KEINE AUSREDEN: TESTAUTOMATISIERUNG IN AGILEN PROJEKTEN

In der Zeit der klassischen Vorgehensmodelle ging Zeitdruck bei IT-Projekten noch auf Kosten der Qualität. In der modernen agilen Welt heißen die Stellschrauben Nutzen, Qualität und Rahmenbedingungen – das sind Kosten, Zeitrahmen und Anforderungen. Bei der Qualität dürfen keine Abstriche mehr gemacht werden. Qualitätsprobleme werden in der agilen Welt häufig mit Hilfe von Testautomatisierung entdeckt und behoben. Dieser Artikel beschreibt das Testvorgehen in agilen Entwicklungsprojekten und gibt Tipps, wie die Qualität in der agilen Welt mit Hilfe von Automatisierung effektiv verbessert werden kann.

die autoren

Ramon Anger
(ramon.anger@capgemini.com)
ist als Technischer Architekt im Bereich Public für Capgemini tätig. Er beschäftigt sich insbesondere mit neuen Technologien, Softwarearchitektur und agilen Methoden.

Frederik Eichler
(frederik.eichler@capgemini.com)
ist als Qualitätsmanager im Bereich Individualsoftware (Automotive) für Capgemini tätig. Er beschäftigt sich mit agiler Softwareentwicklung und Qualitätsmanagement.

Die agile Softwareentwicklung wird dadurch getrieben, dass bestehende Prozesse immer wieder hinterfragt und verbessert werden. Das geschieht in den regelmäßigen Retrospektiven agiler Teams. Ein Ergebnis dieses Prozesses kann sein, dass sich Teams für Automatisierung entscheiden. Das bedeutet, dass manuelle Prozesse überall dort automatisiert werden, wo die manuelle Ausführung keine Vorteile bietet. Dieses Vorgehen ermöglicht es, in jeder Entwicklungsiteration ein qualitätsgesichertes Produkt zu erstellen, das dem Kunden Mehrwert bietet.

In der agilen Softwareentwicklung wird der Prozess, in dem für eine Software ein Build erzeugt, angemessen getestet und ausgeliefert wird, als *Deployment-Pipeline* bezeichnet (siehe Abbildung 1). Tritt bei Build, Test oder Deployment ein Fehler auf, wird dieser sofort sichtbar gemacht, damit er zeitnah beseitigt werden kann. Die Stufen und Übergange einer Deployment-Pipeline können je nach Projekt und Rahmenbedingungen stark von unserem Beispiel abweichen. Anhand der Stufen der hier dargestellten Deployment-Pipeline wollen wir in diesem Artikel die verschiedenen Möglichkeiten zur Automatisierung verdeutlichen und Hilfestellung für die Umsetzung in den eigenen Projekten geben.

Entwickler- und Unit-Tests

Entwicklertests sind Modultests auf kleinster Ebene, um die Korrektheit einer Implementierung nachzuweisen. Häufig werden dazu XUnit-Frameworks (z. B. „Junit“, vgl. [Jun]) eingesetzt. Diese sind gut dokumentiert und lassen sich mit unterstützenden Bibliotheken (z. B. „Hamcrest“, vgl. [Ham]) erweitern, um etwa die Lesbarkeit oder die Wartbarkeit der Tests zu erhöhen. Ein wichtiger Faktor bei der Planung dieser Tests ist die Nutzung von externen Ressourcen, wie Datenbanken oder Nachbarsysteme. Ein Entwicklertest bietet nur dann Mehrwert, wenn er sich mit geringem Tool- und Zeitaufwand vom Entwickler ausführen lässt.

Wir empfehlen dringend, darauf zu achten, dass alle Tests nicht nur lokal auf dem Entwicklersystem, sondern auch in der Build-Umgebung ausführbar sind. Ein Test ist erst abgeschlossen, wenn er auch auf der Build-Umgebung läuft.

