1 Einleitung

Software ist allgegenwärtig. Nahezu jedes komplexere Produkt ist heute softwaregesteuert und auch viele Dienstleistungen stützen sich auf Softwaresysteme. Software und Softwarequalität sind daher ein entscheidender Wettbewerbsfaktor. Ein Unternehmen, das neue oder bessere Software in kürzerer Zeit in sein Produkt integrieren bzw. auf den Markt bringen kann (Time-to-Market), ist seinen Mitbewerbern überlegen.

Agile Entwicklungsmodelle versprechen eine schnellere »Time-to-Market« bei gleichzeitig besserer Ausrichtung an den Kundenanforderungen und nicht zuletzt bessere Softwarequalität. So erstaunt es nicht, dass heute in Unternehmen zunehmend agile Methoden eingesetzt werden – auch in großen, internationalen Projekten und in Produktentwicklungseinheiten großer Konzerne, quer durch alle Branchen. In den meisten Fällen bedeutet dies den Umstieg von der Entwicklung nach V-Modell auf die agile Entwicklung mit Scrum[1].

Die Umstellung auf »agil« und das nachhaltige produktive agile Arbeiten sind jedoch nicht ganz einfach. Insbesondere dann, wenn mehr als nur ein Team davon betroffen ist. Jedes Teammitglied, das Projektmanagement, aber auch das Management in der Linienorganisation muss teils gravierende Änderungen gewohnter Abläufe und Arbeitsweisen vollziehen. Dabei sind Softwaretest und Softwarequalitätssicherung ganz entscheidend daran beteiligt, ob ein Team, eine Softwareabteilung oder ein ganzes Unternehmen agiles Entwickeln langfristig erfolgreich beherrscht und so die erhofften Vorteile nachhaltig realisieren kann.

Zu den populären agilen Entwicklungsmethoden gibt es eine Fülle auch deutschsprachiger Literatur. Einige empfehlenswerte Einführungen, z.B. zu Scrum, finden sich im Literaturverzeichnis dieses Buches. In der Regel wird das Thema »agile Softwareentwicklung« in diesen

1. Umfrage [URL: Testpraxis] und Studie [URL: StatusQuoAgile].

Büchern aus Sicht des Entwicklers und Programmierers betrachtet. Demgemäß stehen agile Programmiertechniken und agiles Projektmanagement im Vordergrund. Wenn das Thema Testen erwähnt wird, geht es meistens um Unit Test und zugehörige Unit-Test-Werkzeuge, also im Wesentlichen um den Entwicklertest. Tatsächlich kommt dem Testen in der agilen Entwicklung aber eine sehr große und erfolgskritische Bedeutung zu und Unit Tests alleine sind nicht ausreichend.

Dieses Buch möchte diese Lücke schließen, indem es agile Softwareentwicklung aus der Perspektive des Testens und des Softwarequalitätsmanagements betrachtet und aufzeigt, wie »agiles Testen« funktioniert, wo »traditionelle« Testtechniken auch im agilen Umfeld weiter benötigt werden und wie diese in das agile Vorgehen eingebettet werden.

1.1 Zielgruppen

Verstehen, wie Testen in agilen Projekten funktioniert.

Das Buch richtet sich zum einen an Leser, die in das Thema agile Entwicklung erst einsteigen, weil sie künftig in einem agilen Projekt arbeiten werden oder weil sie Scrum in ihrem Projekt oder Team einführen wollen oder gerade eingeführt haben:

- Entwicklungsleiter, Projektmanager, Testmanager und Qualitätsmanager erhalten Hinweise und Tipps, wie Qualitätssicherung und Testen ihren Beitrag dazu leisten können, das Potenzial agiler Vorgehensweisen voll zu entfalten.
- Professionelle (Certified) Tester und Experten für Softwarequalität erfahren, wie sie in agilen Teams erfolgreich mitarbeiten und ihre spezielle Expertise optimal einbringen können. Sie lernen auch, wo sie ihre aus klassischen Projekten gewohnte Arbeitsweise umstellen oder anpassen müssen.

Wissen über (automatisiertes) Testen und agiles Qualitätsmanagement erweitern

Ebenso angesprochen werden aber auch Leser, die bereits in agilen Teams arbeiten und eigene »agile« Erfahrungen sammeln konnten und die ihr Wissen über Testen und Qualitätssicherung erweitern wollen, um die Produktivität und Entwicklungsqualität in ihrem Team weiter zu erhöhen:

- Product Owner, Scrum Master, Qualitätsverantwortliche und Mitarbeiter mit Führungsfunktion erfahren in kompakter Form, wie systematisches, hoch automatisiertes Testen funktioniert und welchen Beitrag Softwaretester im agilen Team leisten können, um kontinuierlich, zuverlässig und umfassend Feedback über die Qualität der entwickelten Software zu liefern.

