Manfred Rätzmann

Software-Testing

Galileo Computing

Inhalt

Äußerst subjektives Vorwort

Als ich vor einigen Jahren anfing mich intensiver mit dem Thema Testen von Software zu befassen, war da zunächst mal nur eine große Verunsicherung. Einerseits hatte ich meine Programme natürlich immer sorgfältig getestet oder »ausprobiert« und hatte auch nie ein größeres Qualitäts-Desaster erlebt. Zwar tauchten auch nach der Auslieferung immer noch Fehler auf ... Na klar, etliche Sondersituationen waren auch nicht so richtig abgefangen ... Aber im Großen und Ganzen liefen die Sachen recht ordentlich. Auf der anderen Seite beschlich mich aber bald das Gefühl, dass meine Vorgehensweise beim Testen nicht viel mit dem zu tun hatte, was in den einschlägigen Büchern zum Thema beschrieben und gefordert wurde. Ich schob das zum Teil auf die andere Dimension von Projekten, die dort beschrieben wurde, oder auf das eher akademische Umfeld der Autoren, aber so ganz klein war mein damaliges Gesellenstück ja auch nicht – ein Projekt, an dem ca. 20 Entwickler beteiligt waren. Es folgte eine Phase »*das muss jetzt alles anders werden*«, gefolgt von »*jetzt weiß ich, wie's geht, ich mach's bloß noch nicht*« bis schließlich zu »*alles Quatsch, funktioniert ja überhaupt nicht!*«. Die Praxis hatte mich wieder.

Was ich inzwischen gelernt habe ist, meinen Erfahrungen und Einschätzungen zu vertrauen. Was man bei der Softwareentwicklung so tut und was man so lässt, welche Prioritäten man setzt und in welcher Reihenfolge man vorgeht – das alles ensteht im Laufe der Zeit aus Erfahrungen bei der Projektarbeit, eigenen und fremden Ideen, Diskussionen, Lesestoff, Versuch und Irrtum – kurz, aus der andauernden Beschäftigung mit dem Thema. Die führt dann auch beinahe zwangsläufig dazu, dass man bestimmte Vorlieben entwickelt. So macht es mir bedeutend mehr Spaß, die Ursachen eines Fehlers aufzuspüren als Funktionen, die schon abgeprüft sind, nach jeder Programmänderung erneut zu überprüfen. Solche Aufgaben versuche ich deshalb sofort zu automatisieren. Ich bin halt Softwareentwickler, und Softwareentwickler wollen alles, was lästig ist, automatisieren!

Ich denke, dass Tester und Entwickler einander immer ähnlicher werden – Entwickler testen und Tester entwickeln. Die Tatsache, dass Testautomation Softwareentwicklung ist (und zwar eine höchst spannende Unter-

abteilung davon), wird sich in den nächsten Jahren immer weiter rumsprechen. Die Qualitätssicherung wird verstärkt in den Softwareentwicklungsprozess integriert, auch das führt zu einer weiteren Annäherung von Testern und Entwicklern. Die benötigte Qualifikation wird für beide die gleiche sein, Unterschiede sind eher mentaler Art – aber da greife ich vor, das kriegen wir alles ein wenig später.

Ich habe dieses Buch zwar alleine geschrieben, aber ich lebe nicht alleine auf dieser Welt. Deshalb kann es nicht ausbleiben, dass einige Leute von einem solchen Projekt ebenfalls betroffen sind. Bei diesen Leuten möchte ich mich bedanken, dafür, dass sie mich unterstützt und ertragen haben im letzten halben Jahr. Danke, Moni, dass ich mich einfach so ausklinken konnte. Danke, Wolfgang, dass du seit Jahren mein Gesprächspartner zum Thema Softwareentwicklung bist (und mich immer mal wieder auf den Boden zurückholst). Danke, Jan, für das unermüdliche Korrekturlesen und die vielen Hinweise. Danke, Alf, für den Kick, der zu diesem Buch geführt hat. Danke, Judith, für dein Vertrauen und dein Engagement für dieses Buch. Dank an alle, die mir geholfen haben, danke, dass es euch gibt!

1 Rapid Application Testing

»Das geht nicht mit rechten Dingen zu«, rief der Hase: »Noch einmal gelaufen, wieder herum!« Und ab ging er wie der Sturmwind, dass ihm die Ohren am Kopfe flogen. Als er nun oben am Acker ankam, rief ihm dort der Igel entgegen: »Ich bin schon da!«
(aus: Hase und Igel, frei nach [Schröder1840])

1.1 Alles fließt

Die Zeiten ändern sich. Gerade in der Disziplin der Softwareentwickler ändert sich immer alles. 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. Dies hat auch Auswirkungen auf den Herstellungsprozess von Software. Wenn sich das Umfeld ständig ändert, darf die Erstellung der Software für dieses sich ändernde Umfeld kein Hemmschuh sein. Jede Form von bürokratischem Overhead in der Software-Entwicklung wird im Internet-Zeitalter in Frage gestellt.

Testen in Internet-Zeit

Das Testen von Software kann davon nicht unberührt bleiben. Die Grundlage der Testplanung und aller Metriken waren lange Zeit die funktionale Überdeckung und die Anweisungsüberdeckung (siehe im Kapitel »Tests in der täglichen Praxis« den Abschnitt »Testabdeckung«). Nachdem man eingesehen hatte, dass man nicht alles testen kann, wurden Methoden zur Aufdeckung von Redundanzen innerhalb der Testfälle und der Testdaten entwickelt. Außerdem Methoden zur Bewertung der Risiken, die man eingeht, wenn man bestimmte Teile der Software weniger intensiv testet als andere. Weil das alles aber immer noch nicht reicht, um mit der zunehmenden Beschleunigung Schritt halten zu können, fing man an, immer mehr Aspekte des eigenen Vorgehens in Frage zu stellen.

1.1.1 Software-Entwicklung als Spiel

Seit dem überwältigenden Siegeszug des objektorientierten Paradigmas in der Software-Entwicklung hat weniges die Gemüter derart erregt wie der Wandel der Vorgehensweise bei der Erstellung von Software.

Während vor zehn Jahren die Meinung vorherrschte, Computer-Aided Software Engineering, kurz CASE genannt, sei die Lösung aller Probleme (vor allem wegen der teuren Werkzeuge?), wird Software-Entwicklung heute von Leuten wie Alistair Cockburn [Cock02] als Spiel in einer Gruppe beschrieben, das zielorientiert, zeitlich und inhaltlich abgegrenzt und kooperativ abläuft. Die Gruppe besteht aus den Projektsponsoren, Managern, Spezialisten, Technikern, Designern, Programmierern, Testern, kurz allen, die am Erfolg des Projektes Interesse haben. Das Ziel des Spiels ist es in den meisten Fällen, das benötigte System so schnell wie möglich zu erstellen. Manchmal liegt der Fokus aber auf anderen Dingen wie Einfachheit der Bedienung, Fehlerfreiheit oder Absicherung gegen Haftungsansprüche Dritter.

Agile Prozesse

Diese von Cockburn dargestellte und begründete Sichtweise auf die Vorgänge in einem Software-Entwicklungsprozess fügt sich nahtlos ein in die aktuelle Diskussion um die Vorgehensmodelle der Software-Entwicklung. 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 [FowlerHighsmith01].

1.1.2 Die Kunst, ein Programm zum Absturz zu bringen

Das Spiel zwischen Software-Tester und Software-Entwickler erinnert an den Wettlauf zwischen dem Hasen und dem Igel auf der kleinen Heide bei Buxtehude [Schröder1840].

Das Spiel der Tester

Die Regeln des Spiels 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 bezeichnen als die Kunst, ein Programm zum Absturz zu bringen.