Entwicklertests müssen neben Positivtests (erwartete Nutzung der Module) auch Negativtests (erwartete Fehlbenutzung der Module) und Tests der definierten Randbereiche einschließen. Sie sind White-Box-Tests, um bestimmte Fehlersituationen herstellen zu können. Dazu ist Wissen über die Implementierung notwendig.

Um das Feedback an die Entwickler nicht unnötig zu verzögern, muss es möglich sein, sämtliche Entwicklertests innerhalb weniger Minuten auszuführen. Eine sehr kurze Ausführungszeit für alle Entwicklertests hat zwei weitere Vorteile:

- Die Tests können bei jedem Check-In ins Versionsverwaltungssystem ausgeführt werden. Schlägt ein Test fehl, wird der Check-In verweigert. Der fehlgeschlagene Test muss erst grün werden, bevor eingecheckt werden darf.
- Spielt *Continuous Deployment* – die zeitnahe Produktivsetzung einer Änderung – für das betroffene Produkt eine Rolle, minimiert eine kurze Ausführungszeit aller Tests die Zeitspanne, bis die Änderung live gehen kann.

Abb. 1: Eine vereinfachte Deployment-Pipeline (vgl. [Hum10]).

Wenn in diesen Testläufen ein Test fehlschlägt, ist es offensichtlich, dass eine der Bedingungen, nach denen die Softwarekomponenten funktionieren sollen, verletzt wurde. Unabhängig davon lohnt es sich, dem Testen noch weitere statische Qualitätskontrollen nachzulagern. Diese können dem Team Rückmeldung über Probleme in der Software geben. In Java-Projekten setzt man etwa Open-Source-Tools wie „Findbugs“ (vgl. [Sou]) und „Checkstyle“ (vgl [Che]) ein, um offensichtliche Programmierfehler oder Verletzungen der Programmierrichtlinien aufzudecken.

Sind alle Modultests erfolgreich durchgelaufen und alle statischen Analysen vollzogen, ist der Build nun an einem Punkt in

Tipp 1: Testergebnisse müssen sichtbar sein.
Es gibt keinen Mehrwert für das Team, wenn sich die Teammitglieder nicht selbstständig über den Stand der Testergebnisse informieren können. Hier helfen moderne CI-Umgebungen, wie Jenkins, die alle Testergebnisse historisiert und auf Testfall-Ebene in einem Web-Interface darstellen. Nur so kann ein Entwickler die Eigenverantwortung für die von ihm durchgeführten Änderungen und somit die Effekte auf das Gesamtsystem übernehmen.

der Pipeline, wo wir davon ausgehen, dass wir richtig programmiert haben. Es ist aber noch nicht getestet, dass wir auch das Richtige programmiert haben.

Automatisierte Entwicklertests für Artefakte wie Konstruktoren, *Getter* und *Setter* sind aus unserer Sicht nur geeignet, um die Testabdeckung künstlich zu erhöhen. Natürlich können bei der manuellen Erstellung dieser Artefakte Fehler gemacht werden, die durch einen entsprechenden Unit-Test aufgedeckt werden können. Aber wer schreibt Getter und Setter heute noch manuell? Anstatt sich einer Gesamt-Testabdeckung von 93 % zu rühmen, die die Geschäftslogik bei genauer Betrachtung nur zu 60 % testet, empfehlen wir, den Fokus der Entwicklertests auf die Geschäftslogik zu richten – dahin, wo der tatsächliche Wert von Entwicklertests liegt. Eine hohe Codeabdeckung trifft hier keine Aussage über die Codequalität, eine niedrige Codeabdeckung kann hingegen ein Indikator für fehlende Tests sein.

