Mit Stellenanzeigen

iX DEVELOPER

3/2013

Zum Download:

Über 6 GByte Software für Entwickler

Visual Studio Team Foundation Server 2012 mit Update 2, Eclipse, NetBeans, IntelliJ IDEA, Qt, Perforce, CI-Server, Build-Tools, viele kleine Helfer, Buchauszüge, 39 Episoden des Software-ArchitekTOUR-Podcasts

Bessere Software

Qualität für den gesamten Software-Lebenszyklus:

Continuous Delivery, Agile ALM, DevOps

Release-Management:

Status quo der Versionskontrolle

Einfache Tool-Integration

Zeitgemäße Builds

Test-Driven Development:

Die wichtigsten Testarten

Qualität durch Code-Reviews

Requirements Engineering:

Wie gute Anforderungen entstehen

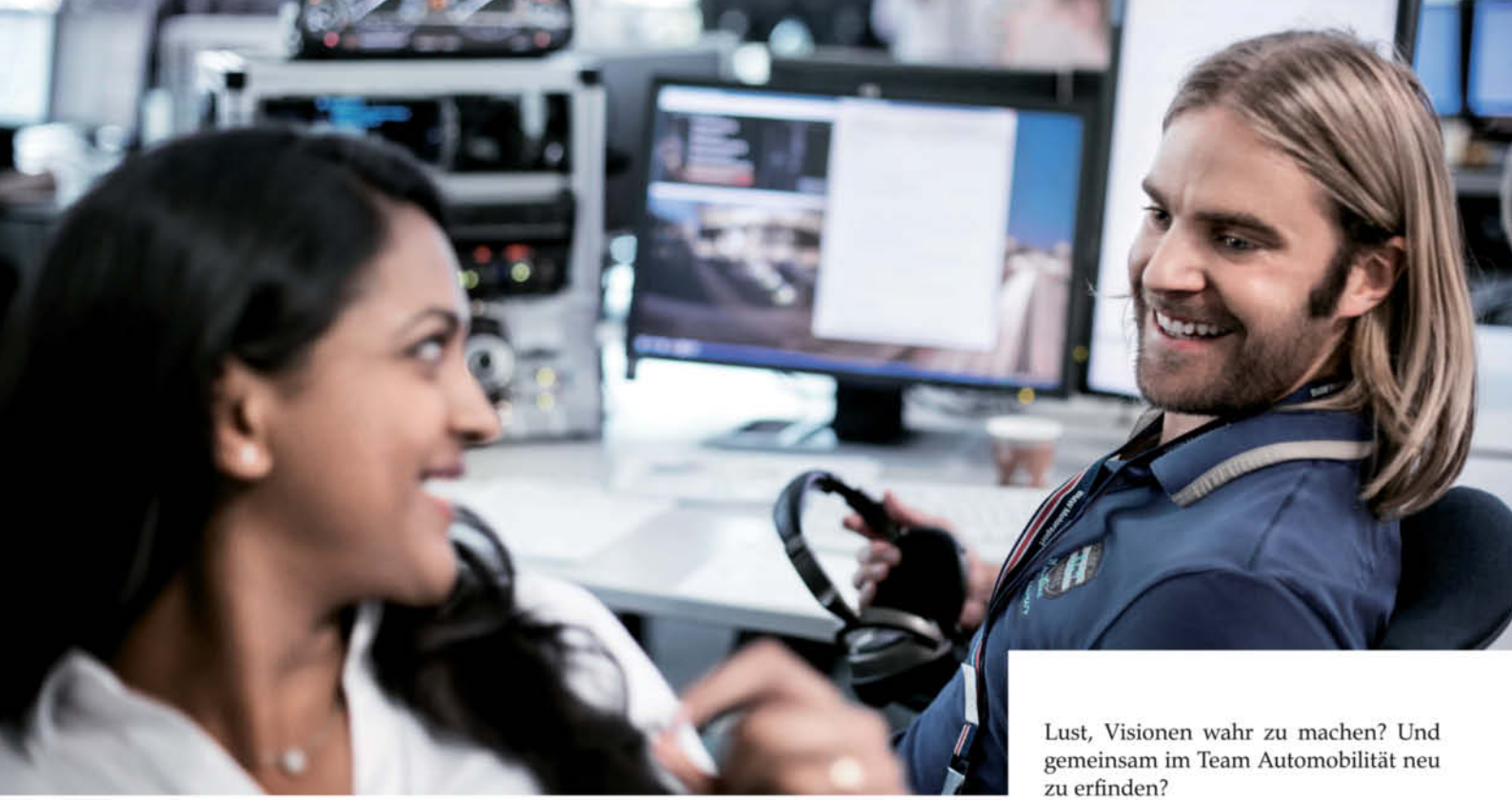
Dokumentation in agilen Projekten

Qualitätssicherung:

Was sich durch die Cloud ändert

Agilität und Prozessreifegrad-Modelle

präsentiert von: heise Developer www.heise-developer.de



Lust, Visionen wahr zu machen? Und gemeinsam im Team Automobilität neu zu erfinden?

AUTOMOBILGESCHICHTE SCHREIBT MAN HEUTE IN C++.

TEILEN SIE MIT UNS IHRE LEIDENSCHAFT FÜR ZUKUNFTSTECHNOLOGIEN.

90 % aller Innovationen im Automobil basieren auf Elektronik und Software. Deshalb sind gerade hier große Freiräume und vernetztes Denken gefordert, um wirklich Neues zu erschaffen. Bis hin zu flexiblen Arbeitsmodellen, die es unseren IT-Spezialisten ermöglichen, unkonventionelle Wege zu gehen. Um weiterhin die Grenzen des bisher Möglichen zu verschieben, suchen wir zur Verstärkung unseres Software-Labs in Ulm mehrere ...

Software Developer (m/w)
Stellenreferenz: 76734

Software Developer (m/w) Mobile Applications
Stellenreferenz: 78682

Software Developer (m/w) Innovative Software Solutions
Stellenreferenz: 78679

Software Developer (m/w) Driving Dynamics Software
Stellenreferenz: 78687

Erfinden Sie gemeinsam mit uns die Zukunft des Automobils. Als Software-Entwickler bei der BMW Car IT GmbH programmieren Sie nicht nur richtungsweisende Innovationen – Sie progammieren Emotionen. Schwerpunkte der Entwicklung bilden Softwareprojekte für die Fahrzeugvernetzung und das integrierte Datenmanagement sowie Embedded-Software und Linux-basierte Open Source Software. Wenn Sie Begeisterung für zukunftsorientierte Technologien und ein abgeschlossenes Informatikstudium mitbringen, sind Sie in unserem Think Tank genau richtig. Wir bieten Ihnen die Sicherheit und Perspektiven der BMW Group – und genügend Freiraum für Ihre Ideen und Entwicklung.

Bewerben Sie sich online auf diese Stelle unter bmwgroup.com/softwarejobs. Dort erfahren Sie auch mehr über uns als Arbeitgeber, unsere Einstiegs-Programme und weitere Stellenausschreibungen.
Oder gehen Sie über den QR Code direkt ins Bewerbungs-System.

www.facebook.com/bmwkarriere

BMW GROUP
Recruiting

Rolls-Royce
Motor Cars Limited

Neue Dreifaltigkeit

„Rückruf von UMTS-Handys, Softwarefehler kann Daten löschen." Oder: Ein Automobilhersteller „ruft weltweit etwa 2,5 Millionen Autos in die Werkstätten zurück. Der Grund ist ein Softwareproblem, das zu Schäden am Automatikgetriebe führen kann". – Das sind zwei reale Beispiele aus einem Artikel dieses Sonderhefts zur Softwarequalität. Ist es so weit gekommen, sind offensichtlich Fehler übersehen worden, die im Nachgang hohe Kosten verursachen. Wie diese Softwarefehler zustande gekommen sind oder was die verantwortlichen Entwickler übersehen haben, spielt hier erst mal keine Rolle. Es ist aber zu vermuten, dass die Schäden vermeidbar gewesen wären.

Vielleicht sind die obigen Warnmeldungen der Tatsache geschuldet, dass sich die Anwendungsentwicklung, zuerst beeinflusst durch das Web 2.0, dann durch die App-Entwicklung für mobile Geräte und neuerdings die Cloud, zunehmend verändert hat. Und zwar von einer als starren Einheit gesehenen Softwareentwicklung zu einer sich stetig verändernden Instanz, die von häufigen Releases mit kurzen Intervallen und Sprints durchsetzt ist. So gut diese Veränderung im Sinne der Wettbewerbsfähigkeit sein mag, birgt sie auch Risiken hinsichtlich der Qualität von Software.

Hier kommen die zentralen Schlagwörter dieses Sonderhefts ins Spiel: Das Dreigestirn aus Agile ALM, Continuous Delivery und DevOps ist in jüngster Zeit ins Zentrum des Interesses vieler Softwareentwickler gerückt. Von den damit verbundenen agilen Tools und Prozessen versprechen sich Programmierer, aber auch Administratoren, die Wünsche ihrer Kunden schneller als bisher und trotzdem qualitativ hochwertig bedienen zu können. Sie implizieren Vorgehen, die näher an der Entwicklerpraxis als das klassische schwergewichtige Application Lifecycle Management sind. Das macht sie so attraktiv, wie ich als Ausrichter der Continuous Lifecycle 2013 anhand der großen Zahl an Einreichungen von Vorträgen und der Flut an Mails von am Thema der Konferenz Interessierten feststellen konnte. Denn die Mitte November stattfindende Veranstaltung wird sich genau mit diesen drei Schlagwörtern auseinandersetzen.

Insbesondere Continuous Delivery führt die bislang aufwendigen und langsamen Build-, Integrations-, Test- und Verifikationsprozesse zu einer schlanken und schnellen sogenannten Development Pipeline zusammen. Vereinfacht ausgedrückt soll das Vorgehen die Veröffentlichung von Software so stark automatisieren, beschleunigen und vereinfachen, dass sie häufig, ja sogar kontinuierlich erfolgen kann, ohne dadurch an Qualität zu verlieren.