Das Spiel der Entwickler

Die Entwickler tun ihr Möglichstes, den Testern dieses Erfolgserlebnis zu verweigern. Sie gewinnen das Spiel, wenn sie bei jeder kritischen Programmsituation, auf die die Tester stoßen, wie der Igel sagen können:

»Ick bün all hier"! Das bedeutet nicht, dass ein Programm in allen erdenklichen Situationen klaglos seinen Dienst tun muss. Aber gleichgültig, welche Gemeinheiten sich die Tester ausdenken, das Programm sollte auf eine vorhersehbare und stabile Weise reagieren – zum Beispiel mit einer Fehlermeldung und einem Hinweis, wie der Fehler behoben werden kann.

1.1.3 Fehlergewichtung

Das Versagen eines Programms zeigt sich nicht immer so krass wie bei einem Absturz. Subtilere Versager sind Ablauf- oder Ausgabefehler, Rechenfehler, Bedienbarkeitsprobleme 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 für eine generelle Vierer-Skala der Fehlerwichtigkeit, die sich an der Benutzbarkeit der Software ausrichtet. Danach staffeln sich die Fehler in der Rangfolge ihrer Wichtigkeit in:

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

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

Die schwerwiegendsten Fehler sind nach dieser Skala diejenigen, die die Benutzer und Administratoren des Programms an ihrer Arbeit hindern und die sich nicht umgehen lassen. Alles, was dazu führt, dass die Benutzer oder Administratoren Umwege gehen müssen, um zum Ziel zu kommen – so genannte Work-Arounds –, kommt an zweiter Stelle. Danach dann alles, was lästig und ärgerlich ist, und schließlich die Schönheitsfehler.

Die vorgesehene Benutzung des Programms muss dafür irgendwo definiert sein, zum Beispiel in Use Cases, Anforderungskatalogen oder etwa in der Formulierung eines Menüpunktes. Die bloße Idee einer Nutzung (»Man könnte doch auch Folgendes machen ...«) reicht nicht aus, um zum Fehler zu erklären, wenn etwas nicht klappt.

Projektziele beachten

Letztendlich wird diese Rangfolge der Wichtigkeit aber von den Zielen des Projektes und den an diesen Zielen interessierten Personen (Stakeholder) festgelegt. So können zum Beispiel in einer Konkurrenzsituation, in der es entscheidend auf das Erscheinungsbild 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.

1.2 Das Rapide am Rapid Application Testing

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 Software-Test Company »Satisfice« und Autor mehrerer Bücher zum Thema Testen beschreibt auf der Homepage seiner Firma [Satisfice] die Unterschiede zwischen seiner Vorgehensweise, die er »Rapid Testing« nennt, und herkömmlichem Testen. Er zählt dazu folgende Punkte auf, in denen »Rapid Testing« sich hauptsächlich vom herkömmlichen formalen Testansatz unterscheidet:

- **Mission**

  Rapid 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 (Skills) des Testers zu niedrig. Rapid 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 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 möglicherweise problematischen Punkte in der Reihenfolge ihrer Wichtigkeit abgeprüft.

- **Erfahrung**

  Um auch beim Testen nicht in einer Analyse-Paralyse zu verharren, empfiehlt Rapid Testing den Testern, ihren Erfahrungen zu vertrauen. Während im herkömmlichen formalen Testansatz die Erfahrungen der Tester meist unbewusst und damit auch ungeprüft einfließen, sollen die Tester im Rapid Testing-Ansatz ihre Erfahrungen sammeln, festhalten und durch bewussten Einsatz überprüfen.

- **Forschen**

  »Rapid Testing is also rapid learning«, schreibt James Bach [Bach01]. Der Testgegenstand wird während des Tests erforscht. Der nächste Test leitet sich aus den Ergebnissen des vorherigen Tests ab. Exploratives Testen dringt so schneller zu den Kernproblemen der Software vor als vorgeschriebenes (scripted) Testing.

- **Zusammenarbeit**

  Testen im herkömmlichen Verfahren ist häufig eine einsame Tätigkeit. Eine wichtige Technik beim Rapid Testing ist »Pair-Testing«, also zwei Leute, ein Computer. Diese Technik wurde vom eXtreme Programming übernommen und funktioniert laut James Bach beim Rapid Testen ebenfalls sehr gut.

»Rapid Testing« ist vor allem also exploratives Testen (siehe in Kapitel 2 den Abschnitt über Teststrategien). Für den professionellen Tester, der das zu testende Programm erst beim Test kennen lernt, ist das explorative Testen häufig die einzige Möglichkeit, die anfallende Arbeit zu bewältigen. Die Testprozeduren für den generellen Funktions- und Stabilitätstest für das Logo »Certified for Microsoft Windows« wurden von Bach nach dem Ansatz des explorativen Testens entwickelt [Microsoft99].

1.2.1 Integriertes Testen

James Bach beschreibt »Rapid Testing« aus der Warte des professionellen Testers. Ich habe »Application« hinzugefügt und Rapid Application Testing (RAT) daraus gemacht, um die Verwandtschaft mit dem Rapid Application Development (RAD) aufzuzeigen. Beides betrifft den gesamten Herstellungsprozess – RAT nicht nur das Testen und RAD nicht nur das Erstellen der Oberfläche, wie man bei mancher Beschreibung meinen könnte –, und beides hat das gleiche Ziel, nämlich keine Zeit zu verschwenden.

Rapid Application Testing ist integriertes Testen, das heißt, es ist in den Software-Entwicklungsprozess integriert und nicht ein zweiter, nebenher laufender Softwarevalidierungsprozess. Zur Erstellung eines Produktes oder Zwischenproduktes im Software-Entwicklungsprozess gehört auch immer das Prüfen und Testen, so wie zum Kochen das Abschmecken gehört.

Rapid Application Testing bietet dazu verschiedene Strategien und Techniken an. Wichtige Strategien sind zum Beispiel »Testen durch Benutzen« und eingebettete Tests. Mehr darüber lesen Sie im Kapitel »Tests ganz allgemein gesehen« und dort im Abschnitt über Teststrategien. Alle Strategien und Techniken haben gemein, dass sie sich an dem Risiko orientieren, das vom jeweiligen (Zwischen-)Produkt für das Gelingen des gesamten Software-Entwicklungsprozesses ausgeht.

1.2.2 Gut Ding will Weil haben

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 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-and-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.

1.3 Tester und Entwickler

1.3.1 Der Stand der Dinge

Warum ist das Wissen um Strategien, Techniken und Hintergründe beim Testen eigentlich für Entwickler, Softwarearchitekten, Analysten, Designer, das heißt für alle am Entwicklungsprozess beteiligten Personen so wichtig? Reicht es nicht, wenn sich die Tester damit auskennen? Die Antwort ist so einfach wie zwingend: Weil in vielen Projekten gar keine Tester vorgesehen sind. Wenn die Projektbeteiligten trotzdem qualitativ hochwertige Software ausliefern wollen, müssen sie selber ran.

Jeder testet

»Tester« wird heute verstärkt als Rolle betrachtet, die von einzelnen Personen je nach Bedarf eingenommen wird. Da schlüpft die Entwicklerin nach getaner Entwicklungsarbeit oder zwischendurch in die Rolle der Testerin und testet die Anwendung »mal eben«. In den meisten Projekten gibt es jemanden, der die Software entworfen hat und hauptsächlich über Änderungen und Erweiterungen entscheidet. Dieser Jemand wacht dann häufig mit Argusaugen darüber, dass der Entwurf eingehalten wird und »sein Programm« das tut, was sie oder er konzipiert hat. Oder der Projektleiter möchte für die nächste Freigabe nicht seinen Kopf hinhalten, ohne das Programm vorher zu testen. 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.

Rapide heißt nicht ad hoc.