Automatisierte Akzeptanztests

Die Idee hinter Akzeptanztests ist, einen möglichst großen Teil der Spezifikation – also indirekt die Anforderungen – über automatisierte Testfälle abzudecken. Hier hat es sich als gute Praxis erwiesen, bereits bei der Spezifikation eines Systems bestimmte Testszenarien mit zu berücksichtigen. Dabei kann es sich – je nach Level der Agilität im Projekt – um im Test nachvollzogene User-Storys oder um explizit in der Spezifikation angegebene Testaspekte und Beispiele handeln. Es liegt in der Natur dieser Tests, dass sie mehr Ressourcen als einfache Modultests benötigen. Dennoch muss im Vorfeld klar abgegrenzt werden, welche Nachbarsysteme und Ressourcen wirklich für einen Akzeptanztest notwendig sind und welche Systeme durch *Mockups* abgebildet werden können. Ziel muss es sein, die Akzeptanztests mindestens einmal am Tag durchzuführen. Agile Entwicklungsmethoden wie *Behaviour Driven Development* (vgl. [Dan]) gehen hier noch weiter: Akzeptanzkriterien sind wesentlicher Bestandteil jeder User-Story, d. h. Akzeptanztests werden Teil der Spezifikation.

Die Tools, die sich für Akzeptanztests einsetzen lassen, sind aufgrund der erhöhten Komplexität – je nach Projekt – sehr unterschiedlich. Häufig lässt sich das grundsätzliche Zusammenspiel von verschiedenen Anwendungsfällen auch mit den gängigen Unit-Test-Werkzeugen (z. B. „XUnit“) abbilden. In Web-Projekten bieten sich zusätzlich Tools wie „Selenium“ (vgl. [Sel]) an, um die Oberfläche zu testen. Um in Projekten mit nativen Benutzungsoberflächen komplexe Anwendungsfälle durchzuführen, setzen wir unter anderem „Squish“ (vgl. [Fro] und [adz11]) ein.

Für all diese Tools gilt der Grundsatz, Logik und Mechanik zu trennen. So kann etwa das Wissen, wie eine Oberfläche implementiert ist, in einem eigenen Objekt gespeichert werden (in Selenium sind dies so genannte *Page Objects*). Der Akzeptanztest kann nun dieses Objekt nutzen, um ein Anwendungsszenario durchzuführen. Wird während der weiteren Entwicklung beispielsweise eine *Drop-Down*-Liste durch mehrere *Radio-Buttons* ersetzt, so muss der Testfall dies nicht wissen. Es wird lediglich das *Page Object* angepasst und alle Testfälle funktionieren weiterhin ohne Probleme. Sollte man sich entscheiden, einen Testfall gegen eine andere Benutzungsoberfläche oder nur gegen einen Service auszuführen, so kann diese Trennung von Logik und Mechanik sogar dabei helfen, dass die Testfälle eins zu eins weiter benutzt werden können und nur die *Page Objects* ausgetauscht werden müssen.

Es empfiehlt sich, die Test-Suiten, in denen Akzeptanztests normalerweise zusammengefasst werden, regelmäßig neu zu ordnen. Test-Suiten sollten nicht die Historie des Projekts, sondern die in der Software umgesetzte Fachlichkeit widerspiegeln und danach gruppiert werden. Daher lohnt sich oft ein Refactoring.

Das „U“ in „User Interface“ macht es merklich komplizierter, Tests auszuführen. Ein menschlicher Nutzer wartet auf einen Dialog oder interpretiert eine Sanduhr als Fortschritt. Ein automatisierter Test kann dies von sich aus nicht – er wartet nicht auf eine Nachbedingung und schlägt fehl. Für diese bekannten und gelösten Probleme gibt es Synchronisierungsmechanismen, die dem Testteam aber bekannt sein müssen. Diese Details können den Unterschied zwischen guten und schlechten Akzeptanztests ausmachen. Ein sporadisch fehlschlagender Test ist oft nicht nur nutzlos, er erzeugt auch erheblichen Aufwand im Team. Ein Akzeptanztest ist dann gut geschrieben, wenn ein roter Testfall bedeutet, dass die Software nicht funktioniert. Es lohnt sich vor dem Aufsetzen von solchen automatisierten Akzeptanztests, einen Experten für das jeweilige Gebiet zu konsultieren oder – wenn möglich – gleich in das Team zu holen.