Was sich so einfach anhört, hat natürlich seine Tücken in der Praxis. Deswegen möchten wir Ihnen mit diesem Heft eine Anregung bieten, Ihre Entwicklungs- und Qualitätssicherungsprozesse zu prüfen und zu überdenken sowie bei Bedarf anzupassen – im Sinne Ihrer Arbeit, Ihrer Projektbeteiligten und Ihrer Kunden.

iX und *heise Developer* wünschen viel Spaß bei der Lektüre dieses mittlerweile sechsten *iX Developer*-Sonderhefts.

ALEXANDER NEUMANN

Softwarequalität

Softwarequalität ist vielschichtig, erst recht bei Betrachtung über den kompletten Lebenszyklus. Sie lässt sich nur mit einem ganzheitlichen Ansatz optimieren. Wichtig sind hier die geeignete Auswahl von Maßnahmen, eine offene Herangehensweise und eine veränderte Mentalität. Am besten gelingt das durch die gemeinsame Ausrichtung von Zielen, Prozessen und Werkzeugen.

ab Seite 7

Lifecycle-Tools

Mit neuen Werkzeugen wie Git, Gerrit und Gradle, aber auch mit zeitgemäßen Techniken zur Tool-Integration erfährt Application Lifecycle Management einen neuen entwicklerzentrierten Anstrich, durch den das bislang meist als schwergewichtig erachtete ALM endlich auch Unterstützung durch die maßgeblich in die Softwareentwicklung involvierten Akteure erhält.

ab Seite 43

Continuous Delivery

Build-Management und Continuous Integration wurden unlängst um neue Ideen ergänzt, die unter dem Namen Continuous Delivery zusammengefasst werden. Zusätzlich trifft man nun vermehrt auf Begriffe wie Deployment Pipeline und DevOps. Alle haben das Ziel, schnellere und bessere Releases zu ermöglichen.

ab Seite 81

Test

Neben den weitverbreiteten Unit-Tests gibt es eine Reihe weiterer Testgattungen, die zunehmend an Verbreitung gewinnen. Sie einzusetzen ist die eine Sache, wichtiger hingegen ist die dahinterstehende Testidee oder gar die richtige Mischung von beispielsweise Unit-, Regressions-, Akzeptanz- und Fuzzing-Tests.

ab Seite 107

Testing

Diverses

Sonstiges

Alle Links: www.ix.de/ix1315SSS Artikel mit Verweisen ins Web enthalten am Ende einen Hinweis darauf, dass diese Webadressen auf dem Server der iX abrufbar sind. Dazu gibt man den iX-Link in der URL-Zeile des Browsers ein. Dann kann man auch die längsten Links bequem mit einem Klick ansteuern. Alternativ steht oben rechts auf der iX-Homepage ein Eingabefeld zur Verfügung.

Auf der Heft-DVD

Visual Studio Team Foundation Server 2012 mit Update 2

Microsofts Entwicklungsumgebung für das Application Lifecycle Development bietet Teams ein umfassendes Verwaltungstool-Paket für den gesamten Lebenszyklus von Anwendungen. Das auf der DVD bereitgestellte ISO ist eine 90-tägige Testversion.

Außerdem:

- Eclipse 4.2.2 Classic: das zweite Service Pack der aktuellen Version der Java-Entwicklungsumgebung
- NetBeans 7.3: Oracles Open-Source-IDE in der umfassenden Ausgabe zur Entwicklung für Java, C/C++ und PHP
- IntelliJ IDEA 12.1.3: JetBrains' quelloffene Community Edition der als „polyglott" bezeichneten Entwicklungsumgebung
- Qt Framework 5.0.2/Qt Creator 2.7.1: das C++-Framework zur plattformübergreifenden Oberflächenentwicklung samt darauf abgestimmter IDE

Tools

Build

Apache Buildr, Gradle, SBT (Simple Build Tool), Apache Maven

Continuous Integration

CruiseControl, Eclipse Hudson, Jenkins

Versionskontrolle

Git, Mercurial, Bazaar, Subversion

Perforce

Auf der DVD findet man die Client- und Server-Komponenten, die den Kern eines Perforce-Versionsmanagements ausmachen. Der Client ist die GUI P4V. Es handelt sich um die vollständige Version der Software ohne funktionale Einschränkungen. Sie kann kostenlos auch für kommerzielle Zwecke verwendet werden, wobei sich maximal 20 Benutzer mit insgesamt 20 Workspaces einrichten lassen.

Kleine Helfer

Checkstyle, Findbugs, FitNesse, Gerrit, Hamcrest, Liquibase, NCrunch, PMD, Sandcastle, Sandcastle-Help-File-Builder, Vagrant

Literatur

Auszüge aus den dpunkt-Büchern:

- Git – Dezentrale Versionsverwaltung im Team, Grundlagen und Workflows (René Preißel, Bjorn Stachmann)
- Soft Skills für Softwaretester und Testmanager (Heinz Hellerer)
- ATDD in der Praxis – Eine praktische Einführung in die akzeptanzgetriebene Softwareentwicklung (Markus Gärtner)
- Testen in Scrum-Projekten – Leitfaden für Softwarequalität in der agilen Welt (Tilo Linz)
- Abenteuer Softwarequalität – Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement (Kurt Schneider)

SoftwareArchitekTOUR

39 Episoden des renommierten Podcasts auf heise Developer zu Themen wie:

- Ausbildung zum Architekten
- Die Rolle des Softwarearchitekten
- Webarchitekturen
- Product Line Engineering
- Cloud Computing
- Systems Engineering
- Systematischer Einsatz von Designtaktiken
- Testing
- Funktionale Programmierung
- Architektur-Reviews
- Architekturdokumentation
- Serviceorientierte Architekturen
- Modellgetriebene Softwareentwicklung
- Patterns
- Entwicklung für Embedded-Systeme mit mbeddr
- Barrierefreiheit
- ROCA (Resource-Oriented Client Architecture)
- Leichtgewichtige Webapplikationen (für Mobile)
- Business Process Management
- NoSQL

Hinweis für Käufer

- PDF- und iPad-Version: In der *iX*-App finden Sie einen Button zum Download des DVD-Images.
- PDF-E-Book: Folgen Sie im Browser der unter „Alle Links" angegebenen URL.

Alle Links: www.ix.de/ix1315006

Introducing into ...

Softwarequalität ist vielschichtig, erst recht bei Betrachtung über den kompletten Lebenszyklus. Sie lässt sich nur mit einem ganzheitlichen Ansatz optimieren. Wichtig sind hier die geeignete Auswahl von Maßnahmen, eine offene Herangehensweise und eine veränderte Mentalität. Am besten gelingt das durch eine gemeinsame Ausrichtung von Zielen, Prozessen und Werkzeugen.

Softwarequalität über den gesamten Anwendungslebenszyklus

Agil ausgerichtet

Michael Hüttermann

Die Integration aller Projektphasen, -aktivitäten und -rollen über den vollständigen Anwendungslebenszyklus ist essenziell, um die Softwarequalität hochzuhalten und die Wahrscheinlichkeit für einen Projekterfolg zu erhöhen. Doch wie bettet sich Softwarequalität in Application Lifecycle Management und Agile ALM ein?

Man unterscheidet interne und externe Softwarequalität. Erstere beschreibt die Beschaffenheit des Codes. Ist er nicht wart- und erweiterbar, tun sich zu einem späteren Zeitpunkt Missstände auf. „Technical Debt" beschreibt in diesem Zusammenhang die Schuld, die das Team zukünftig abtragen muss (s. Artikel auf S. 148). Aktivitäten wie Clean Code und Refactoring helfen, die Software in einer guten internen Qualität zu halten. Es gilt hierfür eine einfache Regel: Ohne interne Qualität gibt es mittelfristig nur wenig bis gar keine externe Qualität. Diese wiederum umfasst die dem Benutzer zur Verfügung stehenden Funktionen. Sie erkennen direkt: Dem schnellen Erfolg (beispielsweise eine rechtzeitige Lieferung einer Funktion) die interne Qualität zu opfern hat nicht selten zur Folge, dass weitere Lieferungen viel länger dauern oder die externe Qualität signifikant leidet.

Doch Qualität ist mehr, sie ist erst mal alles, wofür der Auftraggeber zahlt. Er entscheidet, was für ihn Qualität in einem konkreten Kontext bedeutet. Dennoch gibt es Unterschiede, und nicht jedes Qualitätsattribut hat den gleichen fachlichen Rahmen. Die Testabdeckung oder auch die Kaffeeversorgung sind beispielsweise unterstützende Attribute. Sie scheinen in vielen Fällen notwendig, aber nicht hinreichend für den Erfolg. Häufig ist es das Fehlen unterstützender Attribute, das erst Auswirkungen hat, und zwar negative. Die führenden Attribute sind die Eigenschaften, die tatsächlich über die Manövrierbarkeit des Projekts beziehungsweise der Software oder die „Time to market" Auskunft geben.

Da es nicht möglich ist, Qualität post mortem in eine programmierte Software zu injizieren, ist es umso wichtiger, dass qualitätssteigernde Maßnahmen von Anfang an Teil des Entwicklungsprozesses sind. Qualität sollte somit inhärenter Teil des Entwicklungs- und Auslieferungsprozesses und der entwickelten Software sein.

Qualität und Metriken

Die Definition von Qualität ist eine tragende Säule von ALM. Folgende Aufforderungen können für Klarheit sorgen:

1. Definiere, was Qualität im jeweiligen Kontext bedeutet.
2. Beschreibe Qualität mit Szenarien, die Qualität in Kontext setzen.
3. Unterscheide zwischen führenden und unterstützenden Qualitätsattributen.
4. Respektiere die Abhängigkeiten zwischen den Qualitätsattributen.
5. Identifiziere die Qualitätsattribute, die für den Erfolg von besonderer Bedeutung sind.
6. Verflechte Qualität als inhärenten Teil des Prozesses, der alle Aktivitäten und Rollen verbindet.

