Technische Aspekte des erfolgreichen Testens von Software in Unternehmen

Tim A. Majchrzak

Institut für Wirtschaftsinformatik
Westfälische Wilhelms-Universität Münster
Münster, Germany
tima@wi.uni-muenster.de

Zusammenfassung Um Softwareprodukte von hoher Qualität zu erstellen ist das Testen unerlässlich. Da es sich bei Softwaretests um eine teure Aufgabe handelt, die nur schwierig zu beherrschen ist, muss ihre organisatorische Einbettung wohlüberlegt sein. Wir haben mit regionalen Unternehmen zusammengearbeitet, um ihre individuellen Stärken und Schwächen hinsichtlich der Entwicklung und insbesondere des Testens von Software kennenzulernen. In der Folge war es uns möglich, erfolgreiche Vorgehensweisen ("best practices") abzuleiten und Empfehlungen zu formulieren. In diesem Artikel wird das Projekt und die gewählte Forschungsmethodik vorgestellt. Danach werden fünf Empfehlungen vorgestellt, deren Fokus auf technischen bzw. technologischen Aspekten des Testens liegt. Es wird insbesondere auch berücksichtigt, welchen Einfluss Programmierpraktiken sowie -paradigmen bzw. die Wahl der Programmiersprache haben. Für jede Empfehlung wird erörtert, unter welchem Gegebenheiten sich ihre Umsetzung anbietet.

1 Einführung

Das Streben nach hoher Softwarequalität ist keine neue Erscheinung. Es wird bereits seit Jahrzehnten versucht, die Qualität mit Hilfe verschiedener Techniken und der Entwicklung neuer Technologien zu erhöhen. Dementsprechend ist der Begriff *Software Engineering* [1] bereits in den 1960er Jahren geprägt worden. Die *Softwarekrise* währt schon seit den 1970ern [2].

Es sind vor allem Großprojekte, deren scheitern Software unzuverlässig erscheinen lässt. Als Beispiel kann der NASA Mars Climate Orbiter angeführt werden. Er verglühte 1999 in der Atmosphäre des Planeten, nachdem metrische und imperiale Einheiten ohne Konvertierung zwischen Teilsystemen ausgetauscht wurden [3]. Dieses Problem hätte durch umfangreiche Tests entdeckt werden können. Unglücklicherweise lassen sich zahlreiche Projekte finden, die aufgrund fehlerhafter bzw. nicht ausreichend getesteter Software scheiterten [4].

Aber nicht gescheiterte Großprojekte, sondern alltägliches Versagen von Software stellt das wirkliche Problem dar [5]. Offenbar wächst die Komplexität von Softwaresystemen schneller, als Maßnahmen zur Beherrschung entwickelt werden [6]. Probleme finden sich in allen Arten von Entwicklungsprojekten. Allerdings

scheitern Projekte natürlich nicht generell; vielmehr entwickeln viele Unternehmen Software mit großem Erfolg. Wir regen daher an, aussichtsreiche Entwicklungsprojekte zu beobachten und erfolgreiche Vorgehensweisen abzuleiten.

In einem Projekt in Zusammenarbeit mit regionalen Unternehmen wollten wir erfahren, was Softwareentwicklung *erfolgreich* macht. Nach Herausarbeitung des Status quo haben wir dazu die gewonnenen Erkenntnisse ausgewertet und schließlich Handlungsempfehlungen abgeleitet. Wir haben für jede Empfehlungen erarbeitet, unter welchen Bedingungen und mit welchen Voraussetzungen sie umzusetzen ist. Denn die tatsächliche Umsetzung ist selbst für bekannte Vorgehensweisen nicht notwendigerweise einfach. Vielmehr ist Kontextwissen erforderlich, das sie für Unternehmen nutzbar macht. Die in diesem Artikel präsentierten Empfehlungen haben einen technischen Fokus oder beziehen sich auf bestimmte Technologien. Handlungsempfehlungen für die organisatorische Einbettung des Softwaretestens haben wir in [6] beschrieben.

Dieser Artikel ist wie folgt strukturiert. Abschnitt 2 führt in das Projekt ein. Die Forschungsmethodik wird in Abschnitt 3 skizziert. Unser Ordnungsrahmen zur Kategorisierung wird in Abschnitt 4 vorgestellt. In Abschnitt 5 diskutieren wir fünf technische Handlungsempfehlungen. Schließlich ziehen wir in Abschnitt 6 ein Fazit und weisen auf zukünftige Arbeiten hin.

2 Hintergrund

Münster und das umgebende Münsterland liegen in Nordrhein-Westfalen. Die Region weist eine hohe Zahl von Unternehmen mit Bezug zur Informationstechnologie auf. Die meisten dieser Firmen sind mittelständisch und entwickeln Software. Einige der größeren Firmen entwickeln zwar ebenfalls Software, dies erfolgt allerdings nur zur Unterstützung der eigenen Geschäftsprozesse. Es handelt sich dabei vor allem um Finanzdienstleister. Alle Unternehmen sind Mitglieder der Industrie- und Handelskammer, die das *Institut für Angewandte Informatik* (IAI) unterstützt. Das IAI ist der Westfälische Wilhelms-Universität angeschlossen und vereint die interdisziplinäre Arbeit von Ökonomen und Informatikern.

Der regelmäßige Austausch von Forschern des IAI und interessierten Unternehmen dient dazu, typischer Probleme der Wirtschaft aufzudecken. Auf dieser Weise erfuhr das IAI von einer generelle Unzufriedenheit mit dem Testen von Software. Die meisten Unternehmen versuchten, die Qualität der entwickelten Software zu erhöhen und gleichzeitig Kosten einzusparen. Nur bedingt war ihnen dabei bekannt, wie diese Ziele zu erreichen waren. Zudem fehlt Unternehmen in der Regel die Zeit, neue Technologien auszuprobieren oder Änderungen der Prozesse zu evaluieren. Nichts desto trotz war festzustellen, dass im Münsterland sehr erfolgreich Software entwickelt wird. Folglich zogen wir zwei Schlüsse [6]: Keines der Unternehmen hat einen *perfekten* Testprozess; alle stehen vor mehr oder weniger schweren Problemen. Jedes Unternehmen hat allerdings individuelle Stärken entwickelt, die dabei helfen, *gute* Software zu entwickeln.

Schließlich wurde das IAI-Projekt zur Verbesserung des Softwaretestens initiiert. Als Ziele wurden festgelegt, dass zunächst der Status quo des Testens in

der Region ermittelt werden sollte, um in der Folge erfolgreiche Vorgehensweisen ("best practices") zu identifizieren. Aus diesen sollten dann Handlungsempfehlungen abgeleitet werden. Es wurde erwartet, dass Stärken komplementär sind und dementsprechend zwar einige bereits bekannte Vorgehensweisen, aber auch viele interessante neue Ansätze gefunden werden konnten.