- Programmierer, Tester und andere Mitglieder eines agilen Teams erfahren, wie sie hoch automatisiertes Testen realisieren können, und zwar nicht nur im Unit Test, sondern auch im Integrations- und im Systemtest.

Das Buch beinhaltet viele praxisorientierte Beispiele und Übungsfragen, sodass es auch als Lehrbuch und zum Selbststudium geeignet ist.

1.2 Zum Inhalt

Kapitel 2

Kapitel 2 gibt einen knappen, vergleichenden Überblick über die derzeit populärsten agilen Vorgehensmodelle Scrum und Kanban. Leser mit Managementaufgaben, die ihr Projekt oder ihre Unternehmenseinheit auf »agil« umstellen wollen, erhalten hier eine Zusammenfassung der wichtigsten agilen Managementinstrumente. Dem gegenübergestellt wird das Vorgehen in Projekten, die sich an klassischen Vorgehensmodellen orientieren. Dies vermittelt einen Eindruck über die Veränderungen, die mit der Einführung agiler Ansätze einhergehen. Leser, die Scrum und Kanban schon kennen, können dieses Kapitel überspringen.

Kapitel 3

Kapitel 3 zeigt auf, welche leichtgewichtigen Planungs- und Steuerungsinstrumente in Scrum anstelle des klassischen Projektplans zum Einsatz kommen. Denn »agil« zu arbeiten, bedeutet keineswegs »planlos« zu arbeiten. Auch Kapitel 3 richtet sich primär an Leser, die auf agile Entwicklung umsteigen. Die Erläuterungen und Hinweise, welchen Beitrag die jeweiligen Planungsinstrumente zur »konstruktiven Qualitätssicherung« und damit zur Fehlervermeidung liefern, sind jedoch auch für Leser mit agiler Projekterfahrung wertvoll.

Kapitel 4

Kapitel 4 behandelt das Thema Unit Tests und »Test First«. Es erklärt, was Unit Tests leisten und wie Unit Tests automatisiert werden. Systemtester, Fachtester oder Projektbeteiligte ohne oder mit geringer Erfahrung im Unit Test finden hier Grundlagen über die Techniken und Werkzeuge im entwicklungsnahen Test, die ihnen helfen, enger mit Programmierern und Unit-Testern zusammenzuarbeiten. Programmierer und Tester mit Erfahrung im Unit Test erhalten hilfreiche Tipps, um ihre Unit Tests zu verbessern. Ausgehend von diesen Grundlagen wird Test First (testgetriebene Entwicklung) vorgestellt und die hohe Bedeutung dieser Praktik für agile Projekte erklärt.

Kapitel 5

Kapitel 5 erklärt Integrationstests und »Continuous Integration«. Auch Programmierer, die ihren Code intensiv mit Unit Tests prüfen, vernachlässigen dabei oft Testfälle, die Integrationsaspekte überprüfen. Daher werden in diesem Kapitel zunächst wichtige Grundlagen

über Softwareintegration und Integrationstests vermittelt. Anschließend wird die Continuous-Integration-Technik vorgestellt und erläutert, wie ein Continuous-Integration-Prozess im Projekt eingeführt und angewendet wird.

Kapitel 6 Kapitel 6 erörtert Systemtests und »Test nonstop«. Aufbauend auf den Grundlagen über Systemtests werden wichtige Techniken für manuelle und automatisierte System- und Akzeptanztests im agilen Umfeld erläutert. Anschließend wird aufgezeigt, wie auch Systemtests effizient automatisiert und in den Continuous-Integration-Prozess des Teams eingebunden werden können. Kapitel 6 ist dabei nicht nur für Systemtester und Fachtester gedacht, sondern auch für Programmierer, die besser verstehen wollen, welche Testaufgaben jenseits des entwicklungsnahen Tests im agilen Team gemeinsam zu bewältigen sind.

Kapitel 7 Kapitel 7 stellt klassisches und agiles Verständnis von Qualitätsmanagement und Qualitätssicherung gegenüber und erläutert die in Scrum »eingebauten« Praktiken zur vorbeugenden, konstruktiven Qualitätssicherung. Der Leser erhält Hinweise und Tipps, wie Qualitätsmanagement »agiler« realisiert werden kann und wie Mitarbeiter aus Qualitätssicherungsabteilungen, Qualitätsmanagementbeauftragte und andere Qualitätssicherungsspezialisten ihr Know-how in agilen Projekten einbringen und so einen wertvollen Beitrag für ein agiles Team leisten können.

