TESTAUTOMATISIERUNG MIT CONTINUOUS INTEGRATION: IST MANUELLES TESTEN NOCH SINNVOLL?

Testautomatisierung und „Continuous Integration" sind Bestandteile hoch-qualitativer Softwareentwicklung. Bei dieser kann erhebliche Effizienz für Projekte und den Entwicklungs-Workflow erzielt werden. Dieser Artikel beschäftigt sich mit den Möglichkeiten von Testautomatisierung und illustriert die möglichen Konzepte der einzelnen Testarten.

der autor

Mark Paluch
(mpaluch@paluch.biz)
ist in der Softwareentwicklung mit den Schwerpunkten „Clean Code" und „Architektur Java Enterprise Applications" tätig.

Mit wachsenden Entwicklungsteams, Aufgabenstellungen und Komponenten steigt zunehmend die Komplexität von Softwareentwicklungsprojekten. Die Codebasis gewinnt an Masse, die Anzahl der Testfälle steigt und die unterschiedlichen Fehlerfälle nehmen zu. Wird ein Fehler behoben, so taucht an einer anderen Stelle ein neuer Fehler auf. Neben Terminen und Systemabnahme ist die Qualität der Softwaresysteme ein wichtiger Bestandteil in Projekten. Daher muss ein bestimmtes Qualitätsniveau erreicht und beibehalten werden.

Eine Möglichkeit, dies zu erreichen, sind ausreichendes Personal und Ressourcen. Auf Dauer stellt sich diese Variante jedoch als sehr kostspielig heraus. Robert C. Martin dokumentiert die Kosten eines großen Internet-Unternehmens für die manuelle Testausführung von über einer Million US-Dollar pro Testphase (vgl. [Mar11]). Die hohen Kosten führen irgendwann dazu, dass das Management nach 50 % Kostenreduktion verlangt und sich der Testmanager fragt, welche Hälfte der Testfälle er auslassen soll.

Dabei unterteilen sich die Möglichkeiten in Projekten nicht nur in schwarz und weiß. *Continuous Integration (CI)* bietet die Basis, neben einfachen Unit-Tests (vgl. [Ham04]) auch komplexere Tests automatisiert auszuführen.

CI als Test

CI ist in Projekten schon lange nicht mehr das reine Zusammenführen von Code und Artefakten, wie es in im *eXtreme Programming (XP)* (vgl. [Bec99]) beschrieben wurde. Heute versteht ein Entwickler unter CI mindestens den automatisierten Erstellungsprozess (Build-Prozess), an dessen Ende ein Kompilat in paketierter Form entsteht. Die Orchestrierung des Build-Prozesses erfolgt entweder zeitgesteuert oder ereignisgetrieben (Quellcode wird in das Versionsverwaltungs-System eingecheckt). Auf einem Server oder in einer Server-Farm wird das entsprechende CI-System (vgl. [Wik]) betrieben, in dem ein oder mehrere CI-Jobs eingerichtet sind. Üblicherweise gibt es zwei Sorten von Jobs:

- Regulärer Build-Job
- Nightly oder Integration Build-Job

Die Aufgabe des regulären Jobs ist das Auschecken des Quellcodes und das Übersetzen in ein Kompilat. Idealerweise werden nach der Übersetzung bereits Unit-Tests ausgeführt. Die Übersetzung ist schon bereits eine Art Test. Der Test besteht dabei aus der Syntax- und Abhängigkeitsprüfung des Codes. Sind defekte oder nicht vorhandene Abhängigkeiten im Quellcode referenziert, so schlägt die Übersetzung fehl. Gleiches gilt für Syntaxfehler. Damit entspricht die Übersetzung einem Test, ob der Haupt-Entwicklungszweig instabil geworden ist. In Projekten, die keine CI einsetzen, fallen Instabilitäten erst beim nächsten Auschecken auf. Die Entwickler sind blockiert und der Haupt-Entwicklungszweig muss erst repariert werden, bevor die Entwicklung voranschreiten kann. CI deckt diese Probleme sehr viel früher auf, da reguläre Build-Jobs den Anspruch haben, innerhalb von wenigen Minuten durchzulaufen.