Die Heterogenität der teilnehmenden Unternehmen und der Softwareentwicklung im Allgemeinen ist sowohl Fluch als auch Segen. So lassen sich Vorgehensweise finden, die als Handlungsempfehlungen von hohem Wert für Unternehmen sind. Gleichzeitig erfordern die meisten Empfehlungen bestimmte Voraussetzungen, damit eine Umsetzung überhaupt möglich wird. Aus diesem Grund haben wir einen Ordnungsrahmen entwickelt, der Unternehmen helfen soll, die für sie relevanten Handlungsempfehlungen auszuwählen. Er wird in Abschnitt 4 erläutert.

Die beiden Ziele des Projekts, also die Ermittlung des Status quo und das Ableiten von Handlungsempfehlungen, lassen sich individuell darstellen. Daher bezieht sich dieser Artikel auf Handlungsempfehlungen, insbesondere auf solche, die technischen Aspekte des Testens oder verwendete Technologien betreffen.

3 Forschungsmethodik

Die für das Projekt gewählte Forschungsmethodik musste zwei Ziele miteinander verbinden. So sollte die Forschung akademischen Ansprüchen genügen, aber für Unternehmen direkt verwertbare Ergebnisse liefern. Aus diesen Gründen kommt ein gestaltungsorientierter Ansatz zum Einsatz, der in der englischsprachigen Literatur als *design science* bezeichnet wird. Das typischer Vorgehen zielt darauf ab, bisher ungelöst Probleme auf neue oder einzigartige Art und Weise anzugehen und effizienter oder effektiver zu lösen [7]. Dabei ist klar, dass es unmöglich ist, einen *idealen* Testprozess zu beschreiben oder eine ganzheitliche Beschreibung von Testmethoden zu liefern. Vielmehr sollen Empfehlungen erarbeitet werden, die möglichst viele typische Probleme adressieren und zu deren Lösung beitragen.

Erfolgreiche Vorgehensweisen werden sicherlich nicht gefunden, wenn Projektteilnehmern ein einfacher Fragebogen vorgelegt wird. Um die Probleme aus Sicht der Teilnehmer beleuchten zu können, haben wir semi-strukturierte Experteninterviews durchgeführt. Durch diesen qualitativen Ansatz, dem nur ein grober Interview-Leitfaden zugrunde lag, erfuhren wir, *wie* in den Unternehmen getestet wird. Im Verlauf der Interviews wurden dann Stärken und Schwächen der einzelnen Unternehmen thematisiert und schließlich erfolgreiche Vorgehensweisen mit den Teilnehmern erörtert.

Die von uns ermittelten Handlungsempfehlungen zielen nicht darauf ab, mit theoretischer oder praktischer Literatur zu Softwarequalität zu konkurrieren. Vielmehr sollen sie den vorhandenen Kenntnisstand ergänzen, der nicht für die Lösung aller in Unternehmen anzutreffenden Probleme ausreicht. In dieser Arbeit wird technischen Aspekten ein besonderer Fokus gegeben. Auch wenn organisatorische Fragen für erfolgreiches Testen gelöst werden müssen, sollte darauf geachtet werden, dass IT-Forschung versucht, neue Erkenntnisse in Bezug auf Informations*technologie* zu gewinnen [8].

Das grundlegende Vorgehen während des Projekts stellt sich wie folgt dar. Zunächst wurden Unternehmen, die das IAI unterstützen, kontaktiert und für Interviews geeignete Mitarbeiter identifiziert. Dabei wurden sowohl Führungskräfte als auch technisch ausgebildete Angestellte ausgewählt. Im zweiten Schritt folgten die Interviews. Mit kleinen Unternehmen wurde in der Regel nur ein Termin vereinbart. Große Unternehmen wurden hingegen mehrfach besucht, so dass Gespräche mit verschiedenen Mitarbeitern geführt werden konnten. In den Interviews sollte dabei geklärt werden, bei *wem* die Testverantwortung liegt, *wann* getestet wird, *was* dabei in die Test eingeschlossen ist (z. B. die graphische Benutzerschnittstelle oder die Geschäftslogik), *welche* Methoden genutzt wurden und *wie* das Testen im Allgemeinen angegangen wird. Des Weiteren haben wir versucht, möglichst viel über den Einsatz von *Testwerkzeugen* zu erfahren.

Nach Ermittlung des Status quo wurde mit den Teilnehmern diskutiert, auf welche generellen Probleme sie gestoßen sind und welche erfolgreichen Vorgehensweisen im Unternehmen entwickelt wurden. In diesem Zug wurde auch erörtert, welche Verbesserungen wünschenswert waren. Schließlich wurden potentiell ableitbare Handlungsempfehlungen besprochen. Dieser letzte Teil der Interviews wurde sehr offen gehalten, so dass zahlreiche Ideen ausgetauscht und viele interessante Ansätze besprochen werden konnten. In der Folge wurden die gewonnenen Erkenntnisse von uns verdichtet und anschließend analysiert. Darauf aufbauend konnte der Status quo gezeichnet und Handlungsempfehlungen zusammengestellt werden. Dabei achteten wir insbesondere darauf, die Bedingungen festzustellen, unter denen Empfehlungen einsetzbar sind.

4 Ordnungsrahmen

Um Handlungsempfehlungen für ein Thema, dass sehr komplex ist und zahlreiche Abhängigkeiten und Querverbindungen aufweist, nutzbar zu machen, bedarf es einer leistungsstarken Kategorisierung. Wirklich nützlich und für Unternehmen interessant werden Empfehlungen erst, wenn erklärt wird, *wie* sie zu nutzen sind und welche Voraussetzungen für die Nutzung erfüllt sein müssen. Aus diesem Grund wird der von uns entwickelte Ordnungsrahmen [6] verwendet.

Der *Anspruch* einer Empfehlung beschreibt, wie umfangreich organisatorische Veränderungen für die Umsetzung ausfallen. *Grundsätzliche* Empfehlungen sind so einfach umzusetzen, dass jedes Unternehmen sie befolgen sollte. Wenn der Aufwand darüber hinaus deutlich steigt, werden sie als *fortgeschritten* eingeordnet. Der *Zielzustand* beschreibt Ideale, die nur mit größtem Aufwand erreicht werden können. Da sie häufig die Initiierung eines Prozesses kontinuierlicher Verbesserungen erfordern, sollten Unternehmen abwägen, ob sie dieses Ziel anstreben wollen. Die erreichbaren Vorteile sind auf lange Sicht aber erheblich.