Die Kluft zwischen den formalen Anforderungen und der Projektpraxis beim Testen (aber nicht nur dort) gibt es wohl schon sehr lange, wahrscheinlich schon seit es Software gibt und die Notwendigkeit von Tests erkannt wurde. Vor allem in den ärmeren Projekten, die sich aus Gründen von Budget- und/oder Zeitknappheit auf das Wichtigste beschränken müssen, ist »Ad-hoc-Testen« an der Tagesordnung und bereitet allen Projektbeteiligten häufig Bauchschmerzen. Es bleibt ja so vieles ungetestet, denn eigentlich müsste man doch alles testen, oder?! Diese Unsicherheit rührt daher, dass das Ziel des Spiels »Software-Entwicklung« – die Mission also – den nur unbewusst am Spiel teilnehmenden Personen nicht klar ist. Auch die eigene Rolle in diesem Spiel und die damit verbundenen Aufgaben bleiben unklar.

Testen wird nicht als Mittel zur Risikominimierung begriffen, sondern als Anspruch, dessen Herkunft und Umfang vage bleibt. Man ahnt nur, dass man ihm nie gerecht werden wird. Daraus entsteht Unsicherheit über das, was zuerst getestet werden sollte. Diese Unsicherheit versuchen die Beteiligten mit formalen Methoden zu überwinden – so wie es in den Büchern gelehrt wird. Zum formal korrekten Erstellen der im »IEEE Standard for Software Test Documentation« [IEEE829] vorgesehenen Dokumente »*Testplan, Testentwurf, Testfallspezifikation, Testprozeduren, Testobjektverzeichnis, Testlog, Testvorfallsbericht und Testabschlussbericht*« fehlt aber erst recht die Zeit. Die Tester (falls im Projektbudget überhaupt vorgesehen) testen »irgendwas«, die Entwickler stören sich daran, dass die

Tester Meldungen zu eher marginalen Problemen verfassen (»Ich bin froh, dass das Teil läuft, und da kommt der mit so was!«), schließlich wird ausgeliefert und auf Gott vertraut.

1.3.2 Theorie und Praxis

Chem Kaner et al. vergleichen in ihrem Standardwerk »Testing Computer Software« [Kaner99] das Testen von Software mit der experimentellen Überprüfung wissenschaftlicher Theorien. Tester sind die Praktiker des Entwicklungsteams. Sie bauen Versuchsanordnungen auf, mit denen sie ihr mentales Modell des Programms bestätigen wollen. Oder mit denen sie nachweisen wollen, dass das Programm unter bestimmten Bedingungen versagt. Tester sind skeptisch. Sie betrachten das Programm mit Benutzeraugen, klopfen die Theorien der Entwickler auf Praxistauglichkeit ab.

Entwickler sind die Theoretiker in diesem Vergleich. Sie entwickeln die Ideen und können sich nur schwer vorstellen, an welchen Stellen die raue Wirklichkeit ihrer idealen Welt in die Quere kommen könnte. Entwickler glauben gerne, dass die ganze Welt so funktioniert, wie sie sich das vorstellen. Sie betrachten Programme – und auch Bugs – als interessante Aufgaben und Herausforderungen. Praxisnutzen ist ihnen weniger wichtig, es sei denn, sie setzen ihr Programm selber ein.[1]

Diese Charakterisierung mag etwas extrem sein, trifft jedoch den Kern der Sache. Wer beides betrieben hat – eigene Entwicklungsarbeit und das Testen fremder Programme – weiß, dass man sich als Tester und als Entwickler durchaus so fühlen kann, wie Kaner beschreibt.

Daraus nun zu schließen, dass zum Testen und Entwickeln von Software unbedingt verschiedene Personen gehören, ist für die Praxis eines Software-Entwicklungsprojektes oft wenig hilfreich. In vielen Projekten sind die Entwickler nun mal die Einzigen, die das Programm vor der Auslieferung überhaupt ausprobieren. Testen kann man das häufig kaum nennen, eher ein nochmaliges Bestätigen der Gültigkeit ihrer theoretischen Überlegungen (siehe oben).

1 Wenn Entwickler Programme für den Eigengebrauch schreiben, hapert's allerdings meistens mit der Benutzerfreundlichkeit.

Die Lösung kann nicht sein, die Situation und deren Hintergründe zu ignorieren und auf einer formalen Trennung zwischen Testen und Entwickeln zu bestehen. Vielen Projekten, deren Vorzug gerade der direkte Kundenkontakt und flexibles Reagieren auf Änderungswünsche und Fehlermeldungen ist, würde ein solcher Formalismus sofort das Genick brechen. Besser ist es, die Realität zu akzeptieren und den Beteiligten Techniken und Werkzeuge an die Hand zu geben, mit denen sie die Situation nicht nur meistern, sondern ihren Wettbewerbsvorteil durch hohe Qualität weiter ausbauen können.

1.3.3 Sein und Bewusstsein

Wie fühlen Sie sich?

Wie fühlt man sich, wenn man in einem Programm einen Fehler entdeckt hat? Wenn es gar das eigene Programm ist oder das Produkt des eigenen Teams – und der Auslieferungstermin direkt bevorsteht? Es gibt durchaus unterschiedliche Arten, auf eine solche Situation zu reagieren. Die einen haben es schon immer gewusst, dass die Entwickler dieses Programmteils Nieten sind, die noch nicht mal daran denken, so was Naheliegendes abzuprüfen (würde mir nie passieren, weil ...). Die anderen möchten am liebsten gar nicht hinschauen, weil sie wissen, dass der Fehler, der sich dort in seiner ganzen Schönheit zeigt, zu einer weiteren Projektverzögerung führt (und wir liegen doch schon so weit zurück!). Wieder andere finden es einfach lästig, so einen Kleinkram als Fehler zu melden, weil sie wissen, dass sie nach der Behebung des Fehlers eigentlich die Korrektur wieder überprüfen müssten.

Mission possible

O.k. – für seine Gefühle kann niemand. Man muss sie vielleicht nicht so deutlich zeigen, dass andere davon in Mitleidenschaft gezogen werden. Man könnte aber auch versuchen, sich mit ein bis zwei Gedanken zum Thema die eigene Aufgabe als Tester in diesem Projekt oder für diesen Programmteil, das Ziel des Spiels, die Mission, klar zu machen.

Bestimmte Tests sind dazu da, Fehler zu finden. Andere Tests sollen zeigen, dass bestimmte Anforderungen durch das Programm erfüllt werden. In beiden Fällen gilt: Je eher, desto besser. Je eher ein Fehler gefunden wird, desto eher kann er behoben werden. Je eher entdeckt wird, dass etwas fehlt, desto eher kann es ergänzt werden. Verdrängen und ignorieren hilft in beiden Fällen wenig.

Wenn ich einen Fehler finde, tue ich dem Projekt damit etwas Gutes. Wenn der Fehler ein gravierender Fehler ist, tue ich etwas sehr Gutes. Denn ein Projekt steht und fällt mit dem frühzeitigen Aufdecken der gravierenden Fehler, der so genannten Show-Stopper.

Vier Augen sehen mehr als zwei.

Ein Entwickler, der das eigene Programm testet, ist in gewisser Weise behindert. Unterbewusste Ängste halten ihn davon ab, bestimmte Programmteile intensiv zu überprüfen. Betriebsblind starrt mancher minutenlang auf den eigenen Sourcecode, ohne den Fehler zu sehen, der sich darin verbirgt. Nicht überprüfte Gewissheiten versperren allzu oft den Blick auf mögliche Fehlerursachen.

Anders sieht es aus, wenn Entwickler die Programme ihrer Kollegen anschauen, zum Beispiel beim Pair-Programming, als Gutachter bei Peer- oder Gruppen-Reviews oder in der Rolle des Benutzers. Als Benutzer eines fremden Moduls tritt auch beim normalen Entwickler der »Schau'n-mer-mal«-Reflex auf. Die öffentliche Schnittstelle des Moduls wird als das genommen, was sie ist: eine Benutzerschnittstelle. Ihr wird instinktiv misstraut, die Behauptungen der Schnittstellenbeschreibung (falls vorhanden) werden auf Korrektheit überprüft und so weiter.

