rapid application testing

von manfred rätzmann

mehr zum thema:
www.stickyminds.com
www.satisfice.com

„RAPID APPLICATION TESTING"

Nicht nur die Anforderungen der Kunden, auch die technischen Möglichkeiten, das Aussehen der Programme und die Art, wie Mensch und Software zueinander finden, sind einem ständigen Wandel unterzogen. Das Testen von Software kann davon nicht unberührt bleiben. Der Artikel beschreibt Möglichkeiten, auch im Testprozess auf sich ständig ändernde Anforderungen und die in der Praxis übliche Zeit- und Ressourcenknappheit zu reagieren.

Agile Prozesse sind gefragt, also Prozesse, die auf Änderungen schnell reagieren und die dazu mehr auf Kommunikation, Selbstorganisation und Auslieferung real existierender Software setzen als auf Planung, Überwachung und Dokumentation des Herstellungsprozesses (vgl. [Fow01]). Alistair Cockburn beschreibt Softwareentwicklung als Spiel in einer Gruppe, das zielorientiert, zeitlich und inhaltlich abgegrenzt und kooperativ abläuft (vgl. [Coc02]).

Die Regeln des Spiels zwischen Software-Entwickler und -Tester sind einfach: Die Tester versuchen, so schnell wie möglich die wichtigsten Fehler zu finden, das heißt nachzuweisen, dass das Programm unter bestimmten Voraussetzungen versagt. Deutlichstes Zeichen für ein Versagen ist der gemeine Absturz: eine allgemeine Schutzverletzung oder ein Einfrieren („Nichts geht mehr") des Programms. In diesem Sinne kann man *Rapid Application Testing* durchaus als die Kunst bezeichnen, ein Programm zum Absturz zu bringen.

Das Versagen eines Programms zeigt sich nicht immer so krass wie bei einem Absturz. Subtilere Versager sind Ablauf- oder Ausgabefehler, Rechenfehler, Benutzbarkeitsprobleme bis hin zu übergroßer Sorglosigkeit im Umgang mit vertraulichen Daten. Da nicht alle Fehler gleich wichtig sind, sollte eine Rangfolge der Fehlergewichtung zu den bekannt gegebenen Spielregeln zählen. Hier ein Vorschlag, wie sich die Fehler in der Rangfolge ihrer Wichtigkeit – ausgerichtet an der Benutzbarkeit der Software – staffeln lassen:

- Fehler, die eine vorgesehene Benutzung unmöglich machen,
- Fehler, die dazu führen, dass eine vorgesehene Benutzung nur auf Umwegen möglich ist,
- Fehler, die einen lästigen und unnötigen Mehraufwand bei der Benutzung des Programms verursachen und
- Fehler, die das Erscheinungsbild der Software beeinträchtigen.

Benutzung bedeutet hier immer den gesamten Umgang mit dem Programm, also auch Installieren und Administrieren.

Letztendlich wird diese Rangfolge der Wichtigkeit aber von den Zielen des Projekts und den an diesen Zielen interessierten Personen (*Stakeholder*) festgelegt. So können zum Beispiel in einer Konkurrenzsituation, in der es entscheidend auf das Aussehen der Software ankommt – ob diese also jugendlich oder seriös, frech oder abgeklärt, unterhaltsam oder kompetent wirkt – Unstimmigkeiten im Erscheinungsbild viel wichtiger sein als fehlende Funktionalität.

Das Rapide am „Rapid Application Testing"

Die Grundlage der Testplanung und aller Testmetriken war lange Zeit die funktionale Überdeckung und die Anweisungsüberdeckung. Nachdem man eingesehen hatte, dass man nicht alles testen kann, wurden Methoden zur Aufdeckung von Redundanzen innerhalb der Testfälle und der Testdaten entwickelt und außerdem Methoden zur Bewertung der Risiken, die man eingeht, wenn bestimmte Teile der Software weniger intensiv getestet werden als andere.