Ein *Nightly Build* ist eine Erweiterung regulärer *Build-Jobs*. Im Nightly Build werden zeit- und ressourcenintensive Aufgaben durchgeführt. Dazu gehören unter anderem Integrations- und Akzeptanztests, das Aufsetzen einer Ausführungsumgebung, statische Codeanalysen sowie Performancetests. Einige Projekte, insbesondere Großprojekte, sind aufgrund der Struktur und Organisation auf Nightly Builds beschränkt. Die Entwicklungsfortschritte von ca. einem Tag werden im Nightly Build vereint. Schlägt der Build fehl, so sind die Ergebnisse erst am nächsten Tag verfügbar. Der Nachteil – im Unterschied zum regulären Build – ist, dass die Entwickler den Aufgabenkontext erst wieder herstellen müssen. Kurze Builds, die wenige Minuten nach dem Einchecken ein Ergebnis liefern, erfordern keinen Kontextwechsel bzw. -aufbau, was zu einer höheren Entwicklungseffizienz führt.

Bei der Testautomatisierung im Rahmen von CI ist eine Teststrategie erforderlich. Das International Software Testing Qualifications Board (ISTQB, vgl. [Bla12]) spezifiziert zwar vier Strategien, die jedoch nur für eine funktionale Test-Herangehensweise anwendbar sind. CI erfordert ein Phasenmodell (vgl. [Hum10]) für die Ausführung. So müssen die Testarten aufeinander aufbauen: Performance-Tests tragen nicht sonderlich zur Qualitätsaussage bei, sofern ein Unit-Test fehlgeschlagen ist. Umgekehrt sagt ein Performance-Test nicht unbedingt etwas darüber aus, ob die Anwendung funktional korrekt implementiert ist. Die Testphasen sichern aufeinander aufbauend zunehmend die Qualität. Das übliche Phasenmodell sieht wie folgt aus (**siehe auch Abbildung 1**):

- Übersetzung
- Unit-Tests
- Paketierung
- Gegebenenfalls Verteilung auf die Laufzeitsysteme und Smoke-Test
- Integrationstests
- Akzeptanztests
- Performance-Tests
- Manueller Testlauf

Unit-Tests

Unit-Tests (realisiert mit [JUnit] oder [Tes]) sind die kleinste Testeinheit. In einem Unit-Test werden einzelne Einheiten (Module, Klassen, Komponenten) getestet, die die Integrität eines Moduls verifizieren. Unit-Tests sind im Gegensatz zu Integrations- oder Performance-Tests unabhängig von externen Laufzeitumgebungen und Res-

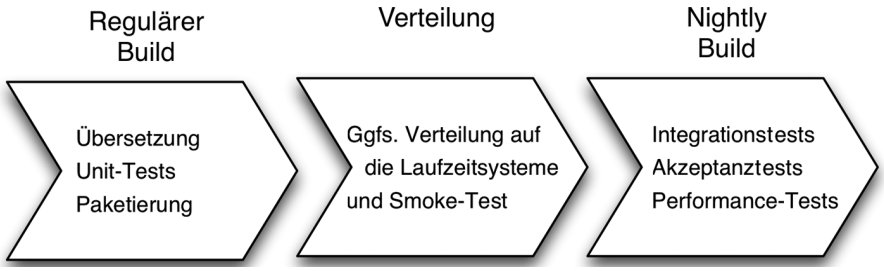

Abb. 1: CI-Phasen.

sourcen, wie z. B. Datenbanken oder entfernten Schnittstellen (vgl. [Ham04]).

Bei der testgetriebenen Entwicklung (vgl. [Bec02]) entstehen Unit-Tests direkt bei der eigentlichen Entwicklung. Diese Tatsache macht klar, dass die Entwickler Unit-Tests implementieren. Auch die Anpassung bestehender Tests gehört zum Arbeitspaket der Entwicklung. Die Automatisierung erfolgt durch den Einsatz von *Continuous Integration Software* (vgl. [Wik]) im Build. Im Code unterscheidet man zwischen Test- und Produktiv-Code. Werden die Regeln der testgetriebenen Entwicklung eingehalten, so ist eine 100%ige Testabdeckung sehr wahrscheinlich. Gute Softwareprodukte haben eine Testabdeckung von 80-90 %. Es gibt auch Produkte mit 100 % Testabdeckung, wie z. B. „FitNesse“ (vgl. [Fit]). Das 100 %-Ziel kann mit klassischen Mitteln nur durch enormen Aufwand erreicht werden und ist daher teuer. Üblicherweise wird bei ca. 80-90%iger Testabdeckung die Grenze erreicht, bei der Tests mehr Kosten als Nutzen verursachen. Trivialer Code kann effektiver durch eigene Mini-Frameworks getestet werden (z. B. generischer Test für *Getter* und *Setter*). Das Hauptaugenmerk bei Unit-Tests liegt daher auf der Anwendungslogik und nicht auf dem Test von externen Komponenten und trivialem Code.