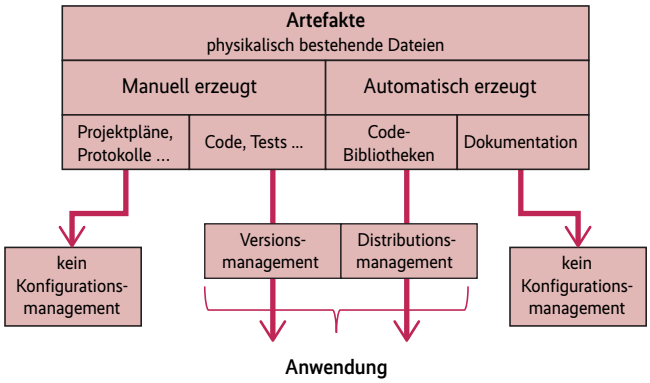
Unterstützende Attribute wie die Testabdeckung können wichtig sein sowie wertvolle Erkenntnisse liefern. Führende wie die Durchlaufzeit verbinden Entwicklung beziehungsweise Betrieb und geben aufschlussreich Auskunft über Wettbewerbsfähigkeit, Flexibilität und „Time to market".

Die Nutzung einer Versionskontrolle zum Vorhalten von Code, Build-Skripten et cetera gehört mittlerweile zum Standard. Ein Software Configuration Management (SCM) beschreibt Pro-

zesse und Verfahren, wie mit Änderungen (engl. Changes) umzugehen ist (s. Abb. 1). SCM ist weit mehr als ein Versionskontrollsystem (VCS). Es ist eine eigene Disziplin mit Prozessen und Werkzeugen. Die vier Kernaufgaben des SCM sind:

- die Identifikation von Konfigurationseinheiten
- die Kontrolle (im Sinne unterstützender Begleitung) von Änderungen an Konfigurationseinheiten
- Audits zur Konfiguration überwachen Korrektheit, Vollständigkeit und Konsistenz von Baselines
- Status Accounting: das Reporting der Konfigurationseinheiten über ihren Lebenszyklus hinweg

Begriffe wie Changesets (in sich konsistente, logisch verbundene Änderungen im VCS), Baselines (spezielle, konsistente Softwareversionen, auf denen sich reproduzierbar Operationen und Änderungen anwenden lassen) und Konfigurationseinheiten (Einheiten, die ausgeliefert oder bei der Erstellung dieser Einheiten eingebracht werden) prägen das SCM vieler Projekte. Es hilft, die Rahmenbedingungen der Projekte zu verbessern. Eine Konzentration allein auf technische Aspekte und Konfigurationseinheiten, so wie es im traditionellen SCM der Fall ist, greift nicht selten zu kurz, obgleich das eine essenzielle Voraussetzung ist. Entscheidende Aspekte kommen so zu kurz. Beispielsweise der für einen Kunden wichtige Aspekt, in kurzen Zyklen qualitativ hochwertige Software bereitzustellen.



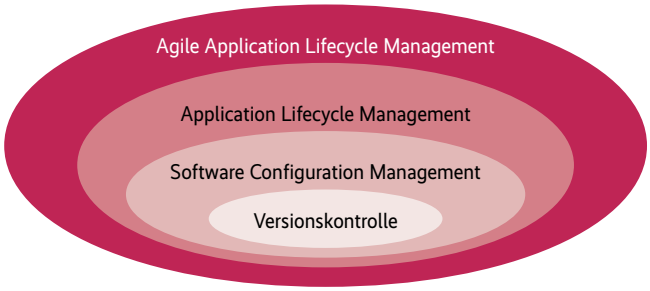
Systematisches Software Configuration Management handelt vom Umgang mit automatisch und manuell erstellten Konfigurationseinheiten, insbesondere ob und wo sie reproduzierbar gespeichert werden (Abb. 1).

Systematisches Change-Management: Von SCM zu Agile ALM

ALM erweitert ein SCM auf den kompletten Softwarelebenszyklus (s. Abb. 2). Mit der Definition von Anforderungen startend, über Code, Test und Auslieferung in den Betrieb und die Wartung verbindet ALM alle Projektrollen und -aktivitäten sowie organisatorische Einheiten. Der ganzheitliche, integrative Ansatz lässt sich durch ein Werkzeug allein (ALM-Suite) nicht meistern. Vielmehr sind Barrieren zu überwinden, die in historisch gewachsenen Projekt- und Abteilungskonstellationen entstanden sind. Nicht selten sind es dabei die führenden Fachkräfte von gestern, die sich über die Jahre suboptimal fortgebildet haben sowie mit neuen Ansätzen und einem geänderten Anforderungsprofil konfrontiert werden. Die IT-Welt dreht sich weiter, und das Altbewährte zu verteidigen, ohne offen für Neues zu sein, reduziert die Wettbewerbsfähigkeit.

ALM verknüpft Projektphasen wie den Umgang mit Anforderungen, die Entwicklung, das Testen, die Produktivsetzung und die Wartung. Es begleitet die Aktivitäten der jeweiligen Stakeholder und sorgt für die Integration mit denen der anderen Stakeholder. Ein agiles ALM implementiert und verknüpft Disziplinen wie Change-, Build-, Requirements-, Integrations-, Release- und Test-Management nun mit agilen Herangehensweisen. Dabei kommen Vorgehen wie Scrum und Kanban sowie „leichtgewichtige“, zumeist Open-Source-Werkzeugketten zum Einsatz.

Agiles ALM fördert Werte wie Respekt, Offenheit, Strategien wie kontinuierliche Integration und Auslieferung sowie Methoden à la Scrum oder Kanban. Es lässt sich als ein angereichertes ALM verstehen, das wiederum auf ein SCM mit einfacher Versionskontrolle aufbaut. Ob ALM nun in Generationen eingeteilt, wie manchmal praktiziert, und dabei zusätzlich mit „+“-Zeichen oder Ziffern versehen wird, ist unerheblich. „Agilität“ muss nicht selten als Alibi für ein hemdsärmeliges, unsystematisches Vorgehen herhalten. Agiles ALM ist ein Weg, agiler Softwareentwicklung Struktur zu geben. Es lässt sich als Verfahren und Mentalität beschreiben, um

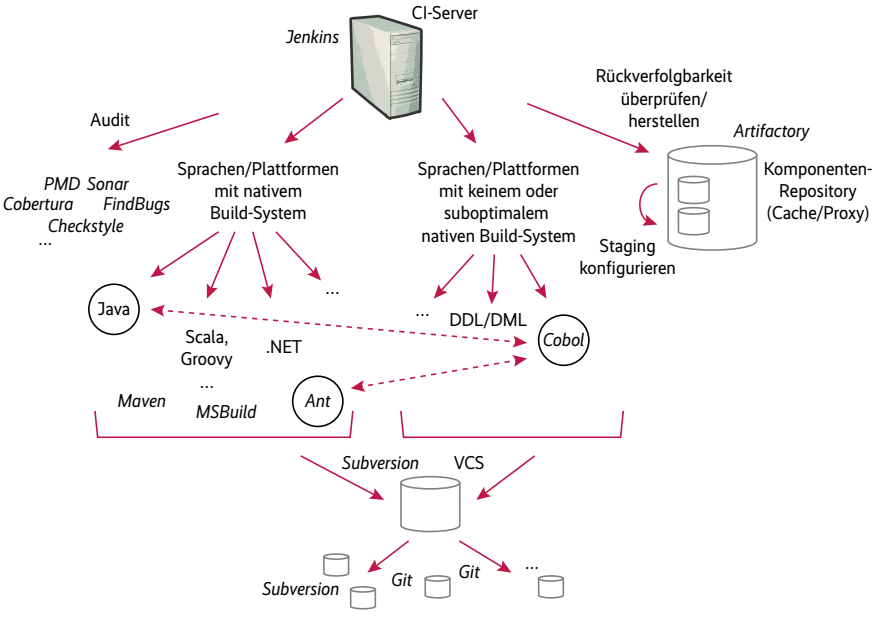


Die ALM-Zwiebel: SCM, die Basis für ALM, basiert auf Versionskontrolle. Agiles ALM erweitert ALM mit agilen Werten und Verfahren (Abb. 2).

- Prozess-, Technik- und Organisationsbarrieren zu überwinden und Silos zu minimieren,
- verschiedene Artefakttypen sowie Projektphasen und -rollen zu verbinden,
- leichtgewichtige, optimalerweise Open-Source-Werkzeugketten zu integrieren,
- die Beziehung zwischen Artefakten rückverfolgbar zu machen (z. B. durch Task-basierte Entwicklung) und
- Task-basierte Aktivitäten zu ermöglichen, die an konkreten Anforderungen ausgerichtet sind, wobei alle Änderungen zu konkreten Anforderungen zurückzuverfolgen sind.

Schnell ein ALM auf das Werkzeugetikett, und schon kann es alle Probleme lösen – so einfach ist es leider nicht. Die „One size fits all“-Werkzeug-Suite gibt es leider ebenfalls nicht. Umso mehr ist es heute von Interesse, Werkzeuge einzusetzen, die über offene Schnittstellen und Architekturen verfügen. Sie lassen sich miteinander integrieren. Als Ergebnis daraus sind es zunehmend leichtgewichtige Werkzeugketten, die als ALM-Gesamtpaket eingesetzt werden. Statt sich von nur einem Softwarehersteller abhängig zu machen, ist häufig eine Integration von Werkzeugen wie Jenkins, Artifactory, Sonar, Gradle oder JIRA die bessere Option. Hier hat ein Umdenken eingesetzt, von proprietären Tools hin zu einer Open-Source-Herangehensweise, die eine stetige Beobachtung des Markts nach den aktuellen Best-of-breed-Werkzeugen voraussetzt.