Tipp 3: Tests müssen wertvoll sein.
Beim Automatisieren von Tests muss immer darauf geachtet werden, dass sie einen wirklichen Mehrwert für das Team und für die Abnahme der Software bieten. Wenn ein Testszenario unwahrscheinlich ist und im Fehlerfall zu keinem größeren Schaden führt, ist es eine valide Abwägung, solch einen Test nicht zu implementieren (oder bedeutend seltener als einmal pro Tag auszuführen).

Gerne wird überlegt, ein Feature in einer Iteration zu entwickeln und es in der nächsten Iteration zu testen. Das empfiehlt sich jedoch grundsätzlich nicht. Ein automatisierter Akzeptanztest gehört zur aktuellen Iteration. Wird darauf verzichtet, kann das korrespondierende Arbeitspaket in der Iteration nicht abgeschlossen werden. Eine einzige Ausnahme ist vorstellbar: Die betroffene Funktionalität wird in der nächsten Iteration erweitert und erst dann können automatisierte Tests sinnvoll formuliert werden. Problematisch wird diese Vor-

Tipp 2: Besser ein automatisierter Testfall als gar keiner.
Gerade wenn mit der Automatisierung begonnen wird, scheint dies oft eine unlösbar aufwändige Aufgabe. Hier lohnt sich allerdings jeder kleine Schritt, der dem Team Feedback gibt und Zeit bei der abschließenden Software-Qualitätssicherung spart. Es bietet sich an, ein Backlog von Testfällen für die Automatisierung anzulegen, diese nach Kritikalität zu sortieren und Stück für Stück zu automatisieren. Wenn neue Arbeitspakete begonnen werden, muss die Automatisierung gleich mit geplant werden.

Tipp 4: Abnahmetests sollten immer grün sein – Zero Bug Policy.
Auch wenn das Release noch ein paar Wochen entfernt sein mag, ist es sehr wichtig, Testfälle nicht länger als notwendig fehlschlagen zu lassen – auch wenn man die vermutliche Ursache kennt. Ein fehlschlagender Testfall kann durchaus Probleme verdecken, die erst nach Behebung der offensichtlichen Ursache zu Tage treten. Ziel des Teams muss es sein, die Testfälle auch während der Entwicklung grün zu halten. Grün bezieht sich hier auf die Signalfarben, die traditionell in Build-Umgebungen eingesetzt werden.

gehensweise, wenn die Erweiterung durch Umpriorisierung doch nicht stattfindet und die Tests weiterhin manuell durchgeführt werden müssen.

Aus unserer Erfahrung lohnt sich eine sprechende, nachvollziehbare Zuordnung von Anforderungen (*User-Storys*) und Akzeptanztests. Wird eine Anforderung verändert oder erweitert, fließt der Aufwand für eine notwendige Anpassung bestehender Akzeptanztests mit in die Schätzung ein. Gibt es außerhalb des Entwicklungsteams Interessenten für die Akzeptanztest-Kriterien, erhalten diese wertvolles Feedback zu den Auswirkungen der Änderung. Können Anforderungen und Akzeptanztests einander nicht zugeordnet werden, kommt das böse Erwachen bei der Entwicklung: Die Änderungen dauern fünf Minuten, das Anpassen der fehlgeschlagenen alten Akzeptanztests zwei Tage.

Manuelle Tests

Manuelle Tests sind auch in agilen Projekten wichtig. Sie schließen die Lücke zwischen automatisierten Tests und aufwändig oder gar nicht automatisiert prüfbaren Anforderungen. Dabei können manuelle Tests zeitaufwändig und fehleranfällig sein. Auch in der agilen Welt neigen Projektteams dazu, manuelle Tests zu vernachlässigen, wenn die Zeit am Ende einer Iteration knapp wird. Das führt zu einem gefährlichen Trugschluss: Wenn es akzeptabel ist, manuelle Tests im Notfall wegzulassen, sind sie scheinbar nicht wichtig.

