Markus Gärtner

(markus.gaertner@it-agile.de)
arbeitet als Berater, Trainer, Coach und Tester für die it-agile GmbH in Hamburg. Der Autor von ATDD by Example (Addison-Wesley) hilft agilen Teams beim leichgewichtigen Testen, vornehmlich mit ATDD und explorativem Testen.

„Wir brauchen mehr Qualität“ – Systemische Effekte hinter den unfairen Fragen

„Wieso hat unsere QS diesen Bug eigentlich nicht gefunden?“, „Warum dauert das Testen so lange?“, „Wir sind doch jetzt agil.“ Kennen Sie diese Phrasen? Haben Sie eine Antwort oder passende Reaktion darauf? Das Problem mit derartigen Bemerkungen ist, dass sie unfair sind. Ihnen unterliegt eine Interpretation des Fragenden, die nicht in Frage gestellt wird. Hätte beispielsweise die Testabteilung diesen Fehler wirklich finden können – in der gegebenen Zeit, mit den verfügbaren Informationen, unter dem bestehenden Termindruck? Sollte das Testen wirklich kürzer dauern? Und zu guter Letzt: Was bedeutet es eigentlich, agil zu sein? Die Phrasen setzen all dies als gegeben voraus und fragen lediglich nach einer Lösung für das wahrgenommene Problem. „Wir haben zu viele Bugs“, „Wir brauchen mehr Qualität“ und „Wir haben ein Testproblem“ sind dabei leider zu häufig die falschen Reaktionen. Woran liegt das?

Matthew Heusser stellt in einem Artikel fünf verschiedene Wege vor, wie Tester mit solchen unfairen Fragen umgehen können (vgl. [Heu2009]). Darunter fällt beispielsweise die versteckte Voraussetzung herauszufordern. „Warum sollte das Testen schneller gehen?“, „Sollen wir weniger testen?“, „Sollen Tester nur drei statt fünf Tests pro Funktion durchführen?“, „Wir haben zwar diesen Bug nicht gefunden, aber dafür diese 20 anderen.“, „Welchen von diesen 20 anderen sollen wir dafür opfern, dass wir zukünftig diesen einen wichtigen Bug finden werden?“ Dies sind alles valide Reaktionen auf obige Fragen. Sie machen die Opportunitätskosten des Testens (vgl. Kapitel 4 in [Heu2011]) deutlich. Wenn wir einen Test ausführen, können wir die aufgewendete Zeit nicht mehr dafür nutzen, andere Tests auszuführen. Damit kaufen wir letzten Endes Zeit auf Kosten von Risiko. Jeder Test, den wir ausführen, adressiert ein Risiko. Wenn wir einen Test nicht ausführen, öffnen wir dem damit verbundenen Risiko Tür und Tor. Wir müssen also herausfinden, welche Risiken wir unbedingt adressieren müssen und mit welchen wir leben können. Was hilft uns das konkret bei unseren Problemen?

„Wir haben zu viele Bugs“

Zu viele Fehler, die an den Kunden ausgeliefert wurden, sind erst mal ein Symptom. In zwei Blog-Einträgen beschreibt Michael Bolton, woran das liegt (vgl. [Bol2009a], Bol2009b]). Stellen wir uns einmal ein Testteam vor, das in Sessions von 90 Minuten testet. Mit Vorbereitungszeit, Testdurchführung, Testauswertung und Protokollierung der Ergebnisse benötigt ein Tester durchschnittlich 2 Minuten pro manuellem Test. Bei Fehlern gibt es allerdings etwas mehr zu tun. Für einen Bug muss der Tester den Fehler isolieren, Follow-up-Tests durchführen, um mehr über die Konsequenzen zu erfahren, und vor allem einen Bug im Defect-Tracking-Tool einstellen. Dafür benötigt das Team im Durchschnitt 8 Minuten[1]) .

Angenommen unser fiktives Testteam arbeitet für ein sehr, sehr gutes Entwicklungsteam. Dieses arbeitet nahezu perfekt, und zwar so perfekt, dass die Tester keine Fehler im Produkt entdecken können[2]). Das bedeutet, dass das Testteam in 90 Minuten 45 Testfälle durchführen kann. In einen durchschnittlichen Arbeitstag passen ca. 4 derartige Sessions von ununterbrochenem Testen, sodass wir in einer Woche bei 900 ausgeführten Testfällen enden. Das sind 900 Testfälle, die uns mehr Informationen über das Produkt liefern (siehe Tabelle 1).