Qualität hochzuhalten erfordert, gemeinsame Anreize zwischen Entwicklung und Betrieb zu schaffen sowie übergreifende Metriken zu nutzen. Ansätze dazu sind der Fokus auf die Zykluszeit („cycle time“, manchmal mit „lead time“ gleichgesetzt, auch: Durchlaufzeit) und Stapelgröße („batch size“). ALM



Das Ökosystem der kontinuierlichen Integration. Heterogene Sprachen und Plattformen werden fortlaufend integriert. Quellen und Build-Skripte liegen in der Versionskontrolle, Binärdateien in einem Komponenten-Repository (Abb. 3).

möchte beides minimieren, es werden also häufiger Änderungen bereitgestellt, die dafür kleiner sind.

Die Zykluszeit beschreibt die Zeit vom Start bis zum Ende des Prozesses, beispielsweise der Umsetzung einer Änderung bis hin zu deren Produktivsetzung. Wenn von einer Zykluszeit von wenigen Sekunden die Rede ist – Web-2.0-Unternehmen stellen alle paar Minuten Änderungen in Produktion – ist damit nicht gemeint, dass in wenigen Minuten Anforderungen erhoben und beschrieben, Lösungen spezifiziert und umgesetzt sowie Implementierungen ausgerollt werden. Vielmehr ist die Zykluszeit in der Regel die Messzahl, wie häufig sich Änderungen auf eine Software in Produktion systematisch aufspielen lassen. Eine geringe Qualität ist dabei, wenn die Zykluszeit eher im Monats- als im Minutenbereich liegt.

Immer und immer wieder integrieren

Etwas allgemeiner: Unter der Zykluszeit versteht man die Zeit, die vergeht, wenn ein Produkt durch eine Arbeitsstation läuft, bis der Prozess von vorne beginnt. Die Kennzahl beschreibt, wie oft Arbeitsaufträge an eine Produktionseinheit gegeben werden. Sie ist ein Messwert, der abteilungsübergreifend ausdrückt, wie schnell und manövrierfähig die Unternehmung ist. Die Zykluszeit kommt als Basismetrik für Planung und Prozessverbesserung zum Einsatz. Sie umfasst gewöhnlich die Zeit zwischen einer Eintrittsbedingung und der Produktivstellung der daraus resultierenden Änderung. Die Bedingung kann sein: Identifikation eines Bugs in der Software in Produktion, ein neuer Geschäftsfall oder ein neu entwickeltes Feature. Es ist wichtig, die Stapelgröße kontextsensitiv zu definieren. In enger Abhängigkeit zur Durchlaufzeit steht die Stapelgröße. Sie umfasst den Umfang der produktiv gestellten Änderung.

Kleine Änderungen lassen sich schneller bereitstellen als größere. Das Risiko fehlerhafter Verteilungen und Bugs in der Software ist durch häufigere Bereitstellungen kleiner. Gründe umfassen: besser eingeschliffene Prozesse, handhabbare Änderungen (mit einfacher Rückverfolgbarkeit von Fehlern) und komfortablere Auslieferungsprozesse bis hin zur Wegoptimierung von Rollbacks, indem Änderungen inklusive Bugfixes und Produktionsdefekte ausschließlich „vorwärts“ ausgerollt werden. Die Vorbedingung für gute Zykluszeiten und Stapelgrößen ist die kontinuierliche Integration.

Kontinuierliche Integration beschreibt die Fähigkeit, jederzeit einen kompilierbaren, lauffähigen Stand aus der Versionskontrolle auschecken zu können, mit der Prämisse, dass die Entwickler jeden Tag, am besten mehrfach, ihre Änderungen der zentralen Versionskontrolle hinzufügen. Damit ist nicht gemeint, dass die Entwickler einen Branch-Föderalismus betreiben, sondern die Änderungen in einem zentralen Strom der Versionskontrolle integriert werden. Ob dabei eine zentrale (z. B. Subversion) oder dezentrale (z. B. Git, Mercurial, Bazaar) Versionskontrolle zum Einsatz kommt, ist zunächst mal nebensächlich. Abbildung 3 zeigt übliche Aspekte bei der kontinuierlichen Integration.

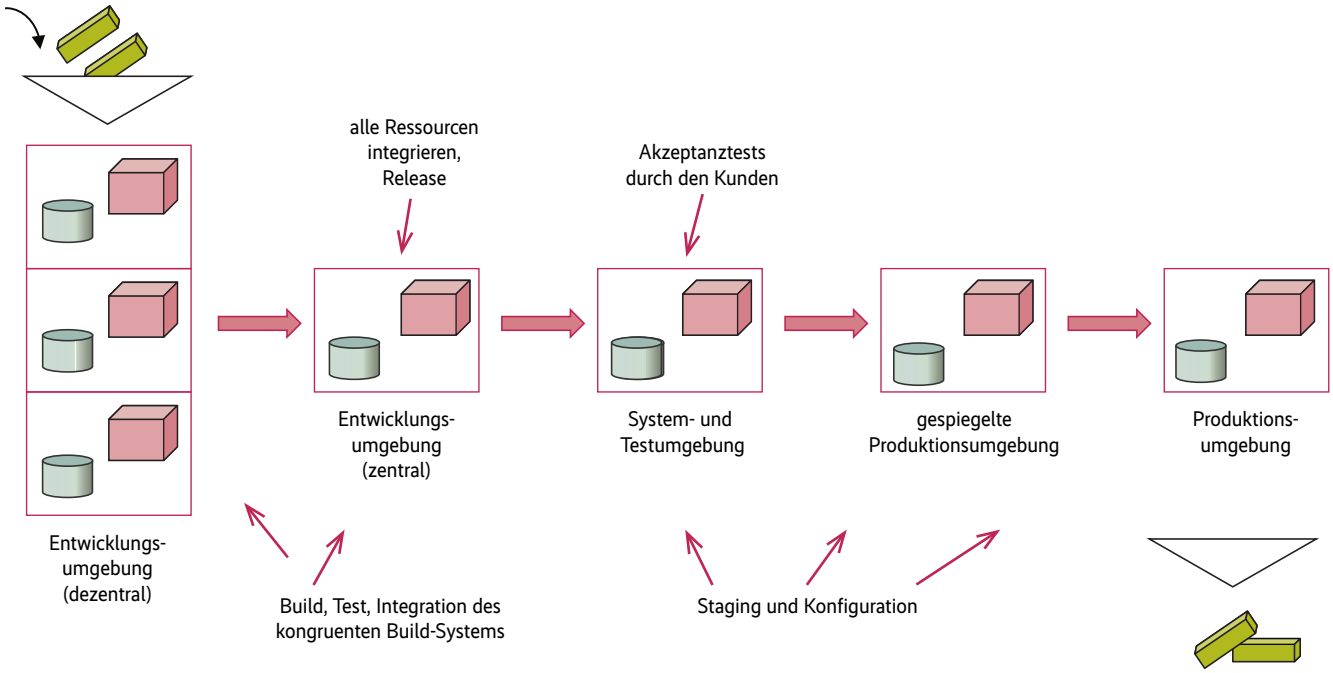
Die Software zu „bauen“ wird häufig mit Build-Management umschrieben. Kompilieren, Linken, Paketieren ist in der Regel dessen Bestandteil, und das hoffentlich für alle Artefakttypen. Für viele Typen stehen native, also plattformspezifische Build-Werkzeuge zur Verfügung. Für Artefakttypen, für die es keine nativen Build-Werkzeuge gibt, lassen sich andere geschickt einsetzen. Beispielsweise kann Cobol- oder Microsoft-Code gänzlich ohne proprietäre, kommerzielle Software gebaut werden. Auch das Deployment auf Testumgebungen und Testing wird häufig zum Build-Management dazugezählt, wenn es nicht eine eigene Disziplin darstellt.

Mit Softwarearchitektur gepaart

Build-Management hat eine starke Tangente zur Softwarearchitektur. Liegt ein architektonischer Monolith vor, ohne jegliche Schichten und Komponenten, ist er in aller Regel auch monolithisch zu bauen. Das resultiert in langen Build-Zeiten und schlechter Wart-/Erweiterbarkeit. Außerdem lassen sich einzelne Teile der Software nicht separat bauen. Eine Build-Engine, eine Art Web-2.0-kompatible Version einer Cron-Tab, hält das Ganze zusammen. Sie baut und testet regelmäßig die Software und verteilt sie auf Zielumgebungen. Sie visualisiert Ergebnisse und ermöglicht die Definition von Abhängigkeiten im Build-Prozess. Vertreter dieser Werkzeugzunft sind beispielsweise Jenkins, Hudson, TeamCity, CruiseControl und Bamboo.

Eine Schwester der kontinuierlichen Integration ist die kontinuierliche Inspektion. Wie oben festgestellt, lässt sich Qualität nicht post mortem in die Software einspeisen. Stattdessen ist Qualität inhärenter Bestandteil im Prozess, beispielsweise durch Einsatz manueller oder automatischer Audits. Es kommen Werkzeuge zum Einsatz wie das frei verfügbare SonarQube (ehemals Sonar).

Die zweite Schwester der kontinuierlichen Integration ist das kontinuierliche Deployment, also das fortlaufende Verteilen der Software auf Zielumgebungen. In diesem Kontext findet auch häufig die kontinuierliche Auslieferung (Continuous Delivery) Erwähnung. Bei ihr geht es darum, dass es die technische Mög-



Staging: das systematische Durchlaufen einer Pipeline von Start (Anforderungen werden definiert) bis Ende (Auslieferung der Umsetzung) (Abb. 4)

lichkeit gibt, jederzeit eine Softwareversion in die Produktion zu bringen. Doch wie wird aus einem Deployment ein Release, oder ist das etwa das Gleiche?