Externe Abhängigkeiten werden durch Test-Dummys (*Mocks*) simuliert. Die Basis für die Testfälle liefern die Anforderungen an den Produktivcode selbst. Ziel ist es, jede Anweisung im Produktivcode entsprechend zu testen (direkt oder indirekt).

Bei einigen dieser Abhängigkeiten sind die Test-Dummys besonders teuer. Dort lohnen sich gegebenenfalls integrative Tests, bei denen diese Abhängigkeit zur Laufzeit des Gesamtsystems gegeben ist.

Unit-Tests benötigen nur einen geringen Umfang an Testdaten: mit ihnen werden die Funktionen/Methoden/Kontrollstrukturen des Softwaresystems getestet. Unit-Tests sind daher Tests auf sehr technischer Ebene. Unterschiedliche Konstellationen (z. B. auf Basis einer Entscheidungsmatrix) werden durch Akzeptanztests getestet. Unit-Tests bieten unterschiedliche Messwerte, die ausgewertet werden können:

- Anzahl der Tests
- durch Tests abgedeckte Quellcode-Zeilen bzw. Methoden (Testabdeckung)
- Laufzeit der Tests

Die Qualität der Tests hingegen lässt sich derzeit noch nicht messen. Eine geringe Testqualität wird sichtbar, sobald im Produktivcode eine Änderung stattfindet, jedoch kein Test fehlschlägt. Durch statische Codeanalysen kann zwar geprüft werden, dass gewisse Merkmale und Prüfungen in den Tests vorhanden sind, jedoch ist dies keine zuverlässige Methode. Je größer die Testabdeckung und die Anzahl der Tests sind, desto größer ist in der Regel auch die Qualität des Softwaresystems. Eine Garantie hierfür gibt es nicht.

Die Ausführung von Unit-Tests ist im Umfeld von CI ein Pflichtbestandteil. Unit-Tests sind idealerweise in wenigen Sekunden durchgeführt und sichern somit die Basisfunktionalität der Anwendung ab. Schlägt ein Unit-Test während der Ausführung fehl, deutet dies auf einen Defekt im Quellcode hin und der Build muss fehlschlagen.

Smoke-Tests

Smoke-Tests bezeichnen eine Testart, die nicht direkt mit der Entwicklung korreliert. Ein Smoke-Test verifiziert, dass ein Softwareartefakt in seiner Laufzeitumgebung erfolgreich installiert und gestartet werden kann. Meist verbergen sich dahinter das Hochfahren des Systems und einige wenige grundlegende bzw. missionskritische Testfälle (z. B. Login, Aufruf des Hauptmenüs, das Prüfen von Datenbank-Verbindungen und der Zugriff auf externe Dienste).

Smoke-Tests gehören zur Installation eines Systems und werden direkt im Anschluss an die Installation ausgeführt. Jede Ausführungsumgebung bringt ihre eigenen Daten mit sich (Server-Namen, Datenbank-Daten usw.). Testdaten haben umgekehrt in Produktivsystemen keinen Platz. Daher muss der Smoke-Test mit umgebungsspezifischen Referenzdaten umgehen können. Pro Umgebung muss für den Smoke-Test eine Menge von Daten vorgehalten und konfiguriert werden, gegen die die Tests ausgeführt werden können. Automatisierung eignet sich ideal, um exakte Vergleiche auszuführen. Eine menschliche Interpretation (beispielsweise der Ausgleich von Werten aufgrund von Ungenauigkeiten) der Testergebnisse gehört jedoch nicht zu den Aufgaben von einfachen Tests.

Integrationstests

Integrationstests sind die nächste Stufe beim Test von Software. Im Integrationstest agieren mehrere Komponenten miteinander. Wurde der Unit-Test mit erfolgreichem Ergebnis abgeschlossen, so kann der Integrationstest das Zusammenspiel der Komponenten verifizieren. Im abstrakten Sinne sind Integrationstests die Basis für die später folgenden Akzeptanz- und Performance-Tests.

Integrationstests setzen auf die Schnittstellen der Anwendung (Oberfläche, REST-/SOAP-Schnittstellen, Programmierschnittstelle) und bedienen die Anwendung entsprechend den Anwendungsfällen.