Zugegeben, ein Entwicklungsteam, das keine Fehler produziert, ist utopisch. Hin und wieder hört man von einigen Teams, die sehr nahe an die null Fehler herankommen, aber gar keine Fehler sind unrealistisch. Was würde passieren, wenn unser fiktives Testteam auf ein Entwicklungsteam trifft, bei dem es in 90 Minuten einen einzigen Fehler findet? Dieses Team wendet vielleicht testgetriebene Entwicklung mit

[1]) Falls Ihnen 8 Minuten für die ganze Arbeit zu wenig erscheinen, rechnen Sie die Beispiele doch gleich mal mit anderen Werten durch.

[2]) Eine interessante Frage hierbei ist, wenn wir wissen, dass unser Entwicklungsteam keine Fehler produziert, müssen wir dann trotzdem dessen Software testen.

Entwicklungsteam	Testdesign und -ausführung	Fehlersuche und -Reporting	Tests pro Session	Tests pro Woche
A	90 Minuten	0 Minuten	45	900

Tab. 1: Bei Team A handelt es sich um ein sehr, sehr gutes Entwicklungsteam.

Entwicklungsteam	Testdesign und -ausführung	Fehlersuche und -Reporting	Tests pro Session	Tests pro Woche
A	90 Minuten	0 Minuten	45	900
B	80 Minuten	10 Minuten	41	820

Tab. 2: Bei Team B erzielt unser Testteam leicht andere Ergebnisse.

Refactoring und Continuous Integration auf einem hohen Level an. Durch das ständige Feedback ist dieses Team in der Lage, sehr hochwertige Software zu produzieren, aber hin und wieder verlieren sie den Blick für das große Ganze.

Unser Testteam wäre hier in der Lage, in 90 Minuten den Test mit dem Bug durchzuführen. Dabei werden 2 Minuten auf den Test und 8 Minuten auf die Fehlersuche und das Reporting verwendet. Es verbleiben also 80 Minuten, in denen das Testteam keinen Fehler findet. Das sind 40 weitere Tests, die das Testteam hier durchführen kann. Durchschnittlich auf die Woche hochgerechnet landen wir somit bei insgesamt 820 Tests. Anders gerechnet benötigt das gleiche Testteam eine Woche und etwas weniger als einen halben Tag, um die gleiche Anzahl an Testfällen durchzuführen (siehe Tabelle 2).

Ich weiß nicht, wie es um Ihr Entwicklungsteam steht. Ich halte einen Fehler pro 40 Tests für relativ gut. Wie sieht es mit einem Entwicklungsteam aus, bei dem unser fiktives Testteam in 90 Minuten durchschnittlich 5 Bugs finden kann. 5 Bugs in 90 Minuten ist nicht ganz so viel und hört sich im Falle der meisten Firmen realistisch an. Hier würden die Tester also 5 Fehler finden und dafür 50 Minuten aufwenden. Die restlichen 40 Minuten schaffen sie dann noch gerade so 20 Tests, in denen keine Probleme gefunden werden. Das sind somit 25 Tests pro 45 Minuten oder 500 Tests pro Woche. Damit braucht das gleiche Testteam für die gleiche Menge an Tests fast die doppelte Zeit im Vergleich zum fehlerfreien Team A (siehe Tabelle 3).

Natürlich ist das erst mal nur eine Milchmädchenrechnung. In einigen Projekten, z. B. im regulierten Umfeld, kommen zur Testausführung noch der Testentwurf und das Protokollieren der Testergebnisse hinzu. Außerdem gibt es ein signifikantes Problem mit der Rechnung: Dass die Tester die Fehler melden, bedeutet noch lange nicht, dass diese auch ordnungsgemäß beseitigt werden. Häufig werden weitere Fehler beim Beheben von vorherigen Fehlern eingebaut. Dadurch leitet sich dann letzten Endes noch mehr Arbeit für unser fiktives Testteam ab, wenn es darum geht, die behobenen Fehler nachzutesten.