Releasing: Die Software produktiv stellen

Das fachliche Releasing sollte mit dem technischen verknüpft werden. Fachliches Releasing sind die Aktivitäten, die Arbeitsergebnisse miteinander verknüpfen und diese dem Benutzer bereitstellen, wobei ein Deployment nicht zwingend zu einem Release führen muss und sich beides voneinander trennen lässt, zum Beispiel durch den Einsatz von Feature Toggles (siehe auch Artikel auf S. 88). Technisches Releasing umfasst die Aktivitäten, die die Artefakttypen technisch Richtung Produktion bringen. Es kommen dabei Rezepte wie der Einsatz von Delivery Pipelines und Baselines sowie die Nutzung von Release-Containern zum Einsatz, also technische und semantische Klammern um ein Release.

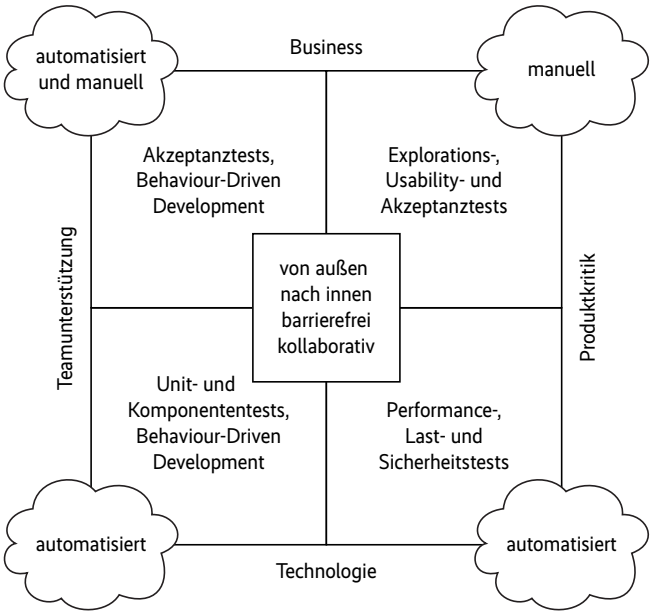
Eine Pipeline ist eine Verkettung von (technischen) Einzelschritten, um Änderungen von der Entwicklung bis in die Produktion zu bekommen (s. Abb. 4). Eine übliche Pipeline fängt damit an, dass ein Continuous Build fortlaufend auf Änderungen im Versionskontrollsystem horcht. Änderungen werden kompiliert, paketiert und im Anschluss auf definierte Metriken abgeglichen. Es kommen dabei Werkzeuge wie Jenkins und Sonar zum Einsatz. Sind die Änderung beziehungsweise der Reifegrad der Anwendung gut genug, die Pipeline bis hier zu durchlaufen, wird eine Baseline erstellt. Das ist – vereinfacht gesagt – ein eingefrorener Stand von Konfigurationseinheiten, auf den weitere Changes reproduzierbar und verfolgbar angewendet werden. Auch kann der Stand als Grundlage für weitere Schritte in der Pipeline dienen.

Spätere Schritte in der Pipeline umfassen erneutes Prüfen der nichtfunktionalen Anforderungen und die Verteilung auf Umgebungen, insbesondere der Produktionsumgebung. Die ganze Pipeline wird nicht unbedingt bei jedem Checkin/Commit durchlaufen. Stattdessen ist ein Task-basierter Ansatz zu präferieren, der die Bereitstellung von Releases an fachlich funktionalen Funktionen und nicht an technischen Commits ausrichtet. Die Metapher einer Pipeline hinkt ein wenig, da deren Ende und Anfang verbunden sein sollten, um Rückkopplungen aus der Produktion in die weitere Entwicklung einfließen lassen zu können. Denn nur in Produktion befindliche Software liefert ein aufschlussreiches Bild über den technischen Reifegrad (durch Release auf das Zielsystem) und den fachlichen (durch die Nutzung des Kunden/Anwenders). Auch Testen ist eine Disziplin, die die Integration bedient, und hat einige nun zu beobachtende Nuancen.

Von Kollaboration und Testen: Die Test-Quadranten

Eine kollaborative Herangehensweise ist häufig an den Test-Quadranten ausgerichtet (s. Abb. 5 auf der nächsten Seite). Q1 umfasst beispielsweise Unit- und Komponententests. Testgetriebene Entwicklung bedeutet das iterative (= Miniprojekte) und inkrementelle (= scheibchenweise) Entwickeln von Software, bei stetigem Refaktorisieren (äußeres Design/API bleibt unverändert, innere Struktur wird verbessert) sowie den Test-first-Ansatz (also zunächst Tests schreiben, dann fachlichen Code – oder zumindest in enger Verzahnung) (s. Artikel auf S. 108). Behavior-Driven Development (s. Artikel auf S. 112) ist eine andere, neuere Herangehensweise, die mehr das fachliche Verhalten in den Vordergrund stellt. Q2 sind Akzeptanztests, die in der Regel über die Oberfläche fachliche Anforderungen testen. Akzeptanztestgetriebene Entwicklung (ATDD) stellt ein TDD also in einen fachlichen Kontext. Häufig sind es Funktions- und Regressionstests. Q1 und Q2 sollen das Team während der Entwicklung unterstützen. Auf der anderen Seite setzen Q3 und Q4 schon ein signifikantes Inkrement voraus, prüfen also einen fertigen Reifegrad. Dabei kommen explorative Tests genauso zum Einsatz wie Performance-Tests. Insgesamt gilt: Die Testkategorien sind nicht in Stein gemeißelt, sondern sollten kontextsensitiv hinterfragt werden.

Während ihrer täglichen Arbeit befinden sich Entwicklung und Betrieb häufig in Konflikt miteinander. Erstere möchte ihre Änderungen beim Kunden schnell im produktiven Einsatz sehen. Letzterer ist primär an der Stabilität der Produktionsumgebung interessiert. Jede Änderung daran ist ein Risiko, das die Stabilität gefährdet. Die Kluft zwischen Entwicklung und Betrieb kommt auf mehreren Ebenen zum Tragen:



Die Test-Matrix: ein Versuch, die multiplen Test-Disziplinen systematisch zu beschreiben und in Beziehung zu setzen (Abb. 5)

- die Kluft bei den Anreizen, resultierend aus unterschiedlichen Zielen.
- die Kluft bei den Prozessen, resultierend aus dem Umgang mit Änderungen.
- die Kluft bei den Werkzeugen, resultierend aus der Tatsache, dass Entwicklung und Betrieb häufig jeweils eigene Werkzeuge nutzen.

Alle drei Punkte sind das Resultat aus einer Mikrooptimierung: Die jeweilige Gruppe verbessert ihre Arbeitsweise, um ihre persönlichen Ziele (viele Änderungen bzw. Stabilität) zu verfolgen. Als Konsequenz fungieren Entwicklung und Betrieb häufig als Silos, also zwei orthogonale Abteilungen, die suboptimal, in manchen Fällen gar nicht, zusammenarbeiten. In Unternehmen gibt es recht unterschiedliche Rahmenparameter und historische Entwicklungen, die Klüfte entstehen lassen oder vergrößern. Ein Beispiel ist der Fokus auf „Management by Objectives“, ein Zusammenstoß agiler Praktiken und konservativer Herangehensweisen, sowie unterschiedliche Werkzeuge wie nginx, OpenEJB und Windows auf Rechnern der Entwickler und Apache, JBoss und Linux auf Produktionsmaschinen.

Bitte inklusive Betrieb: DevOps

DevOps, ein Kofferwort aus „development“ und „operations“, will sich dieser Probleme annehmen, indem Anreize aneinander ausgerichtet und angeglichen sowie Prozesse und Werkzeuge gemeinsam und kollaborativ genutzt werden. Es weitet zudem die Nutzung agiler Praktiken auf den Betrieb aus. Als Ergebnis rücken Entwicklung und Betrieb näher zusammen. DevOps ist kein neues Projektprofil („nun suchen wir einen DevOps Engineer“), keine neue Organisationseinheit („schnell DevOps einrichten, eine neue Pufferabteilung zwischen Entwicklung und Betrieb“) und auch keine Werkzeug-Suite („das DevOps-Etikett drauf, und alle Probleme sind gelöst“). Es lässt sich vielmehr als ein Baukasten verstehen.

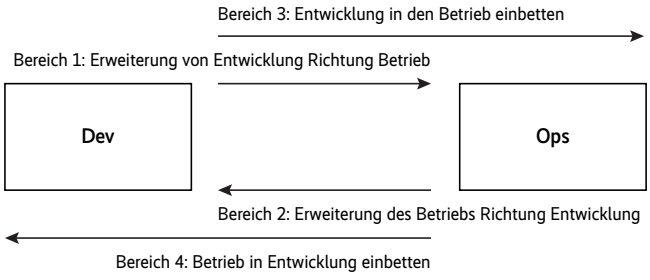
Um DevOps zu kategorisieren, lassen sich Aktivitäten und Wechselwirkungen in vier Bereiche unterscheiden (s. Abb. 6). Im ersten Bereich wird die Entwicklung in Richtung Betrieb ausgedehnt. Das bedeutet nicht, dass Entwickler mit Root-Rechten auf Produktionsmaschinen arbeiten. Beispiele umfassen vielmehr die Nutzung des IT-Konfigurations-Management-Tools Puppet, um Umgebungen aus Code heraus zu provisionieren, oder die von Entwicklung und Betrieb gemeinsame Nutzung von Kanban. Kanban ist ein Vorgehen, das die Anzahl der parallelen Arbeit limitiert. Dadurch werden Stapelgrößen und Durchlaufzeiten verkürzt und Herausforderungen schneller sichtbar. Warum ist der Einsatz von Kanban für DevOps von gesteigertem Interesse? Das hat im Wesentlichen zwei Gründe:

- Für Bereiche, die durch starke Arbeitsteilung und Spezialisierung gekennzeichnet sind, ist Kanban häufig attraktiver als andere agile Methoden, die häufig fordern, dass die Teams aus Generalisten bestehen und keine Wissensinseln vorhanden sind.
- Da Wartung und IT-Betrieb durch viele Unterbrechungen und regelmäßige Notfälle gekennzeichnet sind, sind ungestörtes Arbeiten und Iterationen fester Länge wie in Scrum kaum möglich. Hier kann Kanban die bessere Wahl darstellen, insbesondere weil die Service-Arten in Kanban gut zum Alltag von Systemadministratoren und Wartungsteams passen.