Ebenso wichtig ist die *Projektgröße*. *Kleine* Projekte werden meistens von einem einzelnen Team bearbeitet, das für Entwicklung und Testen zuständig ist. In *mittleren* Projekten werden diese Aufgaben in der Regel auf mindestens zwei Teams vergeteilt. *Große* Projekte weisen eine veränderte Struktur auf und vereinen häufig die Arbeitskraft hunderter Mitarbeiter. Auch könnten weitere Abtei-

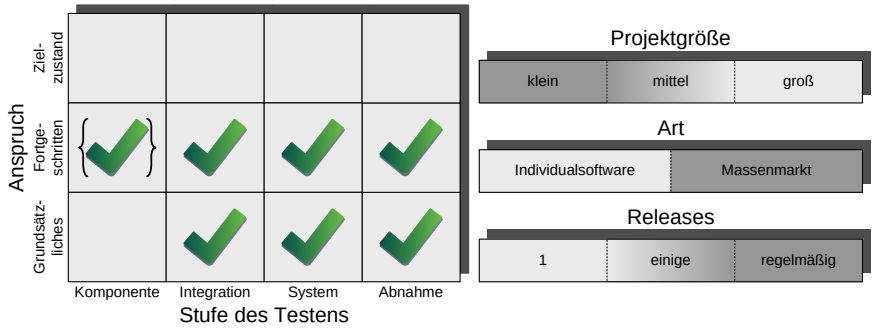

Abbildung 1. Beispielhafte Anwendung des Ordnungsrahmens

lungen beteiligt sein. Bei der Einordnung der Projektgröße sollte der Charakter des Projekts betrachtet werden, weniger die tatsächlichen Mitarbeiterzahlen.

Grob lassen sich zwei *Arten* von Softwareprodukten unterschieden. *Individualsoftware* wird gemäß Vertrag für einzelne Kunden implementiert, die oftmals in das Projekt eingebunden werden. Software für den *Massenmarkt* wird hingegen für einen größeren Kundenkreis entwickelt und über einen längeren Zeitraum angeboten. Folglich spielen hier Regressionstests eine große Rolle. Viele – aber nicht alle – Empfehlungen sind für beide Arten von Software nützlich.

Eine weitere Kategorisierung ist anhand der Zahl der *Releases* möglich. Es wird zwischen *einem*, *einigen* und *regelmäßigen* Releases unterschieden. Letzteres bedeutet, dass ein Produkt über Monate oder Jahre aktualisiert wird.

Die letzte Determinante teilt das Testen in *Stufen*. Der Literatur folgend [9] werden *Komponenten-*, *Integrations-*, *System-* und *Abnahmetest* unterschieden.

Wie in Abb. 1 dargestellt, bilden Anspruch und Stufe eine Matrix. Ein Häkchen zeigt dabei an, dass eine Handlungsempfehlung für die entsprechende Kombination Bedeutung hat. Wird es in Klammern dargestellt, ist die Bedeutung nebenrangig bzw. eine Implementierung für diese Kombination optional. Die anderen drei Determinanten sind als Balken dargestellt. Eine dunkelgraue Schraffur weist darauf hin, dass die Empfehlung unter dieser Bedingung gilt. Ein Farbverlauf deutet auf eine eingeschränkte Relevanz hin. In Abb. 1 ist die beispielhafte Verwendung des Ordnungsrahmens dargestellt:

- Die Handlungsempfehlung bezieht sich auf den Integrations-, System- und Abnahmetest. Sie hat sowohl einen grundsätzlichen als auch einen fortgeschrittenen Anspruch. Die Empfehlung sollte also umgesetzt werden, aber nicht notwendigerweise direkt im vollen Umfang.
- Das Häkchen beim Komponententest ist in Klammern dargestellt. Er profitiert zwar von einer Umsetzung, aber in geringerem Maße als andere Phasen.
- Die Handlungsempfehlung bezieht sich vor allem auf kleine und mittelgroße Projekte. Der Übergang der Färbung bedeutet, dass die Relevanz für solche Projekte noch gegeben ist, die kleinen Projekten ähnlich sind.
- Ferner bezieht sie sich ausschließlich auf Massenmarkt-Software. Theoretisch könnte sich hier für die individuelle Entwicklung ein Farbverlauf finden. Das hieße, sie würde ebenfalls profitieren, wenn auch in nicht so hohem Maße.

– Die entwickelte Software sollte zumindest in einigen Releases erscheinen, oder dauerhaft aktualisiert werden.

Für eine endgültige Umsetzungsentscheidung benötigen Unternehmen natürlich weitere Informationen. Der dargestellte Ordnungsrahmen kann aber helfen, einen Überblick zu bekommen und wichtige Voraussetzungen abzuschätzen.

5 Technische Empfehlungen

5.1 Moderne Entwicklungsumgebung

Die erste Handlungsempfehlung ist sicherlich wenig überraschend. Wir regen die Nutzung moderner Entwicklungsumgebungen an, insbesondere einer integrierten Entwicklungsumgebung (IDE) die anpassbar ist und sich über Plug-ins erweitern lässt. Die sich daraus ergebenden Vorteile sind sehr vielfältig. Die Nutzung bietet sich vor allem an, weil Software zur Entwicklung ohnehin genutzt wird und viele IDEs oder zumindest Plug-ins für sie *frei* erhältlich sind.

Der Einsatz einer IDE dient nicht nur dem Testen, kann aber signifikant zur Steigerung der Softwarequalität beitragen. Darüber hinaus unterstützt es Entwickler bei alltäglichen Aufgaben. Dies hat zur Konsequenz, dass sich weniger Programmierfehler durch Nachlässigkeiten einschleichen. Tester werden durch solche Fehler dann nicht aufgehalten, sondern können sich auf das Finden schwerwiegender Probleme konzentrieren. Diese Empfehlung dient folglich dem Testen, auch wenn sich nicht alle Vorteile *direkt* darauf beziehen.

Für Entwickler in kleinen Firmen ist es üblich, moderne IDEs zu nutzen. Unserer Erfahrung nach nutzen Unternehmen sie nicht notwendigerweise. Sobald die Wahl der Entwicklungssoftware nicht mehr in den Händen der Entwickler liegt, sondern es Richtlinien oder sogar Vorschriften gibt, kommt Software zum Einsatz, die hinter den aktuellen Möglichkeiten bleibt. Dies ist insbesondere dann zu beobachten, wenn PCs zentral installiert werden. Die Entwickler tauschen in einem solchen Fall nur selten die Programme aus und sind in vielen Fällen dazu noch nicht einmal in der Lage, etwa aufgrund Abhängigkeiten in der Systemlandschaft. Die von einigen Projektteilnehmern geschilderte Lage erinnerte uns an die 1980er Jahre. Entwickelt wurde ohne *Syntaxhervorhebung* (syntax highlighting) und die Entwicklungsumgebung bot kaum unterstützende Funktionalität. Die Dokumentation war nicht zugänglich. Vielmehr benutzen Entwickler direkt das Internet oder griffen auch für einfachste Fragen auf Bücher zurück. Am ungünstigsten stellte sich allerdings das Fehlern eines Debuggers dar. Debuggen erfolgte mithilfe von print()-Anweisungen, die zur mehr oder weniger zufälligen Ausgabe von Quelltextfragmenten führen.