Ziel des *Rapid Application Testing* ist es, die schwerwiegendsten Fehler möglichst schnell zu finden. Deshalb heißt das Vorgehen auch „Rapid Application Testing" und nicht etwa „Total Application Testing". Alle Fehler zu finden, kann ebenfalls ein interessantes Spiel sein – die Frage ist allerdings meistens, wer die Kosten dafür übernimmt.

James Bach, Inhaber der Firma Satisfice (vgl. [Bac]), propagiert „Rapid Testing" für externe Tester. Seine Beschreibung des *Rapid Testing* beginnt mit den unten aufgeführten Erkenntnissen zu Mission, Können, Risiko und Erfahrung beim Testen von Software. *Rapid Application Testing* übernimmt und erweitert diesen Ansatz für die Testverfahren des Entwicklungsteams selbst. Es unterscheidet sich vom herkömmlichen formalen Testansatz hauptsächlich in den folgenden Punkten:

der autor

Manfred Rätzmann (E-Mail: m.raetzmann@raetzmann-gmbh.de) ist Softwareentwickler und Berater mit den Schwerpunkten Testtechniken, Anwendungsdesign, Komponentenbau und Framework-Entwicklung. Er ist Autor des Buches „Software Testing" ([Rät02]).

Mission

Rapid Application Testing beginnt nicht mit einer Aufgabe wie „Erstelle die Testfälle", sondern mit einer Mission, zum Beispiel: „Finde die wichtigsten Fehler schnell." Welche Aufgaben zur Erfüllung der Mission zu erledigen sind, hängt vom Inhalt der Mission ab. Keine der im formalen Testansatz als notwendig erachteten Tätigkeiten gilt als unverzichtbar, alle Tätigkeiten müssen ihre Nützlichkeit in Bezug auf die Mission nachweisen.

Können

Der herkömmliche formale Testansatz bewertet die Bedeutung des Könnens, des Wissens und der Fertigkeiten der Testerin oder des Testers zu niedrig. *Rapid Application Testing* erfordert Wissen um den Testgegenstand und die möglichen Probleme beim Einsatz ebenso wie die Fähigkeit, logische Schlussfolgerungen zu ziehen und aussagekräftige Versuche zu entwickeln.

Risiko

Der herkömmliche Testansatz strebt eine möglichst hohe funktionale und strukturelle Überdeckung an. *Rapid Application Testing* zielt auf die wichtigsten Probleme zuerst. Dazu wird zunächst ein Verständnis dessen, was passieren kann, und welche Auswirkungen es hat, wenn es passiert, erarbeitet. Anschließend werden die mögli-

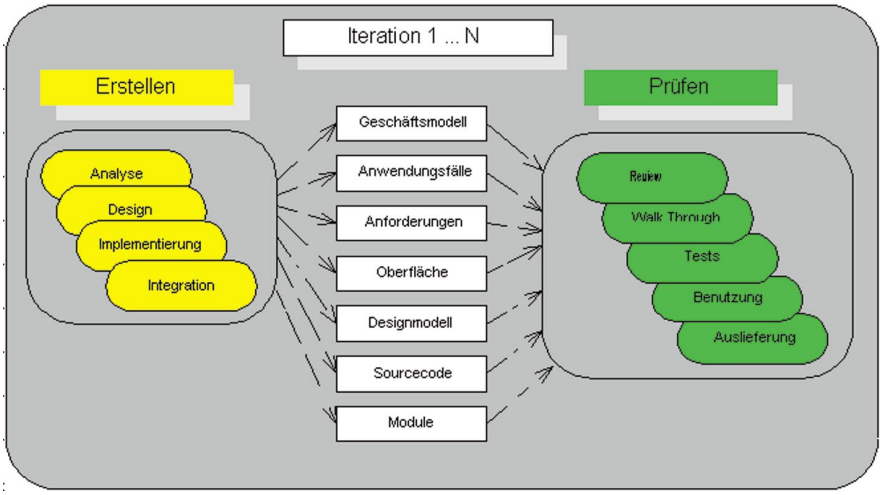

Abb. 1: Produkte werden mit Prüfmethoden verknüpft

cherweise problematischen Punkte in der Reihenfolge ihrer Wichtigkeit abgeprüft.