Was dieses Zahlenbeispiel allerdings deutlich macht, ist, dass ein Testteam unabhängig von den individuellen Fähigkeiten alleine aufgrund der Tatsache, dass es mit unterschiedlichen Entwicklungsteams zusammenarbeitet, eine vollständig andere Außenwahrnehmung haben kann. Bei Team A betrug die Testzeit gerade einmal die Hälfte der Testzeit von Team C. Der einzige Unterschied besteht in der Anzahl der vom jeweiligen Entwicklungsteam produzierten Bugs.

Dadurch wird deutlich, dass sehr große Effekte bei der Verbesserung in der Software-Entwicklung durch fortschrittliche Entwicklungspraktiken wie testgetriebener Entwicklung, Continuous Integration und von akzeptanztestgetriebener Entwicklung erbracht werden können. Das Beispiel zeigt allerdings auch, dass die Wahrnehmung, unsere Tester müssten nur schneller testen, oftmals ein Trugschluss ist. Natürlich hängt viel von den Fähigkeiten des Testteams ab. Je eher jedoch Fehler im Entwicklungsprozess vermieden werden, desto schneller kann das Entwicklungsteam insgesamt vorankommen und desto weniger Folgefehler gelangen in das Produkt. Dies ist ein Grund, warum Software-Entwicklung höhere Qualität durch bessere Entwicklungspraktiken, wie sie beispielsweise von agilen Methoden favorisiert werden, leisten kann.

Systemische Effekte

Die beschriebenen Effekte sind so genannte systemische Effekte. Diese treten in komplexen Systemen auf, wenn es Feedback-Zyklen gibt. Dabei unterscheidet man zwischen

Entwicklungsteam	Testdesign und -ausführung	Fehlersuche und -Reporting	Tests pro Session	Tests pro Woche
A	90 Minuten	0 Minuten	45	900
B	80 Minuten	10 Minuten	41	820
C	40 Minuten	50 Minuten	25	500

Tab. 3: Bei Team C findet unser Testteam mehr Fehler und kann dadurch weniger Tests pro Woche durchführen.

positiv verstärkenden und degenerierenden Feedback-Zyklen. Erstere werden im Volksmund auch Teufelskreis genannt. Eine Aktion aus der Wahrnehmung A heraus führt zu einem Zustand, der die Wahrnehmung von A verstärkt. Beispielsweise führt Überlastung bei der Arbeit dazu, dass ich noch mehr Fehler in meinen Code einbaue. Durch mehr eingebaute Fehler steigt aber auch der Projektdruck und damit meine eigene Überlastung, sodass ich noch mehr Fehler produziere. Degenerierend wird der Feedback-Zyklus, wenn ich mich damit abfinde, dass ich langfristig besser dastehe, wenn ich ausgeruht auf der Arbeit erscheine.

Für unser Problem mit den vielen Bugs müssen wir allerdings den Fokus auf das gesamte System der Software-Entwicklung ausdehnen. Schließlich kann und wird es so sein, dass unser Testteam zwar außerordentlich gut ist, aber mit der Anzahl an Fehlern, die unsere Entwickler einbauen, schlüpfen auch mehr und mehr Fehler durch das Sicherheitsnetz, das unsere Tester bereitstellen. Damit erreichen letzten Endes auch Fehler den Kunden, und der spielt in der systemischen Betrachtung eine wesentliche Rolle.

Normalerweise fragt ein Kunde nach neuen Features; er möchte neue Funktionen für seine Software haben. D. h., die Software soll ein bestimmtes Problem für ihn lösen. Diese „normalen" Anfragen nennt John Seddon „Regular Demand", also den regulären Bedarf des Kunden (vgl. [Sed2005]). Doch wenn unsere Software Fehler enthält, generieren wir als Software-Unternehmen beim Kunden einen weiteren Bedarf. Der Kunde gibt sich selten damit zufrieden, wenn er schlechte Software bekommt. Viel wahrscheinlicher ist, dass er Nachbesserung verlangt. Dies ist der so genannte „Failure Demand" (vgl. [Heu2011]). Der Kunde kommt wieder und wieder zurück und verlangt irgendwann vielleicht nicht nur, dass der Fehler behoben wird, sondern will auch eine Art Kompensation für seinen Verdienstausfall. Im schlimmsten Fall tritt der Kunde allerdings überhaupt nicht mehr auf uns zu, sondern wandert zur Konkurrenz ab.

