modul fertig gestellt ist, werden – nach dem Gutdünken des Entwicklers – einige Tests vorgenommen. Etwaige Fehler werden direkt behoben, oder unstrukturiert notiert und im Anschluss an den Test behoben. Dasselbe Vorgehen erfolgt bei der Integration einzelner Module und schließlich auch beim Systemtest. Eine solche Herangehensweise ist extrem ineffizient und unvereinbar mit den Forderungen des fünften Kapitels nach einer ganzheitlichen Sicht auf den Testprozess und einer strukturierten Vorgehensweise.

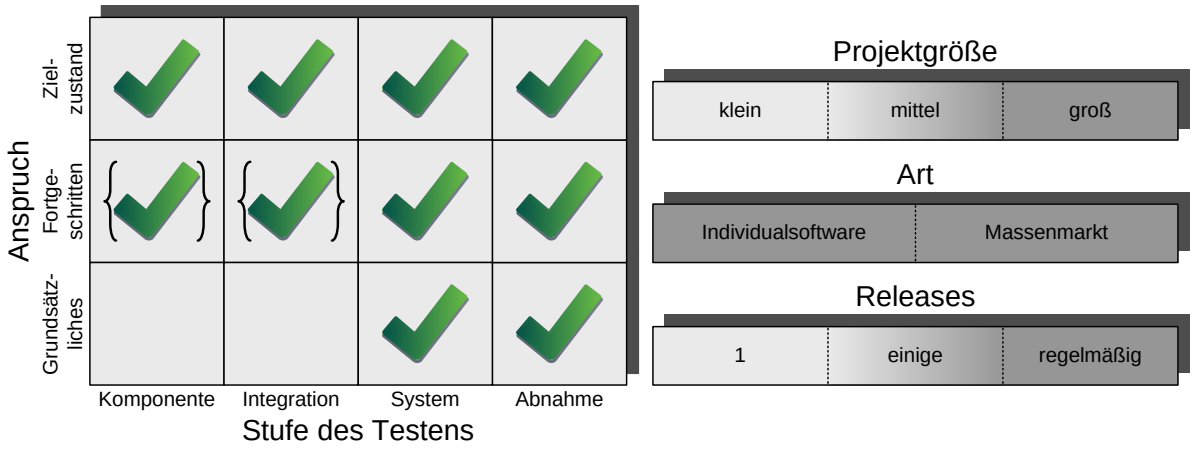

Abbildung 6.2.: Einordnung der Handlungsempfehlung

Als eine Lösung für dieses Problem, die gleichzeitig zahlreiche weitere Vorteile bietet und einen hohen Effizienzgewinn verspricht, bietet sich die Einführung einer Testfallverwaltung an. Grundsätzlich sollte eine Software zur Verwaltung von Testfällen die folgenden Funktionen bieten:

- Erfassung und Kategorisierung von Testfällen. Wichtig dabei ist, dass die auszufüllenden Merkmale eines Testfalls anpassbar sind. Es gilt, einen Kompromiss zwischen unbedingt notwendigen Eigenschaften eines Testfalls und optionalen Informationen zu finden, der eine effiziente Arbeit erlaubt. Idealerweise bietet die Software Vorschlagsfunktionen, um die Eingabe repetitiver Daten zu beschleunigen und die Arbeit für die Benutzer somit angenehm zu gestalten.

- Festlegung des Status von Testfällen, idealerweise mit der Möglichkeit, Aufgaben zuzuweisen und verantwortliche Rollen oder Mitarbeiter festzulegen.

- Möglichkeiten zur verteilten Arbeit. Mit mehreren Arbeitsplätzen sollte zur gleichen Zeit auf das System zugegriffen werden können.

- Mit Benachrichtigungsfunktionen können Mitarbeiter auf noch zu erledigende Arbeiten hingewiesen werden bzw. über Statusänderungen von Testfällen informiert werden, für die sie zuständig sind.

Die wichtigste Aufgabe der Software ist also die Formalisierung und Strukturierung der Verwaltung von Testfällen. Bisher mündlich getroffene Vereinbarungen und ungenau festgelegte Verantwortlichkeiten werden exakt festgelegt. Dies entlastet nicht nur die Mitarbeiter, die genau über Ihre Aufgaben informiert sind und diese gegebenenfalls sogar aufgrund ihrer Fähigkeiten in einem gewissen Rahmen im System selbstständig auswählen können. Vielmehr schafft es auch einen klaren Überblick über den Bearbeitungsstand der Software und die Testabdeckung. Somit lassen sich Lücken beim Testen ebenso nachweisen, wie eine vollständige Abdeckung aller gewünschten Tests. Insbesondere bei größeren Projekten bietet dies unschätzbare Vorteile. Aber auch schon bei kleinen Projekten ist der Effizienzgewinn durch die Formalisierung nicht zu unterschätzen. Der dem gegenüberstehende Erfassungsaufwand kann durch eine intelligente Software sehr in Grenzen gehalten werden, so dass zu erwarten ist, dass sich der Einsatz einer Testfallverwaltung schnell bezahlt macht. Für die Einbindung von Regressionstests ist die Strukturierung von hohem Wert. Während etwa in Excel-Tabellen dokumentierte Testfälle nur mit Aufwand wieder genutzt werden können, ist der erneute Durchlauf eines Satzes von Testfällen direkt aus der Testfallverwaltung im Idealfall ebenfalls möglich.

Optional bieten sich folgende Funktionen an:

- Die Anforderungsverwaltung kann direkt in das System integriert werden. Somit können aus sprachlichen Anforderungen Programmfunktionen abgeleitet und diesen dann Testfälle zugewiesen werden. Dies legt Verantwortlichkeiten klar fest (vgl. Kapitel 5.5) und kann auch helfen, besonders fehleranfällige Module zu finden. Gegebenenfalls ist es sogar möglich, während des Testens auf lückenhafte Anforderungen zu schließen.

- Für Softwareprodukte, die nicht einmalig entwickelt werden, sondern für die eine ständige Wartung und Weiterentwicklung erfolgt, ist die Einbindung eines so genannten *Bug Trackers* sinnvoll, also einer Software für die Meldung, Erfassung und Verwaltung von (vermuteten) Fehlern in der entwickelten Software. Erfasste

Fehler können wiederum mit Testfällen verknüpft werden. Auch bietet sich an, die Beseitigung der Fehler über die Software einem Entwickler zuzuweisen und direkt auch den zuständigen Tester auszuwählen. Werden Fehler zusätzlich priorisiert, ist es unmöglich, wichtige Fehler zu übersehen.

Das Ziel bei der Einführung einer Software für die Testfallverwaltung sollte also sein, eine Schnittstelle zwischen einzelnen Prozessschritten zu schaffen. Bisher informelle und damit ungenaue, wenig nachprüfbare und kaum wiederholbare Prozesskomponenten werden durch die Software abgebildet und Informationen strukturiert bereitgestellt. Bei allen operativen Testvorgängen steht die Software am Anfang und Ende des Vorgangs. Zudem bildet sie eine gemeinsame Sprachbasis für alle am Testen, streng genommen sogar alle an der Softwareentwicklung beteiligten Mitarbeiter.

In vielen Unternehmen findet zwar eine Verwaltung von Testfällen statt, diese ist aber nur semi-strukturiert, der Testverwaltungsprozess umfasst Medienbrüche oder zumindest unstrukturierte Übergänge zwischen unterschiedlichen Softwaresystemen (vgl. Kapitel 3). Aufgrund der Vorteile, die eine dedizierte Software bietet, ist der Übergang zu einer solchen dringend zu empfehlen. Ebenfalls ist dazu zu raten, diese Software möglichst stark mit weiterer zum Einsatz kommender Software für Anforderungsmanagement und Testen zu vernetzen, so dass ein automatisierter Datenaustausch möglich ist. Somit können Prozesse nicht nur viel effizienter ablaufen, auch die Informationsfülle (und gleichzeitig die Selektionsfähigkeit) steigt. Falls benötigt, können zusätzliche Informationen nach wie vor in Dokumentenform verwaltet werden. Idealerweise sollten diese dann allerdings auch über die Software zur Testfallverwaltung erreichbar sein. Ein Beispiel hierzu wären Word- oder Excel-Dokumente mit weiterführenden Informationen, die als „Anhänge“ zu Testfällen in der Software hinterlegt werden. Als Idealzustand sollte ein ganzheitliches System angestrebt werden, das etwa auch statistische Auswertungen erlaubt. Ein solches System wurde bereits im Rahmen der organisatorischen Handlungsempfehlungen ausführlich motiviert (vgl. Kapitel 5.13).

Die Einführung der Testfallverwaltung kann schrittweise erfolgen. Dies ist im Rahmen eines Projektes möglich, bei dem eine gewisse Vorlaufzeit, in der die Projektbeteiligten die Software kennen lernen, vertretbar ist. Alternativ können zunächst einzelne Testphasen in der Software verwaltet werden. Sowohl ein *bottom-up*- (beginnend mit Komponententests) als auch ein *top-down*-Ansatz (beginnend mit dem System- oder sogar beim Abnahmetests) sind denkbar. Bei einer schrittweisen Einführung sollten die Risiken auch auch die potentiellen Verzögerungen aktueller Projekte sehr klein gehalten werden können. Ein *Return-on-Investment* (ROI) sollte sich innerhalb kürzester Zeit einstellen. Die genaue Einführungsstategie muss im Vorfeld festgelegt werden. Sie kann im Rahmen dieser Broschüre nicht weitergehend diskutiert werden, da sie stark von den Gegebenheiten, der Ausgestaltung des Testprozesses, der aktuellen Verfügbarkeit von Mitarbeitern etc. abhängig ist. Die Frage, ob eine Testfallverwaltung eingeführt werden sollte, wenn sie noch nicht im Einsatz ist, sollte aber ebenso mit „Ja!“ beantwortet werden, wie die Frage, ob bisher unstrukturiert verwaltete Informationen in eine den gesamten Testprozess umfassende Testfallverwaltung eingebunden werden sollten.

6.4. Aufbau einer Testfalldatenbank

Auf den ersten Blick scheinen ein Systeme zur Verwaltung von Testfällen und eine Testfalldatenbank redundant zu sein. Tatsächlich ist es möglich, beide Funktionen in einem System zu vereinigen. Funktional unterscheiden sich die beiden dennoch. Die Testfallverwaltung dient vor allem der Strukturierung und der Dokumentation. Die Testfalldatenbank hingegen ist stärker technisch geprägt. Sie sammelt Testfälle in ihrer ausführbaren Form und stellt weitere für das Testen benötigte Komponenten wie Teststümpfe oder Testdatenbanken bereit. Das Ziel der Nutzung einer Testfalldatenbank ist die Erhöhung des Anteils wiederverwendeter Tests. Daher bildet sie eine wichtige Voraussetzung für umfangreiche Regressionstests.

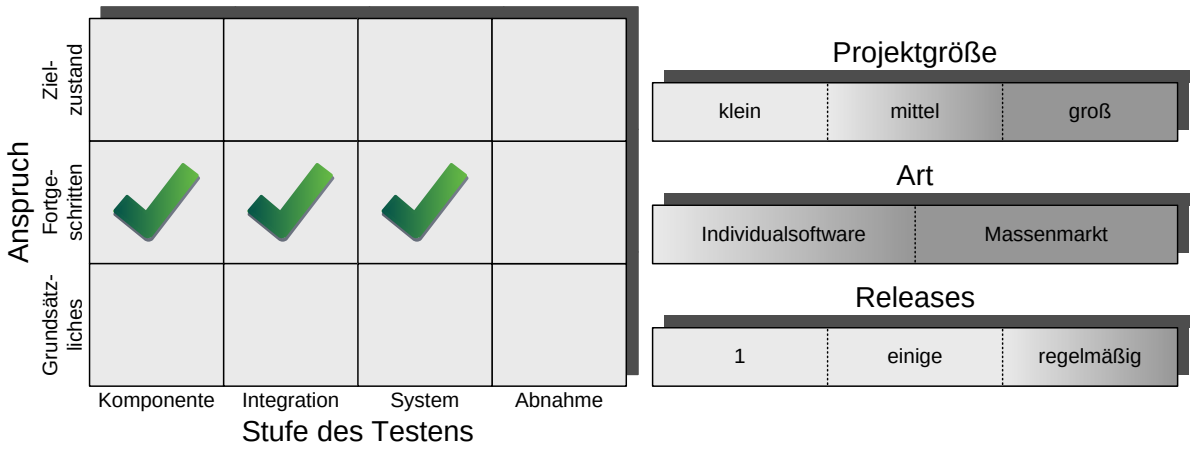

Abbildung 6.3.: Einordnung der Handlungsempfehlung

Testfalldatenbanken sind häufig in Werkzeuge integriert. Sie können aber auch selbst erstellt sein oder gehören idealerweise zu umfangreichen Testfallverwaltungssystemen. Testfälle sollten strukturiert abzulegen sein und einfach wieder aufgerufen werden können. Idealerweise hat das zugrunde liegende System Zugriff auf die zu testende Software und kann somit Testläufe direkt anstoßen. Dabei ist es hilfreich, wenn es Teststümpfe automatisch einbinden kann. Als nützlich hat sich auch erwiesen, wenn das System konsistente Testdatenbanken zur Verfügung stellen kann. Werden verschiedene Testläufe auf derselben Datenbank durchgeführt, besteht die Gefahr, dass Ergebnis nur schwierig zu reproduzieren sind oder im Nachhinein unerklärlich erscheinen. Daher sollte die Datenbank vor jedem Testlauf in einem definierten und konsistenten Zustand sein.

Die Testfalldatenbank bietet über die reine Wiederverwendung von Testfällen hinaus noch weitere Vorteile. Eine gängige Strategie zum Testen ist etwa der Vergleich einer alten, als funktionierend bekannten Version einer Software, und einer neuen Version derselben. Dabei kann ein definierter Testfallsatz zum Einsatz kommen. Liefert dieser bei den Läufen auf der alten und der neuen Version unterschiedliche Ergebnisse, sollte nach der Ursache dafür geforscht werden. Analog gilt dies für Testdatenbanken: Hier wird zunächst die alte Version mit einer definierten Datenbank getestet. Dann folgt die

neue Version und arbeitet auf einer Datenbank, die wieder auf den Ausgangszustand zurückgesetzt wurde. Anschließend werden die beiden Datenbanken verglichen. Sind sie nicht identisch, deutet dies auf Probleme hin. Sind sie identisch, aber die alte Version der Software enthielt Fehler, die sich im Datenbestand abzeichneten, sind diese Fehler offenbar in der neuen Version nach wie vor enthalten.

Testfalldatenbanken sind auch dann nützlich, wenn einzelne Programmteile als gemeinsame Bibliotheken genutzt werden sollen. Nach jeder Veränderung können sie dann mit kleinem Aufwand getestet werden. Unzulässige Änderungen, etwa an den definierten Schnittstellen, oder neu hinzugekommene Fehler fallen somit auf, bevor die neue Bibliothek auf produktive Systeme aufgespielt wird. Im Allgemeinen zeigt sich die Stärke einer Testfalldatenbank immer dann, wenn Regressionstests sinnvoll sind oder ähnliche Programmfunktionen getestet werden. Regressionstests sind vor allem für wiederkehrende Releases bestehender Softwareprodukte sinnvoll bzw. als stetig wiederholende Tests im Rahmen der Entwicklung umfangreicher Applikationen. Die Vorteile beim Test ähnlicher Programmfunktionen lassen sich am besten Anhand eines sehr einfachen Beispiels erklären: Für ein Softwaresystem X besteht eine Bibliothek, die Datensätze in der Form N sortiert. Der Algorithmus hat eine Laufzeit, die im schlechtesten Fall quadratisch ($\mathcal{O}(n^2)$) ist, in der Regel aber $\mathcal{O}(n \cdot \log(n))$ entspricht (wie etwa Quicksort). Für das System Y soll ein neuer Algorithmus implementiert werden, da aufgrund der erwarteten Daten und derer Sortierung erwartet wird, dass der *worst case* häufig eintritt. Zum Testen des Sortierverfahren ist es nun möglich, dieselben Testfälle wieder zu verwenden. Die Komplexität muss statistisch oder – besser – formal ermittelt werden[1]. Zum Testen allerdings sind die für den ersten Algorithmus erstellten Testfälle vermutlich völlig ausreichend — der Aufwand für das Testen sinkt somit drastisch. Ohne eine Testfalldatenbank wäre es wahrscheinlich viel zu aufwendig, die Testfälle zunächst zu suchen und dann für den neuen Algorithmus anzupassen.

6.5. Modularisierte und dienstorientierte Entwicklung

Bereits im fünften Kapitel wurde als Mittel zur Verringerung der Komplexität eine Modularisierung vorgeschlagen. Auch in technischer Hinsicht ist diese sehr empfehlenswert. Dabei sollte auf eine klare Schnittstellendefinition geachtet werden. Einzelne Module werden so zu diensterbringenden Einheiten. Dies verringert nicht nur die Komplexität bei der Entwicklung. Vielmehr wird auch das Testen einfacher.

Die genaue Strategie und die Möglichkeiten zur Modularisierung richten sich stark nach der verwendeten Programmiersprache bzw. dem Programmierparadigma. Zudem gibt es häufig unterschiedliche Stufen der Modularisierung. In objektorientierten Sprachen etwa bildet die *Klasse* die kleinste Einheit, die als Modul zu bezeichnen ist. Als nächste Stufe können z. B. Namensräume oder Pakete (*Packages*) betrachtet werden. Wiederum darüber angesiedelt sind etwa Bibliotheken. Module vereinigen, unabhängig von der Stufe, zusammengehörige Funktionalitäten.

[1] Heapsort und Mergesort bieten z. B. auch im worst case eine Laufzeit von $\mathcal{O}(n \cdot \log(n))$. In der Praxis könnte Quicksort dennoch vorzuziehen sein.

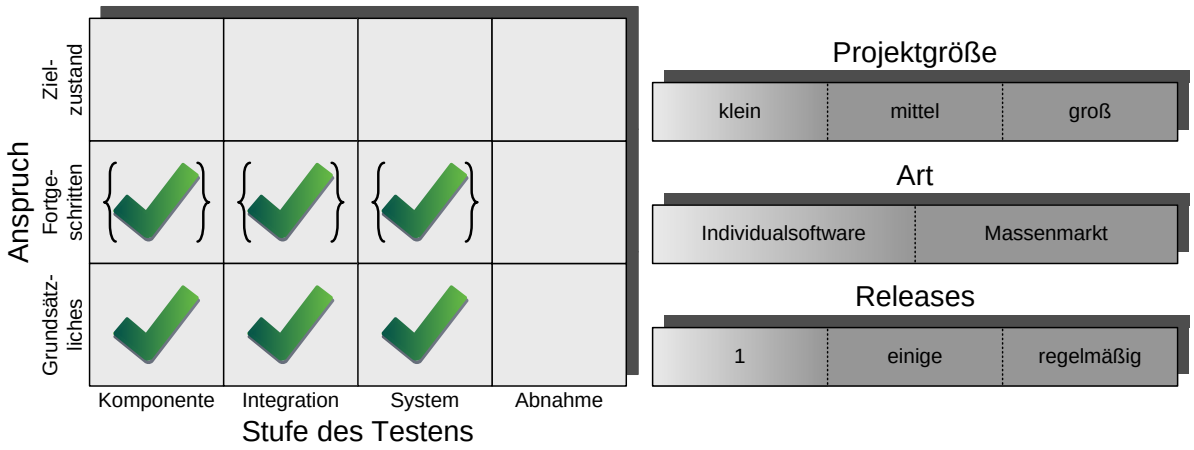

Abbildung 6.4.: Einordnung der Handlungsempfehlung

Was dabei als zusammengehörig gilt, ändert sich mit der Stufe, da höhere Stufen ein höheres Abstraktionsniveau bieten. So könne etwa die Darstellung eines Datums als eine Klasse `Datum` in Java modularisiert werden. Sie würde zusammen mit ähnlichen Klassen in einem Paket zur Darstellung und Berechnung von Zeiträumen abgelegt. Dieses wiederum könnte zu einer Bibliothek gehören, die die Erfassung von Arbeitszeiten abbildet. Diese Vorgehensweise biete unschätzbare Vorteile für die Wiederverwendung: Überall, wo eine Arbeitszeiterfassung benötigt wird, muss nur die entsprechende Bibliothek eingebunden werden. Aber auch das Paket oder die einzelne Klasse könnte an geeigneter Stelle wiederverwendet werden. Klar ist dabei, dass nötige Änderungen wahrscheinlicher werden, je Tiefer die Abstraktionsstufe ist.

Die Wiederverwendung hat erhebliche Vorteile für das Testen. Bereits eingesetzte Module besitzen häufig ein hohes Qualitätsniveau. Darüber hinaus sind Testfälle bereits vorhanden, so dass selbst bei einer Anpassung der Bibliothek nur eine Anpassung der Testfälle, nicht aber das Erstellen derselben von Grund auf nötig wird. Die durch Modularisierung erreichte Abstrahierung hilft zudem, den Überblick zu behalten und den Blick auf das Wesentliche zu Beschränken. Wird etwa die Integration einer Bibliothek in ein Gesamtsystem getestet, so sollten Implementierungsdetails keine Rolle mehr spielen. Testfälle werden vielmehr in Bezug auf die dokumentierten und klar spezifizierten Schnittstellen geschrieben.

Der Begriff der *Serviceorientierten Architektur* (SOA) hat in den letzten Jahren eine hohe Bedeutung erlangt. Ganz in diesem Sinne ist es empfehlenswert, alle Module dienstorientiert zu gestalten. Dies bedeutet, dass jedes Modul – unabhängig von seinem Abstraktionsniveau – spezifizierte und dokumentierte Schnittstellen bietet, an denen es bestimmte Dienste anbietet. Die oben angesprochene Klasse `Datum` etwa könnte die Möglichkeit bieten, ein Datum auf verschiedene Arten und Wiesen anzulegen, z. B. durch Übergabe von Tag, Monat und Jahr oder durch die Anweisung, den Tag zum Aufrufzeitpunkt darzustellen. Als Dienste könnte sie die Rückgabe des Datums in verschiedensten Formaten anbieten, etwa auch als Unix-Zeitstempel (*Timestamp*).