Bereich zwei umfasst die Ausdehnung des Betriebs nach Entwicklung. Typische Beispiele dafür sind die vom Betrieb für die Entwicklung bereitgestellten Log-Dateien oder das Monitoring. Das Aufsetzen und Fortschreiben von Monitoring, verzahnt mit der Entwicklung, ist essenziell, um insbesondere die Umsetzung nichtfunktionaler Anforderungen frühzeitig zu begleiten. Monitoring-Werkzeuge wie Nagios gewinnen Daten auf Systemen und Services und bieten eine Sammlung von Modulen zur Überwachung von Netzen, Hosts und Diensten. Auch fachliches Monitoring (beispielsweise die Visualisierung offener Geschäftstransaktionen) ist möglich und sinnvoll. Konsequent verfolgt, kann das gar zu einem „monitoring-driven development“ der Anwendung führen.

Der dritte Bereich schließt Entwicklung in den Betrieb ein. Ein übliches Beispiel dafür ist das gemeinsame Ausrichten an Zielen, insbesondere produktionssensiblen nichtfunktionalen Anforderungen aus den Bereichen Stabilität und Kapazität. Der Fokus der Programmierer liegt auf der eigentlichen Entwicklung der Anwendung. Entwicklung heißt aber auch, sich frühzeitig Gedanken zu machen, wie die Anwendung in Produktion laufen soll. Ein einfaches Beispiel: der Einsatz eines Applikationsservers, der in Produktion in der Regel als Cluster genutzt wird. Trifft der Entwickler falsche Designentscheidungen, kann sich das im Betrieb verheerend auswirken.

Schließlich beschreibt Bereich vier das Vorgehen, um den Betrieb in die Entwicklung einzubetten, wie die frühe und regelmäßige Rückkopplung über Design und die Tragfähigkeit der in Entwicklung befindlichen Software und Werkzeugen durch



Vier komplementäre Herangehensweisen, sich DevOps zu nähern. Entwicklung und Betrieb weiten jeweils ihr Feld auf den anderen Bereich oder lagern sich ein. Ziel ist immer, die Zusammenarbeit zu verbessern (Abb. 6).

den Betrieb. Durch intensive, stetige Begleitung wird ausgeschlossen, dass nach langen Monaten der Entwicklung der Betrieb mit Lösungen (z. B. Deployment-Verfahren) konfrontiert wird, die ihm gänzlich neu und suspekt erscheinen.

Infrastructure as Code

DevOps mit Werkzeugen wie CFEngine, Chef oder Puppet gleichzusetzen ist ein häufiger Fehler. Trotzdem ist „Infrastructure as Code“, zum Beispiel mit Puppet, ein wichtiger Aspekt von DevOps. Um was geht es? Ein Thema, das in Softwareprojekten häufig zu kurz kommt, ist die Infrastruktur. Nicht selten baut, testet und verteilt eine raffinierte kontinuierliche Integrationsfarm regelmäßig neue Softwareversionen. Doch die Zielumgebungen werden manuell aufgesetzt und fortgeschrieben. Der hohe Automatisierungsgrad hat ein plötzliches Ende. Oder er wechselt massiv die Richtung: Wurden in der Entwicklung noch andere Verfahren eingesetzt, kommen im Betrieb plötzlich Shell-Skripte zum Einsatz. Nicht selten sollen, wenn auch unbewusst, diese Skripte lediglich ein Werkzeug wie Puppet nachbilden.

Mit Puppet, einer freien Software zur Systemadministration, werden Eigenschaften des Zielsystems eintrittsinvariant beschrieben. Das geschieht deklarativ in Form einer domänenspezifischen Sprache (DSL). Der Nutzer erklärt dem Werkzeug also nicht, wie es etwas erreichen, sondern wie das Zielbild aussehen soll. Verschiedene Laufmodi sind denkbar, etwa ein Puppet Master, der unterschiedliche Nodes versorgt, oder ein Root-Manifest, das angestoßen die Infrastruktur aktualisiert. Als Medium kommen dabei Manifeste zum Einsatz. Versionierte Konfigurationseinheiten umfassen nicht nur die traditionellen Artefakttypen wie Java, sondern enthalten auch Infrastrukturelemente (Pakete, Services, User …). Eine weitere nicht zu vernachlässigende Eigenschaft von Puppet ist das Schnüren von Release-Klammern. Es lässt sich also deklarativ beschreiben, welche Artefakte in welchen Versionen ein Release ausmachen, um auf dem Weg zu fachlich und technisch konsistenten Releases zu kommen.

Fazit

Softwarequalität hat viele Gesichter, erst recht, wenn sie über den kompletten Lebenszyklus betrachtet wird. Sie lässt sich nur mit einem ganzheitlichen Ansatz optimieren. Dabei werden übliche Barrieren und Silos überwunden. Wichtiger noch als eine gute Auswahl von Maßnahmen ist eine offene und kollaborative Herangehensweise und eine veränderte Mentalität. Am besten gelingt das durch eine gemeinsame Ausrichtung von Zielen, Prozessen und Werkzeugen. (ane)

Michael Hüttermann
unterstützt als Delivery Engineer Projekte und gibt Seminare in den Bereichen DevOps, Continuous Delivery und agiles ALM. Seine Bücher „Agile ALM“ (2011) und „DevOps for Developers“ (2012) sind international die ersten, die sich mit den jeweiligen Themen auseinandersetzen.

Die Sicht des Architekten auf Softwarequalität in Unternehmen

Offene Geheimnisse

Robert Bräutigam

Wie lässt sich Softwarequalität erreichen? Sie entsteht mit der Hilfe von Werkzeugen, Standards und Prozessen, vor allem aber Menschen sind für sie verantwortlich. Manche ignorieren diese Tatsache lieber, andere nehmen sie hingegen ernst. Dadurch bekommen diese die Gelegenheit, die Qualität überproportional zum Aufwand zu steigern.

Theoretisch sollte es keinen Artikel über Softwarequalität geben, der nicht mit einer Definition anfängt. Entwickler würden unter dem Begriff meistens Codequalität verstehen und wahrscheinlich Metriken (und Werkzeuge) empfehlen. Zum Beispiel: Wie viel Code ist durch Unit-Tests abgedeckt? Wie steht es um die zyklomatische Komplexität der Methoden? Wie hoch ist der Zusammenhalt des Codes? Wie stabil sind die abstrakten Klassen? Solche Metriken lassen sich relativ leicht definieren, und in den richtigen Händen sind sie wichtige Werkzeuge. Trotzdem sind deren Zahlen nichts weiter als Indikatoren, die auf ein Problem oder Defizit hinweisen können, aber selbst keine eindeutige Aussagekraft haben. Eine Testabdeckung von 100 Prozent sieht etwa in einem Managementbericht oder auf einer Power-Point-Folie überzeugend aus. Jedoch würde jeder Entwickler darauf hinweisen, dass so etwas selten praktisch ist und lediglich bedeutet, dass das Werkzeug keine weiteren Aussagen als die treffen kann, es sei alles während des Tests gelaufen.

Wenn Entwickler sich im Code umsehen, können sie recht einfach erkennen, ob er „gut“ oder „schlecht“ ist – auch ohne Metriken. Nicht nur wegen der tatsächlichen Fehler, die sie sehen, sondern anhand des Gefühls, das sich durch ihre Erfahrungen entwickelt hat. Nicht alles ist dabei objektiv. Was etwa für den einen Entwickler „lesbar“ ist, kann für einen anderen durchaus „unlesbar“ sein. Manche wollen unbedingt, dass „Interface“-Klassen mit „I“ anfangen und manche eben nicht.

Endbenutzer bewerten bei der Qualität meistens die Benutzeroberfläche, also wie leicht es ist, bestimmte Abläufe und Aufgaben zu erledigen oder auch wie schnell man lernen kann, sie zu bedienen. Obwohl man oft versucht, Metriken aufzustellen, die diesen Aspekt spezifizieren sollen, ist das bisher niemandem gelungen. Es lässt sich aber die „Benutzbarkeit“ messen, indem man etwa Oberflächen vom Benutzer bewerten lässt. Leider geht das aber erst nach Fertigstellung der Oberfläche. Für Betreiber bedeutet Qualität nur, wie viel Aufwand benötigt wird, um das System am Laufen zu halten. Das ist messbar, etwa an der Anzahl der Supportanfragen oder der Updates, meistens sogar planbar. Andere Perspektiven haben Tester, Qualitätsmanager, Scrum Master, Projektmanager und auch Architekten. Letztere sind dafür verantwortlich, alle diese Perspektiven im Auge zu behalten, mit den entsprechenden Personen zu reden und alles in ein konsistentes Bild zu gießen. Bei der Kategorisierung hilft ISO 9126, und Risiken und Zielerreichung lassen sich mit Methoden wie ATAM (Architecture Tradeoff Analysis Method) einschätzen.

Für die meisten dieser Perspektiven kann man zumindest etwas messen, Metriken aufstellen und Ziele definieren. Leider garantiert das weder, dass die aufgestellten Metriken richtig angewendet und interpretiert, noch, dass die Ziele tatsächlich erreicht werden. Was bedeutet es, wenn man 80 Prozent Code Coverage fordert? Garantiert das wirklich, dass der Code getestet wird? Natürlich nicht, es hängt davon ab, wie die Tests geschrieben sind.