1.3.4 And the winner is ...

Testern und Entwicklern muss bewusst werden, dass sie eine Rolle in einem Spiel innehaben. Insbesondere wenn beide Rollen von derselben Person oder demselben Personenkreis wahrgenommen werden, tun sich viele schwer, in die scheinbar aggressive oder destruktive Rolle des Testers zu schlüpfen.

Wenn beide Rollen im Spiel aber bewusst wahrgenommen werden, wenn das Ziel des Spiels bekannt ist und die Spielregeln eingehalten werden, dann kann aus der Not eine Tugend werden. Die Tester kennen ihre Mission und stürzen sich ohne Vorbehalte auf ihre Aufgabe, selbst wenn sie soeben noch Entwickler waren. Die Entwickler wissen, was Testen bedeutet, und treffen entsprechende Vorkehrungsmaßnahmen. Ihre Erfahrungen nutzen sie dann später als Tester wieder für verschärfte Prüfungen. Alle wissen, dass nur gefundene Fehler gute Fehler sind, und keiner »nimmt übel«. Wenn doch, hilft häufiger Rollentausch und eine offene Diskussion im Team. Wenn die Entwicklerin zum Tester sagt: »Hey, danke

für die Fehlermeldung. Da wäre ich nie drauf gekommen!«, sind beide auf dem richtigen Weg.

Gewinner des Spiels ist immer das Programm, das getestet wird. Gewinner ist damit auch das Team, das dieses Programm herstellt und das mit seinem Produkt im größeren Spiel innerhalb der eigenen Firma oder auf dem Markt mitwirkt. In seinem sehr lesenswerten Artikel sagt Alistair Cockburn: *»The project has two goals: to deliver the software and to create an advantageous position for the next game. If the primary goal isn't met, the next game may be canceled.«* [Cockburn02]

1.4 Ideen, Techniken und Werkzeuge

Dieses Buch soll Ideen, Techniken und Werkzeuge vorstellen, die zum Rapid Application Testing genutzt werden können. Ideen zum Finden der richtigen Fragen, Techniken für die Versuchsanordnungen zu deren Beantwortung und Werkzeuge zum Aufzeichnen, Interpretieren und Verwalten der Messergebnisse. Das wichtigste Glied in dieser Kette ist das Erste, nämlich das Stellen der richtigen Fragen. Wenn die Fragestellung nicht stimmt, nutzt die cleverste Versuchsanordnung und das empfindlichste Messinstrument nichts.

Ideen, Techniken und Werkzeuge werden benötigt:

- zum Aufdecken von Fehlern
- zum Aufdecken der Fehlerursache(n)
- zum Management des Testprozesses

Nicht alles davon wird von Testern und Entwicklern gleichermaßen eingesetzt. Während Tester zum Aufdecken von Fehlern Techniken und Werkzeuge brauchen, die zur Beobachtung des Programms und zum Vergleich von Ablaufergebnissen eingesetzt werden, brauchen Entwickler Techniken und Werkzeuge zur Analyse des Programmgeschehens. Ideen und Vorstellungen davon, was schief gehen könnte, brauchen beide.

Manager und Projektverantwortliche brauchen Techniken zur Risikoabschätzung, einen Überblick über den aktuellen Stand des Projektes, Zahlen als Entscheidungsgrundlage und zum Vergleich mit den Erfahrungswerten aus anderen Projekten.

Viele der hier vorgestellten Ideen haben mit einem Doppelnutzen zu tun. Doppelnutzen bedeutet, eine Tätigkeit so auszuführen, dass neben dem hauptsächlichen ersten Nutzen ein zweiter Nutzen entsteht. In unserem Fall der Nutzen der permanenten Qualitätssicherung und Qualitätskontrolle.

Doppelnutzen

Weil die Trennung zwischen Entwickler und Tester in der Praxis immer mehr aufgeweicht wird, soll dies kein Buch ausschließlich für Tester sein und auch keine Anleitung zum Debuggen ausschließlich für Entwickler. Beide Sichtweisen sollen diskutiert und die verschiedenen Herangehensweisen, wo immer sinnvoll, miteinander kombiniert werden.

Die Personen, von denen hier die Rede ist, sind im Allgemeinen keine auf ihren Bereich eingeschränkten Spezialisten, sondern Leute, die bei allem Spezialistentum noch über ihren Tellerrand hinausblicken können. Rapid Application Testing hat viel mit der Tatsache zu tun, dass in kleineren bis mittleren Projekten nahezu jeder in die unterschiedlichsten Rollen schlüpfen muss. Die ganz großen Projekte folgen sicherlich eigenen Regeln, die mehr mit Politik als mit Erfolgsorientierung zu tun haben.

Rapid Application Tester kennen ihre Rolle im Spiel, sie wissen um ihre Bedeutung im Projekt, und sie greifen auf jedes verfügbare Mittel zurück, um ihre Versuche zu planen, durchzuführen und auszuwerten.

2 Tests, ganz allgemein gesehen

Qualität bedeutet, dass der Kunde zurückkommt, nicht die Ware.
(Herrmann Titz, Begründer der Hertie-Kaufhäuser)

In diesem Buch werden unterschiedliche Techniken zum Testen von Software, Prüfen von Zwischenergebnissen bei der Erstellung von Software, zur Absicherung des bereits Erreichten bei einer Änderung oder Erweiterung der Software und so weiter vorgestellt. Techniken also, die für ganz unterschiedliche Zwecke entwickelt wurden und doch alle ein Ziel verfolgen: Die Qualität der ausgelieferten Software zu erhöhen.

2.1 Was ist Software-Qualität?

Qualitätssicherung wird generell in produktive und analytische Qualitätssicherung unterteilt. Zur produktiven Qualitätssicherung gehören alle die Maßnahmen, die das Produkt verbessern, zur analytischen Qualitätssicherung gehören die Maßnahmen, die die erreichte Qualität abprüfen.

Software-Testen gehört demnach zur analytischen Qualitätssicherung und dort zu den so genannten dynamischen Prüfungen. Die wichtigsten statischen Prüfungen der analytischen Qualitätssicherung sind Reviews, Code-Analysen und Modellvalidierung. Tests alleine verbessern die Softwarequalität noch nicht. Das wäre so, als könnte man alleine dadurch abnehmen, dass man sich täglich auf die Waage stellt. Tests und andere Prüfungen können nur aufdecken, wo etwas im Argen liegt. Behoben werden müssen die gefundenen Probleme auch noch, bevor die Qualität gesichert ist.

2.1.1 Definitionsversuche

Zum Qualitätsbegriff gibt es viele Aussagen und Definitionsversuche, zum Beispiel die Definition aus der DIN 55350:

> *»Qualität ist die Gesamtheit von Eigenschaften und Merkmalen eines Produktes oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse bezieht.«*

So wenig konkret diese Definition auf den ersten Blick erscheint, ergeben sich aus ihr doch einige wichtige Anhaltspunkte für die Frage, was Software-Qualität eigentlich ist:

»Qualität ist die Gesamtheit ...«

besagt, dass alle Eigenschaften und Merkmale in den Qualitätsbegriff eingehen, nicht nur die irgendwo niedergelegten Anforderungen. Software kann also durchaus alle schriftlich formulierten Anforderungen erfüllen und trotzdem von geringer Qualität sein. Oft ist das erwünschte »Look-and-Feel« oder die möglichst einheitliche Bedienung aller Programmteile nicht als Anforderung irgendwo festgehalten. Dennoch wäre die Qualität einer Software erheblich betroffen, wenn jedes Bildschirmformular völlig anders gestaltet wäre, die Bedeutung von Funktionstasten und Tastaturkürzeln dauernd wechseln würde und so weiter. Neben den formellen Qualitätsmerkmalen gibt es immer eine ganze Reihe von informellen Qualitätsmerkmalen.