Für diese Tests ist ein Verständnis der Abhängigkeiten und Fachlichkeit erforderlich. Testdaten müssen gepflegt werden. Die Testdaten werden vor dem Testlauf in das System eingebracht, danach wird der Integrationstest ausgeführt. Ein Hochfahren und Initialisieren des Systems ist erforderlich. Dies reicht von mehreren Komponenten im Speicher über die Initialisierung eines Clients bis hin zum Aufsetzen eines entfernten Servers (z. B. Web-Anwendungen).

Integrationstests benötigen ein gewisses Maß an externen Abhängigkeiten (Datenbanken, Middleware, Laufzeitumgebungen usw.). Diese Abhängigkeiten müssen entsprechend auf den Test vorbereitet werden. Automatisierte Integrationstests müssen daher diese Abhängigkeiten berücksichtigen und einrichten. Die Daten dafür (Skripte, Daten) sind idealerweise Bestandteil des Softwareprojekts und werden mit jedem Projekt oder Änderungsauftrag an der Software gepflegt und gewartet. Tests, die auf die Validierung der Abhängigkeiten abgestimmt sind, bringen ebenfalls einen Mehrwert. Diese Tests prüfen die Konfi-

guration und Funktion der Abhängigkeiten im Zusammenspiel des Systems.

Die Basis für Integrationstests sind sowohl die öffentlichen Schnittstellen des Systems als auch die einzelnen Anwendungsfälle (Anforderungsspezifikation). Dies könnte im Kontext einer Kundenverwaltung beispielsweise die Neuanlage, Änderung und Suche eines Kundendatensatzes sein. Idealerweise werden die Integrationstests vom Entwicklungsteam auf Basis der Anforderungen implementiert.

Die Ausführung von Integrationstests benötigt zum Teil deutlich länger als die von Unit-Tests. Der Grund hierfür sind Latenz- und Verarbeitungszeiten des Gesamtsystems. Ähnlich wie bei Unit-Tests lässt sich auch bei Integrationstests die Testabdeckung messen. Obgleich dies im Gegensatz zu Unit-Tests komplexer ist, verliert die Anzahl der getesteten Quellcode-Zeilen zunehmend an Bedeutung. Der Grund hierfür ist, dass die Sicht auf das System fachlicher wird.

Integrationstests verhalten sich zu Unit-Tests komplementär. Funktionalität, die durch Unit-Tests nicht getestet werden konnte (z. B. aufgrund der fehlenden Datenbankabhängigkeit), kann in vielen Fällen durch Integrationstests abgedeckt werden. Integrationstests sind eine maschinelle Prüfung von exakt messbaren Werten. Ästhetische Aspekte oder die Wirkung von Oberflächen-Layouts sind Testfälle, die nicht durch Integrationstests geprüft werden können. Diese Black-Box-Tests sind und bleiben manuelle (menschliche) Aufgaben.

Integrationstests können auf unterschiedliche Art und Weise implementiert werden: als alleinstehende Ausführungseinheit, in externen Werkzeugen wie „SoapUI“ (vgl. [Soa]) oder basierend auf gängigen Test-Frameworks wie „JUnit“ (vgl. [JUn]) oder „TestNG“ (vgl. [Tes]). Liegt letzteres vor, können die Testergebnisse durch die vorhandene Integration in *CI- Software* direkt ausgewertet werden.

Scheitern Integrationstests, so heißt dies nicht zwangsläufig, dass ein Fehler im Produktivcode vorliegt. Oftmals sind externe Abhängigkeiten der Auslöser dafür: Festplatte voll, Server-Dienst hängt sich auf, Überlastung von Systemen. Diese Fälle führen zu Fehlalarmen und müssen entsprechend adressiert werden. Schlägt der Build diesbezüglich zu häufig fehl, verliert die CI an Aussagekraft für das System. Die Testfehler können jedoch auch Hinweise auf architektonische oder infrastrukturelle Hindernisse sein, die schon lange vor dem Produktivbetrieb gefunden werden können.

Akzeptanztests

Akzeptanztests führen einen Test aus Anforderungssicht durch. Entstehen im Zuge der Anforderungsanalyse Akzeptanzkriterien, so können diese für die Testfallgestaltung (vgl. [Bla09]) verwendet werden. Idealerweise sind die Akzeptanzkriterien entsprechend für den Test formuliert (siehe Abbildung 2).