Wie diese Dienste intern dargestellt sind, interessiert den Aufrufer nicht. Er benötigt schlicht die Information darüber, welcher Dienste es gibt, und welche Rückgaben sie liefern. Er hat daher auch keine Möglichkeiten, Daten direkt zu verändern. Soll ein Datum veränderlich sein, müsste er vielmehr ein Dienst aufrufen, der dies erledigt. So könnte Datum einen Dienst anbieten, der eine Anzahl an Tagen entgegen nimmt und ein neues Datum zurück liefert, welches das alte Datum verschoben um die Zahl der Tage repräsentiert. Diese Kapselung findet natürlich auch auf höheren Abstraktionsniveaus statt. Die Nutzer der Bibliothek zur Arbeitszeiterfassung müssen nichts über die Dienste wissen, die die Klasse Datum anbietet. Vielmehr rufen sie Dienste auf, um bestimmte Arbeitszeiten zu übermitteln. Schließlich rufen sie z. B. einen Dienst auf, der als Rückgabe die Information liefert, ob ein bestimmter Mitarbeiter mit seinen geleisteten Arbeitsstunden im Soll ist oder ob es Abweichungen gibt.

Die Dienstorientierung von Modulen reduziert die Komplexität drastisch und erleichtert somit nicht nur die Entwicklung wesentlich. Vielmehr profitiert vor allem auch das Testen. Getestet wird schon mit dem Integrationstest aller Regel nach gegen die Schnittstellen. Tests sind dann reine Black Box-Tests. Im Rahmen des Komponententests, bei dem – um bei obigen Beispielen für objektorientierte Sprachen zu bleiben – nur kleinste Module wie Klassen oder Pakete direkt getestet werden, ist es unvermeidlich, dass der Entwickler „seinen“ Code kennt und mit diesem im Hinterkopf testet. Das Tests hier Glass Box- oder Gray Box-Tests sind ist auch sinnvoll, etwa um algorithmische Fehler aufzuspüren. In allen späteren Phasen sind Black Box-Tests gegen die Schnittstellen völlig ausreichend. Ein Blick auf den Quelltext erfolgt erst dann wieder, wenn Fehler oder unerwartete Zustände festgestellt werden. Natürlich gehört es auch dazu, Schnittstellenverletzungen bewusst zu provozieren, um die Robustheit von Modulen zu testen. Nichts desto trotz sinkt durch die Abstrahierung die Komplexität sehr wesentlich. Insbesondere bleibt sie auch in den späten Testphasen, in denen sehr viele Module im Zusammenspiel getestet werden, beherrschbar.

Es ist unmittelbar verständlich, dass im Rahmen des Systemtests einen großen Softwaresystems nicht tausende oder zehntausende einzelner Klassen getestet werden können. Fehler in grundsätzlich Algorithmen sollten zudem in den vorherigen Phasen gefunden worden sein. Treten sie erst durch das Zusammenspiel der Module auf, so ist die Abstraktion kein Hindernis: Vielmehr wird der Fehler offenbar erst gefunden, weil auf einem hohen Abstraktionsniveau getestet wird und die komplexen Zustände das System überfordern.

Zusammenfassend lässt sich sagen, dass dienstorientierten Softwarekomponenten die Zukunft gehört. Die Testbarkeit wird durch den vermehrten Einsatz von Diensten erhöht. Module sollten robust und in sich abgeschlossen gestaltet sein; wichtig ist die klare Spezifikation und Dokumentation ihrer Schnittstellen. Auch für die Automatisierung bietet die Modularisierung Vorteile, da durch die klaren Schnittstellen viel strukturierter und systematischer getestet werden kann. Strukturierte und systematische Tests bieten sich aufgrund der Formalisierbarkeit häufig für automatisierte Tests an. Die Empfehlung für Modularisierung und Dienstorientierung fällt nach den Beobachtungen im Rahmen des zugrunde liegenden Projekts so stark aus, dass selbst die Ablösung bisheriger Programmiersprachen und Paradigmen dringend empfohlen werden muss, wenn diese nicht die

Entwicklung von dienstorientierten Modulen unterstützen.

6.6. Nutzung moderner Paradigmen und Frameworks

Auf der einen Seite erhöht sich die der Softwareentwicklung innewohnende Komplexität ständig, da Systeme immer umfangreicher und die Möglichkeiten erweitert werden. Auf der anderen Seite bieten moderne Programmierparadigmen und Frameworks die Möglichkeit, diese Komplexität besser zu beherrschen und durch höhere Abstraktionsniveaus mehr Übersicht zu schaffen.

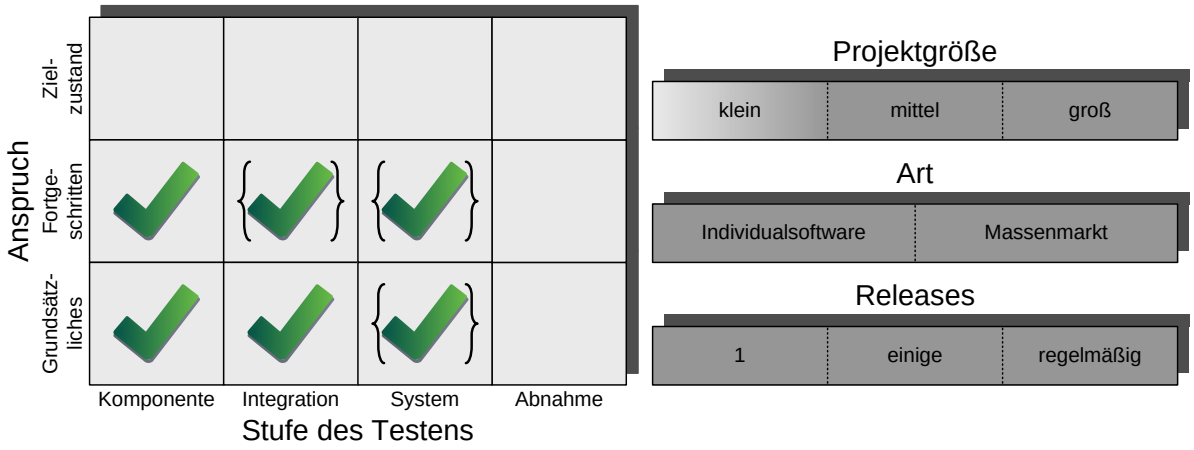

Abbildung 6.5.: Einordnung der Handlungsempfehlung

Unabhängig von einer möglichen Modularisierung ist die Verzahnung unterschiedlicher Schichten einer Applikation bei vielen im Einsatz befindlichen Softwaresystemen sehr hoch. Dies ist im Sinne der Entwicklung moderner dreischichtiger Client-Server- bzw. vierschichtiger Webanwendungen wenig sinnvoll. Vielmehr sollten die einzelnen Schichten einer Applikation klar getrennt werden und nur über Schnittstellen kommunizieren. Dabei ist ferner darauf zu achten, dass nur solche Funktionen in die jeweiligen Schichten fließen, die diesen auch logisch zuzuordnen sind. Dies erleichtert nicht nur die Entwicklung, sondern ist für das Testen sehr sinnvoll. Testfälle werden ab dem Integrationstests für Schnittstellen geschrieben. Sind diese nicht sauber definiert und trennen diese nicht eindeutig die Schichten, ist es sehr schwierig, entsprechende Testfälle zu erstellen. Insbesondere ist es für Tester kaum mehr abzuschätzen, ob tatsächlich alle zu einer Schnittstelle gehörenden Funktionen getestet werden. Besonders gefährlich sind schnittstellenlose Durchgriffe auf Daten: Auf diese Weise kann eine Schicht einer Applikation Zustände einer anderen Schicht modifizieren. Ein solches Verhalten stellt sich für Tester als undeterministisch dar und ist kaum sinnvoll zu testen. Auch erschwert eine unklare Aufteilung in Schichten die Rollenverteilung unter den Testern: Vor allem dann, wenn auf einem realistisch niedrigen Abstraktionsniveau getestet wird, ist es sinnvoll, die Tester gemäß ihren Fähigkeiten einzuteilen, da noch nicht das Verhalten des Systems

als Ganzes untersucht wird, sondern einzelne Funktionen. Finden sich nun etwa gemischt mit der Darstellungslogik Berechnungsfunktionen, wird dies einem auf Oberflächentests spezialisiertem Tester möglicherweise Schwierigkeiten bereiten; gleichzeitig präsentiert sich die Programmlogik als unvollständig, wenn sie gewissermaßen nur Daten aus der Datenhaltung durchreicht, damit diese von der Darstellungslogik aufbereitet werden.

Für eine stärker automatisierte Überprüfung dienstorientierter Systeme (vgl. auch mit dem vorherigen Kapitel) ist zudem eine strikte Trennung der Darstellungslogik von der restlichen Anwendung nötig. Dienste sollten stets so spezifiziert werden, das sie von beliebigen Aufrufern genutzt werden können. Dementsprechend müssen sie etwa so definiert sein, dass der Aufrufer ein weiteres System sein könnte, ebenso wie die Darstellungslogik einer Web-Anwendung oder eines klassischen Client-Programms. Nur wenn Dienste dementsprechend definiert und nutzbar sind, lassen sie sich effektiv Testen. Von Teilnehmern des zugrunde liegenden Projekts erhielten wir mehrfach den Hinweis, dass derartige Systeme angestrebt würden.

Praktisch alle im Einsatz befindlichen Systeme waren allerdings wenig dienstorientiert oder Verstießen gegen einer klare Trennung der Schichten. Dies hing oft mit der Nutzung älterer Progammiersprachen zusammen, in denen die oben beschrieben Konzepte ursprünglich nicht vorgesehen waren. Zudem wurde auch mit modernen Programmiersprachen häufig wenig zukunftsorientiert gearbeitet. Aus diesem Grund ist es sehr empfehlenswert, moderne Programmiersprachen und moderne Programmierparadigmen zu nutzen. Dort, wo auf alte Programmiersprachen zurückzugreifen ist, sollte dennoch versucht werden, möglichst „kompatibel“ zu programmieren. So ist es auch für Systeme, die etwa in COBOL oder Fortran entwickelt wurden, möglich, Dienste anzubieten, über die mit diesen Systemen kommuniziert werden kann. Gegebenenfalls können auch so genannte Wrapper zum Einsatz kommen, die den Zugriff auf Alt-Systeme maskieren und wohldefinierte Schnittstellen anbieten. Auch wenn dies zusätzlichen Programmieraufwand bedeutet, verbessert es die Testbarkeit wesentlich.

Für moderne Programmiersprachen existieren häufig Leitfäden, die die effektive und effizientere Programmierung motivieren. Auch wenn dies zunächst kaum möglich scheint, zeigt die Praxis, dass viele Programmierer an altbekannten Techniken festhalten. Dies gilt selbst dann, wenn sie prinzipiell vom Vorhandensein besserer Herangehensweisen wissen. Insbesondere ist es wenig verbreitet, Programme hinsichtlich ihrer Testbarkeit zu entwickeln. Für vermeidliche minimale Leistungsgewinne werden undurchsichtige Techniken gewählt und für minimale Zeitgewinne auf eine umfangreiche Dokumentation der Quelltexte verzichtet. Beides ist dringend zu vermeiden. Eine undurchsichtige Programmierung ist auch deshalb nicht gerechtfertigt, da moderne Compiler hochgradig komplexe Optimierungen vornehmen. Die explizite Verwendung von Shift-Operationen etwa muss als nicht mehr zeitgemäß gelten – selbst, wenn sie als „hohe Kunst“ angesehen wird. Compiler erkennen Code, der sich etwa durch ein Shiften anstatt einer Multiplikation beschleunigen lässt und erzeugen die entsprechenden Maschinenbefehle automatisch.

Des Weiteren ist dringend zu empfehlen, Literatur zu modernen Programmierparadigmen für die verwendeten Programmiersprachen zu konsultieren. Dies sollte mit Standardwerken wie dem Buch zu *Entwurfsmustern* von Gamma et al. [GHJV95] beginnen, und bei fortgeschrittenen Werken enden. Als Beispiel sei das Werk *Effective Java* [Blo08]

für die Programmiersprache Java genannt. Es enthält nicht nur Hinweise, die auch für fortgeschrittenste Programmierer dienlich sein können, sondern versucht eine zukunftsorientierte Programmierweise zu motivieren und deren Vorzüge aufzuzeigen. Im Sinne der im fünften Kapitel angeregten Schulungsmaßnahmen für Entwickler und Tester sollte demnach auch Platz für die Verbreitung moderner Paradigmen gefunden werden, da sie häufig dem Testen sehr zuträglich sind und darüber Hinaus auch zahlreiche Vorteile für die Entwicklung bieten.

Auch neuere Rahmenwerke (*Frameworks*) können die Testbarkeit von Software erhöhen. Unter Frameworks seien Programmiergerüste verstanden, mit deren Hilfe Anwendungen entwickelt werden. Sie stellen in er Regel bestimmte allgemein benötige Komponenten bereit, abstrahieren bestimmten Techniken, bieten entsprechende Programmierschnittstellen und begünstigen zudem eine modularisierte Entwicklung. Die Vorteile der Modularisierung wurden eingehend erläutert. Die Verwendung standardisierter Komponenten sorgt dafür, dass es *verlässliche Bereiche* in der zu entwickelnden Software gibt. Denn die vom Framework bereitgestellten Funktionen sind häufig in einem sehr breiten Einsatz. Dementsprechend sind sie vielfach getestet worden und idealerweise auf einem sehr hohen Qualitätsniveau. Etwaig verbleibende Fehler sind ebenso wie Workarrounds[2] dokumentiert. Die Abstraktion von bestimmten Funktionalitäten sorgt dafür, dass übliche Programmierfehler vermieden werden und das Testen vereinfacht wird, da auf die Zusicherungen durch das Framework vertraut werden kann. Laut Aussagen der Projektteilnehmer können Frameworks zur Entwicklung verteilter mehrschichtiger Applikationen wie *Enterprise Java Beans* (EJB) oder *Spring* das Testen wesentlich vereinfachen. Gleichzeitig gaben sie zu bedenken, dass das zugrunde liegende Paradigma für Entwickler, die bisher auf andere Arten und Weisen programmiert haben, gewöhnungsbedürftig sei. Techniken wie *Dependency Injection*, also die Konfiguration von Programmmodulen von „außen“, können ebenso wesentliche Vereinfachungen ermöglichen.

Festzuhalten ist, dass die Chancen, die moderne Paradigmen und Frameworks bieten, beachtlich sind. Die Einführung setzt entsprechende Schulungen voraus, ebenso wie die Evaluation geeigneter Techniken und Frameworks. Dazu ist ein hohes Maß an Disziplin gefordert; eine Nutzung außerhalb der Spezifikation eines Frameworks bzw. eine nicht konsequente Nutzung von Paradigmen führt zu keiner Verbesserung der Situation.

6.7. Abstimmung von Testsystemen und Produktionssystemen

Die Entwicklung von komplexen verteilten Anwendungen mit modernen Programmiersprachen und -paradigmen bietet nicht nur Vorteile. Durch die *Entwicklung* auf Arbeitsplatzrechnern, aber den *Einsatz* der Software auf Server- oder Großrechnersystemen ergeben sich häufig Probleme.

[2]Ein Workarround beschreibt eine Möglichkeit, einem Programmfehler ausweichen zu können, ohne auf die Nutzung der betroffenen Funktionalität verzichten zu müssen.

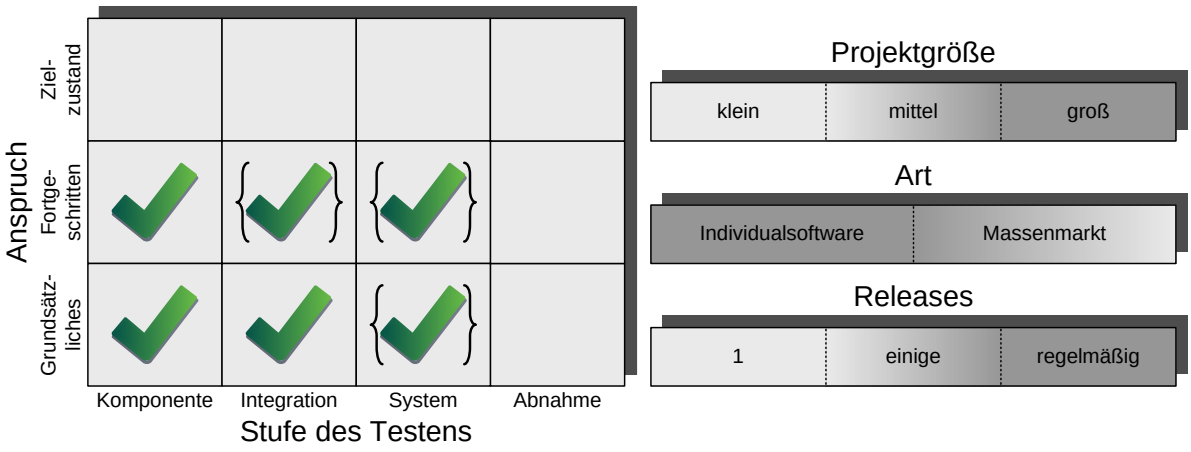

Abbildung 6.6.: Einordnung der Handlungsempfehlung

In frühen Zeit der Softwareentwicklung kamen Programme auf dem System zum Einsatz, auf dem sie entwickelt wurden. Wurden sie auf einem anderen System entwickelt, waren sie erst auf dem Zielsystem lauffähig. Die heutige Wirklichkeit ist anders, was sicherlich sehr viele Vorteile bietet. Allerdings ist es verständlich, dass es bei Unterschieden auf Entwicklungs- und Zielsystem zu Inkompatibilitäten kommen kann. Dabei spielt nicht nur die unterschiedliche Hardware eine Rolle, die anders als erwartet reagieren kann. Auch führt sie zu mitunter erheblichen Leistungsunterschieden, etwa wenn auf Desktop-Computern oder gar Notebooks entwickelt, die fertige Software aber auf Großrechnern ausgeführt wird. Vor allem die zugrunde liegende Software spielt auch eine gewichtige Rolle. Neben unterschiedlichen Betriebssystemen sind auch weitere Komponenten, wie systemweite Bibliotheken betroffen. Weitere Unterschiede betreffen Datenbankmanagementsysteme (DBMS). Deren für Serversysteme gedachte Versionen sind mitunter auf Desktop-Computern erst gar nicht lauffähig. Des Weiteren laufen moderne Applikationen häufig auf so genannten Applikationsservern. Auch hier kann es erhebliche Unterschiede zwischen Entwicklung- und Zielsystem geben. So sind die Funktionen der Applikationsserver zwar häufig standardisiert, aber Produkte verschiedener Hersteller sind dennoch in Teilen inkompatibel. Darüber hinaus besteht oftmals die Möglichkeit, zwischen leichtgewichtigen und schwergewichtigen Ansätzen zu wählen. Wird etwa eine einfache Webanwendung mit Java EE entwickelt, könnte auf dem Entwicklungssystem ein *Tomcat*-Webserver genutzt werden, die alle zum Testen benötigten Container[3] bereitstellt. Das Zielsystem allerdings könnte viele verschiedene Applikationen gemeinsam auf einem IBM *Websphere*-Applikationsserver betreiben. Konflikte sind in diesem Fall fast sicher zu erwarten.

Die Ausführungen für Entwicklungssysteme gelten ebenso schwer für Testumgebungen. Diese sollten den späteren Zielumgebungen möglichst ähnlich sein, selbst, wenn dies zusätzlichen Aufwand bedeutet. Nur so können Probleme durch Inkompatibilitäten

[3]Container bieten den zum Nutzen bestimmte Funktionalitäten benötigten Rahmen. Auf weitere technische Details kann an dieser Stelle verzichtet werden.

ausgeschlossen werden. Zumindest die Tests der späteren Phasen sollten unter möglichst realistischen Bedingungen stattfinden. Ein gängiger Ansatz ist die Nutzung spezieller Entwicklungsversionen der verwendeten Serversysteme. Laut Aussage eines Projektteilnehmers etwa existiert eine „abgespeckte“ Version der Websphere-Laufzeitumgebung, die einem Tomcat-Webserver vorzuziehen sei, wenn das Zielsystem ein Websphere-Server ist. Darüber hinaus können häufig Dienste der Server mitgenutzt werden. So ist es nicht zwingend nötig, DBMS lokal zu installieren. Vielmehr kann auf dem DBMS des Zielsystems eine neue, abgeschottete Datenbank eingerichtet werden, die für Testläufe dient. Moderne Server und noch in viel stärkerem Maße Großrechner bieten überdies umfangreiche Virtualisierungsfunktionen. Somit können sie mit etwas initialem Aufwand als Testsysteme genutzt werden, ohne die parallel laufenden produktiven Anwendungen zu gefährden. Für besonders komplexe oder aufwendige Softwareprojekte, insbesondere solche, die in stetigen Releases erscheinen, kann es nichts desto trotz lohnenswert sein, produktive Umgebungen zu replizieren (siehe Kapitel 5.15).

Ein weiteres Problem von Testsystemen, die nicht identisch mit den Zielsystemen sind, ist die unterschiedliche Leistungsfähigkeit. Verschiedene Probleme, insbesondere, wenn diese mit der Speichernutzung oder mit der parallelen Ausführung zusammenhängen, müssen sich nicht unmittelbar offenbaren und sind auch nur sehr schwierig durch systematisches Testen aufzudecken. Dabei gilt keineswegs, dass ein befriedigendes Leistungsverhalten auf Testsystemen auf mit großzügigerer Hardware ausgestatteter Produktivsysteme zu übertragen ist. Weitere, zunächst nicht betrachtete Abhängigkeiten, wachsende Datenbestände und Ähnliches können unmittelbar oder in der Zukunft zu erheblichen Problemen führen. Tests sind deshalb unbedingt unter *realistischen* Bedingungen durchzuführen. Auch offenbaren sich gerade Fehler in parallel arbeitenden Programmen häufig nur unter bestimmten Voraussetzungen. So genannte *Race Conditions*, in denen sich mehrere Ausführungsstränge eines Programms gegenseitig behindern, zeigen sich möglicherweise erst auf besonders leistungsfähigen Systemen. Auch Aussagen zum zu erwartenden Leistungsverhalten lassen sich nur auf Basis der Ausführung auf dem Zielsystem treffen.