Kapitel 8 Kapitel 8 präsentiert mehrere Fallstudien aus Industrie, Onlinehandel und Unternehmen der Softwarebranche. Diese spiegeln Erfahrungen und Lessons Learned wider, die die Interviewpartner bei der Einführung und Anwendung agiler Vorgehensweisen in ihrem jeweiligen Unternehmen gesammelt haben.

Kapitelstruktur Die Kapitel 2, 3, 7 und 8 erörtern Prozess- und Managementthemen und sprechen demgemäß den an Managementaspekten interessierten Leser an. Die Kapitel 4, 5 und 6 erörtern das (automatisierte) »agile Testen« auf den verschiedenen Teststufen und sprechen den (auch) technisch interessierten Leser an. Dabei werden die Ziele und Unterschiede von Unit Tests, Integrations- und Systemtests ausführlich angesprochen. Denn wie bereits erwähnt, wird Testen leider in vielen agilen Projekten oft mit Unit Tests gleichgesetzt. Abbildung 1–1 illustriert die Kapitelstruktur nochmals:

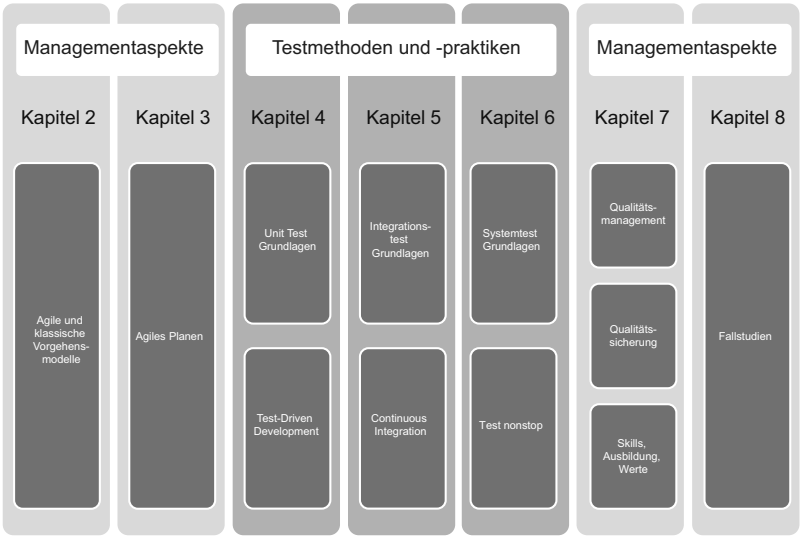

Abb. 1–1
Kapitelstruktur

Cross-Referenz zum ISTQB®-Syllabus

Das Buch deckt den Stoff des *ISTQB® Certified Tester – Foundation Level Extension Syllabus Agile Tester* (Version 2014) ab. Die Gliederung des Buches folgt jedoch nicht der Gliederung des Lehrplans. Auch werden viele Aspekte, die beim Testen in agilen Projekten eine Rolle spielen, im Buch über den ISTQB®-Lehrplan hinausgehend oder zusätzlich behandelt.

Die folgende Cross-Referenz-Tabelle erleichtert es, gezielt den ISTQB®-Lehrstoff anhand der im Lehrplan genannten Lernziele nachlesen zu können:

Tab. 1–1
Cross-Referenz zum ISTQB®-Syllabus

Lernziele nach Certified Tester – Foundation Level Extension Syllabus, 2014	**Kapitel und Abschnitte in diesem Buch**
1. Agile Softwareentwicklung	
1.1 Die Grundlagen der agilen Softwareentwicklung	
FA-1.1.1 Das Grundkonzept der agilen Softwareentwicklung basierend auf dem Agilen Manifest beschreiben können.	■ 2
FA-1.1.2 Die Vorteile des Whole-Team Approach (interdisziplinäres, selbstorganisiertes Team) verstehen.	■ 2.1, 2.4
FA-1.1.3 Den Nutzen von frühen und häufigen Rückmeldungen verstehen.	■ 3.2, 3.4 ■ 4.2, 4.4, 4.5 ■ 5.5 ■ 6.1, 6.2, 6.4, 6.6, 6.8 ■ 7.1, 7.2, 7.2.3, 7.5, 7.5.1, 7.6, 7.6.1, 7.7