Diese Tests bedienen – ähnlich wie Integrationstests – einen gewissen Anteil des Softwaresystems. Im Gegensatz zu Unit-Tests und Integrationstests kann bei der Automatisierung von Akzeptanztests eine Arbeitsteilung erfolgen: Die Entwicklung stellt Schnittstellen-Funktionalität bereit, die von Testern oder Requirements-Ingenieuren bedient werden kann. Die Ansteuerung der Schnittstellen erfolgt durch Frameworks wie „FitNesse“ (vgl. [Fit]), „JBehave“ (vgl. [JBe]) oder „Cucumber“ (vgl. [Cuc]). Diese Frameworks erlauben es, entweder mittels einer domänenspezifischen Sprache (DSL) oder in natürlicher Sprache Testfälle zu definieren. Das Besondere an der Formulierung ist, dass natürliche Sprachanteile zur Steuerung des Test-Frameworks verwendet werden können.

Tests mittels JBehave können auf die entsprechenden Schlüsselwörter reagieren und somit das entsprechende Test-Szenario ausführen.

Die Verlagerung des Abstraktionsniveaus entkoppelt die Tester und Testmanager von der Entwicklung. Ändern sich die Anforderungen, so können diese vor Entwicklungsbeginn entsprechend angepasst werden. Sobald die Implementierung des Produktivcodes abgeschlossen ist, laufen die Tests mit einem erfolgreichen Ergebnis durch.

Mit Akzeptanztests ist nicht nur der Test der Programmierschnittstelle oder externer Schnittstellen möglich. Akzeptanztests können Rich-Clients, Web-Browser via Selenium (vgl.[Sel]) oder Business-Prozess-Engines bedienen. In diesem Kontext wurde das Entwurfsmuster der Treiber-Abstraktion populär. Dieses Entwurfsmuster zieht eine Treiber-Schicht („Test-Driver“) zwischen Anwendung und Test ein, in der Anwendungsfälle gekapselt werden. Die Treiber bedienen die Anwendung und lösen die gewünschten Aktionen und Anwendungsfälle aus. Mit dieser Methode lassen sich Programmierschnittstellen-Änderungen weitestgehend kapseln, was zu einer geringeren Änderungshäufigkeit der Tests führt.

Werden ganze Test-Suiten mit Hunderten von Akzeptanztests ausgeführt, so benötigt ein Testdurchlauf deutlich länger als die Ausführung von Unit-Tests. Für CI bedeutet dies, dass die Akzeptanztests entweder an einen Nightly Build gekoppelt sind oder in einem eigenen Build-Durchlauf (Pre-Rollout-Testing) ausgeführt werden.

Story

```
Given a wizard named Harry
Given a wizard named Hermione
When the wizards meet
Then a spark should occur
```

Akzeptanz-Test Code

```
@Given ("a wizard named $theName")
public void givenWizard(String theName) {
…
}

@When ("the wizards meet")
public void whenMeet() {
…
}

@Then ("a spark should occur")
public void thenSpark() {
…
}
```

Abb. 2: Story und Story-Code realisiert mit JBehave.

Performance-Tests

Performance-Tests spiegeln eine ähnliche Facette wider wie Integrationstests. Geht es bei Integrationstests um Fachlichkeiten und die korrekte Integration, so konzentrieren sich Performance-Tests auf Durchsatz, Reaktionszeit und Lastgrenzen.

In den meisten Fällen benötigen Performance-Tests ein gewisses Maß an Testdaten. Diese Daten müssen vor Beginn der Tests in das System gebracht werden. Anschließend kann ein Testlauf gestartet werden. In den gängigen Testwerkzeugen werden Testpläne modelliert und Performance-Parameter definiert (Anzahl Zugriffe/Sekunde, Anzahl virtueller Benutzer). Die Art der Testaufrufe richtet sich nach den Anforderungen. Die Aufrufe können HTTP-Anforderungen, Messaging-/Datenbank-Tests oder sogar JUnit- oder TestNG-Tests sein. Ein typisches Open-Source-Werkzeug ist JMeter (vgl. [JMe]). Einige Testwerkzeuge unterstützen auch die Modellierung von Klick-Pfaden innerhalb von Oberflächen-Anwendungen.