6.8. Kein Testen auf Produktivsystemen

Aufgrund der hohen Wichtigkeit, auf die von einigen Projektteilnehmern hingewiesen wurde, haben wir uns entscheiden, gesondert zu empfehlen, Produktivsysteme nicht für Tests zu verwenden. Denn dabei auftretende Fehler sind in vielen Fällen völlig inakzeptabel.

Wichtig ist, dabei die Systemgrenzen, mögliche Auswirkungen und Zugriffsmöglichkeiten genau zu beleuchten. Im letzten Kapitel empfahlen wir, Tests durchaus auf produktiv eingesetzten Hardwaresystemen laufen zu lassen, da nur diese realistische Ergebnisse bieten. Zu beachten ist in einem solchen Fall allerdings, dass die zu testende Anwendung tatsächlich von anderen auf dem System laufenden Anwendungen unabhängig ist. Die Virtualisierungsmöglichkeiten von Großrechnersystemen bieten dies etwa an. Für dennoch eintretende Störfälle muss es die Möglichkeit geben, die Folgen zu begrenzen. So

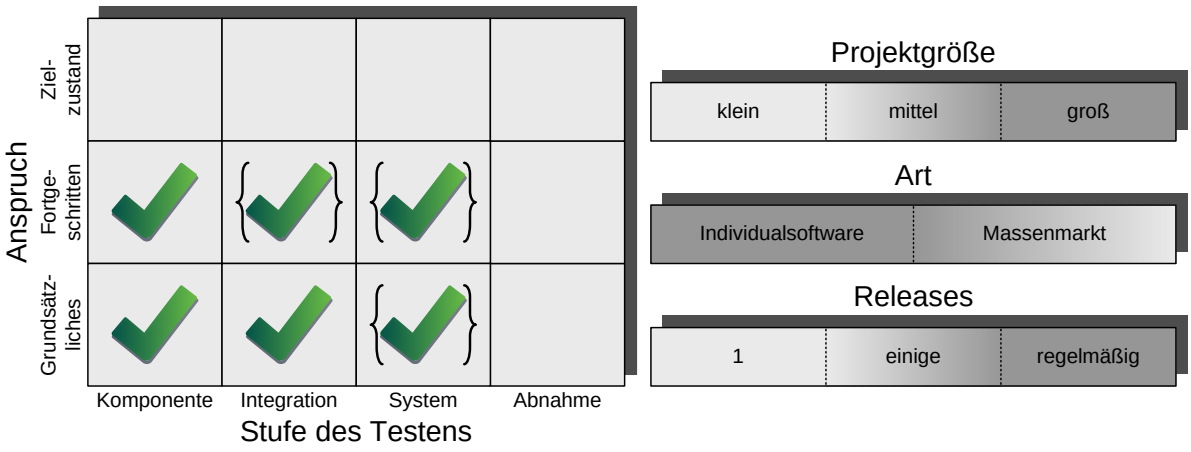

Abbildung 6.7.: Einordnung der Handlungsempfehlung

ist z. B. eine Ressourcenkontrolle sinnvoll, die verhindert, dass außer Kontrolle geratene Testsysteme nicht immer weiter Ressourcen allokieren oder den Großteil der verfügbaren Rechenkapazität für sich beanspruchen, obwohl sie von anderen Anwendungen benötigt würde. Im Allgemeinen ist eine derartige Trennung mit vertretbaren Aufwand möglich, insbesondere unter Zuhilfenahme der immer ausgeklügelten Funktionen zur Virtualisierung.

Viel problematischer stellt sich die logische Trennung dar. Umfangreiche Softwaresysteme sind häufig von weiteren Systemen abhängig. Ein neues System zur Steuerung von Maschinen etwa wird vermutlich Schnittstellen zur Materialwirtschaft haben; ein System, das Vertragsdaten verwaltet, tauscht vermutlich Daten mit Systemen zur Kundenverwaltung aus. Tests sollten so stattfinden, dass dabei keine versehentliche Manipulation der Produktivsysteme möglich ist. Während der Entwicklung ist die wichtigste Maßnahme dafür die logische, besser noch die physikalische Trennung von Test- und Produktivsystemen. Wenn Testsysteme sich in unterschiedlichen Netzwerken befinden, werden viele Probleme von vornherein vermieden. Selbst wenn aber die zu testende Anwendung und abhängige Produktivsysteme auf derselben Hardware laufen, kann eine Verbindung mit entsprechenden Mitteln verhindert werden. So könnte etwa für die virtuelle Maschine, die das Testsystem beherbergt, die Netzwerkschnittstelle deaktiviert oder ein Paket-Filter installiert werden. Besonders wichtig ist, dass nicht nur Maßnahmen ergriffen werden, wenn ein tatsächlicher Zugriff durch das Testsystem zu erwarten ist. Aufgrund der möglichen Fehler und unbekannten Zustände des Testsystems sollte die Absicherung immer aktiviert sein. Darüber hinaus sollten Teststümpfe und Hilfssysteme, die beide die Aufgabe haben, anstelle der produktiven Systemen an die Schnittstellen der zu testenden Software angebunden zu werden, sorgfältig verwaltet und eingesetzt werden.

Lässt sich eine Interaktion mit Produktivsystemen nicht verhindern, so sollten besondere Vorsichtsmaßnahmen getroffen werden. Idealerweise werden nun unkritische Datenbestände bearbeitet, etwa Kundendaten von fiktiven Personen. Für kritische Daten

wird unterdessen ein Schreibschutz gesetzt. Auch bietet es sich an, *Testfenster* zu nutzen, in der die Last durch produktive Abläufe gering ist oder in denen die Systeme für produktive Abläufe sogar gesperrt werden können. Dies könnte etwa am Wochenende oder in der Nacht der Fall sein. Auch wenn dies für mögliche dritte Nutzer der Systeme, wie etwa Partnerunternehmen, zunächst als Hindernis erscheint, kann daraus eine Tugend gemacht werden: Möglicherweise profitieren diese ebenfalls von einem vereinbarten Testfenster, in denen sie ebenfalls neue oder veränderte Systeme testen können.

Nach Abschluss der Tests sind Konsistenzüberprüfungen der Daten unbedingt nötig. Insgesamt bleibt festzuhalten, dass Tests auf Produktivsystemen nur dann durchgeführt werden sollten, wenn es sich unter keinen Umständen verhindern lässt. Ferner sollten Maßnahmen ergriffen werden, die Folgeprobleme verhindern. Dieses sollten im Aufwand angemessen zu dem Wert der Daten sein, die auf den produktiven Systemen verwaltet werden.

Regelungen zur Nutzung von Produktivsytemen für Tests und für die Absicherung der Testsysteme sollten verbindlich festgelegt und allen betroffenen Mitarbeitern mitgeteilt werden. Die Einhaltung dieser Regelungen sollte zudem stichprobenartig kontrolliert werden, wobei die Probedichte mit der Kritikalität der möglicherweise betroffenen Systeme zunehmen sollte. Wichtig ist, dass sich die nötige Sensibilität bei den Mitarbeitern durchsetzt. Die verbreitete Auffassung, dass kurze Probeläufe auf produktiven Systemen gerechtfertigt seien, wenn sie eine erhebliche Beschleunigung eines Tests versprechen und die Risiken gering scheinen, ist abzulehnen. Denn auch wenn derartige Ausnahmen tatsächlich attraktiv sein können und in den meisten Fällen ohne Probleme verlaufen, können Probleme mit einigen weniger Fällen drastische Auswirkungen haben. Durch die fehlende Planung werden Rettungsmaßnahme (etwa der *Rollback* einer Datenbank) erheblich erschwert oder mitunter unmöglich.

Die Nutzung von (gegebenenfalls abgeänderten) Produktivdaten für das Testen birgt ebenso Gefahren. Da es aber auch eine erfolgreiche Strategie darstellen kann, ist es weiter unten als eigenes Kapitel 6.14 aufgeführt. Mögliche Gefahren und geeignete Gegenmaßnahmen werden dort diskutiert.

6.9. Integration der Werkzeuge

Der Einsatz von Werkzeugen für das Testen ist mittlerweile weit verbreitet. Wie in Kapitel 3 dargestellt, ist die Durchdringung des Testprozesses mit Werkzeugen dabei von Unternehmen zu Unternehmen stark unterschiedlich. Auch ist es in verschiedenen Bereich sehr weit verbreitet (etwa für Komponententests), in einigen Bereichen zunehmend stark verbreitet (etwa für Integrations- und Systemtests) und in anderen Bereichen erst kaum genutzt (etwa bezüglich des Testmanagements). Im Allgemeinen lässt sich allerdings feststellen, dass die Integration der Werkzeuge kaum erfolgt. Eine solche ist jedoch sehr empfehlenswert.

Testwerkzeuge sind häufig als alleinstehende Applikationen ausgelegt. Nur in einigen Fällen integrieren sie sich in anderen Entwicklungswerkzeuge, etwa in Form von Plugins für die IDE Eclipse. Gemeinsame Formate oder allgemeine definierte Schnittstellen für

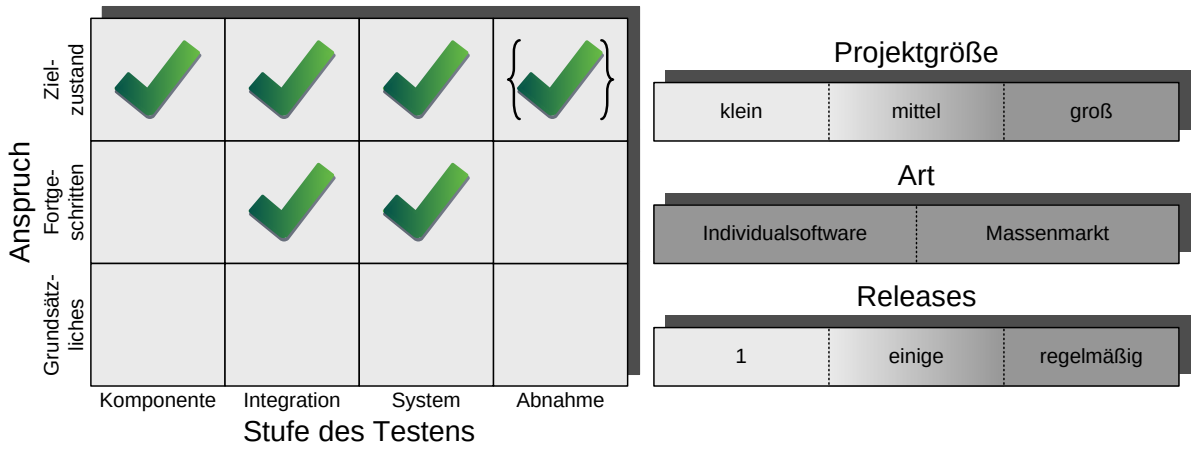

Abbildung 6.8.: Einordnung der Handlungsempfehlung

den Austausch zwischen Werkzeugen existieren nicht. Umfangreiche Testwerkzeuge, wie etwa Programme der IBM Rational-Reihe, bieten zumindest eine Integration mehrerer Funktionen, da sie verschiedene Testfunktionen unter einer Software vereinbaren und die Möglichkeit des Datenaustausch unter den einzelnen Produkten bieten. Aus den Gesprächen mit Projektteilnehmern und den darauf aufbauenden Ausführungen der Kapitel 5 und 6, lässt sich ableiten, dass die Integration von Werkzeugen sehr wünschenswert ist und für sinnvoll erachtet wird. Tatsächlich ist eine solche Integration aber nur sehr rudimentär anzutreffen. Eine Integration dem Testen vor- oder nachgelagerter Systeme (etwa dem Anforderungsmanagement [LS08]) mit Systemen des Testens ist sogar noch seltener.

Undokumentierte Tests haben nur einen Bruchteil des Wertes dokumentierter Tests. Aus diesem Grund ist es sinnvoll, Systeme zur Durchführung von Tests mit den Systemen zur Dokumentation zu vernetzen. Dies befreit Mitarbeiter von lästigen Aufgaben, etwa dem händischen und redundanten Anlegen von Testfällen in beiden Systemen. Vielmehr könnten die ausführenden Systeme Testfälle automatisch in das Dokumentationssystem eintragen und einen Teil der vom Mitarbeiter einzutragenden Daten direkt mitliefern. Dies gilt etwa für den Testerfolg, Laufzeiten und Ähnliches. Das Dokumentationssystem seinerseits bietet eine breite Datenbasis. Diese sollte zur Gewinnung statistischer Informationen genutzt werden (vgl. auch mit dem Kapitel 5.13 über Kennzahlen). Des Weiteren sollte es mit Systemen zur Dokumentation und Verwaltung von Fehlern, so genannten *Bug Trackern* vernetzt werden. Somit können durch das Testen gefundene Fehler direkt mit bekannten Fehlern abgeglichen werden. Dies ist besonders dann sehr wertvoll, wenn Software in regelmäßigen Releases erscheint und auch Kunden die Möglichkeit haben, Fehler zu melden.

Aber auch die Vernetzung der Systeme zur Ausführung einzelner Tests bietet sich an. So ist es etwa sinnvoll, bestimmte Tests früherer Testphasen oder im Rahmen von mehreren Releases früherer Testläufe zu wiederholen. Mit integrierten Werkzeugen wird dies viel einfacher. Ähnlich wie oben beschrieben, kann durch Automatisierungen des

Datenabgleichs und repetitiver Schritte eine hohe Entlastung der Tester erzielt werden. Viele Schritte sind derart aufwendig, dass sie nicht wirtschaftlich von Hand erledigt werden können; gleichzeitig sind sie technisch so einfach, dass sie sich für eine Automatisierung anbieten. Dementsprechend bietet es sich auch an, Systeme zur Verwaltung von Testfällen und Testfalldatenbanken einzubinden. Idealerweise können Testfälle je nach Testlauf in das für die gerade zu erledigende Aufgabe am besten geeignete System importiert werden.

Die Vorschläge dieses Kapitels erscheinen sicherlich zunächst utopisch, wenn aktuell einige wenige Systeme zum Einsatz kommen, die keinerlei Schnittstellen bieten. Nichts desto trotz soll es – in Anlehnung an die Tipps in Kapiteln zum Umgang mit einzelnen Werkzeugen – die Integration einzelner Werkzeuge motivieren und zum Nachdenken anregen. Leider existieren keine umfassenden Lösungen auf dem Markt, was zwangsläufig Eigenentwicklungen nötig macht. Allerdings kann bei der Evaluation neuer Werkzeuge darauf geachtet werden, dass diese Schnittstellen für den Datenaustausch anbieten. Und auch bereits kleinere selbst erstellte Lösungen für Teilfunktionen können in hohem Maße Arbeitserleichterungen bringen. Ein Werkzeug etwa, das direkt aus der Datenbank eines Testfallverwaltungs- oder Dokumentationssystems Daten zur Statistiken verdichtet, ist schnell erstellt und kann dann sukzessive ausgebaut werden. Auf diese Art und Weise kann eine Integration ohne großen Aufwand erreicht werden. Durch das schrittweise Vorgehen entsteht eine breite Erfahrungsbasis. Ein weiteres Beispiel ist die Erweiterung von Werkzeugen für Komponententests. Zu JUnit [13] etwa, einem Komponententestwerkzeug für Java, sind die Quelltexte verfügbar. Eine Erweiterung um eine einfache Funktion, die Testfälle in ein Dokumentationswerkzeug einträgt, sollte schnell zu erstellen sein. Dies könnte ebenfalls nach und nach erweitert werden, so dass die Funktion zusätzlich den Testerfolg und ähnliche Informationen direkt einpflegt.

Auch hinsichtlich der technischen Integration der Testwerkzeuge gilt die Vision, die im Rahmen des fünften Kapitels bezüglich der Motivation von Kennzahlensystemen gezeichnet wurde: Idealziel ist ein System, das Testfallverwaltung, Entwicklung- und Testplanung, Mitarbeiterplanung, Zeiterfassung, Aufgabenverwaltung, Controlling und Management-Cockpit vereint. Jeder, wenn auch kleine Schritt auf dem Weg dorthin bietet unbestreitbare Vorteile, da es Mitarbeiter von repetitiven, wenig anspruchsvollen und „langweiligen“ Aufgaben entbindet und gleichzeitig die Mächtigkeit des Testprozesses erhöht.

6.10. Breite Evaluation von Werkzeugen

Aus Gründen der Lerneffekte, aber auch der effektiven Nutzung von Verträgen, empfiehlt es sich, umfangreiche Testwerkzeuge nur für größere Projekte oder direkt unternehmensweit einzuführen. Grundlage einer solche Einführung sollte eine umfangreiche Evaluation der verfügbaren Werkzeuge sein.

Viele der im Rahmen des Projektes interviewten Unternehmen bringen umfangreichen Testwerkzeugen ein hohes Maß an Skepsis entgegen. Mit umfangreichen Testwerkzeuge sind in dieser Broschüre solche Produkte gemeint, die über die Unterstützung einzelner

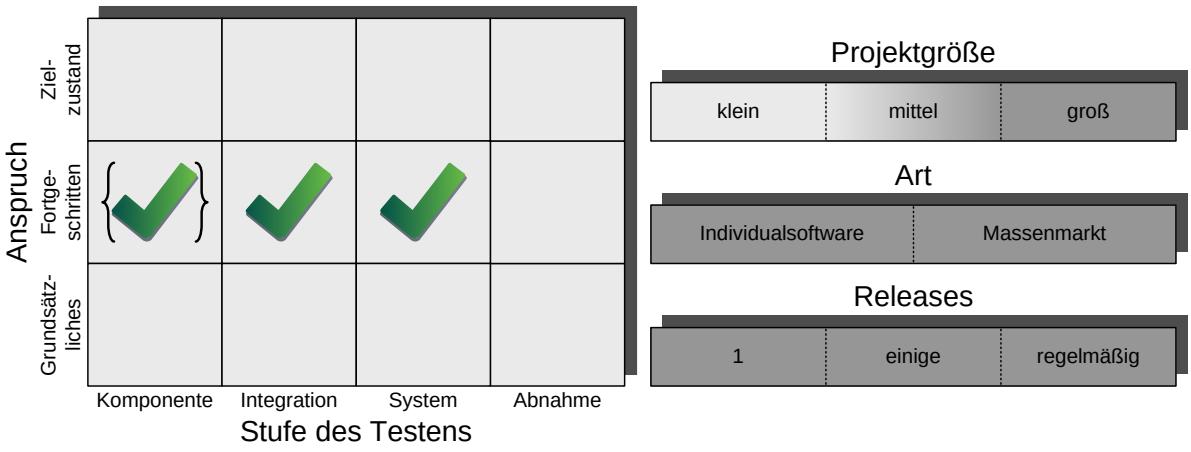

Abbildung 6.9.: Einordnung der Handlungsempfehlung

Testfunktionen hinausgeben (wie etwa „kleine“ Werkzeuge für Komponententests und Ähnliches) und zumindest Teilprozesse des Softwaretestens komplett abbilden. Beispiele hierfür sind diverse Produkte der *Rational*-Reihe von IBM [17] oder Produkte aus dem Bereich *Business Technology Optimization* (BTO) von HP [18] (vormals *Mercury*).

Unternehmen, die Produkte dieser Art nicht einsetzen, haben zwar in der Regel zumindest die Angebote eines Herstellers evaluiert. Diese Evaluation liegt aber häufig einige Jahre zurück und war eher oberflächlich. Sicherlich sind Werkzeuge nicht als Allheilmittel zu sehen. Das sie eine sehr hohe Leistungsfähigkeit erreichen können, steht allerdings mittlerweile außer Frage. Es ist daher zum empfehlen, nicht auf Evaluationsergebnisse aus der Vergangenheit zu vertrauen, sondern gezielt den Einsatz der entsprechenden Produkte zu prüfen. Bereits zwei Jahre nach dem Tests eines Produktes kann dieses sich völlig verändert präsentieren. Für kleinere Werkzeuge, insbesondere Neuentwicklungen aus der Szene der freien Software, gelten noch erheblich kürzere Innovationszyklen. Größere Hersteller ergänzen Ihre Produkte häufig auch durch Akquisitionen.

Die Evaluation sollte, sofern möglich, mehrere Produkte umfassen. Online verfügbare Foliensätze zur Produkt-Demonstrationen durch der Hersteller können dabei nur erste Anhaltspunkte liefern. Bei kleineren Entwicklungsteams ist darüber nachzudenken, Demo-Versionen auszuprobieren. Diese bieten viele Hersteller an. In der Regel sind die Demo-Versionen mit dem verkäuflichen Produkt qualitativ identisch, haben aber quantitative Limitierungen[4]. Auch wenn die Evaluation Mitarbeiter bindet, rechtfertigen die Möglichkeiten, die Werkzeuge bieten können, den entsprechenden Aufwand. Dies bestätigen die Erkenntnis der Unternehmen, die derartige Produkte im Einsatz haben. Für größere Entwicklungsteams empfiehlt sich sogar eine Anfrage beim Hersteller. Dieser ist in der Regel bereit, das Werkzeug vor Ort zu demonstrieren und zu erläutern. Sind technische Einschränkungen der genutzten Programmiersprache, bzw. -umgebung bekannt, sollten vor allem auch Entwickler an einem solchen Treffen teil-

[4]So wäre es z. B. möglich, dass sich nur n Testfälle erstellen und speichern lassen, während die Vollversion eine unbegrenzte Anzahl unterstützt.

nehmen und den Einsatz nach Möglichkeit sogar prototypisch ausprobieren. Für die Evaluation frei verfügbarer Werkzeuge stehen häufig allerdings keine geeigneten Unternehmen zur Verfügung. Hier kann es helfen, ein Vorschlagswesen zu etablieren, bei denen Entwickler und Tester ein „offenes Auge“ für aktuelle Entwicklungen behalten und gelegentlich Zeit erhalten, neue Werkzeuge auszuprobieren. Um redundante Evaluationen zu verhindern, sollte eine Datenbank solcher freien Werkzeuge angelegt werden, in der die Ergebnisse dokumentiert werden.