Erfahrung

Um auch beim Testen nicht in einer Analyse-Paralyse zu verharren, sollen die Tester ihren Erfahrungen vertrauen. Während beim herkömmlichen formalen Testansatz die Erfahrungen der Tester meist unbewusst und damit auch ungeprüft einfließen, sollen die Tester beim *Rapid-Application-Testing*-Ansatz ihre Erfahrungen sammeln, festhalten und durch bewussten Einsatz überprüfen. Dazu dienen ein Gespür für Fehler und Schwachstellen (*Error-Guessing*) genauso wie Aussagen zu Fehlerhäufungen und Fehlerfindungsraten, die aus *Bug-Tracking*-Systemen gewonnen werden.

Flexibilität

Rapid Application Testing ist nicht an vorgegebene Testpläne und Testprozeduren gebunden. Die Ergebnisse der durchgeführten Tests haben einen weit reichenden Einfluss auf die noch auszuführenden Tests. So dringt *Rapid Application Testing* schneller zu den Kernproblemen der Software vor als vorgeschriebenes (*scripted*) Testen.

Integration

Rapid Application Testing ist integriertes Testen, d. h. es ist in den Softwareentwicklungsprozess integriert und nicht ein zweiter, nebenher laufender Software-Validierungsprozess. Alle konstruktiven Maßnahmen sollen einen Doppelnutzen hervorbringen. Doppelnutzen bedeutet eine Tätigkeit so auszuführen, dass neben dem hauptsächlichen ersten ein zweiter Nutzen entsteht – in diesem Fall der Nutzen der permanenten Qualitätssicherung und Qualitätskontrolle.

Produktorientierung

Das klassische Testplanungsmodell orientiert sich am Herstellungsprozess der Software und leitet daraus die Testphasen ab. *Rapid Application Testing* orientiert sich dagegen an den hergestellten Produkten oder Zwischenprodukten.

Produkte des Softwareentwicklungsprozesses entstehen, wenn sie gebraucht werden. Wenn sie entstanden sind, werden sie sofort geprüft. Jedes Produkt wird dabei einer expliziten oder einer impliziten Prüfung unterworfen: explizit durch Prüfverfahren, die auf die spezielle Art von Produkt zugeschnitten sind, und implizit durch kritische Nutzung der entstehenden Produkte im weiteren Entwicklungsprozess. **Abbildung 1** stellt die Verknüpfung von Produkten des Entwicklungsprozesses mit spezifischen Prüfverfahren dar.

Teamwork

Rapid Application Testing erfolgt im Wesentlichen „zwischendurch". Erstellungs- und Testphasen wechseln sich ab, gesteuert durch die Verfügbarkeit einzelner Produkte oder Zwischenprodukte im Entwicklungsprozess. „Tester sein" wird als Rolle betrachtet, die von einzelnen Personen je nach Bedarf eingenommen wird. Alle diese Personen sind darauf angewiesen, schnell zum Thema zu kommen. Die formalen Anforderungen an Testprozeduren, die bei gut ausgestatteten und zeitlich entspannten Projekten durchaus sinnvoll sein können, sind hier eher hinderlich.

Pragmatismus

Rapid Application Testing ist *Gray-Box Testing*. Dieses besteht aus Methoden und Werkzeugen, die Kenntnisse der internen Applikationsstrukturen benutzen, um *Black-Box*-Testfälle zu planen und durchzuführen (**siehe Abb. 2**). Formale Differenzierungen zwischen Funktions- und Strukturtests sind zweitrangig. Alles, was hilft die wichtigsten Fehler schnell zu finden, ist erlaubt. Auch Test-Automatisierung wird nicht als Ziel, sondern als Mittel betrachtet, das seine Nützlichkeit nachweisen muss. Das Ziel ist funktionierende Software.