Performance-Tests eigenen sich ideal, um die Lastgrenzen von Systemen zu evaluieren, Ausführungsumgebungen können gegen die Anforderungen getestet werden. Lastspitzen sowie Überlast-Szenarien lassen sich durch die Anpassung der Last-Parameter simulieren. Bei Performance-Tests spielt neben der Automatisierung auch die Umgebung eine entscheidende Rolle: Die Umgebung sollte eine größtmögliche Ähnlichkeit mit der Produktionsumgebung haben, insbesondere die Basis-Hardware. Ist die Umgebung beispielsweise als Cluster aufgesetzt, so sind im Test nicht unbedingt alle Cluster-Knoten erforderlich. Falls sich jedoch die Testsysteme im Betriebssystem unterscheiden, entsteht beispielsweise eine neue Abweichung. Ein anderes Beispiel sind abweichende Datenbank-Typen oder Provisionierungsmechanismen (manuell vs. automatisiert). Abweichungen dieser Kategorien führen zu unterschiedlichem Verhalten der Systeme.

Performance-Tests-Integrationen im Rahmen von CI sind weniger stark ausgeprägt als die Integration von beispielsweise JUnit oder TestNG. In Jenkins (vgl. [Jen]) oder Sonar (vgl. [Son]) können Performance-Tests mit entsprechenden Kennzahlen ausgewertet und nachverfolgt werden (siehe Abbildung 3 und [Wie10]).

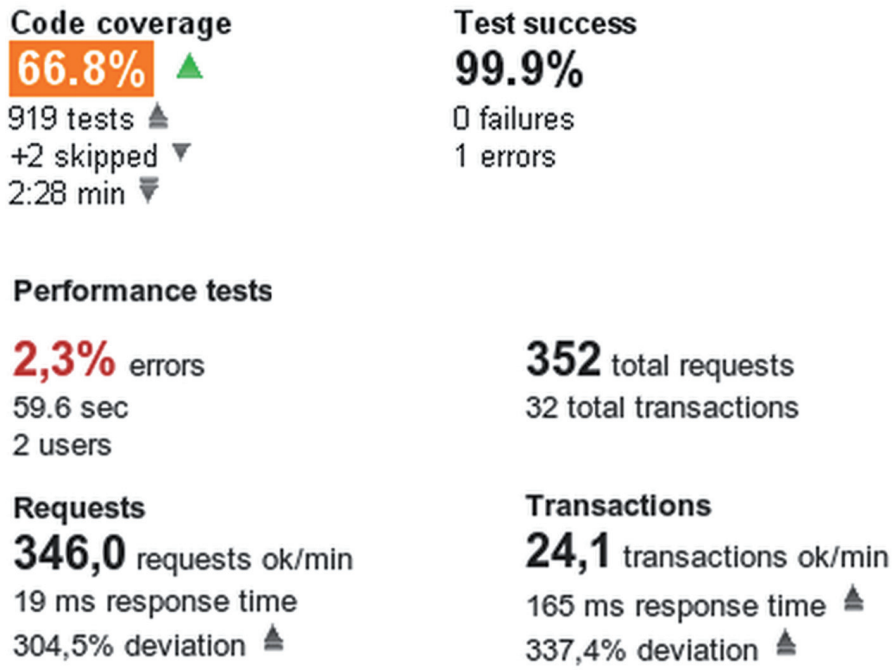

Abb. 3: Kennzahlen automatisierter Tests.

Staging

Eine Test-Automatisierung von Integrations-, Akzeptanz- und Performance-Tests ist nur möglich, wenn eine gewisse Kontrolle der externen Abhängigkeiten möglich ist. Nur so lassen sich verlässlich und wiederholbar Tests ausführen und das Ergebnis gegen ein erwartetes Ergebnis verifizieren. Testautomatisierung führt dazu, dass der Faktor Mensch bei der Ausführung immer weiter in den Hintergrund rückt. Somit sind menschliche Korrekturen oder Reaktionen auf bestimmte Testergebnisse nicht mehr möglich.

Funktionale und nicht-funktionale Tests benötigen eine isolierte und definierte Ausführungsumgebung. Diese Umgebung muss die Möglichkeit bieten, Testdaten in das System einzubringen: Datenbanken, Dateisysteme und vieles mehr. In einigen Fällen sind Testdaten auch an die Umgebung gebunden (z. B. Host-Namen). Dadurch verlieren die Tests ihre Portierbarkeit. Diese ist in der Regel auch nicht erforderlich, weil der Fokus des Tests die Funktionalität ist, nicht die Validierung von Software zu Umgebung. Findet innerhalb des Codes keine Unterscheidung nach der Umgebung statt, dann ist die Funktionalität gewährleistet, sofern ein Test in einer beliebigen Umgebung erfolgreich durchläuft. Es gibt immer wieder Code, der im Testsystem durchläuft, jedoch nicht im Produktivsystem. Gründe dafür sind, dass – je nach Umgebung – unterschiedlicher Code ausgeführt wird oder man sich auf Abhängigkeiten verlässt, die auf anderen Systemen nicht gegeben sind.