Da wir sahen, wie viel produktiver *mit* einer modern IDE entwickelt werden und wie sehr dies die Softwarequalität erhöhen kann, empfehlen wir deren Nutzung dringend. Diese Empfehlung gilt für alle Unternehmen. Sie ist insbesondere für Komponententests hilfreich (vgl. Abb. 2). Diese Empfehlung gilt sogar dann, wenn die vorhandene Entwicklungsumgebung nicht erweitert oder durch Plug-ins ergänzt werden kann und ein Upgrade nötig wird. *Eclipse* etwa, die wohl

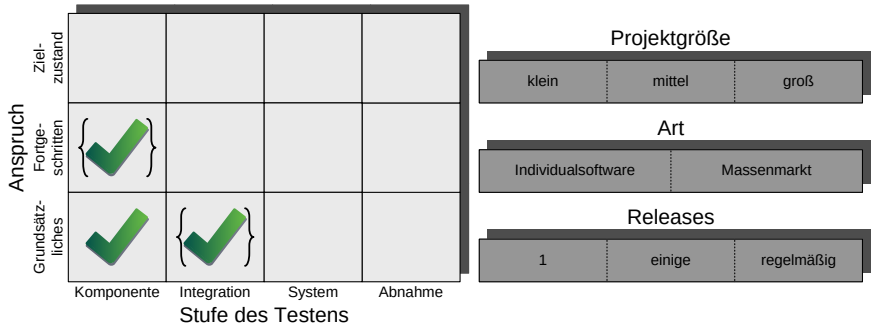

Abbildung 2. Einordnung der ersten Handlungsempfehlung

am weitesten verbreitete und gleichzeitig eine der leistungsstärksten IDEs, unterstützt Java, C/C++ und – über Erweiterungen – viele andere Sprachen. Zur umfangreichen Funktionalität kommt die Möglichkeit des Funktionsausbaus. Es steht eine vierstellige Zahl Plug-ins zur Verfügung (vgl. z. B. [10]).

Um die Leistungsfähigkeit der verwendeten Entwicklungsumgebung einschätzen zu können, empfiehlt sich ein Vergleich mit führenden IDEs. Die von einigen Projektteilnehmern eingesetzten Produkte blieben weit hinter dem zurück, was Eclipse oder Microsoft *Visual Studio* (um den Marktführer für die .NET-Entwicklung zu nennen) bereits vor Jahren leisteten.

Zur üblichen Funktionalität gehören die farbliche Hervorhebung des Quelltextes [11] und automatisierte Empfehlungen während der Eingabe (*code completion*). Des Weiteren lässt sich die Dokumentation direkt einblenden um etwa die Verwendung von veralteten Methoden (*deprecated*) zu verhindern. Auch lässt sich Code direkt auf Fehler überprüfen. Die *partielle Kompilierung* zeigt Hinweise, ohne den Compiler explizit aufzurufen. Software mit Syntaxfehlern kann so erst gar nicht kompiliert und vom Entwickler als *fertig* betrachtet werden.

Auch wenn die semantische Korrektheit unmöglich automatisch garantiert werden kann, lassen sich typische Fehler beispielsweise durch Warnungen vermeiden. Eclipse kann Java-Warnungen einblenden und betroffenen Code gelb unterschlängeln. Eine Variable etwa, die den Wert `null` annehmen kann aber ohne Prüfung verwendet wird, würde markiert werden. Somit kann Code, der `NullPointerException`s provoziert, korrigiert werden – ebenso wie viele weitere, potentielle Probleme. Entwickler gewöhnen sich daran schnell, selbst wenn Warnungen zunächst lästig erscheinen. Überflüssige Warnungen können zudem deaktiviert werden, z. B. durch Annotationen [12].

Als nächste Ausbaustufe bietet sich die Überprüfung so genannter *code rules* an. Kein IDE bietet diese Funktionalität direkt, sie kann aber über Plug-ins nachgerüstet werden. Während Warnungen vom Compiler stammen, verarbeiten diese Werkzeuge den Code selbst. Sie sind nicht nur sehr leistungsfähig, sondern auch anpassbar und somit für die Einführung von unternehmensweiten Programmierkonventionen geeignet. Solche Standard sind sehr empfehlenswert, auch wenn sich Entwickler nicht bevormundet fühlen sollten. Denn zahlreiche Probleme entstehen dadurch, dass mehrere Programmierer am selben Code arbeiten und ihn falsch interpretieren. Unternehmensweite Konventionen beugen

dem vor. Die entsprechenden Werkzeuge können zudem die Einhaltung der empfohlenen Programmierstandards der Hersteller (z. B. [13]) und der Literatur (z. B. [14]) sicherstellen.

In hohem Maße empfehlen wir auch die Nutzung der Debugging-Funktionen moderner IDEs. *Trace Debugger* können den Status einer Programms zu einem beliebigen Zeitpunkt visualisieren. Zeiger können verfolgt und Variablen unmittelbar verändert werden. Ferner ist die Ausführung Schritt-für-Schritt möglich. Darüber hinaus lassen sich Kontroll- und Datenflussgraphen anzeigen. Zusammen mit der Kenntnis moderner Debugging-Techniken [15] ist der Debugger einer modernen IDE ein sehr vielseitiges und leistungsfähiges Werkzeug.

Zusammenfassend raten wir dringend die Nutzung einer modernen IDE an, selbst wenn dies zur Bedingung hat, alte Bibliotheken, Vorgehensweisen, Paradigmen und sogar Programmiersprachen aufzugeben. Im Rahmen dieses Prozesses sollten bindende Vorgaben für die Formatierung von Quelltext sowie Benennungskonventionen festgelegt werden. Die optimale Nutzung einer IDE kann zudem gut durch ein Studium praxisbezogener Literatur wie [12,16,17,18] unterstützt werden. Hier existiert eine Vielzahl zum Teil sehr guter Titel.

5.2 Nutzung moderner Paradigmen und Frameworks

Softwaresysteme werden immer umfang- und funktionsreicher und damit komplexer. Als Möglichkeit zur Beherrschung bietet sich ein höherer Abstraktions- und Modularisierungsgrad an, der durch moderne Programmierparadigmen und Rahmenwerke (*Frameworks*) erreicht werden kann. Deren Einsatz wird allen Unternehmen empfohlen. Echte Vorteile ergeben sich nur für den Abnahmetests und für kleinste Projekte nicht, ansonsten ist durchgehend mit positiven Effekten zu rechnen (vgl. Abb. 3).