Aus den Erfahrungen, die von Projektteilnehmern gemacht wurden, lässt sich ableiten, dass es zahlreiche Schwierigkeiten beim Einsatz umfangreicher Testwerkzeuge gibt. Umso mehr empfiehlt sich ein systematischer und objektiver Evaluationsprozess, der kontinuierlich wiederholt wird, oder dem zumindest kontinuierlich Aktualisierungen folgen. Mitunter haben Werkzeuge in wenigen Jahren mehrere Generationen durchlaufen. *Capture&Replay*-Werkzeuge etwa können als deutlich zuverlässiger gelten, als noch vor einigen Jahren. Es lässt sich leider nicht sagen, ob ein Werkzeug für einen bestimmten Einsatzzweck geeignet ist oder gar, ob es für den angestrebten Einsatzzweck und die genutzte Umgebung überhaupt zufriedenstellende Werkzeuge gibt. Als wesentliche Erkenntnis aus den Interviews lässt sich allerdings festhalten, das insbesondere *die* Unternehmen, deren Testprozesse hochgradig effizient sind, Werkzeuge sehr sorgfältig evaluieren und gegebenenfalls neue Werkzeuge in den Betrieb nehmen. Nützliche Hinweise zur Einführung von Werkzeugen finden sich auch in der Literatur. Empfehlenswert ist etwa Kapitel 12 aus [SRWL08]. Für die Evaluation und Einführung von Werkzeugen existieren ferner Standards wie [ISO08b] und [ISO07].

6.11. Anpassen von Werkzeugen an die Prozesse

Die für das Testen erhältlichen Werkzeuge arbeiten aller Regel nach technikgetrieben. Zwar lassen sie sich mitunter anpassen, dies aber nur mit zum Teil erheblichem Aufwand. Im Allgemeinen geben sie durch die Art und Weise, auf diese sie standardmäßig genutzt werden, bestimmte Prozesse vor. Dies ist nicht im Interesse der Unternehmen, die diese Werkzeuge einsetzen.

Insbesondere von den Projektteilnehmern, die einen definierten Testprozess haben und (zumindest zum Teil) zentralisiert Testleistungen erbringen, erhielten wir den Hinweis, dass dringend davon abzuraten sei, Prozesse an Werkzeuge anzupassen. Vielmehr müssten diese an die im Unternehmen vorzufindenden Prozesse anpassbar sein. Wie bereits in Kapitel 5 motiviert, sollten der Testprozess des Unternehmens und seine Teilprozesse wohldefiniert sein. Auch unterliegen sie einer stetigen Optimierung. Das Anpassungen bei der Einführung neuer Werkzeuge stattfinden, ist daher zu erwarten. Diese ergeben sich schon dadurch, dass mit neuen Werkzeugen weitere Testphasen eingeführt bzw. das Testen an sich ausgebaut und verfeinert wird. Die Anpassung der Prozesse sollte allerdings so erfolgen, dass *optimale* Prozesse angestrebt werden. Eine einfache Adaptierung der durch Werkzeuge vorgegebenen Vorgehensweisen kann dabei höchstens der Ausgangspunkt der Überlegungen sein. Nur in wenigen Fällen wird dieser vorgegeben Prozess optimal sein. Insbesondere ist nicht zu erwarten, dass er sich optimal

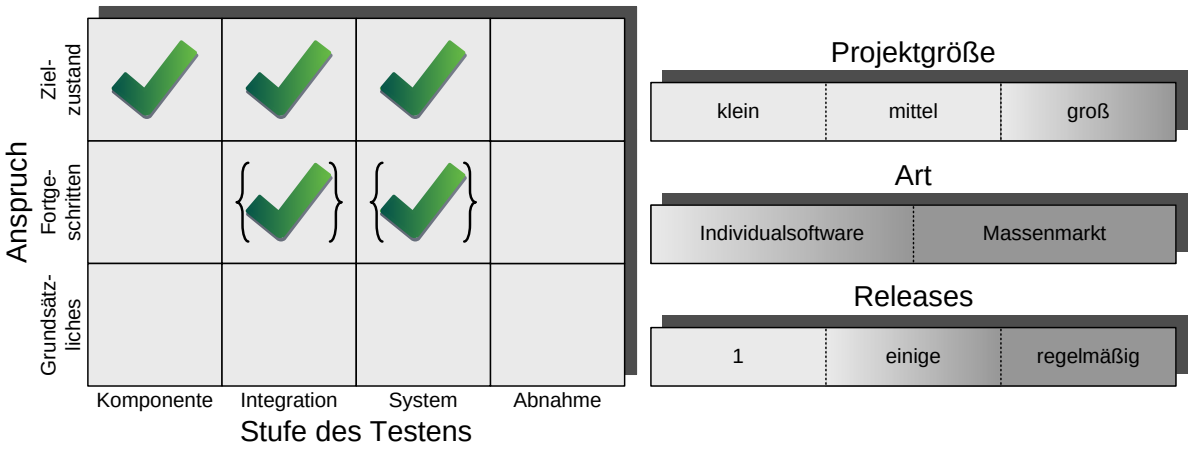

Abbildung 6.10.: Einordnung der Handlungsempfehlung

in den bisherigen Testprozess einfügt. Die Anpassung des Prozesses sollte daher sehr durchdacht erfolgen. Weitere Anpassungen sollten eingeplant werden. Schließlich ist zu erwarten, dass nach einiger Zeit der Arbeit mit neuen oder erweiterten Testphasen bzw. neuen Werkzeugen ergänzende Erkenntnisse vorliegen. Diese legen dann zusätzliche Optimierungen nahe.

Die Grundlage für die Möglichkeit der Anpassung der Werkzeuge an die Prozesse ist technisch geprägt, weshalb diese Maßnahme auch im sechsten und nicht im fünften Kapitel dieser Broschüre Platz gefunden hat. Bereits bei der Evaluation neuer Werkzeuge (siehe Kapitel 6.10), sollte darauf geachtet werden, dass diese keine starren Prozessvorgaben machen. Natürlich lassen sich gewisse Vorgaben naturgemäß nicht vermeiden. Allerdings sollten Werkzeuge niemals statisch eine feste und unveränderliche Arbeitsreihenfolge vorschreiben. Besonders hinderlich ist die Ununterbrechbarkeit von Prozessen sowie die Einschränkung von Prozessschnittstellen. Ließe sich etwa die Dokumentationsfunktion nicht sinnvoll an die Nutzung eines Testwerkzeugs anbinden (siehe Kapitel 6.9), stellt dies organisatorische und technische Einschränkungen durch dieses Werkzeug dar.

Die variable Nutzbarkeit kann von Seiten des Werkzeugs auf niedrigem oder hohen Abstraktionsniveaus bereitgestellt werden. Ein niedriges Abstraktionsniveau böten etwa Funktionen, Teilprozesse unterbrechen zu können und die aktuelle Arbeit zu speichern, oder die Zugriffsmöglichkeit auf Datenbankinhalte zwecks des Imports in andere Systeme. Auf einem hohen Abstraktionsniveau sind wohldefinierte Schnittstellen denkbar und idealerweise eine einfache und möglichst vollständige Anpassbarkeit des Werkzeugs. Ohnehin sollte darauf geachtet werden, dass Werkzeuge mit zunehmender Komplexität zunehmende Möglichkeiten zur Anpassung und Parametrisierung bieten sowie idealerweise Importschnittstellen mitbringen. Sehr geeignet sind in dieser Hinsicht Werkzeuge, die sich in integrierte Entwicklungsumgebungen (IDEs) einfügen, etwa als Plugins.

Durch die Erfahrungen aus dem Projekt lässt sich sagen, dass eher Aufwand für die Anpassung von Software zu rechtfertigen ist, als die Anpassung der eigenen Prozesse. Zudem sollten wenig anpassbare Produkte sehr kritisch betrachtet werden. Denn auch

wenn durch die Anpassung der Software Aufwände entstehen, bietet sich zur gleichen Zeit die Möglichkeit, die eigenen Prozesse zu reflektieren und zu verbessern. Idealerweise kommt ein neues Werkzeug für einige Jahre zum Einsatz und wird sehr rege genutzt. Das Abstellen selbst kleiner Unregelmäßigkeiten lohnt sich langfristig also sehr. Gröbere Unregelmäßigkeiten durch kaum dem eigenem Testprozess gerecht werdende neue Teilprozesse, in denen ein neues Werkzeug zum Einsatz kommt, können bereits kurzfristig sehr Nachteilhaft sein. In beiden Fällen ist der entgangene Nutzen durch besseres Testen deutlich schwerwiegender als der nötig werdende Anpassungsaufwand. Gleichzeitig möchten wir betonen, dass derartige Aufwände kein Grund für eine verzögerte Einführung neuer Werkzeuge oder eine gebremste Weiterentwicklung des eigenen Testprozesses sein sollen. Die zunehmende Komplexität der Softwareentwicklung und der Druck des Marktes erfordern die in dieser Broschüre bereits mehrfach angeregte stetige Weiterentwicklung.

6.12. Nachträgliches Einpflegen der Tests für ältere Software

Durch einen variable angepassten Testprozess ist es normal, dass immer wieder neue Testmethoden und Werkzeuge eingeführt werden. Zudem wächst die Erfahrung der Tester und die Art des Testens wird immer ausgefeilter. Für in der Vergangenheit entwickelte Software stehen diese Verbesserungen grundsätzlich auch zur Verfügung — sie wurde aber natürlich nur mit den damals verfügbaren Methoden und Werkzeugen auf die damals gängige Art und Weise überprüft. Daher kann es sinnvoll sein, Tests nachzuholen. Ähnliches gilt für ältere im Unternehmen betriebene Softwaresysteme (so genannte *Legacy*-Systeme).

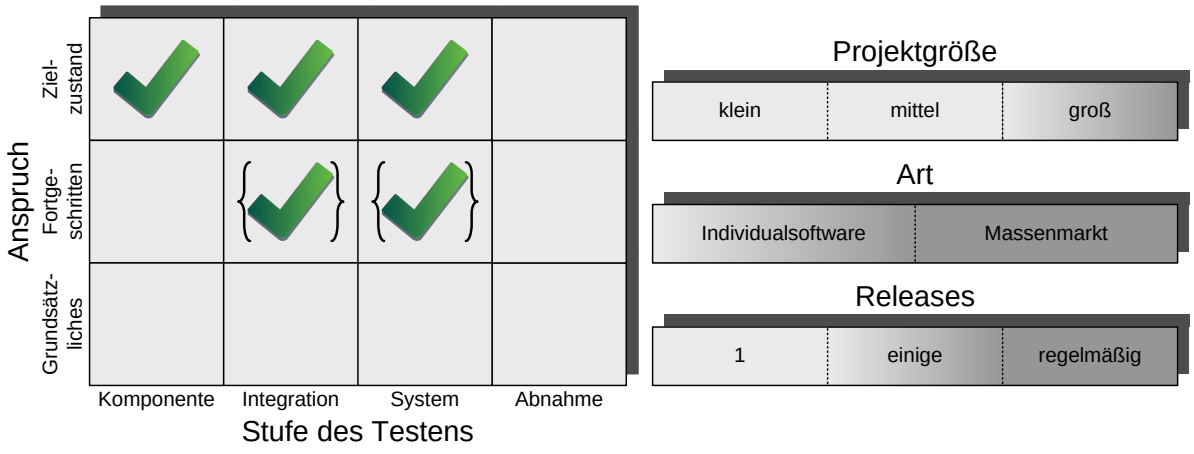

Abbildung 6.11.: Einordnung der Handlungsempfehlung

Wenn Software in mehreren Releases erscheint, ist das nachträgliche Einpflegen weiterer Tests sehr empfehlenswert. Dies gilt selbst für den Fall, dass Tests noch nicht doku-

mentiert wurden. Dies sollte schnellstmöglich nachgeholt werden. Denn je aufwendiger bisherige Tests des Systems waren, desto schwieriger wird eine nachträgliche Dokumentation. Denn Dokumente, Testfälle und Mitarbeiter stehen gegebenenfalls nicht mehr zur Verfügung oder Informationen sind zu ungenau bzw. widersprüchlich. Durch die nachträgliche Pflege kann die Qualität der aktuellen Version wesentlich verbessert werden. Denn im Sinne eines ganzheitlichen Qualitätsanspruchs sollten für aktuelle Produkte alle für erfolgreich befundenen Maßnahmen zur Sicherung der Qualität zum Einsatz kommen. Ja länger allerdings gewartet wird, desto mehr schwindet das Wissen um die Software.

Der zusätzliche Aufwand für das Nachpflegen von Testfällen, die nachträgliche Dokumentation und Ähnliches wird sich aller Regel nach im Laufe des nächsten Releases bezahlt machen. Schließlich werden durch bereits bestehende Testfälle auch Fehler in geänderten Komponenten in neuen Releases gefunden; vorhandene Testfälle und die Dokumentation verhindern redundante Tests. Das Nachpflegen von Testfällen ist schließlich erwartungsgemäß mit geringerem Aufwand verbunden, als das Erstellen neuer Testfälle, die dieselbe Funktionalität abbilden (siehe auch Kapitel 6.4). Dies gilt vor allem dann, wenn die neuen Testfälle durch Tester erstellt würden, die bisher noch keine Berührungspunkte mit der zu testenden Software hatten. Die Intensität der nachträglichen Arbeit ist sicherlich anpassbar. Jede kleinere Änderung des Testprozesses oder die Einführung kleiner Zusatzwerkzeuge muss nicht zu einer kompletten Überprüfung zurückliegender Tests führen. Alle wesentlichen Änderungen sollten aber nachgearbeitet werden.

Kommen im Unternehmen ältere Softwaresysteme zum Einsatz, sollte überprüft werden, ob ein nachträgliches Testen sinnvoll sein kann. Dies gilt insbesondere dann, wenn eine zeitnahe Ablösung dieser Systeme nicht geplant ist. Auch Applikationen, die bereits seit Jahren im Einsatz sind, können kritische Fehler enthalten. Dies gilt umso mehr, wenn neue Systeme an diese angebunden werden sollen und bisher wenig genutzte Funktionalitäten aktiviert werden. In diesen Fällen ist genau zu überprüfen, ob nicht zumindest ein Teil der notwendigen Tests nachträglich erstellt wird. Zum Beispiel könnten häufig genutzte Funktionen einer Oberfläche getestet werden, oder die wichtigsten Schnittstellen einer bisher erst spärlich genutzten Komponente, die in Zukunft aber verstärkt genutzt werden soll. Idealerweise kann der Aufwand dabei sehr in Grenzen gehalten werden, indem ein Teil des Testens automatisiert wird. Häufig sind auch Tests auf einem hohen Abstraktionsniveau ausreichend. Testfälle werden daher in diesem Fall typischerweise als Black Box-Testfälle spezifiziert und aus den Anforderungen an das System oder Schnittstellendokumentationen abgeleitet.

6.13. Regelbasiertes Testen und Entscheidungstabellen

Von einigen der Projektteilnehmer werden so genannte Entscheidungstabellen eingesetzt, um Testfälle zu erzeugen. Aufgrund der guten Erfahrungen, die diese mit der Herangehensweise gesammelt haben, kann eine Allgemeine Betrachtung der Technik empfohlen werden.

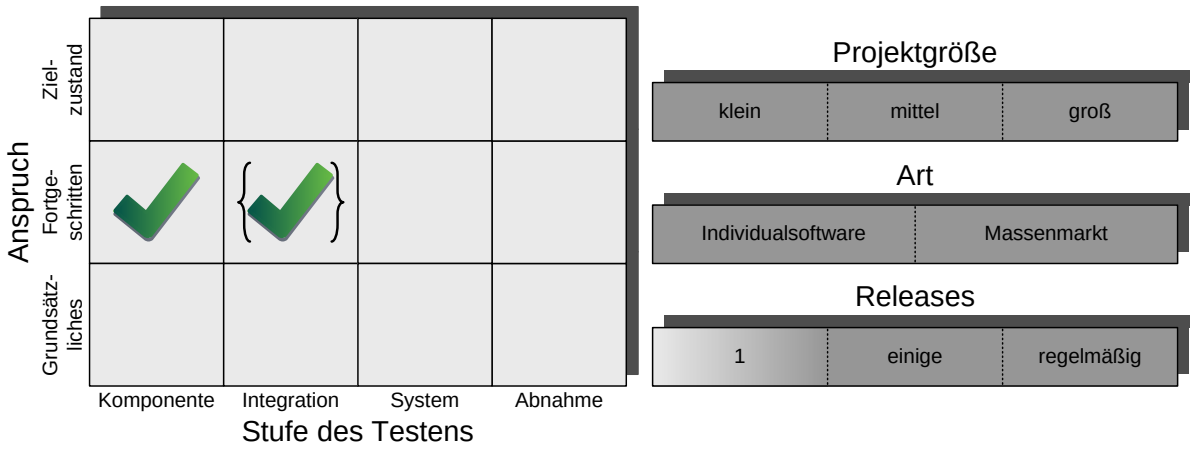

Abbildung 6.12.: Einordnung der Handlungsempfehlung

Bezeichnung	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**
Bedingungen								
System verfügbar	j	j	j	j	n	n	n	n
Korrekte Eingaben	j	j	n	n	j	j	n	n
Benutzer Berechtigt	j	n	j	n	j	n	j	n
Aktionen								
Eingabe speichern	x	x						
Vorgesetzten informieren		x		x				
Eingabe korrigieren			x	x				
Vorgang beenden	x	x			x	x	x	x

Tabelle 6.1.: Beispiel für eine Entscheidungstabelle

Entscheidungstabellen stellen Regelwerke in übersichtlichen Tabellen da. Jede Regel besteht aus Bedingungen und Aktionen. Bestimmten Konstellationen von Bedingungen sind jeweils eine oder mehrere Aktionen zugeordnet. Mögliche Werte für Bedingungen sind normalerweise „trifft zu“ (j), „trifft nicht zu“ (n) oder „interessiert nicht / unbedeutend“ (-). Aktionen können eintreten (x) oder nicht eintreten. Dies soll Anhand eines Beispiels in Tabelle 6.1 dargestellt werden.

Es soll eine einfache Software getestet werden, die Eingaben entgegen nimmt. Sie ist von einem zugrunde liegenden System abhängig, das die Eingaben verarbeitet. Sofern das System verfügbar ist und die Eingaben korrekt sind, werden diese gespeichert. Für den Fall, dass der Benutzer keine Berechtigung hat, wird allerdings sein Vorgesetzter informiert, um die Eingabe gegebenenfalls freizuschalten. In beiden Fällen endet der Vorgang. Falls die Eingaben inkorrekt sind, werden sie automatisch vom System korrigiert. Bei nicht berechtigten Benutzern wird zudem der Vorgesetzte über die inkorrekte Eingabe informiert. Danach kann der Benutzer erneut mit dem System interagieren. In allen Fällen, in denen das System nicht verfügbar ist, wird der Vorgang beendet.

Mithilfe einiger Regel lassen sich Entscheidungstabellen überprüfen und optimieren. Dadurch wird sowohl sichergestellt, dass alle möglichen Fälle abgedeckt sind, als auch dass die Tabelle frei von Redundanz ist. So können in der obigen Tabelle etwa die Regeln T5, T6, T7 und T8 zu einer einzigen Regel zusammengeführt werden, für „Korrekte Eingaben“ und „Benutzer Berechtigt“ wird - gesetzt, um zu verdeutlichen, dass diese Bedingung keine Rolle spielt, falls die Bedingung „System verfügbar“ nicht mit *ja* beantwortet werden kann. Aus den verbleibenden fünf Regeln lassen sich unmittelbar Testfälle ableiten, die dann alle möglichen Zustände abbilden.

Natürlich ist das gewählte Beispiel sehr einfach. Dennoch sind Entscheidungstabellen in der Praxis nutzbar. Viele Sachverhalte lassen sich als Entscheidungstabelle einfacher als zunächst vermutet darstellen, wenn genau beachtet wird, zwischen welchen Funktionen einer Software tatsächlich Abhängigkeiten bestehen und wo diese zu vernachlässigen sind. Zudem wiesen die Projektteilnehmer auf eine effektive Werkzeugunterstützung hin. Entscheidungstabellen müssen zwar in jedem Fall von Hand gefüllt werden, Werkzeuge können diese Arbeit aber komfortabel gestalten. Die Werkzeuge können die benötigten Prüfungen zum Teil automatisch vornehmen und auf Inkonsistenzen hinweisen. Auch die Konsolidierung und Löschung redundanter Bedingungen lässt sich von Tools einfach vornehmen, da die zugrunde liegende Logik sehr einfach ist. Für Menschen wird die Anwendung hingegen mit zunehmender Größe der Entscheidungstabelle mühsam.

Auf folgende Vorteile wurde von den Projektteilnehmern hingewiesen:

- Die Entscheidungstabellen sind prozessorientiert. Eine Überführung in Prozessmodelle bzw. eine Überführung von Prozessmodellen in Entscheidungstabellen ist – zumindest begrenzt – möglich.

- Die einfachen Tabellen lassen sich gut exportieren und weiterverarbeiten. Typische Programme dafür sind Tabellenkalkulationen. Grundsätzlich ist sogar eine komplette Entscheidungstabellen-Applikation auf Basis einer Tabellenkalkulationen denkbar.

- Die Arbeit mit Entscheidungstabellen ersetzt ein willkürliches Vorgehen. An dessen Stelle tritt ein strukturierter Prozess. Auch werden Fälle berücksichtigt und Situationen getestet, an die normalerweise nicht gedacht wird. Zwar achten erfahrene Tester gerade auf solche Fälle, das strukturierte Vorgehen kann aber dazu beitragen, tatsächlich keine Ausnahmen zu übersehen.

- Besonders beim Oberflächentests kann die Arbeitsweise der Tester sehr stark auf die Eingabemaske konzentriert sein. Durch das Erstellen und Ausfüllen der Entscheidungstabellen kann das eigentliche Szenario besser durchdrungen werden, da sie die Arbeit von der Maske löst.