→

Lernziele nach Certified Tester – Foundation Level Extension Syllabus, 2014	Kapitel und Abschnitte in diesem Buch
1. Agile Softwareentwicklung	
1.2 Aspekte agiler Ansätze	
FA-1.2.1 Die Ansätze der agilen Softwareentwicklung nennen können.	■ 2
FA-1.2.2 User Stories schreiben in Zusammenarbeit mit Entwicklern und Vertretern des Fachbereichs.	■ 3.3 ■ 6.1, 6.3.1, 6.3.3, 6.4, 6.8.2 ■ 7.6.2
FA-1.2.3 Verstehen, wie in agilen Projekten Retrospektiven als Mechanismus zur Prozessverbesserung genutzt werden.	■ 2.4 ■ 3.6 ■ 4.5 ■ 5.5 ■ 6.2 ■ 7.2.3, 7.5.1
FA-1.2.4 Die Anwendung und den Zweck von Continuous Integration (der kontinuierlichen Integration) verstehen.	■ 5, 5.5 ■ 6.9
FA-1.2.5 Die Unterschiede zwischen Iterations- und Releaseplanung kennen und wissen, wie sich ein Tester gewinnbringend in jede dieser Aktivitäten einbringt.	■ 3.3, 3.4, 3.5, 3.8 ■ 6.9
2. Grundlegende Prinzipien, Praktiken und Prozesse des agilen Testens	
2.1 Die Unterschiede zwischen traditionellen und agilen Ansätzen im Test	
FA-2.1.1 Die Unterschiede der Testaktivitäten zwischen agilen und nicht agilen Projekten benennen und erläutern können.	■ 2, 2.4 ■ 3.5, 3.7, 3.7.4 ■ 4.5 ■ 5.6 ■ 6.10
FA-2.1.2 Beschreiben können, wie Entwicklungs- und Testaktivitäten in einem agilen Projekt umgesetzt werden.	■ 4 ■ 5 ■ 6
FA-2.1.3 Die Bedeutung von unabhängigem Test in agilen Projekten darlegen können.	■ 6.8, 6.8.1 ■ 7.6.1, 7.7
2.2 Der Status des Testens in agilen Projekten	
FA-2.2.1 Erläutern können, welches Mindestmaß an Arbeitsergebnissen sinnvoll ist, um den Testfortschritt und die Produktqualität in agilen Projekten sichtbar zu machen.	■ 3.7, 3.7.1, 3.7.2 ■ 4.5 ■ 5.6 ■ 6.10 ■ 7.5.1

→

Lernziele nach Certified Tester – Foundation Level Extension Syllabus, 2014	Kapitel und Abschnitte in diesem Buch
2. Grundlegende Prinzipien, Praktiken und Prozesse des agilen Testens	
2.2 Der Status des Testens in agilen Projekten	
FA-2.2.2 Damit vertraut sein, dass sich die Tests über mehrere Iterationen hinweg kontinuierlich weiter entwickeln und daher auch erklären können, warum zum Beherrschen der Risiken im Regressionstest Testautomatisierung wichtig ist.	4.5 5.6 6.2, 6.8, 6.10
2.3 Rolle und Fähigkeiten eines Testers in einem agilen Team	
FA-2.3.1 Verstehen, über welche Fähigkeiten (bzgl. Menschen, Domainwissen und Testen) ein Tester in agilen Teams verfügen muss.	7.7
FA-2.3.2 Wissen, was die Rolle eines Testers in einem agilen Team ist.	2, 2.1 3.7, 3.7.2, 3.7.4 4.2.2 4.5 5.6 6.10 7.5, 7.6, 7.7
3. Methoden, Techniken und Werkzeuge des agilen Testens	
3.1 Agile Testmethoden	
FA-3.1.1 Die Konzepte der testgetriebenen Entwicklung, der abnahmetestgetriebenen Entwicklung und der verhaltensgetriebenen Entwicklung nennen können.	3.3 4.2 5.6 6.1, 6.3.3, 6.5, 6.6, 6.7
FA-3.1.2 Die Konzepte der Testpyramide nennen können.	3.7.3
FA-3.1.3 Die Testquadranten und ihre Beziehungen zu Teststufen und Testarten zusammenfassen.	3.7.3
FA-3.1.4 Für ein vorgegebenes agiles Projekt die Rolle eines Testers in einem Scrum-Team übernehmen.	2, 2.1 3.7, 3.7.2, 3.7.4 4.2.2, 4.5 5.6 6.10 7.5, 7.6, 7.7
3.2 Qualitätsrisiken beurteilen und Testaufwände schätzen	
FA-3.2.1 Qualitätsrisiken in einem agilen Projekt einschätzen.	3.7, 3.7.2 4.1.2, 4.1.4, 4.5 5.5.3 6.2, 6.3.3, 6.8 7.3.1, 7.6.1