Die einzelnen Umgebungen müssen sich in der Kernkonfiguration entsprechen und dürfen keine gravierenden Abweichungen aufweisen: Unterschiedliche Konfigurationsdaten/Hardware, Datenmengen und Firewall-Regeln sind ein sehr häufiger Grund, warum der Test auf einer Testumgebung erfolgreich war, jedoch in der Produktivumgebung scheitert.

In größeren Projekten bzw. Unternehmen gibt es häufig eine Reihe von Testsystemen, die von unterschiedlichen Teams benutzt werden. Die Ausführung der eigenen Tests auf gemeinsam genutzten Systemen führt sehr wahrscheinlich zu Konflikten. Die belegten Ressourcen fehlen bei anderen Teams oder Kollegen. Eine weitere häufige Konsequenz ist, dass die Tester keine isolierten Tests durchführen können, da der Zustand von Daten durch externe Tests verändert wird. Ein möglicher Ausweg ist die eigene Umgebung mit allen erforderlichen Abhängigkeiten. Dieser Ansatz isoliert die beteiligten Teams und Komponenten, ist jedoch wartungs- und ressourcenintensiv. Eine andere Möglichkeit besteht darin, externe Abhängigkeiten als Test-Dummys (*Mocks*) zu simulieren. Dabei können z. B. SOAP/REST/EJB-Schnittstellen gemäß Schnittstellenvertrag implementiert werden, die sich je nach Anfrage unterschiedlich verhalten. Mit dieser Variante kann das Verhalten der Schnittstellen nachempfunden werden.

Möglichkeiten und Ausblick

Testautomatisierung in Kombination mit CI gewährleistet eine kontinuierliche Qualitätssicherung (vgl. [Duv07]). Die Regressionstests decken bei korrekter Anwendung frühzeitig Probleme und Fehler auf. Je höher die Automatisierung

und Testabdeckung, desto schneller, früher und kostengünstiger ist die Qualitätssicherung durchgeführt. Je früher eine Rückmeldung von Qualitätsproblemen an die Entwicklung kommuniziert wird, desto früher können Probleme behoben werden. Abbildung 3 zeigt beispielsweise Kennzahlen einer automatisierten Testausführung an (Sonar-Dashboard). Die schnelle Rückmeldung bewirkt eine Verringerung der Codeänderungen pro Testlauf (vgl. [Hum10]). Die Änderungen können im Vergleich zum Vorgehen ohne CI besser isoliert und nachbearbeitet werden. Die Testautomatisierung kann sogar soweit führen, dass Tester und Testmanager zum untersuchenden Testansatz gelangen: Tests werden auf Basis von Risiko und ästhetischen Aspekten durchgeführt. Der Tester wird entlastet und steht nicht mehr als Engpass zwischen Entwicklung und Freigabe. Damit wird der Weg zu *Continuous Delivery* (vgl. [Hum10]) geebnet und der Entwicklungs-Workflow wird höchst effizient gestaltet (siehe Kasten 1).

Erfahrungen aus der Praxis

Projekterfahrungen aus der Praxis zeigen, dass sehr häufig ein Ideal-Szenario konzipiert wird. Scheitert dieses nicht im Ansatz, sondern erst bei der Umsetzung, ist der Glaube an die Testautomatisierung verloren. Testautomatisierung kann auch anders begonnen werden: Es muss nicht gleich von Anfang an jeder Aspekt abgedeckt sein. Oftmals reichen bereits kleine Automatisierungen, um einen Anfang zu schaffen. Stellt sich die Automatisierung als erfolgreich heraus, kann man aufgrund des Gelernten fortfahren. Dieses Wissen schafft die Grundlage, um fortgeschrittene Problemstellungen zu bewältigen. Scheitern auch die ersten Schritte der Testautomatisierung, so können die Gründe vielfältig sein: Fehlende Unterstützung bzw. Ressourcen, eine zu geringe Wissensbasis oder architektonische Schwierigkeiten. Diese Probleme sind jedoch nicht auf die Testautomatisierung zurückzuführen, sondern müssen durch die eigene Infrastruktur angegangen werden.