Kundenbeteiligung

Rapid Application Testing ist verteiltes Testen. Vertreter des Kunden, des Auftraggebers oder Anwendervertreter werden an Test- und Prüfmaßnahmen beteiligt. Sie übernehmen insbesondere fachliche Tests und Benutzbarkeitsprüfungen. Kurze Iterationen und ein permanenter Dialog zwischen externen Testern und dem Entwicklungsteam integrieren auch externe Tester in den Entwicklungsprozess. **Abbildung 3** zeigt eine sinnvolle Verteilung von Aufgaben oder Verantwortlichkeiten beim Testen.

Teststrategien

Testen und Verbessern

Die am weitesten verbreitete Strategie beim Testen innerhalb des Entwicklungsprozesses ist wohl Testen und gleichzeitiges Verbessern von Software (*Test & Tune*). Testen und Beheben der gefundenen Fehler sind in dieser Strategie nicht voneinander getrennt, vielmehr wird Testen als notwendige Vorarbeit betrachtet, um die Dinge im Programm nachbessern zu können, die noch nicht richtig laufen. Testen und Verbessern ist in dieser Form eine reine Entwicklerstrategie. Vor der Freigabe eines Programmteils probieren der Entwickler oder die Entwicklerin aus, ob ▸

Abb. 2: Gray Box Testing

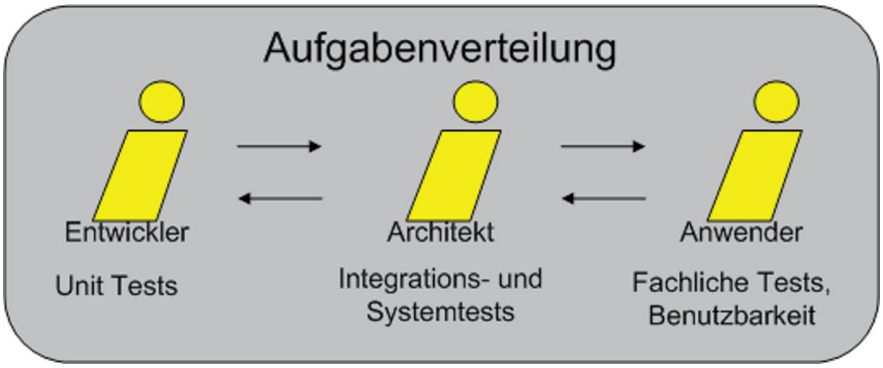

Abb. 3: Aufgabenverteilung beim Testen

alles glatt läuft, und korrigieren dabei sofort die gefundenen Fehler. In dieser Phase beschäftigen sie sich auch verstärkt mit eventuell problematischen Situationen, die während der Entwicklung zurückgestellt wurden, um zunächst den generellen Ablauf sicherzustellen.

Das vom Illinois Institute of Technologie entwickelte *Test Maturity Model (TMM)* sieht diese Strategie sehr zu Unrecht als „TMM Level 1" an – als Ausgangszustand, den es zu überwinden gilt (siehe [IIT]). Die *Test & Tune*-Strategie ist aber unverzichtbar, vor allem wenn man bedenkt, dass ca. 75% des gesamten Codes sich damit beschäftigen, auf Ausnahme- und Fehlersituationen zu reagieren, und nur die übrigen 25% die eigentliche Verarbeitung enthalten (manche Schätzungen geben sogar nur 10% Verarbeitungscode an). Gerade diese Behandlung von Ausnahme- und Fehlersituationen lässt sich am besten mit Versuch, Irrtum, Korrektur und neuem Versuch entwickeln.

Die Entwickler sollten gezielt in Testpraktiken geschult werden um ihre *Test & Tune*-Fähigkeiten zu verbessern. Bei kritischen Abläufen muss natürlich mehr getan werden als *Test & Tune*.

Testen durch Benutzen

Eine der ergiebigsten Teststrategien ist „Testen durch Benutzen". Diese Strategie ist insbesondere dazu geeignet, inhaltliche Aspekte und „weiche" Qualitätsmerkmale abzuprüfen.