Tipp 5: Pair-Testing im UAT.
User-Acceptance-Tests sollten dabei nie durch die Softwareentwickler durchgeführt werden. UAT mittels Pair-Testing hat sich dagegen bewährt. Hier führt ein Benutzer den UAT durch, der Entwickler sitzt daneben und bekommt direkt wertvolles Feedback zur Benutzbarkeit seiner Arbeit. Im Idealfall ist dieser Tester der Kunde selbst oder ein Mitarbeiter mit tiefem fachlichen Wissen, aber keinen weiteren Kenntnissen über den Quellcode.

Auch in der agilen Welt ist es der Zweck von Tests, Fehler in der Software zu finden. Wenn automatisierte Tests dazu nur bedingt in der Lage sind, gibt es lediglich eine Alternative: manuelles beziehungsweise exploratives Testen, bei dem die Tester ihre Kreativität und Spontanität einsetzen, um Fehler zu finden. Ein wesentlicher Nutzen manueller Tests – besonders bei neuen Features – ist die kontextbezogene Perspektive auf die zu prüfende Funktionalität. Denn im Gegensatz zu automatisierte Tests kennt der manuelle Tester deren aktuellen Kontext.

Wenn es Kunde und Budget zulassen, planen wir am Ende einer Iteration für alle Entwickler einen Tag ein, dessen Fokus auf manueller Testdurchführung liegt. Ziel dieses Tages ist es, die Software zerbrechen zu lassen. Gelingt es, den Code durch Benutzereingaben oder Datenkonstellationen zu zerbrechen, wird für die Situation, die dazu geführt hat, ein automatisierter Test geschrieben. Anschließend wird der Mangel nachvollziehbar behoben und mit dem automatisierten Test dafür gesorgt, dass die so gewonnene höhere Robustheit erhalten bleibt.

Plädoyer für Nicht-Automatisierung

Wenn die geforderte Funktionalität implementiert wurde, also sämtliche automatisierte Tests durchgeführt und um eventuell notwendige manuelle Tests der neuen Features ergänzt wurden, schließt sich eine weitere manuelle Stufe an.

Der *User-Acceptance-Test (UAT)* dient dazu, eine Software unter realen Bedingungen zu testen und zu prüfen, ob eine Software effizient und effektiv genutzt werden kann. Software, die für Endanwender gedacht ist, muss bei dieser Testart vom Benutzer selbst geprüft werden. Dazu ist es notwendig, den fachlichen Kontext der zu prüfenden Software zu kennen und einzubeziehen. Testwerkzeuge sind hierzu nicht in der Lage. Aus unserer Sicht ist ein vollständig automatisierter UAT auch in der agilen Welt nicht seriös durchführbar – ausgenommen automatisierte *Smoke-Tests*, die die Vollständigkeit der Funktionalität oberflächlich prüfen. Gelegentlich werden automatisierte *GUI*-Tests (*Graphical-User-Interface*) als UAT bezeichnet. Diese Tests sind funktionale Tests unter Einbeziehung einer Oberfläche und können zum Beispiel im Regressionstest einen wertvollen Beitrag leisten. Sie sind aber kein Ersatz für die Einbeziehung des Menschen als Tester.

Ein Fokus des UAT liegt auf der Installierbarkeit der Software. Diese lässt sich leicht in Kombination mit der oben beschriebenen automatisierten Prüfung auf Vollständigkeit anwenden: Wenn dieser Smoke-Test erfolgreich war, ließ sich die Software offenbar erfolgreich installieren.

Ziel des UAT ist es, die Benutzbarkeit einer Software zu prüfen. Aus eigener Erfahrung wissen wir, dass für einen erfolgreichen UAT in allen Teststufen gewissenhaft gearbeitet werden muss. Für Entwickler gibt nichts Schlimmeres als einen UAT, der von den Testern oder vom Kunden nach wenigen Minuten abgebrochen wird, weil viele offensichtliche Fehler in den vorherigen Teststufen nicht entdeckt wurden und der Software mangelnde Qualität bescheinigt wird.