Qualität entsteht durch Menschen

Die einfache Beobachtung ist: Softwarequalität entsteht durch die Arbeit von Menschen. Entwickler, Architekten, Tester, Prozessmanager, Projektmanager, Scrum Master und so weiter sind dafür verantwortlich, Software gemeinsam zu entwickeln und dabei auf Qualität zu achten. Dazu brauchen sie Werkzeuge, die die Arbeit erleichtern oder sogar erst ermöglichen. Tester brauchen „Bug Tracking“-Werkzeuge, Entwickler IDEs, Versionskontrolle, Bibliotheken, FindBugs, Sonar und Continuous Integration. Sogar Scrum Master brauchen zumindest eine Wand, auf die sie ihre Tasks aufkleben können. Wichtig zu erkennen

ist jedoch, dass die richtigen Menschen ohne die meisten Werkzeuge (oder Prozesse) noch immer Software mit guter Qualität liefern können, aber die richtigen Werkzeuge ohne die richtigen Menschen können das nicht.

Das heißt allerdings nicht, dass alle Entwicklungsteams nur aus den „besten" Programmierern bestehen müssen, das wäre kaum realistisch und würde in den meisten Fällen auch nicht funktionieren. Auf die Teamzusammensetzung kommt es an. Dafür muss man nicht nur individuelles Fachwissen, Persönlichkeit und andere Charakteristika beachten, sondern auch, wie sich diese auf die anderen Teammitglieder auswirken.

Solche Beobachtungen klingen vielleicht trivial, aber es kommt trotzdem oft vor, dass gute Teams aufgebrochen und neue Teammitglieder ohne Absprache ins Team geworfen oder das „zentrale" Menschen, die das Team zusammenhalten, einfach versetzt werden. Die Gruppendynamik wird oft ignoriert, obwohl sie in der psychologischen Fachliteratur gründlich diskutiert wird. Bruce Tuckman [1] stuft das Verhalten und die Entwicklung eines Teams beispielsweise in die folgenden Kategorien ein:

- Forming: die ersten Schritte des Teams, die Menschen lernen einander kennen.
- Storming: Die Lösungsansätze werden diskutiert, es entsteht ein Wettbewerb nicht nur zwischen Ideen, sondern auch zwischen Teammitgliedern. Manche Teams können diesen Zustand nicht mehr verlassen.
- Norming: Die Ziele sind aufgestellt und die Hierarchie zwischen den Teamkollegen festgelegt. Es gibt ein Verständnis, wie sie miteinander kommunizieren können.
- Performing: Teams in diesem Stadium werden hochperformant und funktionieren selbstständig.

Es reicht also nicht, die Menschen nur individuell zu selektieren, um ein Team zu bilden, die Teamdynamik ist immer im Auge zu behalten.

Menschen sind keine Ressourcen

Sätze wie „Ich habe zehn Ressourcen im Projekt", oder: „Ich brauche noch zwei Ressourcen" von einem Team- oder Projektleiter zu hören, wecken das Gefühl, dass er Menschen, Wissen, Persönlichkeit oder Teamdynamik außer Acht lässt. Und weil diese Eigenschaften den größten Einfluss auf die Softwarequalität haben, hat das zur Folge, dass Qualität nicht geachtet wird. Ironischerweise verlangt manchmal derselbe Teamleiter dann „Qualität" von seinem Team.

Ein Projektleiter kann auch Glück haben, nicht aufpassen und trotzdem die richtigen Fachleute ins Team holen, die gut entwickeln und ins Team passen. Es gibt auch welche, die das Wort „Ressource" verwenden, aber dann aufpassen, dass trotzdem die richtigen Menschen zusammenkommen. Es ist nicht schwarz oder weiß, 0 oder 1. Es ist lediglich eine falsche Bezeichnung, die später zu Fehlern oder Missverständnissen führen kann.

Wenn eine Variable im Code zum Beispiel *accountList* heißt, könnte man annehmen, dass es eine Liste ist und man darauf die Methode *isEmpty()* aufrufen kann. Was wäre aber, wenn es tatsächlich ein String ist, der die Liste der Datenbank identifiziert? Hätte vielleicht der Name *accountListKey* besser gepasst, dann hätte man vielleicht nicht versucht *isEmpty()* aufzurufen. So ist es auch wenn ein Projektleiter Menschen als Ressourcen bezeichnet. Man könnte annehmen, dass sie frei austauschbar sind, dass alle mit demselben Tempo arbeiten, dass man sie frei gruppieren oder auch Gruppen frei aufteilen kann. Das alles ist nicht zutreffend für Menschen, sie sind nicht frei austauschbar und arbeiten auch nicht alle mit demselben Tempo. Sie lassen sich

nicht willkürlich gruppieren, außer man ignoriert Fachwissen und Gruppendynamik völlig.

Das Gerede drumherum

Eine komplette mathematische Spezifikation eines Problems ist eigentlich Quellcode, alles andere lässt immer Fragen offen. Es gibt in Unternehmensprojekten nie eine vollständige Spezifikation (nicht mal annähernd), aber das ist nicht unbedingt ein Problem. Die meisten Projekte ändern sich sowieso fortwährend, daher würde es keinen Vorteil bringen, etwas detailliert zu spezifizieren, dass am Ende nicht gebaut wird. Es gibt daher gleich zwei Aufgaben für das Projekt. Die erste: Wie übermittelt es Informationen aus den Köpfen von Leuten, die das Problem kennen, in die Köpfe derer, die die Software bauen werden (Analysten, Entwickler usw.), und zwar so, dass so wenig wie möglich verloren geht oder sich ändert. Die zweite Aufgabe ist sicherzustellen, dass die Entwickler (Architekten usw.) genug Informationen oder Gelegenheiten haben, über Fragen zu entscheiden, die noch nicht spezifiziert worden sind – also die umgekehrte Richtung.

Daher ist die Kommunikation zwischen Teammitgliedern (und manchmal – abhängig vom Teamaufbau – darüber hinaus) mindestens so wichtig wie die Personen selbst. Kommunikation sollte möglichst hindernisfrei und direkt sein. Räumliche und hierarchische Informationswege sind Kommunikationshindernisse. Die Menschen, die nicht die richtigen Informationen haben oder sie nicht leicht bekommen können, werden keine Qualität liefern können. Schlimmer noch: Sie werden fachliche Entscheidungen treffen, die sich später als falsch erweisen, da die Uneindeutigkeit der Spezifikation viel Raum für Fehlinterpretationen lässt.

Ein oft wenig bedachtes Problem beim klassischen Outsourcing ist, dass man die beauftragten Personen des Partners nicht persönlich kennt. Was schon in der eigenen Firma schwierig sein kann, kann beim Partner zur Unmöglichkeit werden. Leider prüfen auch Verfahren wie CMMI (Capability Maturity Model Integration) nur die Prozesse, nicht aber die Menschen und Teams.

Agile – fragile

Das „Agile Manifesto“ wurde geschrieben, um die oben genannten Kenntnisse (zusammen mit ein paar anderen Ideen) formal zu beschreiben. Der erste Punkt im Manifest ist, dass Menschen und Interaktionen wichtiger sind als Prozesse und Werkzeuge, wenn man bessere Software entwickeln will. Die Botschaft ist heute erfolgreich angekommen. Die Anwendung aber ist meistens noch fehlerhaft. Scrum etwa basiert auf diesen Werten und liefert bestimmte Vorgehensweisen, wie sich ein Projekt organisieren lässt, um erfolgreich „agile“ zu sein. Leider wird das häufig missverstanden und nur ein Prozess gestaltet, der Scrum ähnlich ist, aber in Wirklichkeit noch immer die alten Wasserfall- und Projektmanagement-Konzepte nutzt. So wird das Projekt die Vorteile von Scrum nicht nutzen können, und darüber hinaus verliert es auch die „altbewährten“ Methoden. Deshalb nennt man diese Projekte umgangssprachlich „fragile“ statt „agile“.

Es ist wichtig zu erkennen, dass es bei „agile“ nicht nur darum geht, iterativ zu entwickeln, früh und oft zu liefern oder Daily Stand-ups zu machen, sondern in erster Linie um Menschen und deren Zusammenarbeit. Ein eindeutiges Zeichen, dass man Menschen als Ressourcen betrachtet, ist, wenn ihre Arbeit anhand irgendwelcher Metriken gemessen wird. Das kann in einer Fabrik gut funktionieren oder bei Aufgaben, die keine Kreativität benötigen, aber nicht bei der Entwicklung. Werkzeuge und Metriken sind da, um Entwickler zu unterstützen und nicht um sie zu kontrollieren. Manchmal ist es hilfreich (für Entwickler), diese Metriken nicht zu veröffentlichen, weil sie nur andere dazu einladen, jeden Wert zu hinterfragen oder kontrollieren zu wollen.

Wenn der Architekt oder Projektleiter Probleme mit der Qualität hat, hilft es nicht, irgendwelche Grenzwerte für Metriken zu erzwingen. Wenn das Team Software liefert, die zu viele Fehler während der Tests aufweist, ist es sinnlos, die Anzahl von Fehlern vertraglich limitieren zu wollen. Die richtige Reaktion wäre, die eigentlichen Probleme aufzudecken und sie zu beseitigen, nicht die Entwickler zu bestrafen. Es kann sein, dass die Spezifikation nicht detailliert genug ist und die Entwickler niemanden haben, mit dem sie schnell Probleme besprechen können. Es kann auch sein, dass im Team das Fachwissen fehlt oder die Aufgaben einfach zu groß und damit mit zu viel Risiko verbunden sind.

Große Projekte und SOA

Zur Beurteilung der optimalen Teamgröße gibt es viele Ansätze in der Fachliteratur, die Schätzungen liegen meist bei unter 10. Ist das Team zu groß, leidet die Effizienz, ist es zu klein, kann es nicht liefern. Das macht natürlich größere Projekte auf den ersten Blick unmöglich – mit zehn Programmierern ist kein Mautsystem programmierbar. Serviceorientierte Architektur (SOA) ist ein Ansatz, diese Grenzen zu durchbrechen, um auch mit kleinen Teams und kleinen Projekten trotzdem komplexe Systeme zu entwickeln. Weil mit SOA auch Techniken verbunden sind (wie RESTful HTTP, Webservices, WS-*-Spezifikationen, Enterprise Service Bus und andere „Enterprise“-Produkte), werden sie oft mit diesen Techniken verwechselt. Es geht aber um einen grundsätzlichen Denk- und Designansatz.