- Die Vollständigkeit der Tests kann erhöht werden. Idealerweise werden Testfälle sogar schneller erstellt als bei einem Vorgehen ohne Entscheidungstabellen, da das vereinheitlichte Vorgehen die Effizienz erhöht.

Grundsätzlich ist eine hundertprozentige Abdeckung möglicher Testfälle möglich, wenn alle Regeln hinterlegt werden. Gleichzeitig sollten möglichst wenige Regeln hinterlegt werden, damit die Komplexität nicht zu hoch wird. Folglich ist die Arbeit mit Entscheidungstabellen eine Tätigkeit, die mit wachsendem Erfahrungsschatz der betreuten Mitarbeiter immer zuverlässiger wird. Sicherlich lässt sich keine absolute Empfehlung aussprechen; Entscheidungstabellen sind vielmehr eine Option. Aufgrund der sehr positiven Erfahrungen der Projektteilnehmern empfehlen wird aber allen Unternehmen eine Prüfung des Einsatzes von Entscheidungstabellen für ihre Zwecke.

6.14. Testen mit Vergangenheitsdaten

Für Software, die bereits im produktiven Betrieb ist und für die eine neue Version entwickelt bzw. die Funktionalität erweitert wird, bietet sich eine spezielle Form der Testdatengenerierung an. Diese lässt sich ebenfalls für Software nutzen, die in regelmäßigen Releases erscheint. Testdaten für solche Software lässt sich auf Basis von Vergangenheitsdaten erstellen.

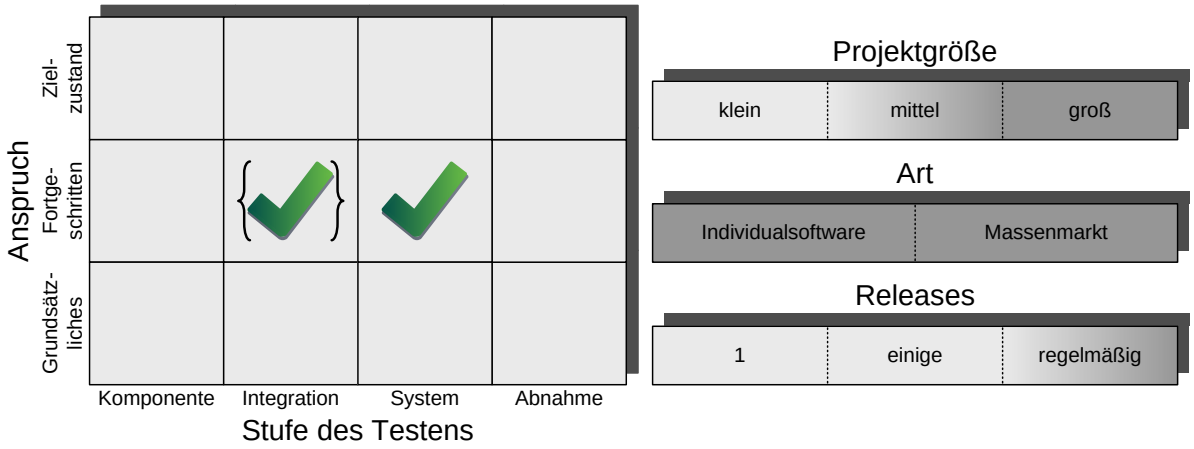

Abbildung 6.13.: Einordnung der Handlungsempfehlung

Vergangenheitsdaten bezeichnen dabei Daten, die bei dem in der Vergangenheit liegenden produktiven Betrieb der Software entstanden oder angefallen sind. Mithilfe solcher Daten lassen sich Ein- und Ausgaben einer Software nachvollziehen. Typischerweise können dafür alle Eingabedaten herangezogen werden, um die Anfragen an das System abzubilden. Für die Verarbeitung werden entweder alle Ausgabedaten, eine Zusammenfassung der Zustandsveränderungen, oder ein Abbild der Datenbank des Systems nach Verarbeitung der Anfragen herangezogen. Typischerweise erstellen die meisten Systeme – insbesondere für kritische Funktionen – Log-Dateien, in denen sie Anfragen dokumentieren. Aus diesen können die Eingabedaten gewonnen werden. Alternativ können an den Schnittstellen Filter installiert werden, die Anfragen protokollieren. Die Protokollierung der Verarbeitung erfolgt entweder über ähnliche Schnittstellen oder über zwei

Abbilder der genutzten Datenbank (vor Verarbeitung der ersten und nach Verarbeitung der letzten Anfrage).

Ein Beispiel ist ein einfaches System zur Verwaltung von Konten. Es basiert auf einer Datenbank, in der Konten als Datensatz aus Name und Kontostand gespeichert sind. Das System nimmt Eingaben der Form `Von, An, Summe` an und verringert den Kontostand von `Von` um die Summe, und erhöht den von `An` entsprechend.[5] Die Arbeit des Systems lässt sich nun nachvollziehen, indem alle Eingabedatensätze protokolliert werden sowie die Liste der Konten und Kontostände einmal für den Beginn der Aufzeichnung und einmal für das Ende vorliegen.

Um die neue Version einer Software zu testen, kann sie mit den Eingabedaten aus der Vergangenheit gespeist werden. Sobald die Verarbeitung abgeschlossen ist, lassen sich die Ergebnisse betrachten. Wird z. B. auf Basis der Datenbank gearbeitet und die Sicherung der alten Datenbank vor der Verarbeitung als Ausgangsbasis benutzt, so sollte die Datenbank der neuen Version nach der Verarbeitung identisch sein mit der Sicherung der alten Datenbank nach der Verarbeitung. Ist dies nicht der Fall, muss geprüft werden, ob in der alten oder in der neuen Version ein Fehler vorliegt. In den meisten Fällen wird dieser Fehler in der neuen Version zu finden sein. Für noch detailreichere Vergleiche kann bei Bedarf auch mit Zwischenergebnissen gearbeitet werden. Die Gefahr, dass sich die Effekte eines oder mehrere Fehler gegenseitig ausgleichen, wird mit zunehmenden Komplexität des Systems und mit einer großen Datenmenge sehr klein und kann in der Regel vernachlässigt werden.

Die besondere Stärke des Testen mit Vergangenheitsdaten zeigt sich bei Performance- bzw. Stresstests. Zwar können auch für funktionale Tests sinnvolle Ergebnisse gewonnen werden. Die Überlegenheit gegenüber anderen Methoden offenbart sich aber bei Leistungsmessungen. Denn das Testen der Leistung von Systemen unter realistischen Bedingungen ist sehr schwierig und aufwendig. Das Testen einzelner Algorithmen ist zwar wenig problematisch, sobald aber ein gesamtes, komplexes Softwaresystem auf sein Verhalten unter Last getestet werden soll, ist entweder eine sehr durchdachte Erzeugung von Eingabedaten vonnöten, oder ein sehr hoher Personaleinsatz. Letzterer kann gegebenenfalls mit normalen Testläufen kombiniert werden, indem sehr viele Tester zur selben Zeit mit dem System arbeiten. Spätestens bei Stresstests, die auch ein Fehlerverhalten des Systems zur Folge haben können, bietet sich dies kaum mehr an, da dann keine Unterscheidung zwischen originär durch die Testfälle und durch die Last verursachten Fehlern möglich ist.

Um Leistungsmessungen vorzunehmen, können die protokollierten Eingabedaten als Testfälle aufgefasst und „abgespielt“ werden. Dabei gibt es ja nach Anforderung mehrere Möglichkeiten. So könnten sie in derselben Geschwindigkeit eingegeben werden, in der sie ursprünglich verarbeitet wurden. Dabei würde dann die Last des Systems beobachtet und z. B. mit Protokollwerten aus der Vergangenheit verglichen. Alternativ kann die Geschwindigkeit der Eingabe variiert werden, ebenfalls unter Beobachtung der Systemlast. Auf diese Weise können unter Umständen Optimierungen an Filtern etc.

[5] Von jeglichen Überprüfungen sei an dieser Stelle abstrahiert. Selbst die Möglichkeit, eine negative Summe einzugeben, spielt für das Beispiel keine Rolle.

vorgenommen werden, um das System über den Test hinausgehend für einen optimalen Durchsatz zu konfigurieren. Mit der Variierung der Eingabegeschwindigkeit lassen sich zudem Flaschenhälse im System aufspüren. Dies ist vor allem dann gut möglich, wenn Testdaten verschiedener Kategorien vorhanden sind, von denen bekannt ist, dass sie unterschiedlich starke Auswirkungen auf das Gesamtsystem haben. Beispiele sind Daten, die aufgrund besonderer Anforderungen zusätzliche Prüfungen und damit die Aktivierung rechenintensiver Untersysteme nach sich ziehen. Schließlich ist es möglich, die Eingaben schnellstmöglich in das System einzugeben. Auch dabei kann das Verhalten des Systems untersucht werden. Bei ausreichend vielen Eingaben wird die Leistungsmessung dabei gleichzeitig Stresstest.

Während die Leistungsmessung bzw. Performcetests darauf abzielen, das Leistungsverhalten eines Systems zu untersuchen und mögliche Schwachstellen aufzudecken, hat der Stresstests die Aufgabe, das Verhalten des Systems in Grenzsituationen zu ermitteln. Dazu zählt etwa festzustellen, ob aufgrund der zu hohen Last Fehler auftreten und Inkonsistenzen in den Daten möglich sind, oder ob das System gemäßigt reagiert (etwa durch Abweisung eingehender Eingaben oder durch Bildung einer Warteschlange und verzögerter Verarbeitung). Zu diesem Zweck lassen sich die Eingaben duplizieren und für zahlreiche Testläufe nutzen. Somit kann nicht nur das unmittelbare Leistungsverhalten ermittelt werden, sondern durch die Simulation von Phasen hoher und niedriger Last getestet werden, ob das System sich nach einer zu hohen Last selbst wieder herstellen kann oder abstürzt.

Für die Arbeit mit Vergangenheitsdaten gelten die in vorangegangenen Kapiteln erläuterten Sicherheitsvorkehrungen. Schon aufgrund rechtlicher Vorschriften wird in vielen Fällen eine Anonymisierung der Daten sinnvoll sein. Die Testsysteme sollten unbedingt von produktiven Systemen gelöst sein, so dass verhindert wird, dass es Auswirkungen auf produktive Systeme geben kann.[6] Nichts desto trotz ist die Nutzung von Vergangenheitsdaten immer dann, wenn eine einfache Generierung simulierter Testdaten schwierig ist oder zu unrealistischen Ergebnissen führt, sehr empfehlenswert. Ihre Domäne liegt vor allem bei den Performance- und Stresstests.

6.15. Parallelbetrieb und Monitoring der alten und neuen Plattform

Der letzte technische Tipp dieser Broschüre schließt sich unmittelbar an die Voraussetzungen der Nutzung von Vergangenheitsdaten an: Er ist dann geeignet, wenn eine neue Version einer bereits im Einsatz befindlichen Software getestet werden soll. Bei sehr kritischen Systemen kann der Test mit Vergangenheitsdaten nicht ausreichend sein. In diesem Fall ist der Parallelbetrieb der alten und neuen Version in Erwägung zu ziehen.

Für den Parallelbetrieb der alten und neuen Version müssen geeignete Kunden bzw.

[6] Um beim Konto-Beispiel zu bleiben: Würden Buchungen an ein weiteres System weitergeleitet, könnte bereits die hundertfache Wiederholung einer Überweisung über 30,00 EUR katastrophale Konsequenzen haben.

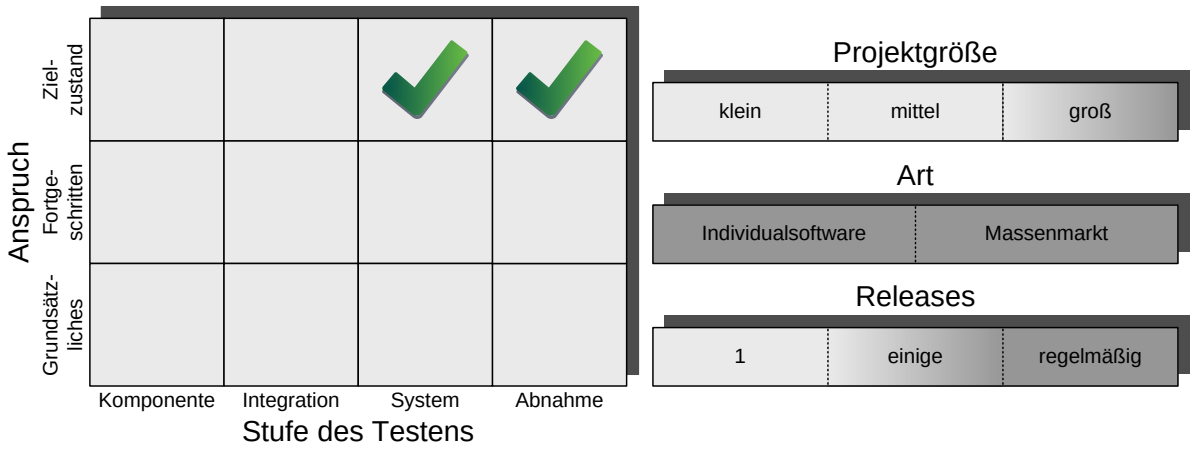

Abbildung 6.14.: Einordnung der Handlungsempfehlung

bei der internen Verwendung der Software geeignete Abteilungen bzw. Mitarbeiter gefunden werden. Zwar sollten diese eher unkritische Aufgaben haben und sich der Tatsache bewusst sein, dass sie mit einem Pilotsystem arbeiten. Gleichzeitig ist diese Empfehlung keineswegs für Phasen vor dem Systemtest gedacht. Schließlich wird die neue Version der Software produktiv genutzt, wenn auch nicht von allen Kunden. Dementsprechend sollte sie auch in dieser Hinsicht getestet worden sein. Der Parallelbetrieb dient nur der Erkennung zusätzlicher Fehler, wenn diese sehr schwierig zu finden sind bzw. inakzeptabel wären. Ziel ist die Erreichung einer außergewöhnlich hohen Qualitätsniveaus. Daher ist der Aufwand ebenfalls sehr hoch.

Während des Parallelbetriebs sollte die neue Software genauestens beobachtet werden. Für diesen Zweck eignen sich zusätzliche technische Lösungen, so genannte *Monitoring*-Werkzeuge. Im Idealfall ist ein direkter Vergleich mit dem alten System möglich, um Abweichungen zu erkennen. Denkbar wäre etwa, das sehr aufwendige, aber grundsätzlich zustandslose Berechnungen von der neuen Software einmal durch ihre eigene Logik vorgenommen und einmal an das alte System weitergeleitet werden. Werden dabei unterschiedliche Ergebnisse ermittelt, sollte die Verarbeitung abgebrochen und die Beobachtungen protokolliert werden. Weitere technische Details des Vergleichs der Systeme gehen über den Rahmen dieser Broschüre hinaus.

Ein weiterer Einsatzzweck des Parallelbetriebs können Leistungsmessung sein, wenn zwar keine kritischen Fehler erwartet werden, das Leistungsverhalten der Software aber extrem kritisch ist. Der Parallelbetrieb sollte dann allerdings erst zum Einsatz kommen, wenn Vergangenheitsdaten bereits genutzt werden, oder wenn deren Nutzung nicht möglich ist. Durch die produktive Nutzung der neuen Version der Software in ausgewählten Bereichen kann ein realistisches Bild der Leistungsverhaltens gewonnen werden. Messungen auf Testsystemen hingegen sind wenig aussagekräftig. Ist die Leistung zufriendenstellend, bietet sich auch die Möglichkeit, zusätzliche Kunden nach und nach auf die neue Version umzustellen. Somit wird ein gemäßigter und sehr „dosierbarer“ Übergang möglich, der bei Leistungsproblemen gegebenenfalls sogar ein Rückgriff

(*Downgrade*) auf die alte Version der Software erlaubt.

Natürlich ist der Parallelbetrieb zweier Versionen einer Software extrem aufwendig. Er erfordert viel Planung und umfangreiche Sicherheitsvorkehrungen. So muss etwa sichergestellt werden, dass der gemeinsame Zugriff auf Datenbanken und Drittsysteme keine Inkonsistenzen oder gegenseitige Blockierungen verursacht. Dies droht vor allem dann, wenn die entsprechenden Systeme für einen exklusiven Zugriff ausgelegt sind. Die Erfahrungen einiger Projektteilnehmer zeigen aber, dass bei besonderen Bedingungen oder bei extremen Anforderungen an die zu testende Software über eine Lösung mit zeitweisem Parallelbetrieb nachgedacht werden sollte. Technische Details sind in hohem Maße von der betroffenen Software und dem Rahmen abhängig, in den sie eingebettet ist. Auch stellt der Parallelbetrieb hohe organisatorische Anforderungen, die jeweils im Detail zu klären sind.

7. Schlussbetrachtungen

In der vorliegenden Broschüre wurde der Status quo des Testens in softwareentwickelnden Unternehmen Nord Westfalens erörtert und darauf aufbauen Empfehlungen für eine Verbesserung des Testens abgeleitet.

Nach der Klärung wichtiger Begriffe wurde daher zunächst dargestellt, wie Unternehmen der Region Software Testen und welche Methoden und Werkzeuge dabei zum Einsatz kommen. Die Ergebnisse entstammen Interviews mit einigen Unternehmen, die sich an dem dieser Broschüre zugrunde liegenden Projekt „Testen“ des Instituts für Angewandte Informatik (IAI). Grundsätzlich wurde dabei eine Zufriedenheit mit dem Status quo festgestellt, ebenso wie der Wunsch einer stetigen weiteren Verbesserung des eigenen Testprozesses. Die Nutzung von Werkzeugen und insbesondere auch die Automatisierung des Testens fallen allerdings recht knapp aus.

Nach der Motivation eines Ordnungsrahmens, mit der sich die folgenden Maßnahmen zur Verbesserung des Testens einordnen lassen, folgte die Darstellung derselben. Ziel war dabei keine holistische Zusammenstellung aller Maßnahmen, die für kommerzielle das Testen von Software nötig sind. Vielmehr basieren alle vorgestellten Maßnahmen auf konkreten Empfehlungen oder im Rahmen des Projektes festgestellten erfolgreichen Strategien der beteiligten Unternehmen. Zur besseren Nutzung wurden die Maßnahmen dabei in organisatorische und technische Empfehlungen aufgeteilt. Aufgrund der direkten Ableitung der Maßnahmen aus den Interviews kann dieser Broschüre nicht die Durchgängigkeit eines Standardwerkes zum Testen von Software erreichen, wenngleich auch zahlreiche Querverweise gezogen werden. Der Praxisbezug und auch die Umsetzbarkeit der Maßnahmen sollte allerdings hervorragend sein.

Die Broschüre zeigt eindrucksvoll, dass auch mit einem relativ kurz laufenden Projekt, das eine Dauer von sechs Monaten hatte, und mit einem eingeschränkten Teilnehmerkreis wertvolle Informationen gewonnen werden können. Auch wenn der Stand des Testens in der teilnehmenden Unternehmen sehr unterschiedlich war, ist zu erwarten, dass für jedes der teilnehmenden Unternehmen wertvolle Hinweise in dieser Broschüre zu finden sind. Zudem bieten die Ausführungen die Möglichkeit, sich die Situation im eigenen Unternehmen zu vergegenwärtigen und kritisch zu reflektieren. Für die Unternehmen der Region sollte die Broschüre eine wertvolle Grundlage zur Erweiterung der eigenen Testprozesse und idealerweise auch zum Überdenken gängiger Vorgehensweisen bieten.

Weiterhin regen die Ergebnisse an, ähnliche Projekte oder auch eine Fortsetzung des Projektes in Angriff zu nehmen. Denn zu den wesentlichen Ergebnissen kann gezählt werden, dass Prozesse rund um die Entwicklung von Software in keinem Unternehmen optimal sind. Gleichzeitig sind die Unternehmen der Region sehr erfolgreich und – auch das beweist das Projekt – gut aufgestellt. Die Chancen, und das Potential, voneinander zu lernen, sind also sehr groß!