Tipp 6: Zeitnaher UAT.
Während der UAT in der klassischen Projektwelt eine abgeschlossene Phase darstellt, bietet es sich in agilen Projekten an, unmittelbar nach dem Abschluss einer Funktionalität Nutzer-Feedback einzuholen. Will der Entwickler in Folge des Feedbacks Änderungen vornehmen, ist seine Arbeit effektiver als nach einem UAT-Feedback mehreren Wochen.

Last und Performance-Tests

Die Teststufe Last- und Performance-Tests dient im Wesentlichen dazu, drei nichtfunktionale Anforderungen an ein System zu prüfen: Antwortzeitverhalten, Lastverhalten und Systemstabilität über einen Zeitraum von beispielsweise 12 oder 24 Stunden. Im Gegensatz zu klassischen Projekten, in denen Last- und Performance-Tests jeweils am Ende einer langen Entwicklungsphase durchgeführt werden, bietet sich in agilen Projekten durch kurze Iterationen die Möglichkeit, früh und kontinuierlich Informationen zum Last- und Antwortzeitverhalten zu erhalten (siehe Abbildung 2).

Noch effektiver ist ein kontinuierlicher Last- und Performance-Test im Rahmen eines *Nightly Builds*. Voraussetzung für die Integration dieser Tests in den *Nightly*

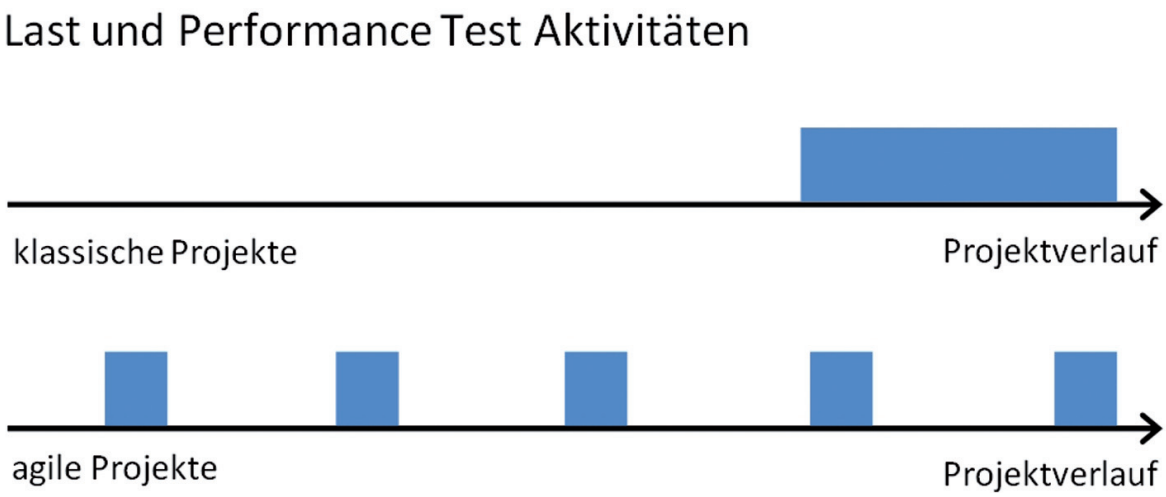

Abb. 2: Last-Tests in agilen Projekten.

Build ist, dass die für Lasttests häufig umfangreichen Testdaten und die Testfälle selbst die Codeänderungen der Entwickler reflektieren. Dies bedingt einen hohen Automatisierungsgrad der Testdaten-Erstellung und eine zeitnahe Aktualisierung der Testfälle. Veränderungen im Last- und Performance-Verhalten einer Software werden so unmittelbar erkannt und nicht erst am Ende einer Iteration.

Um Veränderungen im Last- und Antwortzeit-Verhalten einer Software feststellen zu können, soll diese Stufe so wenig manuelle Schritte wie möglich enthalten, damit aufeinanderfolgende Testläufe untereinander vergleichbar sind und Testergebnisse nicht durch manuelle Eingriffe verfälscht werden. Ändert sich das Last- und Antwortzeit-Verhalten der Software stark beziehungsweise anders als erwartet, lässt sich der konstante Testaufbau als Ursache oftmals von vorn herein ausschließen.