»... die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse bezieht.«

Hier kommen sie ins Spiel, die Anforderungen an ein Programm. Dabei sind wiederum nicht nur die expliziten Anforderungen gemeint, die irgendwo schriftlich festgehalten sind. Ähnlich wichtig sind die impliziten Anforderungen, die sich aus dem Umfeld des Produkts und der Benutzer ergeben. Kaner et al. führen in [KanerBachPetti02] mögliche Quellen solcher impliziten Anforderungen auf:

- Konkurrenzprodukte
- Produkte der gleichen Produktlinie
- Ältere Versionen des Produktes
- Diskussionen im Projekt
- Kommentare des Kunden
- Artikel und Bücher zum fachlichen oder technischen Umfeld
- Interne und allgemeine Style Guides zur Benutzeroberfläche
- Kompatibilität zum Betriebssystem oder zur IT-Umgebung
- Die eigene Erfahrung

Die impliziten Anforderungen sind meistens nicht Gegenstand von Tests oder anderen Prüfungen. Trotzdem sollte man sie nicht außer Acht lassen. Wenn implizite Anforderungen verletzt werden, leidet die Qualität der Software ähnlich wie bei der Verletzung expliziter Anforderungen. Teilweise – insbesondere wenn der Benutzer direkt betroffen ist – sind implizite Anforderungen sogar einklagbar.

Was in der DIN 55350-Definition der Qualität allerdings fehlt, ist der Bezug zum Benutzer. Denn eine Funktion, die für einen erfahrenen Benutzer sehr wohl zur »Erfüllung gegebener Erfordernisse« geeignet ist, kann für einen unerfahrenen Benutzer völlig unbrauchbar sein, da sie ihn überfordert.

Die IEEE-Norm für Software-Qualität (ANSI/IEEE-Standard 729-1983) bezieht die Bedürfnisse des Benutzers ein. Laut Glossar der IEEE Computer Society [IEEEGlossar] ist Software-Qualität:

1. *The totality of features and characteristics of a software product that bear on it's ability to satisfy given needs; for example, conform to specifications.*
2. *The degree to which software possesses a desired combination of attributes.*
3. *The degree to which a customer or a user perceives that software meets his or her composite expectations.*
4. *The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer.*
5. *Attributes of software that affect its perceived value, for example, correctness, reliability, maintainability, and portability.*
6. *Software quality includes fitness for purpose, reasonable cost, reliability, ease of use in relation to those who use it, design of maintenance and upgrade characteristics, and compares well against reliable products.*

Bemerkenswert ist auch die Tatsache, dass die IEEE Computer Society vertretbare Kosten (*reasonable cost*) unter den Qualitätsmerkmalen einer Software aufführt. Auf das Testen angewandt heißt dies nämlich nichts anderes, als dass auch das Testen nicht unendlich weiter gehen darf.

Irgendwann (und zwar ziemlich schnell) wird exzessives Testen die Qualität der Software nicht mehr steigern, sondern mindern – indem es die Kosten in die Höhe treibt, ohne einen entsprechenden Qualitätsgewinn dagegensetzen zu können.

2.1.2 Neue Antworten zum Qualitätsbegriff

W. Mellis beschreibt in »Process and Product Orientation in Software Development and their Effect on Software Quality Management« [WieczMeyerho1] zwei unterschiedliche Qualitätsbegriffe, die sich aus unterschiedlichen Software-Entwicklungs-Modellen ergeben und die auch zu ganz unterschiedlichen Teststrategien und Methoden führen.

Transformative Software-Entwicklung

Transformative Software-Entwicklung überführt einen wohl definierten, verstandenen und relativ stabilen Prozess in einen automatisierten Zustand. Der zu automatisierende Prozess kann dazu genau analysiert und modelliert werden. Grundlage der Software-Entwicklung sind die in der Analyse erkannten Anforderungen. Die entstehende Software ist von guter Qualität, wenn sie sich korrekt verhält, allen Anforderungen entspricht und von den vorgesehenen Benutzern ohne größere Probleme einzusetzen ist. Transformative Entwicklungsprojekte heute sind häufig Inhouse-Projekte größerer Unternehmen zur Erstellung individueller Lösungen. Individualsoftware kann sich ohne Wenn und Aber auf einen Prozess einstellen. Flexibilität ist zweitrangig, dadurch verringert sich die Komplexität einer Software meistens erheblich. Tests lassen sich in solchen Projekten anhand der Anforderungen und des Systemdesigns weit im Voraus planen. Ein chaotischer Aspekt fließt in ein solches Projekt höchstens durch eine große Anzahl von Benutzern mit ihren jeweiligen Sonderwünschen ein. Dieses Chaos kann durch ein geregeltes Anforderungs- und Änderungsmanagement eingegrenzt werden.

Adaptive Software-Entwicklung

Im Gegensatz zur transformativen Software-Entwicklung gibt es bei der adaptiven Software-Entwicklung keinen eindeutigen Prozess, der einfach automatisierbar ist. Das kann verschiedene Ursachen haben: Entweder ist die Software so innovativ, dass mögliche Anforderungen und die praktische Arbeit mit dem System noch unerforscht sind. Oder das Anwendungsgebiet – die Domäne – der Software ist so chaotisch, dass sich kein automatisierbarer Prozess identifizieren lässt. Oder aber, und das ist wohl

die häufigste Ursache, es gibt sehr viele im Prinzip übereinstimmende, in Details jedoch abweichende Prozesse, die vereinheitlicht werden müssen. Das Letztere ist meistens der Fall, wenn Sie Standard-Software entwickeln, die vielen Kunden gefallen und zu deren Arbeitsabläufen kompatibel sein soll.

Adaptive Software-Entwicklung beginnt meistens mit einer Vision. In der Vision passt alles noch einfach zusammen. Danach fängt dann die Detailarbeit an. Dort sind Abstraktionsvermögen und Kreativität gefragt. Dem Softwarekern, der den kleinsten gemeinsamen Nenner der abzubildenden Prozesse darstellt, muss ausreichend Flexibilität mitgegeben werden, damit sich das fertige Programm möglichst flexibel an noch unbekannte Umfeldbedingungen anpassen kann.

Qualität hat hier weniger mit korrektem Verhalten zu tun (keiner weiß, was »korrektes Verhalten« in dem Fall genau ist), auch Übereinstimmung mit den Anforderungen fällt als Qualitätsmerkmal weitestgehend aus, weil die wesentlichen Merkmale noch nicht in detaillierten Anforderungen formuliert sind bzw. weil der nächste Kunde schon wieder andere Anforderungen hat.

Grundvorstellungen

Natürlich gelten auch hier die Grundvorstellungen von Qualität wie Korrektheit, Robustheit, Schnelligkeit und Übereinstimmung mit den impliziten Anforderungen des Anwendungsgebietes. Der Qualitätsbegriff hat in einem adaptiven Software-Entwicklungsprozess aber wesentlich mehr inhaltliche Aspekte als in einem transformativen Entwicklungsprozess. Hohe Qualität heißt hier, dass die Software generell nützlich ist (oder Spaß macht, »cool« ist), dass die Abstraktionen stimmen, dass die Software flexibel ist, sich den unbekannten Anforderungen der Kunden anpassen kann, den Arbeitsablauf nicht behindert, sondern unterstützt, den Kern der Sache trifft und bei den Benutzern »ankommt«. Lauter Dinge also, die sich meistens erst nach Einführung der Software in der Praxis zeigen. Tests dazu sind kaum im Voraus zu planen, da die adaptive Entwicklung eines Programms noch lange, nachdem das Programm in der Praxis eingesetzt wird, ein Forschungsprojekt bleibt.

2.2 Der generelle Ablauf eines Tests

Bevor wir uns in die Testpraxis stürzen, möchte ich in einem kurzen Überblick darstellen, was einen Test eigentlich ausmacht, welche generellen Testverfahren und Testarten es gibt, welche Testphasen im Projekt unterschieden werden und wie und wann andere Qualitätssicherungsmethoden eingesetzt werden können.