Wenn die Korrektheit und Robustheit der Software einigermaßen sichergestellt sind, muss die Software benutzt werden. Nur so lassen sich die wichtigsten Fragen, die sich in einem adaptiven Entwicklungsprozess stellen, beantworten:

- Ist die Software wirklich nützlich? Wie könnte die Nützlichkeit verbessert werden? Welche Funktionalität fehlt, um die Sache richtig rund zu machen?
- Werden alle oder zumindest die wichtigsten Arbeitsabläufe von der Software unterstützt? Fördert das Programm den Fluss der Arbeit oder behindert es ihn?
- Erkennen die Benutzer der Software sich und ihr Arbeitsgebiet wieder? Stimmen die Fachausdrücke, stimmen die Abstraktionen?
- Macht es Spaß, mit der Software zu arbeiten, oder ist es eher „ätzend" und lästig? Fühlen die Benutzer sich überfordert oder langweilen sie sich?

Projektintern können Sie auf Modul- oder Komponentenebene die „Testen durch Benutzen"-Strategie ebenfalls anwenden. Durch Modularisierung, Schichten- oder Komponentenarchitektur wird die Situation des Benutzens bereits in den Entwicklungsprozess eingebaut und entsteht nicht erst an dessen Ende, wenn alles fertig ist. Die Module, Komponenten, Klassen oder Funktionen des einen Entwicklers sollen vom anderen Entwickler genutzt werden, am besten ohne Blick in den Quellcode.

Es wird zwar manchmal gefordert, dass der Quellcode allen Entwicklern „gehören" soll, sprich, dass sich alle Entwickler im Quellcode ähnlich gut auskennen sollen. Im Hinblick auf das „Testen durch Benutzen" ist es aber besser, wenn sich die Kenntnis auf die Schnittstellen, auf die Problembeschreibung und auf den allgemeinen Lösungsansatz beschränkt.

Testen durch Dokumentieren

Auch das Verfassen der Benutzerdokumentation ist eine Möglichkeit, „Testen durch Benutzen" einzusetzen. Die Benutzerdokumentation wird dann nicht vom Entwickler erstellt – zumindest nicht vom Entwickler des beschriebenen Programmteils – sondern von einer Person, die das zu beschreibende Programm tatsächlich als Benutzer erlebt. Ein weiterer Effekt ist, dass die Benutzerdokumentation dann mit höherer Wahrscheinlichkeit die Fragen behandelt, die den Benutzer interessieren.

Wichtige Voraussetzung beim „Testen durch Dokumentieren" ist, dass die Dokumentation nicht einfach aus der Aufgabenbeschreibung oder anderen Analysedokumenten abgeleitet wird. Die Verfasserin oder der Verfasser sollten sich vielmehr bei jedem Satz fragen „Funktioniert das wirklich so, wie ich das hier beschreibe?" und Abläufe durchspielen können. Dazu gehört unter anderem, dass sie eine lauffähige Version der Software zur Verfügung haben und nicht nur mit Screenshots arbeiten müssen. Sie sollten Probleme erkennen können und ihnen nachgehen dürfen. Außerdem sollten sie neben dem direkten Draht zu den Entwicklern Gewicht und Stimme bei der Bewertung des von ihnen dokumentierten Programms haben. Es müssen also Personen sein, die sich auch zu sagen trauen, dass man als Shortcut für die Kopierfunktion doch besser das allgemein übliche Strg+C nehmen sollte.

Eingebettete Tests

Viele Tests lassen sich direkt in den Quellcode einbetten. In den modernen Programmiersprachen stehen ASSERT-Anweisungen oder ähnliches zur Verfügung, mit denen sich die Eingangsbedingungen einer Funktion oder einer Klassenmethode abprüfen lassen. Dieses Konzept ist als *Design by Contract* aus der Sprache Eiffel bekannt. Dort ist es direkt in die Sprache integriert.