Mit kleinen Projekten die Logik durch einheitliche Schnittstellen verfügbar zu machen minimiert nicht nur das Risiko, sondern man erhöht so auch die Geschwindigkeit der Entwicklung und die Wiederverwendbarkeit von Komponenten.

Fazit

Falls Softwarequalität in einem Projekt wirklich relevant sein sollte, gibt es nichts Wichtigeres als die Menschen, die in diesem Team arbeiten, und die Kommunikation zwischen ihnen. Man kann versuchen, noch so teure Werkzeuge und Produkte einzuführen, aber mit dem falschen Team wird das nicht zur Qualität beitragen. Obwohl das eine triviale Feststellung ist, wird es trotzdem oftmals ignoriert oder nicht konsistent angewandt.

Wenn man das erkennt und die Menschen wirklich in den Mittelpunkt stellt, wird man nicht nur bessere Software liefern können, sondern gewinnt auch Einblick in Themen wie Teamaufbau, agile Entwicklung, Outsourcing oder sogar SOA. Man kann diese Themen von einer anderen Perspektive aus betrachten, die deren Anwendung leichter und konsistenter macht. (ane)

Literatur

[1] Bruce Tuckman; Developmental Sequence in Small Groups; Psychological Bulletin, Vol 63, No. 6, 384 – 399

Robert Bräutigam
ist als Senior-Consultant für die MATHEMA Software GmbH tätig. Er ist seit 1999 als Entwickler und Architekt im Java-Enterprise-Umfeld beschäftigt.

Agile ALM – Agilo for trac

Agilo for trac ist ein voll konfigurierbares und flexibles ALM Tool, das speziell für agile Projekte entwickelt wurde. Es bietet Traceability, umfassende Projektkontrolle und Unterstützung aller agiler Rollen.
Agilo for trac ist als freie und Pro Version verfügbar und unterstützt Scrum und Kanban gleichermaßen.

Hoher Nutzen nicht nur für das Entwickler Team

Agilo for trac präsentiert sich mit einer angenehm aufgeräumten Oberfläche im Web Browser mit Zugriff auf alle wichtigen Funktionen. Auch ohne Login stehen den Nutzern ein Wiki, eine Timeline, eine Roadmap, die Ticketdatenbank und allgemeine Suchfunktionen zur Verfügung.

Agiles Arbeiten als Produkt Manager

Produktverantwortliche sind mit Agilo in der Lage, alle Aufgaben, die in einem agilen Projekt anstehen, durchzuführen:

- Anforderungen erfassen, priorisieren und User Stories schreiben
- Sprint und Release Planung erstellen
- Roadmap und Meilensteine definieren
- Kommunikation vereinfachen
- Budgets planen und Business Value beschreiben
- Return on Investment transparent machen

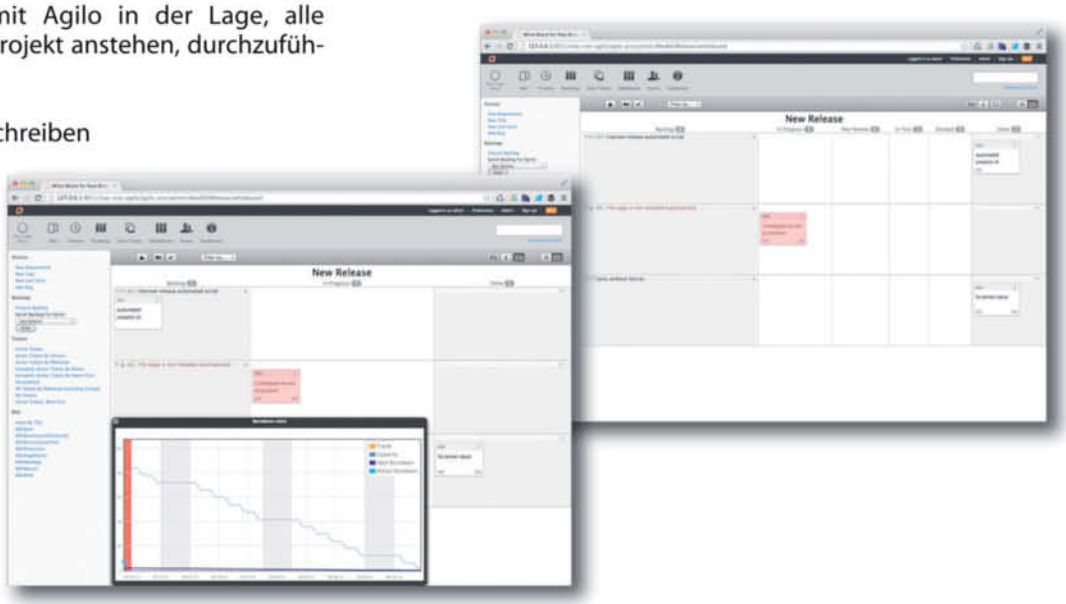
Auch unterstützt Agilo for trac methodische Tools wie z. B. die "Definition of Done".

Arbeiten als agiles Team - Scrum und Kanban

Das flexible Whiteboard lässt sich sowohl für Scrum als auch für Kanban nutzen. Die Spalten des Boards sind beliebig erweiter- und konfigurierbar.
Insbesondere für verteilte Teams ist das Whiteboard ein nützliches Feature, um jederzeit den aktuellen Stand und Fortschritt der Tasks zu sehen.

Darüber hinaus unterstützt Agilo for trac die Teams bei vielen agilen Tätigkeiten:

- Schätzen von User Stories
- Analyse der User Stories und Erstellen damit verbundener Aufgaben
- Commitment
- Bearbeiten der Aufgaben
- Tägliches Update
- Demonstration der Ergebnisse eines Sprints
- Retrospektive der Prozesse



Alle Fakten im Überblick:

Verfügbare Versionen:
- Kostenlose Version mit allen Funktionen aber ohne Pro Features (z.B Whiteboard)
- Pro Version (30 Tage kostenlos und ohne Verpflichtung testen)

Verfügbare Downloads:
- Virtual Machine
- Windows Installer
- Python-Eggs für verschiedene Python Versionen
- Source Code (keine Pro Features)

Technische Informationen:
Licence: Apache Software Licence 2.0
Operating System: all
Released: 12/2007
Latest Update: 05/2013
Language: Python
Version: 1,10
Integrations:
- SVN
- Trac
- Eclipse

agilo software

Hersteller:
www.agilosoftware.com
Free Download:
www.agilofortrac.com
Open Support Group:
http://groups.google.com/group/agilo+

Erfahrungen zur Softwarequalität aus einem aktuellen Projekt

Soufflé de qualité

Felix Ziesel

Qualitätssicherung in einem Softwareprojekt einzuführen und zu erhalten lässt sich mit einem Kochabend mit Freunden vergleichen, an dem alles schiefgeht, was schiefgehen kann. Damit der Abend trotzdem ein Erfolg wird, muss man manchmal improvisieren und manchmal auf Bewährtes zurückgreifen. Einige Best Practices zur Verbesserung des Qualitätssicherungsprozesses am Beispiel eines aktuellen Projekts.

Wie beim Kochen sind die richtigen Zutaten in der entsprechenden Güte eine Grundvoraussetzung. Aber nicht nur sie machen ein leckeres Essen aus, sondern auch die strukturierte Zubereitung und die kleinen Tipps, wie das Gericht möglichst raffiniert gewürzt wird. Es geht um die Fragen: „Was braucht man?“ (Zutaten); „Wie ist vorzugehen?“ (Zubereitung); „Wie geht man mit Herausforderungen um?“ (Tipps und Tricks).

Zuerst zu den Zutaten. Damit der Quality Engineer etwas bewirken kann, ist es unabdingbar, dass er den Softwareentwicklungsprozess kritisch hinterfragen darf und sich seine Änderungswünsche zeitnah umsetzen lassen. Bei entsprechender Teamgröße dürfen es selbstverständlich mehr Quality Engineers sein. Der Qualitätsverantwortliche kann nur effizient testen, wenn er eine wachsende Zahl von Tests automatisieren und in eine mindestens tägliche Regression bringt. Das hat folgenden Vorteil: Die Continuous Integration nimmt die Arbeit durch Verwaltung und Triggern von Builds und Tests ab. Alle Tests müssen in einer neutralen Umgebung lauffähig sein. Insellösungen, die sich einzelne Entwickler zum Laufen ihrer Tests gebaut haben, sollten nicht genutzt werden, sodass sich alle Tests überall gleich ausführen lassen.

Für das Testen braucht man entsprechende Werkzeuge. Zu empfehlen ist ein Zusammenspiel von Unit- und funktionalen Tests. Während sich Unit-Tests normalerweise gut über die IDE starten lassen und durch Continuous Integration unterstützt werden, bieten sich für funktionale Tests eigene Testumgebungen (im folgenden ITE genannt) an (etwa Jubula/GUIdancer, HP QuickTest, Rational Functional Tester). Als Quelle für Testfälle und Ort für das sinnvolle Ablegen der aufgedeckten Mängel fungiert ein Bug-Tracking-System. Wichtig ist darüber hinaus eine strukturierte Dokumentation über den Umfang der Anwendung. Hier zählt weniger ein fortgeschrittenes Werkzeug als vielmehr ein gelebtes Vorgehen.

Zubereitung – Rollendefinition und Rahmenbedingungen

Letztlich stellt die Qualitätssicherung in der Softwareentwicklung eine gemeinsame Anstrengung des gesamten Teams dar. Niemand kann die Qualität alleine gewährleisten. Es gibt zahlreiche Aufgaben innerhalb der Qualitätssicherung, die Entwickler deutlich effizienter durchführen können. Die Aufgabe des Quality Engineer ist es, hier zu koordinieren, zu erklären und so einen Nutzen für das gesamte Team zu erreichen.

Bei einem neuen Projekt sind naturgemäß nicht all diese Zutaten vorhanden. Die erste Aufgabe besteht darin, sie zeit-