Literaturverzeichnis

[Abg] *Abgabenordnung (AO 1977) vom 20. Dezember 2008 (BGBl. I S. 2850, 2856 f.).* – Online: `http://www.gesetze-im-internet.de/ao_1977/index.html`

[Bec00] Beck, K.: *Extreme programming. Die revolutionäre Methode für Softwareentwicklung in kleinen Teams.* München : Addision-Wesley, 2000

[Bil] *Verordnung über Sicherheit und Gesundheitsschutz bei der Arbeit an Bildschirmgeräten (Bildschirmarbeitsverordnung - BildscharbV) vom 4. Dezember 1996 (BGBl. I S. 1841).* – Online: `http://bundesrecht.juris.de/bildscharbv/index.html`

[Blo08] Bloch, J.: *Effective Java.* 2nd. Addision-Wesley, 2008

[DHM98] Dröschel, W. (Hrsg.) ; Heuser, W. (Hrsg.) ; Midderhoff, R. (Hrsg.): *Inkrementelle und objektorientierte Vorgehensweisen mit dem V-Modell 97.* München : Oldenbourg, 1998

[Dij72] Dijkstra, E.: The Humble Programmer. In: *Communications of the ACM* 15 (1972), S. 859–866

[GHJV95] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J.: *Design Patterns. Elements of Reusable Object-Oriented Software.* München : Addision-Wesley, 1995

[Gru] *Grundsätze ordnungsmäßiger DV-gestützter Buchführungssysteme (GoBS 1995) vom 7. November 1995 (IV A 8 – S 0316 – 52/95 BStBl 1995 I S. 738).* – Online: `http://www.bundesfinanzministerium.de/nn_314/DE/`→ `BMF__Startseite/Service/Downloads/Abt__IV/BMF__Schreiben/`→ `015,templateId=raw,property=publicationFile.pdf`

[IEE98a] IEEE, The Institute of Electrical and Electronics Engineers, Inc.: *IEEE Std 1233: IEEE guide for developing system requirements specifications.* New York, 1998

[IEE98b] IEEE, The Institute of Electrical and Electronics Engineers, Inc.: *IEEE Std 829-1998: IEEE Standard for Software Test Documentation.* New York, 1998. – Online: `http://se.inf.ethz.ch/teaching/ss2005/0050/exercises/`→ `REMOVED/IEEE%20Std%20829-1998%20test.pdf`

[ISO94] ISO: *ISO/IEC 12119:1994 Information technology – Software packages – Quality requirements and testing.* Genf, 1994

[ISO05] ISO/IEC: *ISO/IEC 27002:2005: Information technology – Security techniques – Code of practice for information security management.* Genf, 2005

[ISO06] ISO/IEC: *ISO/IEC 25051:2006: Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Requirements for quality of Commercial Off-The-Shelf (COTS) software product and instructions for testing.* Genf, 2006

[ISO07] ISO/IEC: *ISO/IEC TR 14471:2007: Information technology – Software engineering – Guidelines for the adoption of CASE tools.* Genf, 2007

[ISO08a] ISO: *ISO 9241-151:2008 Ergonomics of human-system interaction – Part 151: Guidance on World Wide Web user interfaces.* Genf, 2008

[ISO08b] ISO/IEC: *ISO/IEC 14102:2008: Information technology – Guideline for the evaluation and selection of CASE tools.* Genf, 2008

[Jos09] Díaz, José (Hrsg.): *testing experience – The Magazine for Professional Testers, No. 5.* March 2009. – ISSN 1866-5705

[LS08] Lim, Meike ; Sadeghipour, Sadeg: Werkzeugunterstützte Verknüpfung von Anforderunen und Tests – Voraussetzung für eine systematische Qualitätssicherung. In: *Softwaretechnik-Trends* 28 (2008), Nr. 3, S. 32–33

[o. 92] o. V.: *DO-178B, Software Considerations in Airborne Systems and Equipment Certification.* Radio Technical Commission for Aeronautics (RTCA), 1992

[Sch07] Schuppenhauer, Rainer: *GoDV-Handbuch – Grundsätze ordnungsmäßiger Datenverarbeitung und DV-Revision.* 6. Auflage. München : C. H. Beck, 2007. – ISBN 978-3-406-54313-5

[SRWL08] Spillner, A. ; Rossner, T. ; Winter, M. ; Linz, T.: *Praxiswissen Softwaretest — Testmanagement.* Heidelberg : Dpunkt, 2008

[Toc08] Tochtrop, Gereon: Testexperte als Anforderungsmanager - Ein Erfahrungsbericht. In: *Softwaretechnik-Trends* 28 (2008), Nr. 3, S. 34–37

[YC79] Yourdon, Edward ; Constantine, Larry L.: *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1979. – ISBN 0138544719

Internet-Adressen

[1] http://cs.uni-muenster.de/iai/.

[2] http://cs.uni-muenster.de/iai/foerderkreis/index.html.

[3] http://www.ihk-nordwestfalen.de.

[4] http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf.

[5] http://www.pcwelt.de/start/gaming_fun/archiv/102882/?

[6] http://www.heise.de/tp/r4/artikel/16/16827/1.html.

[7] http://www.spiegel.de/wirtschaft/0,1518,268635,00.html.

[8] http://www.ccc.de/crd/whistleblowerdocs/→
20060731-Gesundheitstelematik.pdf?language=de.

[9] http://www.heise.de/newsticker/meldung/78387.

[10] http://www.heise.de/newsticker/meldung/116836.

[11] http://martinfowler.com/articles/mocksArentStubs.html.

[12] http://weblogs.asp.net/rosherove/archive/2007/09/16/→
mocks-and-stubs-the-difference-is-in-the-flow-of-information.aspx.

[13] http://www.junit.org/.

[14] http://msdn.microsoft.com/de-de/netframework/aa497342(en-us).aspx.

[15] http://www-01.ibm.com/software/awdtools/tester/functional/.

[16] http://www-142.ibm.com/software/dre/ecatalog/→
detail.wss?locale=de_DE&synkey=C108274I57640Y75.

[17] http://www.ibm.com/software/de/rational/.

[18] https://h10078.www1.hp.com/cda/hpms/display/main/→
hpms_home.jsp?zn=bto&cp=1_4011_100__.

[19] http://www.istqb.org/.

[20] http://german-testing-board.info/de/index.shtm.

[21] http://www.knowledge-department.de/UserFiles/File/→
Knowledge%20Department%20-%20Certified%20Tester.pdf.

[22] http://www.knowledge-department.de/→
index.php?surfto=181&pitem=2&citem=3&parent=34&sub=181.

[23] http://www.knowledge-department.de/→
index.php?surfto=180&pitem=2&citem=2&parent=34&sub=180.

[24] http://www.sogeti.de/tpi-test-process-improvement.html.

[25] http://www.sogeti.de/.

[26] http://www.tmmifoundation.org/.

[27] http://www.eclipse.org/.

[28] http://phpeclipse.net/.

[29] http://www.eclipseplugincentral.com/.

[30] http://www.eclipse-plugins.info/eclipse/plugins.jsp.

[31] http://www.tutego.com/java/eclipse/plugin/eclipse-plugins.html.

[32] http://msdn.microsoft.com/en-us/library/ms229045.aspx.

[33] http://java.sun.com/docs/codeconv/.

[34] http://www.bundesbank.de/bankenaufsicht/bankenaufsicht_marisk.php.

[35] http://www.bundesbank.de/download/bankenaufsicht/pdf/marisk/→
071030_rs.pdf.

[36] http://www.bundesbank.de/download/bankenaufsicht/pdf/marisk/→
20090216_kon_0309_MaRisk_Entwurf_der_Neufassung.pdf.

[37] http://www.bsi.de/gshb/deutsch/index.htm.

[38] http://www.bafin.de/cln_109/nn_721290/SharedDocs/→
Veroeffentlichungen/DE/Service/Rundschreiben/2009/→
rs__0903__marisk__va.html?__nnn=true.

[39] http://www.isaca.org/cobit.

[40] http://struts.apache.org/.

A. Hinweise zu Richtlinien

Im Umfeld der Softwareentwicklung existieren zahlreiche Richtlinien und Vorschriften. Zum Teil sind diese generell gültig, zum Teil beziehen sie sich auf Softwareprodukte, die für bestimmte Zwecke bzw. in bestimmten Branchen zum Einsatz kommen. Im Folgenden werden gängige Vorschriften und Gesetze aufgeführt, kurz erläutert und Konsequenzen abgeleitet. Die Zusammenstellung erhebt dabei keinen Anspruch auf Vollständigkeit. Vielmehr wurden solche Richtlinien zusammengestellt, die Projektteilnehmern typischerweise begegnen.

Das Einhalten extern vorgegebener und intern auferlegter Regelwerke wird häufig als *IT-Compliance* bzw. nur *Compliance* zusammengefasst. Hierunter fällt auch die Einhaltung grundsätzlicher Gesetze wie etwa dem **Telekommunikationsgesetz** (TKG), dem **Bundesdatenschutzgesetz** (BDSG) oder dem **Gesetz zur Kontrolle und Transparenz im Unternehmensbereich** (KonTraG).[1] Compliance wiederum lässt sich einer *IT-Governance* bzw. einer *Corporate Governance* unterordnen. Auch wenn die Begriffe Compliance und Governance derzeit inflationär gebraucht werden und die Schlagzeilen der Zeitschriften für (IT-)Management zieren, lohnt sich eine Beschäftigung mit der Thematik. Letztlich ist die Zielsetzung, verpflichtende sowie sinnvolle Regelwerke einzuhalten, Unternehmen zielgerichtet zu führen und Rechtssicherheit zu schaffen. Eine gewisse Zentralisierung und die Einrichtung eines unternehmensweiten Rahmens sind daher sehr empfehlenswert und für Unternehmen ab dreistelligen Mitarbeiterzahl sicherlich unverzichtbar. Die Themen Compliance und Governance bieten allerdings genug Raum für eigene Projekte[2], daher soll eine Vertiefung an dieser Stelle nicht erfolgen.

Hinweise zu Richtlinien mit direkten Konsequenzen für das Testen kamen vor allem aus dem Bereich der Banken und – mit Einschränkungen – Versicherungen. Schließlich unterliegen Finanzdienstleister besonderen Kontrollen und Pflichten bezüglich der Sicherheit ihrer IT-Systeme. Zu nennen sind insbesondere folgende Regelwerke:

- Das BaFin-Rundschreibens zu **Mindestanforderungen an das Risikomanagement** (MaRisk (BA)) [34] bzw. [35] stellt konkrete Anforderungen. So soll sich die „Qualität der technisch-organisatorischen Ausstattung“ unter anderem an der „Risikosituation“ orientieren (AT 7.2, 1). AT 7.2, 3 beschreibt explizit Anforderungen an das Testen: „Die IT-Systeme sind vor ihrem erstmaligen Einsatz und nach wesentlichen Veränderungen zu testen und von den fachlich sowie auch von den technisch zuständigen Mitarbeitern abzunehmen.“ Dabei seien „Produktions- und Testumgebung [...] grundsätzlich voneinander zu trennen“. Die sehr allgemein gehaltene Vorschrift, die „Ausgestaltung der IT-Systeme und [...] IT-Prozesse“

[1] Auf Quellenverweise sei hier verzichtet, da nicht näher auf die Gesetze eingegangen wird.
[2] Möglicherweise sogar im Rahmen des IAI bzw. der IHK Nord Westfalen.

sei auf „gängige Standards abzustellen“ und ihre „Eignung“ sei „regelmäßig“ zu prüfen (AT 7.2, 2), erfordert eine ständige Weiterentwicklung hinsichtlich der IT-Sicherheit und auch des Testens.

- Im Entwurf der **Neufassung von MaRisk (BA)** [36] werden auch Maßnahmen zu IT-Zugriffsrechten gefordert sowie auf den IT-Grundschutzkatalog [37] und die Norm ISO/IEC 27002 [ISO05] (Informationssicherheit) verwiesen.

- Ab Januar 2009 gibt es mit den **MaRisk (VA)** [38] auch eine Fassung für Versicherungsunternehmen.

- Darüber hinaus finden sich viele Vorschriften mit mittelbarem Bezug zum Testen in den Gesetzen zur Kreditvergabe.

Natürlich sind diese Regelungen nicht für alle Unternehmen verpflichtend. Nichts desto trotz kann die Kenntnis derselben nur von Vorteil sein. Insbesondere empfiehlt es sich, eher eine Richtlinie „zu viel“ zu befolgen, als maßgebliche Richtlinien zu übersehen. Gegebenenfalls lässt sich dies sogar kundenwirksam nutzen, indem als Qualitätskriterium die Einhaltung strenger Richtlinien für Finanzdienstleister auch für Produkte angepriesen werden kann, die den Richtlinien eigentlich nicht unterliegen würden.

In der **Abgabenordnung** (AO) [Abg] bzw. den **Grundsätzen ordnungsmäßiger DV-gestützter Buchführungssysteme** (GoBS) [Gru] finden sich unter anderem Regelungen dazu, welchen Maßstäben Systeme zur Buchführung genügen müssen. Diesen Regelungen unterliegen grundsätzlich alle Unternehmen. Neben zahlreichen Forderungen, insbesondere in Hinblick auf die Nachprüfbarkeit und Dokumentation von Vorgängen, werden explizit „Richtlinien für [...] Programmtests“ gefordert, die „den unternehmensspezifischen Besonderheiten Rechnung“ tragen. Auch hier gilt, dass eine Beschäftigung mit den Regelungen über den konkreten Zielbereich (die Buchführung) hinaus sinnvoll sein kann.

Neben Branchenspezifischen Regelungen finden sich auch in IT-Rahmenwerken wie **CobiT** [39] Hinweise auf Vorgehensweisen zum Testen. Empfohlen wurde ebenso das *GoDV-Handbuch – Grundsätze ordnungsmäßiger Datenverarbeitung und DV-Revision* von SCHUPPENHAUER [Sch07]. Der Wert läge vor allem in der Funktion als Ideenlieferant und weniger als konkrete Hilfestellung. Aus diesem Grund lohne sich der Blick in verschiedene Regel- und Rahmenwerke.

Auch die **Bildschirmarbeitsplatzverordnung** (BildscharbV) [Bil] muss für das Testen ins Betracht gezogen werden. Aus der Tatsache, dass Bildschirmarbeitsplätze ergonomisch genutzt werden müssen, ergeben sich Konsequenzen etwa hinsichtlich des Antwortverhaltens von Software. Daraus lassen sich z. B. Ziele für Leistungstests ableiten. Ähnliches gilt für die Norm **DIN EN ISO 9241** [ISO08a], die sich mit Ergonomie auseinandersetzt. Die hieraus abgeleiteten Forderungen können ebenfalls dazu dienen, Testfälle für Oberflächentests sowie für das Leistungsverhalten zu gewinnen.

Abschließend sei noch auf einige Normen verwiesen. Die Norm **ISO/IEC 12119** [ISO94] hat Anforderungen und Tests der Anforderungen von Softwarepaketen zum

Inhalt. Der Nachfolger, **ISO/IEC 25051** [ISO06], geht unter anderem auf die Testdokumentation ein. Eine sehr strenge Norm ist etwa **DO-178B** [o. 92], die Vorgaben für die Softwareentwicklung kritischer Systeme in der Luftfahrt gibt. Neben diesen Normen existieren natürlich zahlreiche weitere. Zum Teil sind diese bei den entsprechenden Handlungsempfehlungen in den Kapiteln 5 und 6 auch direkt angegeben. Eine auf die Unternehmenssituation zugeschnittene Recherche nach maßgeblichen weiteren Normen ist zumindest für große Unternehmen empfehlenswert.

B. Nützliche freie Werkzeuge

Zahlreiche Produkte für die Testfallverwaltung, die Anforderungs- und Fehlerdokumentation und das Testen selbst sind frei verfügbar und könne ohne Lizenzkosten genutzt werden. In vielen Fällen handelt es sich dabei um Open-Source-Software, die bei Bedarf sogar an die eigenen Bedürfnisse angepasst werden kann. Im Folgenden werden viele dieser Werkzeuge kurz vorgestellt, eine Einschätzung bezüglich ihrer Tauglichkeit gegeben, und auf die Möglichkeit sie herunterzuladen hingewiesen.

Die Auflistung im Rahmen dieser Broschüre erhebt keinen Anspruch auf Voll-ständigkeit. Vielmehr werden solche Werkzeuge vorgestellt, die den Autoren entweder durch teilnehmende Unternehmen empfohlen wurden, oder die ihnen bekannt waren und ausprobiert werden konnten. Der Fokus liegt also auf der Qualität der vorgestellten Werkzeuge. Dieses Kapitel des Anhangs wurde vor allem deshalb in die Broschüre aufgenommen, da mehrere der Projekteilnehmern betonten, es mangele an einem Überblick über geeignete Testwerkzeuge. Informationen seien zudem häufig nicht neutral. Auch auf Konferenzen im Umfeld des Testens fänden sich wenige Informationen zu freien Werkzeugen. Darüber hinaus wurde beklagt, dass kommerzielle Werkzeuge häufig bestimmte Vorgehensweisen vorgäben — Open Source-Software sei aufgrund ihrer Anpassbarkeit deutlich flexibler.

Nicht unerwähnt bleiben soll, dass für viele kommerzielle Produkte die Möglichkeit der Evaluation besteht. Die meisten Hersteller bieten entweder eingeschränkte *Trial*-Versionen direkt zum Download an, oder bitten interessierte Nutzer um eine Anfrage. Die Probeversionen sind in der Regel entweder im Funktionsumfang eingeschränkt, erlauben nur eine stichprobenhafte Nutzung (etwa, indem zwar der volle Funktionsumfang zur Verfügung steht, aber die Zahl der erzeugten Testfälle beschränkt ist) oder sind nach einer Probezeit nicht mehr nutzbar. Probeprodukte, die der Hersteller auf Anfrage verschickt, sind zum Teil uneingeschränkt nutzbar. Wenn Interesse an umfangreichen Unterstützungsleistungen besteht, oder keine geeigneten Open Source-Produkte zur Verfügung stehen, sollten die Probemöglichkeiten auf jeden Fall vor einem Kauf genutzt werden.

Werkzeuge aus dem Bereich des modellgetriebenen bzw. -basierten Softwaretests wurden nicht mit aufgenommen. Viele der verfügbaren Werkzeuge befinden sich in einem frühen Stadium oder liegen nur als Forschungsarbeiten vor, sind aber nicht herunterladbar. Zudem sind die häufig domänenspezifisch oder an Voraussetzungen gebunden.

Die meisten der hier aufgeführten Werkzeuge werden derzeit noch weiterentwickelt. Werkzeuge, deren Entwicklung abgeschlossen wurde oder bei denen es seit einigen Jahren offenbar keine Fortschritte mehr gibt, wurden nur in vereinzelten Fällen aufgenommen.[1] Die meisten inaktiven Werkzeugprojekte werden in der Open Source-Szene schnell

[1] Bis auf sehr wenige Ausnahmen finden sich in den Listen keine Werkzeuge, die nicht in diesem oder

durch Folgeprojekte ersetzt, oder finden Nachahmer, die häufig über den ursprünglichen Funktionsumfang hinausgehen. Daraus erklärt sich auch die Zusammenstellung in den folgenden Kapiteln; sie orientiert sich vor allen an den Programmiersprachen C/C++, Java und der .NET-Sprachfamilie.

Eventuell wäre es sinnvoll, eine Liste der Werkzeuge online zur Verfügung zu stellen und sukzessive zu erweitern bzw. mit Erfahrungsberichten anzureichern. Die Autoren dieser Broschüre sind für Kommentare hierzu und Hinweise zu Werkzeugen dankbar. Verweise auf bereits vorhandene Listen finden sich am Ende dieses Anhangkapitels.

B.1. Testfallverwaltung

Aegis
: Aegis ist ein Softwaremanagement-System.
Download: `http://aegis.sourceforge.net/`

JTestCase
: JTestCase dient dazu, *JUnit*-Testfälle in Form von XML-Dokumenten abzulegen. Es bietet damit eine Schnittstelle zwischen Komponententests und der Testfallverwaltung.
Download: `http://jtestcase.sourceforge.net/`

QaTraq
: Eine Web-basierte Testfallverwaltung.
Download: `http://www.testmanagement.com/`

SLAM
: Der Software Lifecycle Artefact Manager (SLAM) ist eine Java-basierte Software zur Testfallverwaltung.
Download: `http://sourceforge.net/projects/slamreq/`

TestLink
: Über eine Browser-basierte Oberfläche lassen sich mit TestLink Testfälle verwalten und Testpläne erstellen. Eine Integration mit Systemen zur Verfolgung von Fehlern wie Bugzilla oder Mantis (siehe B. 2) ist möglich.
Download: `http://www.testlink.org/`

Testopia
: Eine Erweiterung für *BugZilla*, die die Testfallverwaltung integriert.
Download: `http://www.mozilla.org/projects/testopia/`

B.2. Verwaltung von Anforderungen und Fehlern

BugNET
: Ein ASP.NET 2.0-basiertes System zur Fehlerverwaltung.

---

im letzten Jahr aktualisiert wurden.

Download: http://bugnetproject.com/

BugZilla

Mit BugZilla steht eine ausgereifte Software für die Verwaltung von Fehlern zur Verfügung. Es wird Browser-basiert genutzt und kommt auch bei zahlreichen bekannten Projekten zum Einsatz.
Download: http://www.bugzilla.org/

eTraxis

Ein Web-basiertes System zur Verwaltung von Fehlern.
Download: http://www.etraxis.org/

itracker

Ein in Java entwickeltes System zur Verwaltung von Fehlern.
Download: http://sourceforge.net/projects/itracker/

JTrac

Ein in Java geschriebenes, anpassbares Fehlerverwaltungssystem.
Download: http://jtrac.info/

Mantis

Mantis kann nicht nur zur Verwaltung gefundener Fehler verwenden, sondern bietet auch die Möglichkeit der Erfassung von Anforderungen (Feature Requests).
Download: http://www.mantisbt.org/

oops-easytrack

Ein einfaches System zur Verwaltung von Fehlern.
Download: http://sourceforge.net/projects/oops-easytrack/

phpBugTracker

Ein in PHP entwickeltes Fehlerverwaltungssystem.
Download: http://phpbt.sourceforge.net/

Radi

Radi ist ein leichtgewichtiges Testmanagementwerkzeug.
Download: http://sourceforge.net/projects/radi-testdir/

Roundup

Dieses System zur Fehlerverwaltung ist einfach und klein gehalten.
Download: http://roundup.sourceforge.net/

Scarab

Ein weiteres Werkzeug zur Erfassung von Fehlern ist Scarab. Es liegt noch nicht als endgültiges Release vor, wirbt aber mit einer höheren Flexibilität gegenüber BugZilla.
Download: http://scarab.tigris.org/

The Bug Genie

Ein umfangreiches System zur Verwaltung von Fehlern.
Download: http://www.thebuggenie.com/

Trac

Neben einem Wiki und der Erfassung von Fehlern bietet Trac auch Funktionen zum Projektmanagement.

Download: `http://trac.edgewall.org/`

Track+
Ein weiteres Werkzeug zur Verwaltung von Fehlern.
Download: `http://www.trackplus.com/`

Zentrack
Zentrack vereinigt Projektmanagement und Fehlerverwaltung.
Download: `http://www.zentrack.net/`

B.3. Komponententests

Ahven
Eine Komponententestbibliothek für Ada 95.
Download: `http://ahven.sourceforge.net/`

AntUnit
AntUnit dient dem Testen von Projekten, deren Build mit Ant erfolgt.
Download: `http://ant.apache.org/antlibs/antunit/`

Artima SuiteRunner
Dieses Werkzeug kann unter anderem *JUnit*-Testfälle ausführen und danach Berichte erstellen.
Download: `http://www.artima.com/suiterunner/index.html`

AUnit
AUnit ist ein JUnit-Klon für Ada.
Download: `https://libre.adacore.com/aunit/`

BizUnit
Ein Komponententestsystem für .NET.
Download: `http://www.codeplex.com/bizunit/`

blerby
Dieses PHP-basierte Werkzeug führt AJAX-Komponententests aus.
Download: `http://www.blerby.com/project/testrunner`

Cactus
Cactus erweitert JUnit um die benötigte Funktionalität um serverseitigen Java-Code zu testen. So lassen sich etwa Servlets testen, aber auch serverseitige Geschäftslogik, die z. B. mit EJB erstellt wurde.
Download: `http://jakarta.apache.org/cactus/`

cfix
cfix bietet Komponententests für C und C++ und ist für die Entwicklung unter Windows gedacht.
Download: `http://www.cfix-testing.org/`

Cgreen
Cgreen ist ein Komponententestwerkzeug für C.
Download: `http://www.lastcraft.com/cgreen.php`

Check
Ein Unit-Testwerkzeug für C.