In den meisten Programmiersprachen lassen sich zumindest die Vorbedingungen ohne große Probleme auswerten. ASSERT-Anweisungen oder simple IF-Anweisungen zu Beginn einer Methode stellen sicher, dass alle Voraussetzungen für den Start der Methode vorliegen. Die Abprüfung der Vorbedingungen (die Default-Einstellung beim Eiffel-Compiler) reicht für unsere Testzwecke aus. Wenn die Nachbedingungen einer Methode bzw. die Invarianten der Klasse nicht stimmen, fällt dies spätestens dann auf, wenn sie als Vorbedingungen einer Folgemethode abgeprüft werden. Sollten bestimmte Nachbedingungen oder Invarianten nirgends als Vorbedingungen benötigt werden, sind sie vermutlich nicht so wichtig. Ein sorgfältiges Abprüfen der Vorbedingungen aller Funktionen oder Methoden kommt einem permanenten Regres-

sionstest gleich. Testtreiber müssen dann nur noch dafür sorgen, dass alle zu testenden Funktionen aufgerufen werden. Die Rückgabewerte werden vom Testtreiber protokolliert oder ausgeworfene *Exceptions* werden abgefangen. In einer objektorientierten Programmiersprache können solche Testtreiber als Testcode-Methode in der Klasse selbst abgelegt werden. Das hat den großen Vorteil, dass der zu testende Code und der dazu passende Testtreiber ohne zusätzlichen Aufwand gemeinsam verwaltet werden können, da sie sich an einer gemeinsamen physischen Stelle – meistens eine einzelne Quellcode-Datei – befinden. Dieses *One-Source*-Prinzip hat sich bei der Dokumentation bewährt und ist auch bei Code und Treibern zumindest von Unit-Tests sinnvoll.

Test-Automatisierung

Test-Automatisierung ist nicht grundsätzlich abhängig vom Einsatz teurer Werkzeuge. Vieles kann durch eingebettete Tests (siehe oben) automatisch abgeprüft werden. Test-Frameworks, mit denen sich Unit-Tests programmieren und verwalten lassen, können kostenlos aus dem Internet heruntergeladen werden (siehe [xPr]). Skripte lassen sich auch mit Werkzeugen und Skriptsprachen erstellen, die vom Betriebssystem bereitgestellt werden oder als freie Software im Internet verfügbar sind.

Das Erstellen von automatischen Tests bedeutet allerdings zunächst einmal einen erhöhten Zeitaufwand. Die zu bewältigende Lernkurve darf nicht unterschätzt werden. Wenn nur wenig Zeit zum Testen bleibt, ist es keine gute Idee, auf Test-Automatisierung zu setzen. Machen Sie Ihre ersten Automationserfahrungen am besten in frühen Projektphasen oder in zeitlich entspannten Projekten (falls es so etwas gibt) und stellen Sie sich auf eine langsame, kontinuierliche Verbesserung Ihres Automationskonzepts ein. Automatisieren Sie zunächst nur einfache, aber wichtige Tests. Oder automatisieren Sie nur Teile des Testablaufs, wie die Bereitstellung der notwendigen Testumgebung. Im Laufe der Zeit werden Sie mit zunehmender Erfahrung auch komplexere Testszenarien automatisieren können.

Testorientiertes Anwendungsdesign

Hier sind vor allem die Softwaredesigner und Architekten gefragt. Die Testbarkeit von Software muss in viel stärkerem Maße in der Softwarearchitektur selbst berücksichtigt werden. Wenn eine Anwendung sich unkooperativ verhält, kann das beste Test-Tool nicht viel ausrichten. Testorientiertes Anwendungsdesign heißt:

- die gesamte Anwendung ohne Oberfläche steuerbar zu machen,
- Funktionen vorzusehen, die einem Testwerkzeug oder einem Testskript zur Laufzeit die notwendigen Informationen liefern,
- Selbsttests während des Ablaufs vorzusehen,
- allgemein übliche Datenformate zu verwenden, die von verschiedensten Werkzeugen verstanden werden,
- die Anwendung in Komponenten aufzuteilen, die über protokollierbare Nachrichten miteinander kommunizieren,
- Einschübe (*Add-Ins*) und Verzweigungen (*Hooks*) vorzusehen, die einem Diagnosewerkzeug die Beobachtung des Programmablaufs ermöglichen,
- den Aspekt der Testbarkeit ständig im Auge zu behalten und neue Ideen dazu zu entwickeln.