Klassisch wurde Software ohne Modularisierung entwickelt. Auch wenn dies heutzutage nicht mehr üblich ist, findet sich häufig eine starke und starre Verzahnung verschiedener Schichten. Sinnvoll hingegen wäre eine klare Trennung bei der eine Kommunikation nur über definierte Schnittstellen erfolgt. Diese kommt insbesondere Integrations- und Systemtests entgegen. Können sich Tester tatsächlich darauf verlassen, dass Module nur über ihre Schnittstellen angesprochen werden, reichen Tests mit Kenntnis des Modulcodes in der Komponententestphase. Schnittstellenlose Durchgriffe auf Daten hingegen erschweren das Testen wesentlich. Eine Abstraktion einzelner Funktionen ihm Rahmen der Modularisierung erleichtert auch die Rolleneinteilung: Spezialisierte Tester können gemäß ihrer fachlichen Stärken eingesetzt werden.

Um dienstorientierte Systeme zu testen, wie sie im Rahmen dienstorientierte Architekturen (*SOA*) zunehmend häufig anzutreffen sind, ist es nötig, Geschäfts- und Darstellungslogik zu trennen. Dienste sollten stets so spezifiziert werden, dass sie von beliebigen Aufrufern genutzt werden können. Idealerweise sollten sie darüber hinaus zustandslos sein oder vom Status abstrahieren. Für einen Daten liefernden Dienst sollte es unbedeutend sein, ob diese von derselben Anwendung für ihre Geschäftslogik, von der Darstellungslogik zwecks Ausgabe, oder von

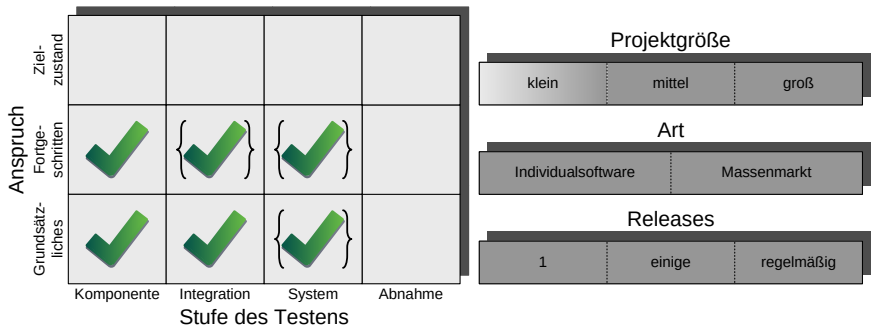

Abbildung 3. Einordnung der zweiten Handlungsempfehlung

einem Drittsystem abgefragt werden – sofern der Zugriff autorisiert ist. Nur dementsprechend definierte und nutzbare Dienste lassen sich effektiv Testen.

Eine fehlende Dienstorientierung ließ sich fast immer feststellen, wenn ältere Programmiersprachen zum Einsatz kamen, mit denen eine Kapselung schwierig ist. Aber auch bei modernen Programmiersprachen werden Frameworks benötigt. Denn ohne diese erfolgt häufig keine klare Schichtentrennung, selbst wenn die Sprache dies bieten würde. Die Nutzung moderner Programmiersprachen und -paradigmen wird dringend angeraten. Für Altsysteme (*legacy*) sollte geprüft werden, inwiefern eine Modularisierung und Dienstorientierung möglich ist. In der Regel ist es – zumindest über Erweiterungen – auch für Programmiersprachen wie COBOL oder Fortran möglich, Dienste anzubieten. In Frage käme auch ein *Wrapper*, der Dienste anbietet und das Altsystem maskiert.

Die große Zahl der angebotenen Literatur zur professionellen Programmierung verdeutlicht, dass auch erfahrene Programmierer die verwendete Sprache häufig weder effektiv noch effizient nutzen. Insbesondere scheint es weit verbreitet, alte Herangehensweise beizubehalten, selbst wenn diese durch *bessere* Möglichkeiten ersetzt wurden. Ein Auffrischen der Kenntnisse kann bei grundlegenden Techniken wie Entwurfsmustern [19,20,21] anfangen. Danach bietet sich Spezialliteratur zur effektiven Programmierung wie [12,16,22] an. Um gut testbaren Code zu schreiben ist es noch nicht einmal nötig, einen kompletten Wechsel auf einen testgetriebenen Entwicklungsprozess [23,24,25] zu vollziehen. Problematisch sind auch komplexe Programmiertechniken, die der Leistungssteigerung von Programmcode dienen sollen. Sicherlich ist es sinnvoll, diese im Rahmen eines Informatik-Studiums zu vermitteln. Auch steht außer Frage, dass es Bereiche gibt, in denen hardwarenah programmiert und in der jeder Kniff zur Leistungssteigerung realisiert werden sollte. Für allgemeine Projekte gilt dies nicht. Moderne Compiler sorgen für Optimierungen, die Bit-Shifts und ausgeklügelte arithmetische Operationen, die kaum lesbaren Code zur Folge haben, unnötig machen. Darüber hinaus werden geringe Effizienzgewinne schnell durch eine mangelnde Wart- und Testbarkeit wieder aufgehoben.

Zur Testbarkeit tragen auch moderne Frameworks bei. Ihre Stärke liegt insbesondere in der Abstraktion bzw. der vereinheitlichten Entwicklung. Darüber hinaus stellen sie häufig benötigte Funktionalitäten bereit. Deren Umsetzung ist bei verbreiteten Frameworks oftmals von hoher Qualität. Meistens ist eine

umfangreiche Dokumentation verfügbar. Laut Aussagen der Projektteilnehmer können Frameworks zur Entwicklung verteilter mehrschichtiger Applikationen wie *Enterprise Java Beans* (EJB) oder *Spring* das Testen wesentlich vereinfachen. Die Einarbeitung erfordert selbst bei leichtgewichtigen Frameworks ein Umdenken bei Entwicklern. Gerade dies kann aber dazu beitragen, unvorteilhafte Programmierpraktiken auf den Prüfstand zu stellen. Die Nutzung moderner Frameworks wird daher sehr empfohlen. Die Einführung sollte konsequent vollzogen und bei Notwendigkeit von Schulungen begleitet werden. Eine undisziplinierte, halbherzige oder gar falsche Verwendung der durch Frameworks zu Softwaresystemen ergänzten Komponenten kann zu massiven Problemen führen.