Download: `http://check.sourceforge.net/`

CppUnit

Portierung von JUnit für die Nutzung in der Programmierung mit C++.
Download: `http://sourceforge.net/projects/cppunit/`

Crosscheck

Crosscheck dient dem Testen von JavaScript. Der Einsatz eines Browser ist nicht nötig.
Download: `http://www.thefrontside.net/crosscheck/`

csUnit

Dieses Werkzeug bietet Komponententests für .NET-Programme.
Download: `http://www.csunit.org/`

CUnitWin32

Ein Unit-Testerkzeug für unter Windows entwickelte C/C++-Programme.
Download: `http://code.google.com/p/cunitwin32/`

cutee

Ein Komponententestwerkzeug für C++ mit dem Anspruch besonders einfacher Nutzbarkeit.
Download: `http://codesink.org/cutee_unit_testing.html`

CxxTest

Ein Komponententestwerkzeug für C++.
Download: `http://cxxtest.tigris.org/`

DDSteps

DDSteps erlaubt die Parametrisierung von JUnit-Tests. Somit kann ein Testfall mehrfach verwendet werden und deckt dabei aufgrund abgeänderter Parameter zusätzlichen Code ab.
Download: `http://www.ddsteps.org/display/www/DDSteps`

DDTUnit

Datengetriebene Komponententests für Java lassen sich mit diesem Werkzeug realisieren.
Download: `http://ddtunit.sourceforge.net/`

depunit

Die Philosophie von depunit ist, dass die Vorgänge um Komponententests vorzubereiten und abzuschließen (*setup* und *tear down*) selbst als Testfälle aufgefasst werden sollen. Es bietet eine dementsprechende Unterstützung an.
Download: `http://code.google.com/p/depunit/`

Ejb3Unit

Um Komponententests für EJB3-Projekte zu realisieren, die auch ohne EJB3-Container ausführbar sind, kann dieses Werkzeug verwendet werden. Der Verzicht auf ein vorheriges Deployment verspricht Produktivitätssteigerungen.
Download: `http://ejb3unit.sourceforge.net/`

JsUnit

Portierung von JUnit für die Nutzung in der Programmierung mit JavaScript.

Download: `http://www.jsunit.net/`

Java Test Runner
Ein Rahmenwerk um funktionale Tests und Stresstests für Java-Programme zu erstellen.
Download: `http://jtrunner.sourceforge.net/JTR/The_JTR_Project.html`

jhammer
jhammer erweitert *JUnit* um wiederholbare Zufallstests.
Download: `http://jhammer.tigris.org/`

JMUnit
Java ME (J2ME)-Applikationen lassen sich mit diesem Werkzeug testen.
Download: `http://sourceforge.net/projects/jmunit/`

JpdfUnit
Mit JpdfUnit lassen sich PDF-Dokumente über das *JUnit*-System testen.
Download: `http://jpdfunit.sourceforge.net/`

JSFUnit
Um Komponententests für Projekte zu erstellen, die Java Server Faces (JSF) verwenden, bietet sich JSFUnit an.
Download: `http://www.jboss.org/jsfunit/`

JSystem
Dieses umfangreiche Komponententestsystem baut unter anderem auf *JUnit* und Ant auf.
Download: `http://www.jsystemtest.org/`

JUnit
JUnit ist das klassische Werkzeug für Komponententests in Java.
Download: `http://www.junit.org/`

JUnitDoclet
JUnitDoclet erleichtert die Erstellung von *JUnit*-Testfällen, indem es automatisch Testfallstümpfe aus dem Quelltext erstellt.
Download: `http://www.junitdoclet.org/`

JUnitEE
Erweiterung von JUnit für die Nutzung in J2EE-Projekten.
Download: `http://www.junitee.org/`

Juxy
Mit Juxy lassen sich Unit-Tests für XSL Transformation (XSLT) in Java realisieren.
Download: `http://juxy.tigris.org/`

moreUnit
Dieses Plugin für die Entwicklungsumgebung Eclipse verspricht eine Vereinfachung der Gestaltung von Unit-Tests.
Download: `http://moreunit.sourceforge.net/index.html`

NUnit
Portierung von JUnit für die Nutzung in der Programmierung mit .NET.
Download: `http://www.nunit.org/`

PHPUnit
Portierung von JUnit für die Nutzung in der Programmierung mit PHP.

Download: `http://www.phpunit.de/`

Pythoscope

Pythoscope ist ein Komponententestwerkzeug für Python.
Download: `http://pythoscope.org/`

QuickCheck

Bei QuickCkeck handelt es sich ursprünglich um ein für die funktionale Programmiersprache Haskell entwickeltes Werkzeug, mit dem sich teilautomatisiert Testfälle erstellen lassen. Das abgeleitete Projekt Functional Java zielt vor allem auf die Verwendung von Closures ab, kann aber auch zur Erweiterung von Unit-Test gute Dienste leisten.
Download: `http://functionaljava.org/`

Weitere Implementierungen existieren z. B. für Perl, Python und Ruby. Als Ausgangspunkt bietet sich hier der Eintrag in der englischsprachigen Wikipedia an:
`http://en.wikipedia.org/wiki/Quickcheck`

ScalaCheck

Ein Komponententestwerkzeug für Scala (und Java).
Download: `http://code.google.com/p/scalacheck/`

ShUnit

Komponententests für die und Unix häufig genutzte Bourne-Shell lassen sich mit diesem Werkzeug erstellen.
Download: `http://sourceforge.net/projects/shunit/`

STAF

Das Software Testing Automation Framework (STAF) dient der Automatisierung von Komponententests.
Download: `http://staf.sourceforge.net/`

Symbian OS Unit

Sogar für das Symbian-Betriebssystem ist ein Komponententestwerkzeug verfügbar.
Download: `http://www.symbianosunit.co.uk/`

Test-Inline

Ein Komponententestwerkzeug für Perl.
Download: `http://search.cpan.org/dist/Test-Inline/`

Testilence

Eine Bibliothek für Komponententests unter PHP 5.
Download: `http://www.testilence.org/`

TestNG

TestNG wurde als Alternative zu JUnit entwickelt. Die Entscheidung zwischen den beiden Werkzeugen ist Geschmackssache. TestNG bietet einige zusätzliche Funktionen. Es soll sich vor allem dann anbieten, wenn zur Initialisierung der zu testenden Software ein größerer Satz von Testfällen ausgeführt werden soll.
Download: `http://testng.org/`

TUT

Ein einfaches und vom Compiler unabhängiges Komponententestsystem für C++.

Download: `http://tut-framework.sourceforge.net/`

UnitTest++
Ein einfaches Unit-Testwerkzeug für C++.
Download: `http://unittest-cpp.sourceforge.net/`

Unity
Unity ist ein Komponententestwerkzeug für C mit Zusatzfunktionen für eingebettete Systeme.
Download: `http://sourceforge.net/projects/embunity/`

vbUnit3
vbUnit3 bietet Komponententests für Visual Basic-Applikationen und COM-Objekte.
Download: `http://www.vbunit.com/`

WSUnit
WSUnit ermöglicht Komponententests von Webservices.
Download: `https://wsunit.dev.java.net/`

XMLUnit
Ein Komponententestwerkzeug für XML. Es liegt in Versionen für .NET und Java vor.
Download: `http://xmlunit.sourceforge.net/`

Neben den vorgestellten *x*Unit-Werkzeugen existieren Portierungen des JUnit-Ansatzes für zahlreiche weitere Programmierspachen. Sie unterscheiden sich zum Teil in ihrem Umfang, bieten aber grundsätzlich die Unterstützung von Komponententests für die jeweilige Programmiersprache und ermöglichen eine testgetriebene Entwicklung.

B.4. Tests der (Web-)Oberfläche

Abbot
Mit Abbot lassen sich Oberflächentests für Java-Programme skripten und ausführen.
Download: `http://abbot.sourceforge.net/`

aost
Das Tellurium Automated Testing Framework ist ein umfangreiches System zum Testen von Web-Oberflächen.
Download: `http://code.google.com/p/aost/`

Canoo WebTest
Canoo WebTest kann genutzt werden, um Tests von Webapplikationen zu automatisieren. Die Tests setzen dabei rein auf der Oberfläche auf und simulieren etwa die Eingabe von Daten in Formularfelder. Tests können in XML oder in Groovy spezifiziert werden.
Download: `http://webtest.canoo.com/webtest`

dogtail
Ein Werkzeug für Oberflächentests, das in Python geschrieben wurde.

Download: `https://fedorahosted.org/dogtail/`

Fit

Fit analysiert von zu testender Software ausgegebene tabellarische Ergebnisse und zieht daraus Schlüsse über die Korrektheit der Berechnung.
Download: `http://fit.c2.com/`

HtmlUnit

HtmlUnit bietet den Zugriff auf von Webserven zurückgesandte Dokumente und die darin enthaltenen Elemente und kann so zur Erstellung von Unit-Tess für Webapplikationen genutzt werden. Diverse „high-level"-Werkzeuge wie etwa *Canoo WebTest* basieren auf HtmlUnit bzw. verwenden es intern.
Download: `http://htmlunit.sourceforge.net/`

HttpUnit

Auch wenn es sich bei HttpUnit streng genommen um ein Werkzeug für Komponententests handelt, haben wir es in diese Rubrik aufgenommen. Denn es emuliert das Verhalten eines Browsers und kann somit zum Testen von Webapplikationen genutzt werden.
Download: `http://httpunit.sourceforge.net/`

Imprimatur

Imprimatur ist ein sehr simples Werkzeug für Web-Tests. Vom Webserver gesendete HTML-Seiten können mithilfe von regulären Ausdrücken ausgewertet werden.
Download: `http://imprimatur.wikispaces.com/`

Jacareto

Ein Capture&Replay-Testwerkzeug für Java.
Download: `http://jacareto.sourceforge.net/wiki/index.php/Main_Page`

Jemmy

Oberflächentests lassen sich mit Jemmy direkt in den Java-Code verankern.
Download: `https://jemmy.dev.java.net/`

jfcUnit

Um JUnit-Test für Swing-Oberflächen auszuführen, kann die Erweiterung jfcUnit genutzt werden.
Download: `http://jfcunit.sourceforge.net/`

Jiffie

Mit Jiffie lässt sich der Microsoft Internet Explorer durch Java-Programme steuern. Auf diese Weise können etwa *JUnit*-Tests verwendet werden, um Web-Oberflächen zu überprüfen.
Download: `http://jiffie.sourceforge.net/`

JWebUnit

JWebUnit verspricht eine Beschleunigung der Entwicklung von Testfällen für *HtmlUnit* bzw. *Selenium*.
Download: `http://jwebunit.sourceforge.net/`

Latka

In XML beschriebene HTTP(S)-Aufruffolgen können mit Latka abgesetzt werden.

Download: `http://commons.apache.org/latka/`

MaxQ

MaxQ ist ein Capture&Replay-Werkzeug für Webanwendungen.
Download: `http://maxq.tigris.org/`

P.A.M.I.E.

P.A.M.I.E. bietet automatisierte Test von Web-Oberflächen. Es steuert dazu den Microsoft Internet Explorer.
Download: `http://pamie.sourceforge.net/`

pywinauto

Dieses Werkzeug bietet eine Capture&Replay-Funktionalität für mit Python entwickelte Windows-Applikationen.
Download: `http://sourceforge.net/projects/pywinauto/`

Sahi

Ein Werkzeug für Web-Tests, das eine Capture&Replay-Funktionalität bietet.
Download: `http://sahi.co.in/w/`

Selenium

Selenium ist ein umfangreiches Rahmenwerk zum Testen von Webapplikationen. Es bietet Automatisierungsmöglichkeiten und ist grundsätzlich unabhängig von der Programmiersprache der zu testenden Applikation. Zusätzliche Komponenten sind für zahlreiche Programmiersprachen verfügbar.
Download: `http://seleniumhq.org/documentation/`

Slimdog

Ein auf *HttpUnit* basierendes Werkzeug, um geskriptete Web-Tests zu erstellen.
Download: `http://slimdog.jzonic.org/`

StoryTestIQ (STIQ)

Als Kombination von *Selenium* und *FitNesse* ergibt sich StoryTestIQ.
Download: `http://storytestiq.solutionsiq.com/wiki/Main_Page`

StrutsTestCase

Um Webapplikationen, die Apache Struts[40] verwenden, mit JUnit testen zu können, kann der Aufsatz StrutsTestCase verwendet werden.
Download: `http://strutstestcase.sourceforge.net/`

Simple Web Testing - Eiffel (swete)

Ein Werkzeug für Oberflächentests von Eiffel-Programmen.
Download: `http://sourceforge.net/projects/swete/`

SWTBot

Mit diesem Werkzeug können Capture&Replay-Tests für Java-Applikationen erstellt werden, die das Simple Widget Toolkit (SWT) nutzen. Es integriert sich in die Entwicklungsumgebung Eclipse.
Download: `http://swtbot.sourceforge.net/index.html`

TagUnit

TagUnit ist ein Komponententestswerkzeug speziell für Oberflächen, die mit Java Server Pages (JSP) erstellt wurden.

Download: `http://tagunit.sourceforge.net/`

UISpec4J
UISpec4J setzt auf *JUnit* und *TestNG* auf und dient dem Testen von Java-Swing-Applikationen.
Download: `http://www.uispec4j.org/`

Watij
Ein weiteres Web-Test-Werkzeug auf Basis von Java. Es orientiert sich an *Watir*.
Download: `http://www.watij.com/`

WatiN
Ein an *Watir* angelehntes Werkzeug für .NET.
Download: `http://watin.sourceforge.net/`

Watir
Ein Web-Testing-Werkzeug auf Basis von Ruby.
Download: `http://wtr.rubyforge.org/`

B.5. Tests der Datenhaltung

databene benerator
Nicht nur Datenbanken, sondern z. B. auch XML- oder CSV-verarbeitende Systeme lassen sich mit diesem Werkzeug überprüfen. Der Fokus liegt dabei auf Lasttests.
Download: `http://databene.org/databene-benerator/`

DbUnit
Für Anwendungen, die eine relationale Datenbank intensiv nutzen ist DbUnit geeignet. Es dient dazu, Datenbanken während Testläufen in definierte Zustände zu bringen. Auf diese Weise sind deterministische Testläufe trotz Einbindung der Datenbank möglich. Ergebnisse werden reproduzierbar. Das Werkzeug ist für Java verfügbar und wird typischerweise im Rahmen von J2EE eingesetzt.
Download: `http://www.dbunit.org/`

Hammerora
Dieses Werkzeug kann Lasttests für Oracle- und MySQL-Datenbanken vornehmen.
Download: `http://hammerora.sourceforge.net/`

NDbUnit
Bei NDbUnit handelt es sich um ein *DbUnit* sehr ähnliches Tool für das .NET-Framework.
Download: `http://sourceforge.net/projects/ndbunit/`

B.6. Generierung von Teststümpfen und Mock-Objekten

ADO.Mock
ADO.Mock simuliert die Datenbank in .NET-Projekten, die ADO.Net nutzen.

Download: http://ado-mock.tigris.org/

Amock

Amock bietet eine aspektorientierte Herangehensweise an die Erzeugung von Mock-Objekten. Das macht es sehr leistungsfähig und befreit es von einigen Einschränkungen, denen auf Reflexion basierende Werkzeuge unterliegen. Allerdings benötigt es AspectJ. Da Projekte in AspectJ-Projekte konvertiert werden müssen, kann dies den Testaufwand steigern. Nichts desto trotz ist das Werkzeug aufgrund der konzeptuellen Stärke sehr interessant.

Download: http://amock.blogspot.com/

AspectJ: http://www.eclipse.org/aspectj/

AMOP

Erstellt Mock-Objekte für C++.

Download: http://code.google.com/p/amop/

ClassMock

ClassMock kann Mock-Objekte für Klassen erzeugen, die Reflexion und Annotationen nutzen.

Download: http://classmock.sourceforge.net/

CMock

CMock erstellt Mock-Objekte bzw. Module für C-Programme.

Download: http://sourceforge.net/projects/cmock/

EasyMock

Mithilfe von EasyMock können Mock-Objekte für Junit-Teste bereitgestellt werden. Es eignet sich insbesondere für die testgetriebene Entwicklung. Die erzeugten Mock-Objekte sind im Allgemeinen robust gegenüber einem Refactoring des Quelltextes. Durch den Einsatz von EasyMock kann während der Komponententests direkt auch ein Test der Schnittstellen erfolgen, was Vorteile für sich anschließende Integrationstests bietet. Hilfreich ist auch, dass bei Aufruf der Mock-Objekte (die in diesem Fall Schnittstellen entsprechen) in einer nicht vorgesehenen Reihenfolge eine Ausnahme (Exception) ausgelöst werden kann.

Download: http://www.easymock.org/

Flex Mock

Mit Flex Mock lassen sich Mock-Objekte für Ruby erstellen.

Download: http://flexmock.rubyforge.org/

Gmock

Dieses Werkzeug erzeugt Mock-Objekte für Ruby.

Download: http://gmock.org/

Google Mock

Das Google C++ Mocking Framework.

Download: http://code.google.com/p/googlemock/

Hippo Mocks

Dieses Werkzeug erstellt Mock-Objekte für C-Programme.

Download: http://www.assembla.com/wiki/show/hippomocks/

jeasytest

jeasytest zielt vor allem darauf ab, Altanwendungen zu simulieren. Es verwendet

AspectJ zur Injektion von Code.
Download: `https://jeasytest.dev.java.net/`

jMock
Einfach zu nutzendes Werkzeug, dass Mock-Objekte für Java zur Verfügung stellt. Insbesondere ist die Benutzung zusammen mit JUnit vorgesehen.
Download: `http://jmock.org/`

JMockit
Umfangreiches Werkzeug zur Erzeugung von Mock-Objekten für Java. Der Funktionsumfang geht über den vieler vergleichbarer Anwendungen hinaus. So lassen sich etwas statische Attribute simulieren.
Download: `https://jmockit.dev.java.net/`

m0cxx0r
m0cxx0r ist eine Bibliothek zum Erstellen von Mock-Objekten für C++.
Download: `http://code.google.com/p/m0cxx0r/`

Mocha
Mocha stellt Mock-Objekte und Teststümpfe für Ruby-Applikationen bereit.
Download: `http://mocha.rubyforge.org/`

MockItNow
Ein weiteres Rahmenwerk um Mock-Objekte für C++-Applikationen zu erstellen.
Download: `http://code.google.com/p/mockitnow/`

MockitoPP
Ein einfaches Werkzeug zur Erstellung von Mock-Objekten für C++.
Download: `http://code.google.com/p/mockitopp/`

MockLib
Ein leichtgewichtiges Werkzeug zur Erstellung von Mock-Objekten. Es ist derzeit für Java und das Google Web Toolkit verfügbar. Versionen für C# und C++ sind in Vorbereitung.
Download: `http://mocklib.sourceforge.net/`

mockpp
mockpp ist ein plattformunabhängiges Testrahmenwerk, das auch Mock-Objekte bereitstellt.
Download: `http://mockpp.sourceforge.net/`

Mockrunner
Auch ohne Applikationsserver lasen sich J2EE-Applikationen mit Mockrunner testen. Es simuliert nicht nur das Verhalten der benötigten Infrastruktur, sondern dient auch als Rahmenwerk für Unit-Tests.
Download: `http://mockrunner.sourceforge.net/index.html`

mocquer
Ein Werkzeug zum Erzeugen von Mock-Objekten für Java.
Download: `https://mocquer.dev.java.net/`

Rhino.Mocks
Rhino.Mocks ist ein Mock-Objekt Rahmenwerk für .NET.