Bei Last- und Performance-Tests sollte aus unserer Sicht das normale Mengengerüst auch einmal überreizt werden. Verkraftet das Online-System den Benutzeransturm des Weihnachtsgeschäfts? Klar, wir testen in jeder Iteration mit deutlich mehr parallelen Benutzern. Bleibt das System auch mit doppelt so vielen Produkten unter Last stabil? Sicher, wir testen mit etwa zehnmal mehr Produkten, als wir im Livesystem online haben. Absolute Sicherheit kann kein Last und Performance-Test bieten, denn kein Test – ob automatisiert oder nicht – kann die Bedingungen der realen Welt widerspiegeln. Wir können uns mit unseren Testszenarien der realen Welt aber angemessen nähern.

Tipp 7: Mikro-Performance-Tuning mit Unit-Tests

Performance-Probleme lassen sich bereits in den ersten Phasen der Entwicklung erkennen. Unit-Test-Frameworks erlauben häufig die Prüfung, ob eine Funktionalität im Unit-Test innerhalb einer vorgegebenen Zeitspanne abgeschlossen wird. Auf diese Weise kann auf der Mikroebene Performance-Tuning unter gleich bleibenden Testbedingungen durchgeführt werden. Die Performance-Auswirkungen von Änderungen im Code können so direkt festgestellt werden. Natürlich ersetzt diese Form der Unit-Tests keinen Performance-Test auf einer produktionsnahen Umgebung.

Produktion

Nach dem erfolgreichem Passieren der vorgelagerten Teststufen steht einer Überführung der Software in den produktiven Betrieb faktisch nichts mehr im Weg. Allerdings kann dieser Schritt in der agilen Welt weitere Prüfungen und Tests beinhalten, mit deren Hilfe die Qualität der Software sichergestellt wird.

Fazit

Durch den Einsatz von automatisierten Tests lassen sich agile Teams bedeutend entlasten, da sie schnelleres Feedback zur ihrem Einfluss auf die Qualität der Software bekommen. Dabei wird nicht nur ein Anwendungsfall nach der Spezifikation umgesetzt – anhand der automatisierten Tests wird täglich aufs Neue bewiesen, dass die Software diesen Anwendungsfall auch ausführen kann.

Einzelne Schritte in Richtung einer automatisierten Deployment-Pipeline lohnen sich auch für traditionelle Softwareprojekte. Dabei müssen diese nicht alles auf den Kopf stellen. Traditionelle Tester verwachsen mehr mit dem Entwicklerteam, indem sie ihnen mit Tipps und Tricks zur Seite stehen. Gleichzeitig bringen sie ihre Expertise bei den immer noch notwendigen manuellen Tests, zum Beispiel im Rahmen des Akzeptanztests und des UAT, ein. Qualität liegt in der agilen Softwareentwicklung im Interesse jedes einzelnen, da jeder seinen Einfluss darauf sehen kann. Eine gut geplante Automatisierung ist ein entscheidender erster Schritt dorthin. ■

Literatur & Links

[Adz11] G. Adzic, Specification by Example: How Successful Teams Deliver the Right Software, Manning 2011

[Che] Checkstyle 5.6, siehe: checkstyle.sourceforge.net/

[Dan] Dan North & Associates, siehe: dannorth.net/introducing-bdd/

[Fro] Froglogic GmbH, Squish – GUI Testing,-Website, siehe: froglogic.com/squish/

[Ham] Hamcrest.org, siehe: hamcrest.org

[Hum10] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, Addison-Wesley Longman 2010

[Jun] Junit, siehe: junit.org

[Sel] SeleniumHQ, siehe: seleniumhq.com

[Sou] Sourceforge.net, FindBugs, siehe: findbugs.sourceforge.net/