5.3 Enge Zusammenarbeit der Entwickler

Aufgrund der hohen Komplexität heutiger Softwareprojekte ist es nicht mehr zeitgemäß, Entwickler unabhängig voneinander Module implementieren zu lassen und diese dann in einer Integrationsphase zusammenzuführen. Vielmehr hat sich im Projekt gezeigt, dass eine enge Zusammenarbeit sehr hilfreich sein kann. Diese kann sich als Paarprogrammierung im Rahmen der agilen Entwicklung [26] darstellen, muss aber nicht unbedingt so weitgehend sein. Generell festhalten lässt sich, dass die niedrigere Effizienz durch eine höhere Personalbindung und mehr Austausch unter den Entwicklern durch eine höhere Code-Qualität und höhere Produktivität überkompensiert werden. Die Empfehlung, auf eine enge Zusammenarbeit hinzuwirken, gilt für alle Unternehmen. Vorteile zeigen sich vor allem für den Komponenten- und Integrationstests. Ein Start ist sehr einfach möglich, wobei sich umfangreiche Ausbaumöglichkeiten ergeben (vgl. Abb. 4).

Die enge Zusammenarbeit verfolgt eine Reihe von Zielen. Durch die gemeinsame Arbeit fallen Fehler schneller auf; besonders komplizierte Module können sogar in Paarprogrammierung implementiert werden. Die Einhaltung von Standards fällt einfacher (siehe auch Abschnitt 5.1). Wissen wird von Entwickler zu Entwickler weitergegeben und ergänzt die Dokumentation; idealerweise findet auch ein Erfahrungsaustausch statt.

Als besonders erfolgversprechend wurde uns von der gegenseitigen Begutachtung der Entwickler berichtet. Module werden nach wie vor in Einzelarbeit erstellt, vor Abschluss aber durch einen oder zwei weitere Mitarbeiter begutachtet. Es bietet sich dabei an, gefundene Fehler direkt zu beheben. Diese Arbeitsweise unter "Vier-Augen" ist nicht nur effektiv, sondern fördert auch den Austausch unter den Entwicklern. Selbst wenn Mitarbeiter ausfallen, findet sich für jedes Modul jemand, der es näher kennt. Ähnlich wie für die bisher vorgestellten Erfahrungen dient die gemeinsame Begutachtung nicht direkt dem Testen. Sie sorgt aber dafür, dass sich Tester auf das Finden schwerwiegender Fehler konzentrieren können, weil viele typische Probleme vor Beginn des Tests gelöst worden sind. Wichtig für die Begutachtung ist, sie als unterstützende und von Entwicklern begrüßte Maßnahme zu implementieren. Führen gefundene Fehler zu Sanktionen oder wird sie zur Leistungsmessung missbraucht, wird sie demotivierend wirken. Die Begutachtung von Quelltext wird auch als *Review* bezeichnet und ist seit

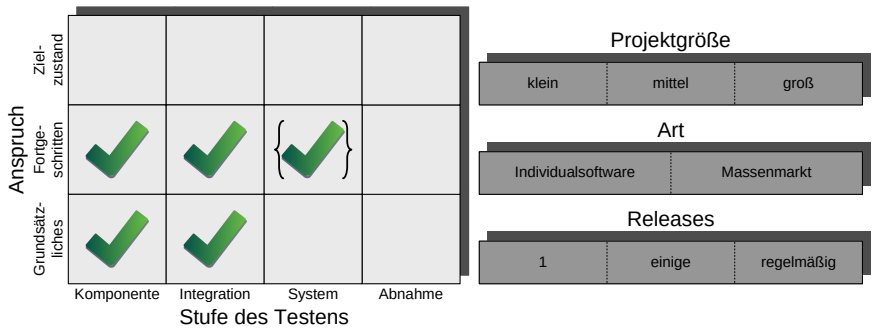

Abbildung 4. Einordnung der dritten Handlungsempfehlung

längerem bekannt [27]. Weitere Informationen finden sich im IEEE Standard für "software reviews" [28] sowie in praktischer Literatur wie [29,30,31].

Wichtig für die erfolgreiches Zusammenarbeit ist, dass die Verantwortlichkeiten klar geregelt sind. Die Zuordnung von Entwicklern und Begutachtern etwa muss mit Bedacht gewählt werden, denn es lassen sich sowohl Argumente für häufige Wechsel der Zuordnung wie für die Etablierung eingespielter Teams finden. Auch muss in diesem Zusammenhang festgelegt werden, welche Module überhaupt begutachtet werden.

5.4 Abstimmung von Testsystemen und Produktivsystemen

Die Nutzung moderner Programmiersprachen und Paradigmen für die Entwicklung komplexer verteilter Applikationen bietet nicht nur Vorteile. Werden Anwendungen auf Arbeitsplatzrechnern entwickelt aber auf Server oder Großrechner gespielt, kann es zu Kompatibilitäts- oder Skalierungsproblemen kommen.

Ursprünglich wurden Programme nur für ein Zielsystem entwickelt. Um auf anderen Plattformen zu laufen, war eine Anpassung nötig. Heutzutage sind Entwicklungs- und Zielplattform für gewöhnlich unterschiedlich. Auch läuft Software nicht mehr nativ auf der Hardware. Hypervisor, Applikationsserver und weitere Komponenten bilden zusätzliche Schichten, was weitreichende Vorteile hat. Als Folge sind Zielsysteme aber häufig leistungsfähiger und auch anders mit Software ausgestattet. In einfachen Fällen gilt dies nur für das Betriebssystem. Weitere Komponenten wie Bibliotheken, Datenbankmanagementsysteme (DBMS) oder auch Applikationsserver unterscheiden sich aber ebenfalls häufig.

Typische Kompatibilitätsprobleme lassen sich gut mit einem Beispiel motivieren. Java EE-Applikationen werden von Applikationsservern bereitgestellt. Auf Großrechnern und Servern kommen dabei Systeme wie IBM *WebSphere* zum Einsatz. Da sie auf "normalen" PCs erst gar nicht laufen, werden dort leichtgewichtige Alternativen wie Apache *Tomcat* verwendet. Auch wenn dies keine Probleme bereiten sollte, zeigt die praktische Erfahrung, dass offenbar durch unterschiedliche Interpretation von Spezifikationen, abweichende Versionen, konfliktäre Bibliotheken und ähnliches Inkompatibilitäten an der Tagesordnung sind.