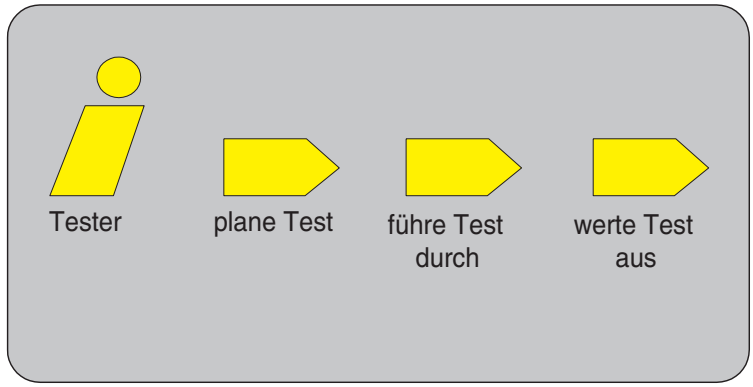

Abbildung 2.1 Genereller Testablauf

Ein einzelner Test läuft eigentlich immer in drei Schritten ab.

- Planung
- Durchführung
- Auswertung

Auch wenn die testende Person diese drei Schritte vielleicht nicht bewusst wahrnimmt, sind sie doch vorhanden – oder anders: Wenn einer der drei Schritte fehlt, kann es sich schwerlich um einen Test handeln.

Testsituation

Wenn sich jemand an einen Computer setzt, ein Programm aufruft und ausprobiert, wie dies und jenes funktioniert, führt sie oder er noch keinen Test durch. Erst wenn er oder sie an einer bestimmten Stelle stutzig wird und den vermuteten Fehler im Programm nachzuvollziehen versucht, kommt eine Testsituation zustande. Diese enthält dann typischerweise die oben angegebenen Schritte Planung (»Wenn ich dieses Feld freilasse und trotzdem auf OK klicke, wird das Programm wahrscheinlich abstürzen...«), Durchführung (»schau'n mer mal ...«) und Auswertung (»Genau: Absturz!«).

2.2.1 Planung

Im ersten Schritt wird festgelegt, was getestet werden soll. Dieser Planungsschritt ist nicht zu verwechseln mit der Testplanung im Großen, die als Teil der Projektplanung durchgeführt wird. Die Planung im Kleinen – in der Literatur auch als Testdesign bezeichnet – identifiziert zunächst den Testfall, legt dann das Vorgehen fest und danach, wie das Testergebnis ermittelt werden soll. Das hört sich alles sehr aufwändig an, ist aber im einfachsten Fall nur die Idee, was getestet werden müsste (»Da hat der Programmierer mit Sicherheit nicht dran gedacht!«), wie man vorgehen will (»Wenn ich dieses Feld freilasse und trotzdem auf OK klicke, ...«) sowie was und wie man prüfen will (»... wird das Programm wahrscheinlich abstürzen.«). Dazu kommt eine Vorstellung vom eigentlich richtigen Ergebnis (»Das müsste das Programm aber abkönnen.«). Ohne diese Vorstellung macht ein Test keinen Sinn. Was soll man mit einem Testergebnis anfangen, wenn man keine Vorstellung davon hat, ob es richtig oder falsch ist?

Risikobasiertes Testen

Rapid Application Testing ist risikobasiertes Testen. Weiter unten im Kapitel »Tests in der täglichen Praxis« habe ich Methoden zur Risikobewertung beschrieben, die anhand von Akzeptanzrisiken und technischen Risiken Testprioritäten festlegen sollen. Aber auch hier ist neben dem methodischen Vorgehen das intuitive Vorgehen ein wichtiger Bestandteil der Testpraxis. Die Fähigkeit eines erfahrenen Testers, Schwachstellen im Programm zu erraten, nennt man im Englischen »Error Guessing«.

Versuchen Sie stets, Tests zu planen, die ein möglichst großes Risiko betreffen. Behalten Sie dabei aber immer das größte Risiko im Auge: dass das Programm überhaupt nicht funktioniert. Der Ausgangspunkt sollte immer sein, dass das Programm unter Gut-Wetter-Bedingungen seinen Dienst tut – der so genannte Best Case . Erst wenn dieser abgehakt ist, kann man darüber nachdenken, was denn so alles passieren könnte.

Hauptfunktionen

Orientieren Sie sich bei der Planung der Tests an den Funktionen des Programms. Teilen Sie die Funktionen in Hauptfunktionen und unterstützende Funktionen auf. Eine Hauptfunktion könnte zum Beispiel sein, dass der Benutzer in einer bestimmten Programmsituation sein Arbeitsergebnis in einer Datei auf seiner Festplatte ablegen kann. Eine unterstützende Funktion dazu ermöglicht es dem Benutzer, das Zielverzeichnis über

einen Button »Suchen« oder »Browse« zu suchen. Konzentrieren Sie sich zunächst auf die Hauptfunktionen. Eine fehlerhaft oder gar nicht arbeitende Hauptfunktion ist für die meisten Benutzer inakzeptabel. Fehler in unterstützenden Funktionen können vom Benutzer eventuell umgangen werden.

Fehlerhäufungen

Mit Tests kann nur das Vorhandensein von Fehlern nachgewiesen werden, nicht die Fehlerfreiheit eines Programms. Wenn man schon 100 Fehler gefunden hat, heißt das nicht, dass nicht noch 1000 Fehler unentdeckt sind. Wenn Sie keinen Fehler mehr finden, kann das zum einen bedeuten, dass die Software fehlerfrei ist (eher selten), zum anderen aber auch, dass Ihre Testmethoden und Testfälle nicht ausreichen, um die tatsächlich noch vorhandenen Fehler zu finden.

Generell gilt: Wo ein Fehler ist, sind auch mehrere. Deshalb sollte die Planung der Tests auch die Ergebnisse der bereits durchgeführten Tests berücksichtigen. Das kann ein Modul betreffen, das sich als besonders fehlerträchtig erwiesen hat, oder den Zugriff auf Ressourcen wie Dateien, Drucker oder Modems, der nicht gut genug abgesichert ist. Oder es betrifft bestimmte Arbeitsabläufe, die nur halbherzig oder fehlerhaft vom Programm unterstützt werden.

Testplanung

Testplanung heißt beim Rapid Application Testing also nicht, dass Sie jeden einzelnen Testfall im Voraus planen und ausformulieren. Das ist meistens nicht sonderlich effektiv, weil Sie dann nicht mehr risikobasiert testen, sondern nach Plan. Der große Plan sollte vielmehr die großen Risikobereiche benennen und die möglichen Teststrategien aufzeigen. Die Feinplanung sollte flexibel bleiben und bereits erreichte Testergebnisse berücksichtigen können.

Alle Testfälle vorab zu planen ist auch dann nicht möglich, wenn Sie das zu testende Programm beim Testen erst kennen lernen. Beim explorativen Testen (siehe unten) wird der weitere Testplan immer von den Ergebnissen der vorherigen Tests abhängig gemacht.

Wenn Sie das zu testende Programm von innen kennen, werden Sie andere Tests planen, als wenn Sie das zu testende Programm nur von der Oberfläche her kennen. Aus dieser Erkenntnis stammt die Unterscheidung zwischen White-Box- und Black-Box-Tests. Dabei ist nicht das eine

Verfahren besser als das andere. Sie haben neben anderen Voraussetzungen auch andere Zielsetzungen. Mehr dazu weiter unten im Abschnitt »Testverfahren«.

2.2.2 Durchführung

Auch die Durchführung eines Tests erfolgt in drei Schritten:

1. Testumgebung bereitstellen
2. Test ausführen
3. Testergebnis ermitteln

Zur Testumgebung gehören eventuell besondere Testdaten, aber auch Metadaten, die das zu testende Programm steuern. In späteren Testphasen gehört vielleicht sogar die komplette Hardware-Umgebung dazu.