Systemisch betrachtet bedeutet das, dass wir selbst mit einem sehr guten Testteam nach wie vor Fehler ausliefern werden, wenn diese nicht frühzeitig während der Entwicklung gefunden werden. Auch ein sehr gutes Testteam ist letzten Endes menschlich und macht Fehler bzw. findet vielleicht die Bugs nicht, die durch andere maskiert werden. Dadurch liefern wir mehr Fehler aus, wenn wir mehr Bugs in die Software in erster Instanz eingebaut haben. Durch die Wirkung beim Kunden entsteht allerdings neben dem ohnehin schon vorhandenen „Regular Demand" ein zusätzlicher „Failure Demand", der das Entwicklungsteam weiter unter Druck setzt. Eine große Anzahl an Bugs haben somit einen systemisch verstärkenden Effekt: Es müssen mehr Fehler vom Testteam gefunden werden, was in mehr Problemen mündet, die unser Entwicklungsteam beheben muss, und mehr Fehler, die an den Kunden gelangen. Insgesamt wächst somit der wahrgenommene Druck unserer Entwickler in doppeltem Ausmaß. Mehr Fehler in der Entwicklung sorgen somit für mehr Druck, durch den wiederum mehr Fehler entstehen. Im Deutschen spricht man in einem solchen Fall von einem Teufelskreis.

Doch ein Entwicklungsteam ist diesem Teufelskreis nicht hilflos ausgeliefert. Es muss aber ein klares Signal bekommen, dass es in der vermeintlichen Krise noch gewissenhafter arbeiten soll. Nur wenn wir es schaffen, weniger Bugs einzubauen und weniger Bugs beim Beheben von Fehlern zu produzieren, kann das Team den Teufelskreis durchbrechen und langfristig den „Failure Demand" so reduzieren, dass es wieder aus dem reaktiven Modus herauskommt.

Tester können frühzeitiges Feedback liefern

Systemisch betrachtet ist es aber genauso fatal zu denken, dass sich nur in der Entwicklung etwas ändern muss, damit sich die Situation des gesamten Teams verbessert. Tester spielen eine wesentliche Rolle in der Software-Entwicklung. Während die Software selbst ein Problem für einen Kunden lösen soll, lösen Tester das Problem, ob die Software wirklich das Problem für den Kunden löst. Dabei sind Tester häufig lediglich die Überbringer von schlechten Nachrichten. Sie weisen auf ein Problem in der Software-Entwicklung hin. Ob und wie das gesamte Team darauf reagiert, steht auf einem anderen Blatt.

Doch welchen Beitrag zur Verbesserung liefern Tester eigentlich? Schenkt man einer nicht repräsentativen Umfrage aus unseren Kursen Glauben, dann sind Tester primär dafür da, das entwickelte Produkt zu kritisieren. Sie finden alle Macken und Makel an unseren Produkten. Sie finden Fehler, bevor die Kunden sie sehen, und lassen uns wissen, wie gut es um unsere Software wirklich bestellt ist. Tester liefern damit Informationen darüber, welchen Grad an Qualität die Software erreicht hat.

Doch wie können Tester dazu beitragen, den „Failure Demand" zu reduzieren? Hier ist ein Paradigmenwechsel im Software-Testen notwendig. Statt die Fehler nach dem Einbauen zu entdecken, müssen Tester ihre Rolle als Qualitätsadvokaten im Projekt von Beginn an wahrnehmen können. Durch das Auffinden von Fehlern vor ihrer Implementierung und Umsetzung in Software kann in den meisten Firmen ein signifikanter Mehrwert entstehen.

In der agilen Software-Entwicklung setzen wir hier beispielsweise auf das gezielte Pairing von Testern und Programmierern. Tester werden zwar oft als erfolglose Entwickler wahrgenommen, doch darf man bei dieser Betrachtung nicht vergessen, dass sie Tester geworden sind, weil sie eins besser können als entwickeln: testen. Durch Pairing werden diese Fähigkeiten gezielt im Team verteilt, und jeder Programmierer wird somit ein Stück weit ein besserer Tester, genauso wie jeder Tester, der mit einem Programmierer pairt, ein Stück weit ein besserer Entwickler wird. Außerdem sind sie so gezielt in der Lage, Feedback zu Problemen im Code noch während dessen Entstehung zu liefern.