Wir empfehlen daher die Abstimmung von Entwicklung- und Testsystemen mit der produktiven Umgebung. *Abstimmung* meint ein durchdachtes Vorgehen

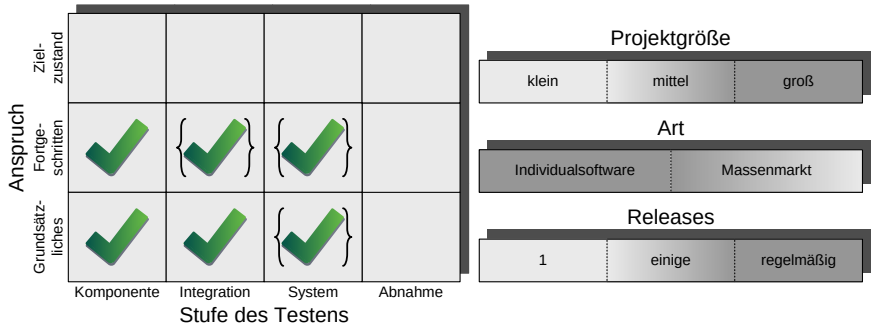

Abbildung 5. Einordnung der vierten Handlungsempfehlung

bei der Auswahl von Hard- und Software. Sie muss ökonomisch vertretbar bleiben – die Anschaffung eines Großrechners zum Testen wird in der Regel unmöglich sein, selbst wenn die Zielplattform ein solcher ist. In der Regel finden sich aber sinnvolle Lösungen. Die Abstimmung der Systeme wird ab mittelgroßen Projekten empfohlen, die in einige Releases münden. Vorteile zeigen sich vor allem in der individuellen Softwareentwicklung und den frühen Testphasen. (vgl. Abb. 5).

Aufgrund der Erfahrungen im Projekt halten wir auch höheren Aufwand für die Abstimmung für gerechtfertigt. Während sich Test- und Produktivsystem mit fortschreiten der Tests ohnehin annähern sollten, bietet es sich an, Kompatibilitätsprobleme so früh wie möglich zu lösen. Im obigen Beispiel kann eine "abgespeckte" WebSphere-Version für PCs genutzt werden. Tomcat als Testsystem sollte dann verwendet werden, wenn Tomcat auch auf dem Server läuft.

Anstatt ein DBMS auf dem Entwicklungssystem zu betreiben, kann fast immer das auf dem Server installierte genutzt werden. Zum Schutz der Daten wird eine neue Datenbank erstellt und vom restlichen Datenbestand entkoppelt. Heutige Server und vor allem Mainframes bieten zudem ausgeklügelte Virtualisierungsmöglichkeiten. Den produktiven Systemen identische Instanzen lassen sich einfach starten, um dann unter realistischen Bedingungen zu testen. Zu achten ist dabei allerdings auf die Ressourcennutzung, so dass Probleme während des Tests keine negativen Auswirkungen auf den Produktivbetrieb haben.

Auch umfangreiches Testen deckt häufig nicht alle Probleme auf. Dies gilt insbesondere für schlecht reproduzierbare Fehler, die in der Speichernutzung oder der parallelen Ausführung begründet liegen. Sie müssen sich auf Testsystemen nicht zeigen. Auch lässt eine akzeptable Leistung auf dem Testsystem keine Rückschlüsse auf die Leistungsfähigkeit auf der Zielplattform zu. Schließlich müssen komplexe Abhängigkeiten und wachsende Datenbestände ins Kalkül gezogen werden. Tests sollten aus diesen Gründen unbedingt auf der Zielplattform erfolgen. Defekte in parallelen Algorithmen – etwa drohende Wettlaufsituation (*race conditions*) – zeigen sich möglicherweise erst auf der leistungsfähigen Zielhardware oder aufgrund eines aggressiveren "Timings" der Zielsoftware.

Trotz aller Werbung für realistische Tests raten wir dringend, *nicht* ohne Vorbereitung auf produktiven Systemen zu testen. Produktive Daten dürfen nicht verändert werden. Die Gefährdung produktiver Daten oder ein "Ausbremsen" dieser Systeme würde alle Vorteile des realistischen Testens zunichte machen.

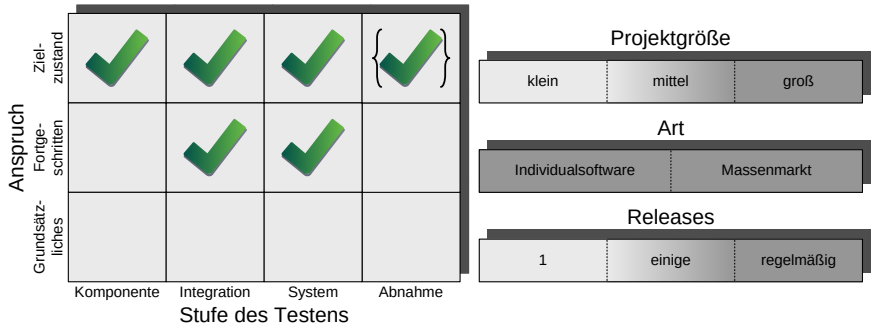

Abbildung 6. Einordnung der fünften Handlungsempfehlung

5.5 Integration der Werkzeuge

Der Einsatz von Testwerkzeugen ist mittlerweile üblich. Während die Nutzung von Unternehmen zu Unternehmen und in den unterschiedlichen Testphasen schwankt, konnten wir generell beobachten, dass Werkzeuge kaum integriert werden. Gerade das ist aber empfehlenswert.

Die meisten Testwerkzeuge sind eigenständige Applikationen. Nur einige integrierten sich als Plug-ins in IDEs. Allgemeine Formate oder Schnittstellen existieren kaum. Nur größere Anwendungen wie etwa die Produkte von *IBM Rational* bieten eine Austauschmöglichkeit von Daten. Diese bleibt meistens auf Produkte derselben Reihe beschränkt. Die Integration ihrer Werkzeuge wurde von vielen Projektteilnehmern angestrebt. Wie empfehlen sie ab mittelgroßen Projekten mit einigen Releases. Der Aufwand für die Integration is beachtlich. Auch wenn letztlich Vorteile für alle Phasen des Testens realisiert werden können, zeigen sich diese zunächst für Integrations- und Systemtests (vgl. Abb. 6).

Die Integration ist auf verschiedenen Ebenen erstrebenswert. Im ersten Schritt bietet es sich an, Systeme für die Dokumentation mit denen zur Testausführung zu verbinden. Undokumentierte Tests sind wenig wert und eine automatische Synchronisation der Ergebnisse mit der Dokumentation entlastet Tester wesentlich. Eine gut strukturierte Dokumentation [32] kann so einfacher erreicht werden. Die wachsende Datenbank kann anschließend für Auswertungen genutzt werden. Werte wie Laufzeiten oder Erfolgsraten sind für Testmanager sehr nützlich. Für regelmäßig aktualisierte Software kann dann als nächstes der *Bug Tracker* eingebunden werden, so dass gemeldete Fehler mit bekannten Defekten und vorhandenen Testfällen abgeglichen werden können.

Die Verbindung von Systemen zur Testausführung unterstützt Regressionstests. Testfälle aus früheren Stufen lassen sich einfach wiederholen. Die Anbindung des Testfallverwaltungsystems macht wiederholte Tests, die manuell nicht denkbar wären, wirtschaftlich. Natürlich ist eine solche Anbindung kompliziert und die Schritte bis zu einer automatisieren Regression mühsam. Nichts desto trotz halten wir die Regression für erstrebenswert. Durch ein Vorgehen in kleinen Schritten zur Verbesserung ist die Integration der Werkzeuge möglich, ohne das dies hohe Kosten oder Arbeitsaufwände verursacht.