Die Testumgebung ist ein wesentlicher Faktor für die Wiederholbarkeit von Tests. Wenn ein Test häufig wiederholt werden muss, um zum Beispiel nach einem Programmtuning Performance-Verbesserungen zu überprüfen, macht es viel Sinn, zumindest die Bereitstellung der benötigten Testumgebung zu automatisieren. Nicht nur um Zeit zu sparen, sondern auch um nichts Wichtiges zu vergessen.

Testschnittstelle

Die Ausführung eines Tests ist sehr davon abhängig, ob für das Programm oder den zu testenden Teil eine Testschnittstelle zur Verfügung steht. Wenn ja, können Sie diese entweder dazu nutzen, den Test interaktiv durchzuführen oder, wenn der Test automatisiert werden soll, ein Programm zu schreiben, das den Test ausführt. Dazu existieren inzwischen Test-Frameworks für fast jede Programmiersprache. Über Test-Frameworks und deren Handhabung finden Sie einiges im Kapitel »Verfahren und Werkzeuge«. Über Testschnittstellen lesen Sie mehr im Abschnitt »Testorientiertes Anwendungsdesign« im Kapitel 4.

Wenn keine Testschnittstelle zur Verfügung steht, bleibt Ihnen wahrscheinlich nichts anderes übrig, als das Programm oder den zu testenden Programmteil über die allgemeine Benutzerschnittstelle zu testen. Während das interaktive Testen in vielen Fällen nur mit einer Benutzeroberfläche möglich ist, stört gerade diese beim automatisierten Testen sehr. Dann ist man nämlich auf die Unterstützung durch ein Werkzeug ange-

wiesen, das Benutzereingaben und Mausaktionen aufzeichnen und wiedergeben kann. Eine eingehende Beschreibung von Automatisierungsmöglichkeiten finden Sie im Abschnitt »Automatisierung von Testabläufen« in Kapitel 4.

Werkzeugunterstützung

Auch beim Ermitteln des Testergebnisses ist man häufig auf Werkzeugunterstützung angewiesen. Bei Programmen, die auf einer Datenbank arbeiten, braucht man zum Beispiel eine Zugriffsmöglichkeit auf die Datenbank außerhalb des zu testenden Programms. Bei verteilten Systemen braucht man eventuell Werkzeuge, die den Netzverkehr aufzeichnen und darstellen können, bei Performance-Tests braucht man Werkzeuge, die die Antwortzeiten eines Programms festhalten können, und so weiter.

Um möglichst schnell zu Aussagen über die Zuverlässigkeit und Robustheit des Programms zu kommen, reichen in den meisten Fällen aber einfache Tests aus. Denken Sie daran, dass ein Test dazu da ist, Probleme aufzudecken. Die Ursache des Problems zu finden ist Aufgabe der Entwickler, nicht der Tester.

2.2.3 Auswertung

Einen Testlauf auszuwerten bedeutet, das eigentlich korrekte Ergebnis mit dem tatsächlichen Ergebnis zu vergleichen. Das eigentlich korrekte Ergebnis sollte zu diesem Zweck eindeutig zu ermitteln sein. Man kann nicht testen, ob ein Programmergebnis korrekt ist, wenn man keine Vorstellung davon hat, wie das korrekte Ergebnis aussieht.

Wenn dokumentierte Vorgaben vorliegen, bilden diese naturgemäß die erste Quelle zur Ermittlung des korrekten Programmverhaltens. Zu den dokumentierten Vorgaben gehören Aufgabenbeschreibungen und Geschäftsmodelle, Anwendungsfall-Diagramme und textliche Beschreibungen der Anwendungsfälle, Layout-Vorgaben zur Oberflächengestaltung, ausgedruckte Beispielreports, die Online-Hilfe, Benutzerhandbuch, Installationsanweisungen oder Ähnliches. Was dort dokumentiert ist, kann direkt überprüft werden.

Fachliche Aspekte

Wenn fachliche Korrektheit überprüft werden soll, ist es unumgänglich, das dazu notwendige Wissen bereitzustellen. Fachliche Aspekte werden in Aufgabenbeschreibungen vielleicht, in der Online-Hilfe oder dem

Benutzerhandbuch aber eher selten erläutert. Das Benutzerhandbuch eines Finanzbuchhaltungsprogramms ist halt nicht dazu gedacht, Finanzbuchhaltung zu erlernen, sondern den Umgang mit diesem speziellen Programm. Als Tester sind Sie in diesem Fall meistens auf jemanden angewiesen, der Bescheid weiß.

Kontrollprogramm

Es kann auch sein, dass Sie in der glücklichen Lage sind, das Ergebnis eines Programms durch ein anderes Programm überprüfen zu lassen. Das ist immer dann der Fall, wenn das zu prüfende Programm eine Ausgabe liefert, die als Eingabe einer nächsten Verarbeitungsstufe oder eines anderen Programms dienen kann. Wenn der Prüfling zum Beispiel eine XML-Datei erstellt, muss diese nicht unbedingt visuell auf formale und inhaltliche Korrektheit überprüft werden. Zumindest um festzustellen, ob ein XML-Dokument »well-formed« ist, reicht es, dieses mit einem Programm zu öffnen, das XML-Dateien lesen kann. Bedenken Sie dabei aber, dass das Kontrollprogramm nicht zu tolerant sein darf. Wenn Fehler im zu prüfenden Dokument vom Kontrollprogramm toleriert oder übersehen werden, ist auch die Auswertung Ihres Testergebnisses unsicher.

Gegenprobe

Generell sollten Sie skeptisch sein, wenn zur Auswertung des Testlaufs ausschließlich die Ausgaben des zu testenden Programms zur Verfügung stehen. Häufig ist es so, dass ein fehlerhaftes Verhalten des Programms durch einen weiteren Fehler ähnlicher Art überdeckt wird. So kann zum Beispiel die fehlende oder falsche Aktualisierung eines neuen Datenbankfeldes einfach übersehen werden, wenn der Entwickler gleichzeitig vergessen hat, das neue Feld in einen damit verknüpften Report zu übernehmen. Versuchen Sie deshalb immer auch, das Ergebnis mit einem anderen Werkzeug zu überprüfen. Bei einem Programm, das seine Daten in einer Datenbank ablegt, sollte zur Ermittlung des tatsächlichen Ergebnisses ein direkter Zugriff auf die Datenbank erfolgen.

2.3 Was tun mit den Testergebnissen?

Das Testergebnis sollte in »geeigneter Form« festgehalten werden. Gut gesagt – nur: Welche Form ist geeignet, welche nicht? Das hängt davon ab, was Sie mit der Dokumentation des Testergebnisses bezwecken. Wenn ein Test glatt durchläuft und keine neuen Fehler aufdeckt, reicht es

in den meisten Fällen festzuhalten, was getestet wurde und wann. Eventuell noch, bei welcher Programmversion.

Wenn ein gefundener Fehler direkt in eine Programmänderung einfließt (Test & Tune), reicht eine kurze stichwortartige Notiz – in anderen Fällen sollte das Ergebnis mehr formal festgehalten werden, zum Beispiel in einer Datenbank, die alle durchgeführten Tests und deren Ergebnisse festhält. Einen Vorschlag zur Organisation einer solchen Datenbank finden Sie im Kapitel 5 im Abschnitt »Qualitätskontrolle«.

Fehlermeldung

Das Testdokument – die Fehlermeldung – sollte neben allgemeinen Angaben (Namen des Testers, falls mehrere Personen testen, Datum des Tests, getestetes Programm, Programmversion) die drei Stufen des Tests dokumentieren, nämlich Planung (worum geht's eigentlich), Durchführung (wie wurde getestet) und Auswertung (was wäre das korrekte Ergebnis, was war das tatsächliche Ergebnis). Dabei sollte so pragmatisch wie möglich verfahren werden. Die folgende Beispielmeldung enthält alle drei Angaben in komprimierter Form: *»Wenn die Pickup-Tabelle leer ist, erscheint momentan beim Programmstart eine Messagebox mit fehlendem Text '[ERR] PickupTableEmpty'. In diesem Fall sollte eigentlich ein Default-Satz angelegt werden.«*

Die Umgebungsbedingungen, unter denen der Test ablief, gehören – soweit sie relevant für das Testergebnis sind – zur Dokumentation der Durchführung. Sie sind insbesondere dann interessant, wenn sich der Fehler in der Entwicklungsumgebung nicht ohne weiteres nachvollziehen lässt. In diesem Fall liefert das unterschiedliche Verhalten von Programmen unter den verschiedenen Umgebungsbedingungen häufig einen entscheidenden Hinweis zur Lokalisierung des Fehlers.