Die Testwerkzeuge sind vom jeweiligen Projekt abhängig. Ein Web-Projekt hat andere Anforderungen als ein Rich-Client oder eine Geschäftsprozess-Plattform. Deshalb muss jedes Projekt seine eigene Auswahl an Testwerkzeugen zusammenstellen (vgl. [Dus99]).

In einem meiner Projekte war das Middleware-Entwicklungsteam anfangs 50 % der Zeit mit Wartung sowie Fehleranalyse und -behebung beschäftigt. Die Gründe hierfür waren einerseits eine fehlende Testautomatisierung und andererseits eine fehlende Test-Infrastruktur. Wurden Teile der Software durch Entwickler im Rahmen ihrer Tätigkeit getestet, so hatte dies Auswirkungen auf ein anderes Testteam. Ein *End-to-End*-Test war bereits in der Entwicklung nicht möglich. Durch eine Qualitätsinitiative wurden Unit-Tests implementiert und um Akzeptanz-, Integrations- und Performance-Tests erweitert. In diesem Rahmen wurde eine Integrationsumgebung mit lauffähigen Softwarertefakten etabliert. Externe Dienste (Datenbanken, SOAP/REST/EJB-Services, ESB usw.) wurden als Test-Dummys („Mocks“) aufgebaut. Kurze Zeit später (nach ca. sechs Wochen) änderte sich das Verhältnis zwischen Wartung und Neuentwicklung dramatisch: Der Wartungsaufwand reduzierte sich auf ca. 10 %, in einer späteren Phase des Projekts verringerte sich der Aufwand erneut auf ca. 2 % des Gesamtaufwands.

Kasten 1: *Projekterfahrung.*

Automatisierte Tests gehören zum Aufgabenpaket der Entwicklung wie das Schreiben von Produktivcode. Ist ein Integrationstest nicht Bestandteil der Aufgabe, ist es wahrscheinlich, dass der Integrationstest auch nicht implementiert wird. Diese Änderung bedarf eines Umdenkens. Mehr Code benötigt anfänglich auch eine längere Implementierungszeit. Diese Investition zahlt sich im späteren Verlauf eines Projekts mehrfach aus durch geringere Fehlerzahlen und eine höhere Qualität. ■

Literatur & Links

[Bec02] K. Beck, Test Driven Development. By Example, Addison-Wesley 2002

[Bec99] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley Professional 1999

[Bla09] R. Black, Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing, Wiley 2009

[Bla12] R. Black, E. van Veenendaal, D. Graham, Foundations of Software Testing ISTQB Certification, Cengage Learning EMEA 2012

[Coh04] F. Cohen, Java Testing, Design, and Automation. A Guide to Maximizing Performance: From Unit Testing to Automated Web Test, Prentice Hall PTR 2004

[Cuc] Cucumber, siehe: cukes.info

[Dus99] E. Dustin, J. Rashka, J. Paul, Automated Software Testing: Introduction, Management, and Performance, Addison-Wesley Professional 1999

[Duv07] P.M. Duvall, S. Matyas, A. Glover, Continuous Integration: Improving Software Quality and Reducing Risk, Martin Fowler Signature Books 2007

[Fit] FitNesse, siehe: fitnesse.org

[Ham04] P. Hamill, Unit Test Frameworks, O'Reilley Media 2004

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

[JBe] JBehave, siehe: jbehave.org

[Jen] Jenkins, siehe: jenkins-ci.org

[JMe] JMeter, siehe: jmeter.apache.org

[Jun] JUnit, siehe: junit.org

[Mar11] R.C. Martin, The Clean Coder: A Code of Conduct for Professional Programmers, Prentice Hall 2011

[Sel] Selenium.org, siehe: seleniumhq.org

[Soa] SoapUI, siehe: soapui.org

[Son] Sonar, siehe: sonarsource.org

[Tes] TestNG, siehe: testng.org

[Wie10] S. Wiest, Continuous Integration mit Hudson/Jenkins: Grundlagen und Praxiswissen für Einsteiger und Umsteiger, dpunkt.verlag 2004

[Wik] Wikipedia, Comparison of continuous integration software, siehe: en.wikipedia.org/wiki/Comparison_of_continuous_integration_software