Gut Ding will Weile haben

Ziel des *Rapid Application Testing* ist es, die schwerwiegendsten Fehler möglichst schnell zu finden. Heißt *Rapid Application Testing* deshalb, dass Programmsituationen und Fehlern, die nur lästig, unnötig oder unschön sind, nicht mehr nachgegangen wird? Dass Fehler auf ewige Zeiten im Programm verbleiben, nur weil sie nicht ganz so kritisch sind? Dass sich niemand mit Schreibfehlern in den Bildschirmformularen aufhalten soll?

Das heißt es natürlich nicht. *Rapid Application Testing* sagt nur, worauf man beim Testen zuerst achten sollte, und nicht, dass man mit dem Bemühen um mehr Qualität aufhören sollte, wenn alle Fehler der Kategorien „Absturz" und „Work-Around erforderlich" gefunden sind. Die wesentlichen Ideen des *Rapid Application Testing* – wie Zielorientierung und Vertrauen auf die Kreativität, das Können und die Erfahrung der Beteiligten, statt Beharren auf formalen Abläufen – sind auch zum Aufspüren von Fehlern, die nur lästig sind, sinnvoll anwendbar. Gerade die Hemmnisse im Fluss des Arbeitsablaufs, die Widersprüche im *Look&Feel*, Schönheitsfehler und wenig elegant wirkende Dialoge können kaum durch vorgefertigte Testfälle gefunden werden.

Man sollte sich allerdings von der Vorstellung lösen, dass diese Ziele ähnlich rapide erreicht werden können wie das Auffinden der schwerwiegendsten Fehler. Es gibt eben Dinge, die Zeit brauchen. Dazu gehört unter anderem, eine Software rund und glatt werden zu lassen.

Rapid Application Testing kann also nur heißen, die Software marktfähig zu machen. Dass nicht alle Software, die auf den Markt kommt, schon rund, glatt oder gar ausgereift ist – und dass sie das auch gar nicht sein kann und muss – brauche ich Ihnen wahrscheinlich nicht zu erzählen.

Alle Test- und Prüfverfahren können nur feststellen, ob ein Produkt von hoher Qualität ist oder welche Qualitätsmängel es aufweist. Qualität herstellen können sie nicht. Deshalb müssen die Vorgehensweisen bei der Erstellung der Produkte und die Test- und Prüfverfahren einander ergänzen. Im Idealfall unterstützt die Vorgehensweise bei der Erstellung eines Produkts gleichzeitig eine Prüfung der Qualität aller in das Produkt einfließenden Vorprodukte. Das hört sich schwieriger an, als es ist. Was Sie hierfür brauchen, sind engagierte und interessierte Mitarbeiter, eine offene Diskussionskultur im Projekt und schließlich ein einfacher Prozess, der unnötige Formalismen vermeidet und jederzeit Änderungen an seinen Ergebnissen zulässt. ■

Literatur & Links

[Bac] J. Bach (Satisfice Inc.), diverse Artikel zum Thema Rapid Testing, siehe: www.satisfice.com

[Coc02] A. Cockburn, games programmers play, in: Software Development, February 2002

[Fow01] M. Fowler, J. Highsmith, The Agile Manifesto, in: Software Development, August 2001

[IIT] Illinois Insitute of Technologie (Herausgeber des „Test Maturity Model" (TMT)), siehe: www.iit.edu

[Kan02] C. Kaner, J. Bach, B. Pettichord, Lessons Learned in Software Testing, Wiley, 2002

[Rät02] M. Rätzmann, Software-Testing, Galileo Computing, 2002

[xPr] Sammelseite zum Herunterladen von xUnit-Test-Frameworks: www.xprogramming.com/software.htm