Download: `http://ayende.com/projects/rhino-mocks.aspx`

Test-MockObject
Test-MockObject erzeugt Mock-Objekte für Perl-Skripte.
Download: `http://search.cpan.org/~chromatic/Test-MockObject/`

Unitils
Unitils baut auf diverse Java-Testwerkzeuge auf und bietet nicht nur eine Komponententestfunktionalität, sondern kann auch Mock-Objekte berietstellen. Des Weiteren unterstützt es das Entwicklungsrahmenwerk Spring.
Download: `http://www.unitils.org/`

Weiterführende Informationen inklusive einer Liste von Mock-Implementierungen für zahlreiche Programmiersprachen finden sich im *mock objects weblog* auf
`http://www.mockobjects.com/`

B.7. Last- und Leistungstests

Apache JMeter
Mit dem JMeter lassen sich Leistungstests und -messungen vornehmen. Während das Werkzeug ursprünglich für den Einsatz mit Webservern (also zur Simulation von HTTP-Anfragen) gedacht war, wurde der Leistungsumfang erweitert. So lassen sich auch Webservices (per SOAP), Datenbanken (per JDBC) oder Mailserver testen. Darüber hinaus zeichnet es sich durch seine Erweiterbarkeit aus.
Download: `http://jakarta.apache.org/jmeter/`

CLIF
Ein umfangreiches Testerkzeug für Leistungsmessungen in Java-Projekten.
Download: `http://clif.ow2.org/`

curl-loader
curl-loader ist ein Werkzeug für massive Stresstests von Webservern und kann tausende parallele Zugriffe simulieren.
Download: `http://curl-loader.sourceforge.net/`

Faban
Ein umfangreiches Werkzeug zum Testen von Webanwendungen.
Download: `http://faban.sunsource.net/`

Funkload
Dieses auf Python-Skripten basierende Werkzeug bietet umfangreiche Möglichkeiten zum Testen von Webanwendungen.
Download: `http://funkload.nuxeo.org/`

httperf
httperf kann genutzt werden, um umfangreiche Leistungstests von Webservern bzw. Webapplikationen vorzunehmen.
Download: `http://sourceforge.net/projects/httperf/`

JUnitPerf
JUnitPerf bietet eine Erweiterung von JUnit-Tests um die Möglichkeit, Zeit- und

Lastmessungen vorzunehmen.
Download: `http://www.clarkware.com/software/JUnitPerf.htm`

Seagull
Seagull kann für Last- und Leistungstests von Protokollen des IP Multimedia Subsystems (IMS) genutzt werden.
Download: `http://gull.sourceforge.net/`

SIPp
Ein Lasttestwerkzeug für auf dem Session Initiation Protocol (SIP) basierende Systeme.
Download: `http://sipp.sourceforge.net/`

SLAMD
Verteilte, massive Last- und Stresstests für Java lassen sich mit diesem Werkzeug realisieren.
Download: `http://www.slamd.com/`

The Grinder
Dieses leichtgewichtige Lasttestwerkzeug kann für alle Applikationen genutzt werden, die eine Java-API bieten.
Download: `http://grinder.sourceforge.net/`

Neben den Werkzeugen, die Last- und Leistungstests vornehmen können, existieren zahlreiche Anwendungen, die in den Bereich der *Profiler* fallen. Diese Werkzeuge sind in der Regel passiv und beobachten das Verhalten von Software. So lassen sich etwa Speicherlecks finden. Zusammen mit automatisierten Testwerkzeugen oder Skripten, die Testfälle ausführen, können sie zur Leistungsmessung genutzt werden. Dabei ist vor allem nützlich, dass sie die gemessene Leistung in der Regel nur wenig Verzerren und gleichzeitig eine sehr genaue Auswertung bieten.

B.8. Messung der Codeabdeckung

Cobertura
Cobertura kann die Abdeckung von Zeilen und Zweigen für Tests ganzer Java-Projekte berechnen.
Download: `http://cobertura.sourceforge.net/`
Ein lesenswerter Artikel zur Arbeit mit Cobertura:
`http://www-128.ibm.com/developerworks/java/library/j-cq01316/→ ?ca?dnw-704`

CodeCover
CodeCover kann als Plugin in Eclipse integriert werden und bietet umfangreiche Informationen zur Code-Abdeckung für Projekte, die in Java oder COBOL entwickelt werden.
Download: `http://codecover.org/`

CoverageMeter
CoverageMeter misst die Codeabdeckung von Tests für C/C++.

Download: `http://www.coveragemeter.com/`

Coverclipse

Auf die Visualisierung der Code-Abdeckung von JUnit-Tests spezialisiert ist Coverclipse.

Download: `http://coverlipse.sourceforge.net/`

djUnit

djUnit misst die Codeabdeckung von JUnit-Tests.

Download: `http://works.dgic.co.jp/djunit/`

EMMA

Mit EMMA lässt sich die Code-Abdeckung in Java-Projekten messen. Das Werkzeug bietet umfangreiche Auswertungsfunktionen. Es kann sehr helfen, die Effektivität von Testfällen nachzuvollziehen.

Download: `http://emma.sourceforge.net/`

EclEmma

Um EMMA in Eclipse zu integrieren, kann EclEmma genutzt werden. Dadurch kann z. B. die Abdeckung direkt im Quellcode-Fenster von Eclipse hervorgehoben werden.

Download: `http://eclemma.org/`

Flexcover

Dieses Werkzeug kann die Codeabdeckung für Adobe Flex- and AIR-Applikationen messen.

Download: `http://code.google.com/p/flexcover/`

Javelina

Ein weiteres Werkzeug zum Messen der Codeabdeckung.

Download: `http://sourceforge.net/projects/javelina-cc`

JSCoverage

Um die Codeabdeckung für Testfälle für JavaScript zu messen kann dieses Werkzeug verwendet werden.

Download: `http://siliconforks.com/jscoverage/`

NoUnit

NoUnit hilft die Qualität von *JUnit*-Testfällen zu bewerten. Es zeigt an, welche Klassen und Methoden getestet werden und ob dies direkt oder über Umwege erfolgt.

Download: `http://nounit.sourceforge.net/`

rcov

Misst die Codeabdeckung von Testfällen für Ruby.

Download: `http://eigenclass.org/hiki.rb?rcov`

Spike PHPCoverage

Dieses Werkzeug kann die Codeabdeckung von Testfällen für PHP messen.

Download: `http://developer.spikesource.com/projects/phpcoverage`

TCAT

Mit den Versionen für C/C++ bzw. für Java lässt sich jeweils die Codeabdeckung durch Testfälle bestimmen.

Download: http://www.soft.com/TestWorks/stwindex.html

XRadar

Umfangreiches Analysewerkzeug für Java.

Download: http://sourceforge.net/projects/xradar/

B.9. Überprüfung der Code-Qualität

C Code Analyzer

CCA ist ein Werkzeug, das potentielle Sicherheitslücken in C-Programmen aufspürt.

Download: http://www.drugphish.ch/~jonny/cca.html

Checkstyle

Checkstyle kann dazu genutzt werden, Java-Code auf die Einhaltung festgelegter Code-Konventionen sein. Dies können etwa die offiziellen Sun Code Conventions (http://java.sun.com/docs/codeconv/) sein. Das Werkzeug ist sehr konfigurierbar, so dass auch eigene Regeln festgelegt werden können.

Download: http://checkstyle.sourceforge.net/

CheckThread

Potentielle Probleme in Java-Programmen, die mehrere Threads verwenden, können durch die statische Analyse mit diesem Werkzeug aufgedeckt werden.

Download: http://checkthread.org/index.html

Classycle

Statische zyklische Dependenzen können ein Zeichen von schlechtem objektorientierten Code sein. Dieses Werkzeug spürt derartige Abhängigkeiten auf.

Download: http://classycle.sourceforge.net/

cppcheck

Ein statisches Analysewerkzeug für C und C++. Es prüft auf Speicherlecks, Pufferüberläufe und Ähnliches.

Download: http://cppcheck.sourceforge.net/

Daikon

Daikon entdeckt Invarianten in Programmen, die in den Sprachen C, C++, Eiffel, IOA, Java, und Perl geschrieben sind, sowie in Arbeitsblättern aus Tabellenkalkulationen sowie in weiteren Datenquellen.

Download: http://groups.csail.mit.edu/pag/daikon/

Dependency Finder

Erstellt Metriken und analysiert Abhängigkeiten von Java-Programmen.

Download: http://depfind.sourceforge.net/

DoctorJ

DoctorJ vergleicht (JavaDoc-)Dokumentationen mit dem zugehörigen Quellcode und weist auf mögliche Probleme hin.

Download: http://www.incava.org/projects/java/doctorj/index.html

FindBugs

Mithilfe von FindBugs können zahlreiche häufige Programmier- und Designfehler

bei der Programmierung in Java vermieden werden. Das Werkzeug analysiert Klassen (oder auch ganze Projekte) und listet „verdächtige“ Stellen im Quelltext auf. Diese können dann überprüft werden. Auch wenn sich manche Meldung als Fehlalarm herausstellt, hilft FindBugs bei der Vermeidung vieler typischer Fehler, die bei der manuellen Kontrolle kaum auffallen.
Download: `http://findbugs.sourceforge.net/`

Flawfinder
Ein statisches Analysewerkzeug für C/C++.
Download: `http://www.dwheeler.com/flawfinder/`

FxCop
Microsoft FxCop ist ein statisches Analysewerkzeug für .NET-Programme.
Download: `http://code.msdn.microsoft.com/codeanalysis`

Hammurapi
Hammurapi nimmt eine statische Codeanalyse vor, um potentielle Probleme aufzudecken.
Download: `http://www.hammurapi.biz/`

JavaNCSS
Um die Code-Qualität beurteilen zu können, kann es hilfreich sein, einige Metriken zu kennen. Dieses Kommandozeilenwerkzeug kann zu deren Ermittlung genutzt werden.
Download: `http://www.kclee.de/clemens/java/javancss/`

Java PathFinder
Ein Werkzeug zur Verifikation von Java-Dateien.
Download: `http://javapathfinder.sourceforge.net/`

JDepend
JDepend erstellt eine Vielzahl von Metriken für Java-Projekte.
Download: `http://www.clarkware.com/software/JDepend.html`

Jlint
Dieses Werkzeug zur statischen Codeanalyse verarbeitet auch größere Projekte sehr schnell und liefert Hinweise auf potentielle Fehler und Probleme.
Download: `http://jlint.sourceforge.net/`

JNorm
JNorm analysiert Java-Programme und findet Code-Abschnitt, die durch Aufrufe bekannter freier Bibliotheken ersetzen werden können. Auf diese Art und Weise soll Programmierfehlern vorgebeugt werden.
Download: `http://www.jnorm.org/index.html`

LAPSE
Dieses Werkzeug ist spezialisiert auf die Analyse von J2EE-Projekten und liefert Hinweise auf potentielle Sicherheitslücken.
Download: `http://suif.stanford.edu/~livshits/work/lapse/index.html`

Lint4j
Lint4j führt eine statische Analyse des Codes durch und weist auf potentielle Probleme, etwa mögliche Deadlocks, Probleme beim Threading und mögliche Leistungs-

engpässe hin.
Download: `http://www.jutils.com/`

Perl-Critic

Dieses statische Analysewerkzeug prüft Perl-Skripte auf die Einhaltung gängiger Programmierstrategien.
Download: `http://search.cpan.org/dist/Perl-Critic/`

PHP-sat

Sucht potentielle Sicherheitslücken in PHP-Anwendungen.
Download: `http://www.program-transformation.org/PHP/PhpSat/`

PMD

PMD untersucht Java-Code auf Probleme wie mögliche Fehler, toten Code bzw. duplizierten Code.
Download: `http://pmd.sourceforge.net/`

Radar

Erstellt Metriken und stellt diese grafisch aufbereitet dar.
Download: `http://xradar.sourceforge.net/`

Sparse

Die Analyse von C-Code ist mit Sparse, einem semantischen Parser, möglich.
Download: `http://www.kernel.org/pub/software/devel/sparse/`

Spike PHPCheckstyle

Dieses Werkzeug prüft PHP-Skripte auf die Einhaltung von Programmierregeln.
Download: `http://developer.spikesource.com/projects/phpcheckstyle/`

Splint

Splint überprüft C-Programme auf Programmierfehler und Sicherheitslücken.
Download: `http://splint.org/`

StyleCop

Mit Microsoft StyleCop können Programmierregeln für C#-Projekte festgelegt und deren Einhaltung überwacht werden.
Download: `http://code.msdn.microsoft.com/sourceanalysis/`

Testability-explorer

Der Testability-explorer analysiert Java Bytecode und gibt Hinweise darauf, wie schwierig es sein wird, Testfälle für den betroffenen Code zu erzeugen.
Download: `http://code.google.com/p/testability-explorer/`

UCDetector

Ein Werkzeug um toten Code in Java-Projekten aufzuspüren.
Download: `http://www.ucdetector.org/index.html`

Yasca

Yasca fasst die Funktionalität mehrere freier Analysewerkzeuge wie *FindBugs*, *PMD* und *Jlint* zusammen.
Download: `http://www.yasca.org/`

B.10. Spezielle Testwerkzeuge

AllPairs
Mit AllPairs lassen sich paarweise Testfälle erstellen. Paarweises Testen basiert auf der Beobachtung, dass die meisten Defekte in Software durch das Zusammenspiel maximal zweier Parameter ausgelöst werden. Durch paarweises testen werden alle Kombinationen aus zwei Parametern abgedeckt. Dies reduziert die Zahl der Testfälle im Verhältnis zu der für alle möglichen Kombinationen benötigten Menge erheblich.
Download: `http://apps.sourceforge.net/trac/allpairs/`

AutoIt
AutoIt ist kein Testwerkzeug im eigentlich Sinne. Es ermöglicht vielmehr, Aktionen auf der Windows-Oberfläche als Skripte abzulegen. Somit kann es für Tests von Programmoberflächen instrumentiert werden.
Download: `http://www.autoitscript.com/autoit3/`

Concordion
Mit Concordion lassen sich Akzeptanztests erstellen und automatisieren. Die Besonderheit ist, dass die Tests zunächst in natürlicher Sprache formuliert und danach durch Steuerungshinweise instrumentiert werden. Neben der Java-Version existieren Portierungen für .NET, Python und Ruby.
Download: `http://www.concordion.org/`
Portierungen: `http://www.concordion.org/Ports.html`

Dumbster
Dumbster simuliert das Verhalten eines SMTP-Servers und kann somit genutzt werden, um alle Arten von Applikationen, die E-Mails versenden, zu testen.
Download: `http://quintanasoft.com/dumbster/`

Eclipse Test & Performance Tools Platform
Die Eclipse TPTP bietet ein Rahmenwerk für diverse Arten von Tests.
Download: `http://www.eclipse.org/tptp/`

FitNesse
FitNesse erzeugt aus Tabellen Testfälle. Dazu werden Kombinationen von Eingabeparametern zusammen mit der erwarteten Ausgabe in tabellarischer Form abgelegt und mit sogenanntem „fixture code“ zur Ausführung gebracht.
Download: `http://www.fitnesse.org/`

Gallio
Gallio führt Testfälle für .NET-Programme aus, die mit einer Vielzahl anderer Werkzeuge erzeugt worden sind.
Download: `http://www.gallio.org/`

Infinitest
Infinitest führt Testfälle während der Entwicklung immer wieder aus und gibt auf die Weise sehr frühzeitig Hinweise, wenn eine Änderung Probleme verursacht. Es lässt sich in die Entwicklungsumgebungen Eclipse und IntelliJ integrieren.
Download: `http://www.infinitest.org/`

iTKO LISA Test

LISA ist ein Black-Box-Werkzeug, mit dem sich insbesondere SOA-Architekturen testen lassen. Die Tests sind dabei auf Geschäftsprozesse fokussiert und abstrahieren von technischen Details. Insbesondere sollen sich heterogene Systemumgebungen testen lassen. Leider ist LISA – im Gegensatz zu den anderen hier vorgestellten Werkzeugen – nicht frei verfügbar. Aufgrund des besonderen Fokus wurde es dennoch in diese Liste aufgenommen.
Informationen:: `http://www.itko.com/site/products/lisatest.jsp`
Deutscher Vertrieb: `http://penta-t.com/`

iValidator
iValidator ist ein Integrationstestwerkzeug für Java.
Download: `http://ivalidator.org/`

Jameleon
Jameleon fasst mehrere Java-Testwerkzeuge wie *JUnit* oder *HTMLUnit* zusammen und bietet die Möglichkeit verschiedener Arten von Tests.
Download: `http://jameleon.sourceforge.net/`

JBehave
JBehave bietet die – aus Sicht der eigenen Entwicklern - Weiterentwicklung der testgetriebenen Entwicklung, die verhaltensgetriebene Entwicklung.
Download: `http://jbehave.org/`

JsTester
JsTester erlaubt das Testen von JavaScript innerhalb von Java.
Download: `http://jstester.sourceforge.net/`

jTracert
Mit jTracert können UML-Sequenzdiagramme aus Java-Quellcode abgeleitet werden. Dies kann beim Verständnis des Codes erheblich helfen.
Download: `http://code.google.com/p/jtracert/`

Jumble
Die Güte von Testfällen lässt sich durch Mutationstests ermitteln. Jumble bietet diese Funktionalität für *JUnit*-Testfälle.
Download: `http://jumble.sourceforge.net/`

KeY
Mithilfe von KeY soll die formale Verifikation von objektorientierten Programmen ermöglicht werden. Es ist mit einem Theorembeweiser für die erstrangige dynamische Logik für Java ausgestattet.
Download: `http://www.key-project.org/`

parallel-junit
JUnit-Testfälle werden sequentiell abgearbeitet. Mithilfe von parallel-junit ist hingegen eine parallele Verarbeitung möglich.
Download: `https://parallel-junit.dev.java.net/`

PHPSpec
PHPSpec ermöglicht verhaltengetriebenes Testen in PHP.
Download: `http://www.phpspec.org/`

StoryQ
StoryQ ermöglicht verhaltengetriebenes Testen in C#.

Download: `http://www.codeplex.com/storyq/`

T2 Framework
Das T2-Werkzeug versucht Testfälle aus in Java-Quelltexten gefundenen Spezifikationen abzuleiten.
Download: `http://www.cs.uu.nl/wiki/WP/T2Framework/`

TESTARE
Ein Java-Testwerkzeug für verteilte Applikationen.
Download: `https://testare.dev.java.net/`

Texttest
Texttest vergleicht die Ausgaben von Programmen mit den zu erwartenden Ergebnissen und liefert so Hinweise auf Fehler.
Download: `http://texttest.carmen.se/`

B.11. Weiterführende Links

Die Vielzahl verfügbarer Open Source-Werkzeuge ist extrem groß, so dass die Darstellung in dieser Broschüre zwar einige sehr interessante Applikationen umfasst, aber viele andere Werkzeuge darüber hinaus ebenfalls eines Blickes wert sein könnten. Im Folgenden sind daher einige Webseiten aufgeführt, die Listen von Werkzeugen bzw. einen Überblick bestimmter Werkzeuge zusammengestellt haben. Die Listen können neben frei verfügbaren allerdings auch kostenpflichtige Werkzeuge enthalten. Zu beachten ist überdies, dass viele der in den Listen enthaltenen Werkzeuge nicht mehr weiterentwickelt werden und mitunter dem technischen Stand von vor einigen Jahren entsprechen können.

Testmanagement:

- `http://www.opensourcetesting.org/testmgt.php`
- `http://www.aptest.com/resources.html#mngt`
- `http://www.softdevtools.com/modules/weblinks/viewcat.php?cid=51`
- `http://www.testingfaqs.org/t-management.html`

Verwaltung von Anforderungen und Fehlern:

- `http://www.opensourcetesting.org/bugdb.php`
- `http://www.aptest.com/bugtrack.html`
- `http://www.aptest.com/resources.html#reqs` (Anforderungsverwaltung)
- `http://www.softdevtools.com/modules/weblinks/viewcat.php?cid=50`
- `http://www.testingfaqs.org/t-track.html`

Last- und Leistungstests:

- http://softwareqatest.com/qatweb1.html#LOAD
- http://www.opensourcetesting.com/performance.php
- http://www.aptest.com/resources.html#app-perf (Applikationsebene)
- http://www.aptest.com/webresources.html#web-perf (Web-Ebene)
- http://support.microsoft.com/default.aspx?scid=kb;EN-US;q231282 (Werkzeuge von Microsoft für Webserver-Stresstests)
- http://www.softdevtools.com/modules/weblinks/viewcat.php?cid=49
- http://www.testingfaqs.org/t-load.html

Testwerkzeuge für Java:

- http://softwareqatest.com/qatweb1.html#JAVA
- http://java-source.net/open-source/web-testing-tools (Fokus auf Webtest-Werkzeuge)
- http://java-source.net/open-source/testing-tools
- http://java-source.net/open-source/code-coverage (Fokus auf die Codeabdeckung)
- http://www.opensourcetesting.org/unit_java.php
- http://www.aptest.com/resources.html#app-jsource

Testwerkzeuge für C/C++:

- http://www.opensourcetesting.org/unit_c.php

Testwerkzeuge für .NET:

- http://www.opensourcetesting.org/unit_dotnet.php

Weitere Programmiersprachen:

- http://www.opensourcetesting.org/unit_ada.php (Ada)
- http://www.opensourcetesting.org/unit_javascript.php (JavaScript)
- http://www.opensourcetesting.org/unit_perl.php (Perl)
- http://www.opensourcetesting.org/unit_php.php (PHP)
- http://www.opensourcetesting.org/unit_python.php (Python)
- http://www.opensourcetesting.org/unit_ruby.php (Ruby)

- http://www.opensourcetesting.org/unit_sql.php (SQL)
- http://www.opensourcetesting.org/unit_tcl.php (TCL)
- http://www.opensourcetesting.org/unit_xml.php (XML)
- http://www.opensourcetesting.org/unit_misc.php (Weitere Sprachen)

Komponententests

- http://www.aptest.com/resources.html#app-source
- http://www.softdevtools.com/modules/weblinks/viewcat.php?cid=47
- http://www.testingfaqs.org/t-unit.html

Funktionale Tests

- http://www.opensourcetesting.org/functional.php
- http://www.aptest.com/resources.html#app-func
- http://www.aptest.com/webresources.html#web-func (Funktionale Tests für Webanwendungen)
- http://www.softdevtools.com/modules/weblinks/viewcat.php?cid=48

Paarweises Testen

- http://pairwise.org/ (Informationen)
- http://pairwise.org/tools.asp (Liste der Werkzeuge)

Weitere Werkzeuge:

- http://www.aptest.com/resources.html#app-embed (Testwerkzeuge für eingebettete Systeme)
- http://www.aptest.com/resources.html#app-data (Tests der Datenbank)
- http://www.aptest.com/resources.html#api (API-Tests)
- http://www.aptest.com/resources.html#comm (Tests für die Kommunikation)
- http://www.testingstuff.com/ (Keine Kategorisierung)
- http://www.softdevtools.com/modules/weblinks/viewcat.php?cid=94 (Codeabdeckung)
- http://www.softdevtools.com/modules/weblinks/viewcat.php?cid=111 (Überprüfung der Code-Qualität)

- http://www.testingfaqs.org/t-gui.html (Oberflächentests)
- http://www.testingfaqs.org/t-static.html (Statische Analysewerkzeuge)
- http://www.testingfaqs.org/t-eval.html (Codeabdeckung)
- http://www.wob11.de/programme.html (Barrierefreiheit)

Förderkreis der Angewandten Informatik
an der Westfälischen Wilhelms-Universität Münster e.V.

Einsteinstraße 62
D-48149 Münster
Tel.: +49 251 83 33797
Fax : +49 251 83 33755

ISSN 1868-0801