Leider existieren keine Lösungen für eine umfangreiche Werkzeugintegration. Entsprechende Zusätze müssen selbst entwickelt werden, wobei Neuanschaffungen hinsichtlich ihrer Integrationsfähigkeit geprüft werden sollten. Schon kleine Werkzeuge, etwa für die Umwandlung von Daten, können viel manuelle Arbeit ersparen. Ein Beispiel ist ein Programm, das Ergebnisse aus der Testdokumentation extrahiert, aggregiert und Statistiken daraus erstellt. Es lässt sich schnell implementieren und kontinuierlich ausbauen. Dementsprechend sind vor allem Werkzeuge mit verfügbaren Quelltexten für die Integration geeignet. Sie lassen sich schnell den eigenen Wünschen anpassen und mit Schnittstellen versehen.

Die volle Integration bietet zahlreiche Vorteile. So könnte ein *Test Controlling* eingerichtet werden, das einen Überblick über den Testprozess verschafft und Kennzahlen berechnet [6]. Als Vision ergibt sich die Integration eines Systems, das Testfallverwaltung, Entwicklung- und Projektplanung, Testterminierung, Mitarbeiterzuweisung, Zeiterfassung, Aufgabenverwaltung und Controlling umfasst, sowie eventuell sogar ein *Management Cockpit*.

6 Fazit und zukünftige Arbeit

Wir haben in diesem Artikel die Ergebnisse eines Projektes vorgestellt, in dem wir zusammen mit regionalen Unternehmen Handlungsempfehlungen für die Softwareentwicklung und insbesondere das Testen erarbeitet haben. Dazu haben wir den Projekthintergrund beschrieben, einen Ordnungsrahmen zur Kategorisierung vorgestellt und fünf Empfehlungen mit technischem Fokus beschrieben.

Moderne IDEs unterstützen die Entwicklung wesentlich. Sie ermöglichen ein Testen, dass schwerwiegende Fehler sucht, anstatt sich mit Nachlässigkeiten im Code beschäftigen zu müssen. Die Nutzung moderner Paradigmen und Frameworks trägt dazu bei, strukturierter zu entwickeln und die Komplexität von Software zu beherrschen. Arbeiten Entwickler eng zusammen fördert dies den Wissensaustausch und dient ebenso dem Testen. Die Abstimmung von Test- und Produktivsystemen beugt Problemen durch Inkompatibilitäten vor. Eine Integration der Testwerkzeuge ist sehr aufwändig, kann aber die Leistung und Effizienz von Tests stark erhöhen.

Das diesem Artikel zugrunde liegende Projekt ist noch nicht beendet. Vielmehr sollen die Ergebnisse in Zukunft mit den Teilnehmern gemeinsam betrachtet und diskutiert werden. Idealerweise könnte eine quantitative Studie angeschlossen werden, die der Validation der Handlungsempfehlungen dient.

Literatur

1. Naur, P., Randell, B.: Software Engineering: Report of a conf. spon. by the NATO Science Committee, Garmisch, Germany. Scientific Affairs Division, NATO (1969)
2. Dijkstra, E.: The humble programmer. Comm. of the ACM **15** (1972) 859–866
3. NASA: Mars climate orbiter mishap investigation board phase I report (1999)
4. Kopec, D., Tamang, S.: Failures in complex systems: case studies, causes, and possible remedies. SIGCSE Bulletin **39**(2) (2007) 180–184

5. Charette, R.N.: Why software fails. IEEE Spectrum **42**(9) (2005) 42–49
6. Majchrzak, T.A.: Best practices for the organizational implementation of software testing. In: Proc. of the 43th Annual Hawaii International Conf. on System Sciences, Computer Society Press (2010) To appear.
7. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. MIS Quarterly **28**(1) (2004)
8. Orlikowski, W.J., Iacono, C.S.: Research commentary: Desperately seeking the "IT" in IT research–a call to theorizing the IT artifact. Info. Sys. Research **12**(2) (2001) 121–134
9. Watkins, J.: Testing IT: an off-the-shelf software testing process. Cambridge University Press, New York, NY, USA (2001)
10. o. A.: Eclipse plugin central. Online: http://www.eclipseplugincentral.com/.
11. Cowlishaw, M.F.: Lexx—a programmable structured editor. IBM Journal of Research and Development **31**(1) (1987) 73–80
12. Bloch, J.: Effective Java. 2nd edn. Prentice Hall, Upper Saddle River (2008)
13. Sun Microsystems, Inc.: Code Conventions for the Java Programming Language. Online: http://java.sun.com/docs/codeconv/.
14. Sutter, H., Alexandrescu, A.: C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series). Addison-Wesley (2004)
15. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann, San Francisco (2009)
16. Meyers, S.: Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition). Addison-Wesley Professional (2005)
17. Conway, D.: Perl – Best Practices: Die deutsche Ausgabe. O'Reilly, Köln (2006)
18. Ford, S.: Effizienter Programmieren mit Visual Studio: 250 Tipps, um Ihre Produktivität zu verbessern. Microsoft Press, Unterschleißheim (2008)
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addision-Wesley, München (1995)
20. Bishop, J.: C# 3.0 Entwurfsmuster. O'Reilly, Köln (2008)
21. Freeman, E., Freeman, E., Sierra, K., Bates, B.: Entwurfsmuster von Kopf bis Fuß. O'Reilly, Köln (2006)
22. Nagel, C., Evjen, B., Glynn, J., Skinner, M., Watson, K.: Professional C# 2008. Wrox Press Ltd., Birmingham (2008)
23. Beck, K.: Test-Driven Development by Example. Addison-Wesley (2002)
24. Newkirk, J.W., Vorontsov, A.A.: Test-Driven Development in Microsoft .Net. Microsoft Press, Redmond, WA, USA (2004)
25. Westphal, F.: Testgetriebene Entwicklung mit JUnit & FIT. Dpunkt Verlag, Heidelberg (2005)
26. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (1999)
27. Fagan, M.E.: Design and code inspections to reduce errors in program development. IBM Systems Journal **15**(3) (1976) 182–211
28. IEEE: IEEE Std 1028: IEEE standard for software reviews, New York (1998)
29. Wong, Y.K.: Modern Software Review. IRM Press (2006)
30. Wiegers, K.E.: Peer reviews in software. Addison-Wesley, Boston (2002)
31. Thayer, R.H.: Software Reviews, Inspections and Audits: A Standards-based Guide. IEEE Computer Society, Washington, DC, USA (2009)
32. IEEE: IEEE Std 829-2008: IEEE standard for software and system test documentation, New York (2008)