Wichtig ist auch, dass bei einem Wiederholungstest nach der Fehlerkorrektur die gleichen Bedingungen wieder hergestellt werden können. Dazu braucht der Tester dann selbst die notwendigen Angaben zur Wiederherstellung der Testumgebung.

2.3.1 Gegentest

Die Ergebnisse der Tests führen normalerweise zur Überarbeitung des Programms und zur Fehlerkorrektur. Im Anschluss daran sollte derselbe

Test noch einmal durchlaufen werden, damit man sicher sein kann, dass 1. der Fehler richtig korrigiert wurde und 2. der richtige Fehler korrigiert wurde. Weil diese Kontrolle der Fehlerkorrektur eine aufwändige Sache ist, die zudem keine neuen Erkenntnisse hervorbringen soll, wird diese Pflichtübung allerdings gerne ausgelassen.

Nichts verdrängen

Beim Testen findet man häufig Fehler, hinter denen man gar nicht her war. Das heißt, auch bei einer risikobasierten Testplanung fallen Testergebnisse an, die ein weniger risikobehaftetes Programmverhalten betreffen. Das sind einfach all die vielen Kleinigkeiten, die einem aufmerksamen Tester nebenbei auffallen und die nicht verdrängt, sondern ebenfalls gemeldet werden sollen. Der Gegentest dieser weniger wichtigen Testergebnisse kann dann aber entfallen. Auch der Gegentest sollte sich am Risiko des jeweiligen Programmverhaltens orientieren, damit Sie als Tester keine Zeit mit dem Überprüfen eher nebensächlicher Korrekturen verlieren.

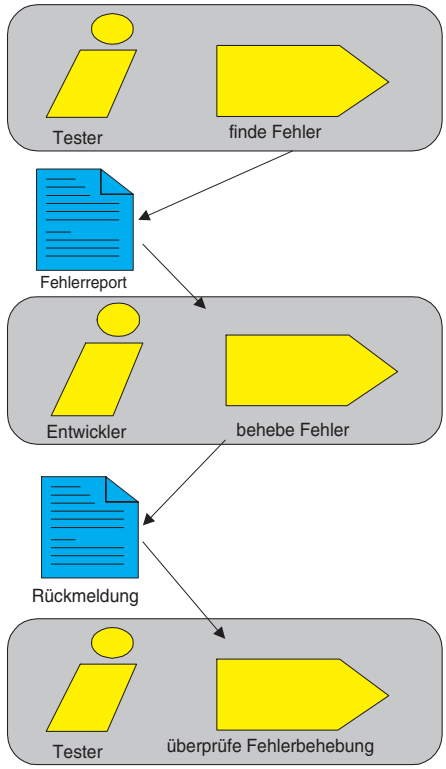

Abbildung 2.2 Test, Fehlerbehebung und Gegentest

Der in Abbildung 2.2 dargestellte Weg wird demnach nur für die Fehlermeldungen und Korrekturen komplett durchlaufen, die wichtig genug sind. Der dargestellte direkte Weg ist auch nur zu empfehlen, wenn die Zahl der Fehlermeldungen gering ist, etwa im letzten Abschnitt eines Software-Lebenszyklus. In allen anderen Fällen werden die Fehlerreports zunächst gesammelt. Bei der Freigabe der nächsten Programmversion werden die behandelten Punkte dann gemeinsam zurückgemeldet. Dazu wird im Allgemeinen eine zentrale Datenbank eingesetzt. Wenn die Tester keine Mitglieder des Entwicklungsteams sind, sollte außerdem jemand im Entwicklungsteam mit der Vorabqualifizierung der eingehenden Fehlermeldungen und der Überprüfung der ausgehenden Rückmeldungen beauftragt werden.

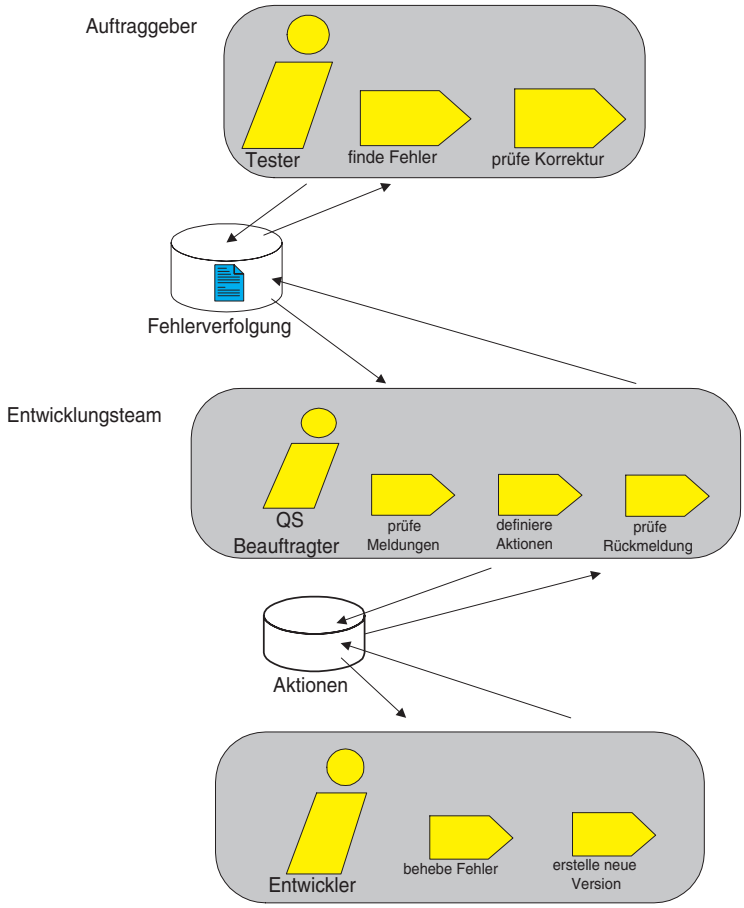

Abbildung 2.3 Überprüfung von Fehler- und Rückmeldungen

Es bietet sich auch an, der in Abbildung 2.3 dargestellten Rolle des QS-Beauftragten auch das Änderungsmanagement anzuvertrauen (siehe »Testfallverfolgung« im Kapitel 5). Achten Sie aber darauf, für diese Rolle dann auch ausreichend Ressourcen zur Verfügung zu stellen, um an dieser Stelle nicht einen neuen Flaschenhals zu installieren. In hektischen Zeiten ist ein QS-Beauftragter mit zwei bis drei Testern auf der einen Seite und drei bis vier fleißigen Entwicklern auf der anderen Seite gut ausgelastet.

2.3.2 Liste der bekannten Fehler

Wenn eine Fehlerbehebung aus irgendwelchen Gründen nicht möglich ist, haben Sie mit den Testergebnissen trotzdem noch eine Liste in Händen, die aussagt, wie zuverlässig die Software in bestimmten Situationen arbeitet. Eine solche Liste kann zum Beispiel als »bekannte Fehler« Eingang in das Handbuch finden. Zusätzlich lohnt es sich immer, die am häufigsten gemachten Fehler bei der Entwicklung festzustellen. Anhand dieser Angaben können Sie Entscheidungen zu notwendigen Aus- und Weiterbildungsmaßnahmen treffen. Oder Ihren Software-Entwicklungsprozess ändern, andere Tools einsetzen etc.

2.4 Teststrategien

... (Ende der Leseprobe)