Darüber hinaus sollten Tester auch Teil des Backlog Groomings sein und so frühzeitig Feedback zu kommenden Funktionen in der Software liefern können. Akzeptanztestgetriebene Entwicklung bindet sogar Tester ganz bewusst in so genannte „Specification Workshops" zusammen mit Programmierern und Product Ownern ein, damit frühzeitig in der Entwicklung von Anforderungen nicht nur Akzeptanzkriterien identifiziert werden können, sondern auch Bugs noch auf der Story-Karte selbst behoben werden. Tester spielen hier vor allem mit ihrem kritischen Denken eine führende Rolle. Letzten Endes geht es um den Traum der traditionellen Software-Entwicklung, von Beginn an mit genügend fehlerfreien Anforderungen zu arbeiten.

Durch den Paradigmenwechsel von „Fehler finden" hin zu „Frühzeitiges Feedback liefern" wird auch die Position von Testern in der Software-Entwicklung deutlich: so früh zu Beginn des Projekts wie möglich. Mittel- bis langfristig bedeutet das allerdings auch, dass wir auf andere Fähigkeiten bei der Ausbildung von Testern setzen müssen als das alleinige systematische Finden von

Fehlern. Menschliche Kommunikation ist leider in vielen Fällen alles außer systematisch.

Krisenmanagement

Schon Albert Einstein hat bemerkt, dass man Probleme niemals mit derselben Denkweise lösen kann, durch die sie entstanden sind. Ein Stück weit trifft das auf das Problem mit zu vielen Bugs und zu wenigen Testern zu. Es bringt nichts, mehr Tester mit dem gleichen Problem zu beschäftigen, wenn dadurch lediglich der „Failure Demand" erhöht wird. Auf der anderen Seite haben Manager die Möglichkeit, einen echten Unterschied zu machen, indem sie es gar nicht erst zulassen, dass „Failure Demand" entsteht. Technische Exzellenz wie testgetriebene Entwicklung und eine so genannte „Zero Bug Policy" tragen ihren Mehrwert dazu bei. Und ohne klare Aussage vom Management wird sich keine Änderung einstellen; weder auf Teamebene noch in der Unternehmenskultur.

Wichtig hierbei ist es zu verstehen, warum ich mich eigentlich dauernd im Krisenmanagement befinde (vgl. [Hen2001]), und dass ich systemisch immer eine Alternative habe, die ich wahrnehmen kann und sollte. Besserer Code löst kurz- und langfristig sicher mehr Probleme als mehr Testressourcen lösen können. Allerdings müssen Tester auch lernen, wie sie sich viel früher als bislang gewohnt in den Entwicklungsprozess einfinden können.

Für den Weg aus der Krise ist es zu guter Letzt wichtig zu verstehen, wie wir uns dort hineinmanövriert haben. Der Beginn jeder Krise ist schließlich stets das Ende einer Illusion. Umso wichtiger ist es, diese Illusion zu verstehen, sowie die Hintergründe, wie man sich in diese Fallen gebracht hat – und wie man wieder herauskommt. ■

Referenzen

[Bol2009a] Why is testing taking so long? – Part 1, Michael Bolton, 2009, siehe http://www.developsense.com/blog/2009/11/why-is-testing-taking-so-long-part-1/.

[Bol2009b] Why is testing taking so long? – Part 2, Michael Bolton, 2009, siehe http://www.developsense.com/blog/2009/11/what-does-testing-take-so-long-part-2/.

[Hen2001] Why are my pants on fire?, Elisabeth Hendrickson, 2001, siehe http://testobsessed.com/wp-content/uploads/2011/04/wampof.pdf.

[Heu2009] 5 ways to answer executives' unfair software test, QA questions, Mattew Heusser, 2009, siehe http://searchsoftwarequality.techtarget.com/tip/5-ways-to-answer-executives-unfair-software-test-QA-questions.

[Heu2011] How to reduce the cost of software testing, Matthew Heusser, Govind Kulkarni, Auerbach Publications, 2011, ISBN 978-1439861554.

[Sed2005] Freedom from Command and Control: Rethinking Lean Service, John Seddon, Productivity Press, 2005.