→

Lernziele nach Certified Tester – Foundation Level Extension Syllabus, 2014	Kapitel und Abschnitte in diesem Buch
3. Methoden, Techniken und Werkzeuge des agilen Testens	
3.2 Qualitätsrisiken beurteilen und Testaufwände schätzen	
FA-3.2.2 Testaufwand auf Basis des Iterationsinhalts und der Qualitätsrisiken schätzen.	■ 3.7 ■ 4.1.2 ■ 5.2, 5.2.2 ■ 6.2, 6.8, 6.10
3.3 Techniken in agilen Projekten	
FA-3.3.1 Die relevanten Informationen interpretieren können, um Testaktivitäten zu unterstützen.	■ 3.1, 3.2, 3.3 ■ 6.6 ■ 7.3.1
FA-3.3.2 Den Fachbereichsvertretern erklären können, wie testbare Abnahmekriterien zu definieren sind.	■ 3.3 ■ 6.1, 6.3, 6.3.3, 6.4.2, 6.6, 6.7
FA-3.3.3 Für eine vorgegebene User Story abnahmetest-getriebene Testfälle (ATDD) schreiben können.	■ 3.3 ■ 6.3, 6.3.3, 6.4.2, 6.4.3
FA-3.3.4 Auf Basis von vorgegebenen User Stories mithilfe von Blackbox-Testentwurfsverfahren funktionale und nicht funktionale Testfälle schreiben können.	■ 4, 4.1, 4.2 ■ 5, 5.1, 5.2 ■ 6, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7
FA-3.3.5 Explorative Tests durchführen können, um das Testen eines agilen Projekts zu unterstützen.	■ 6.3, 6.3.1, 6.3.2
3.4 Werkzeuge in agilen Projekten	
FA-3.4.1 Verschiedene für Tester verfügbare Werkzeuge gemäß ihrem Zweck und den Aktivitäten in agilen Projekten kennen.	■ 3.7.2, 3.7.4 ■ 4.3, 4.4, 4.5 ■ 5.3.1, 5.5, 5.6 ■ 6.4, 6.5, 6.7

Fallbeispiel, Checkfragen und Übungen

Viele, wenn nicht die meisten agilen Ideen, Techniken und Praktiken sind, wenn man den Ausführungen in der entsprechenden Literatur folgt, einfach nachzuvollziehen. Auch die Ideen, Hinweise und Tipps der folgenden Kapitel werden dem Leser vielleicht zunächst einfach erscheinen. Die Knackpunkte stellen sich erst in der Praxis bei der Umsetzung heraus. Das Buch geht auf diese Hürden ein, und damit der Leser praktisch nachvollziehen und »erleben« kann, wo die Herausforderungen stecken, sind folgende Elemente im Text zu finden:

- Ein durchgängiges Fallbeispiel, anhand dessen die jeweils vorgestellten Methoden und Techniken veranschaulicht werden.
- Checkfragen und Übungen, anhand derer der Leser am Ende eines Kapitels den besprochenen Stoff rekapitulieren kann, aber auch

seine Situation und sein Agieren im eigenen Projekt kritisch hinterfragen kann.

1.3 Fallbeispiel

Dem Fallbeispiel des Buches liegt folgendes fiktives Szenario zugrunde: Die Firma »eHome-Tools« entwickelt Systeme zur Hausautomation. Das Funktionsprinzip solcher Systeme ist folgendes:

Fallbeispiel eHome-Controller

- **Aktoren:**
  Lampen und andere elektrische Verbraucher werden mit elektronischen Schaltern verbunden (sog. Aktoren). Jeder Aktor ist (per Kabel- oder Funkverbindung) an einen Kommunikationsbus angeschlossen und über diesen »fernsteuerbar«.
- **Sensoren:**
  An den Bus können zusätzlich elektronische Temperaturfühler, Windmesser, Feuchtigkeitssensoren usw. angekoppelt werden, aber auch einfache Kontaktsensoren, die z.B. geöffnete Fenster erkennen und melden.
- **Bus:**
  Schaltkommandos an die Aktoren, aber auch Statusmeldungen der Aktoren und Messwerte der Sensoren werden in Form von sogenannten Telegrammen über den Bus von und zum Controller übertragen.
- **Controller:**
  Der Controller sendet Schaltkommandos an die Aktoren (z.B. »Licht Küche ein«) und empfängt Statusmeldungen der Sensoren (z.B. »Temperatur Küche 20 Grad«) und Aktoren (z.B. »Licht Küche eingeschaltet«). Er ist in der Lage, ereignisgesteuert (also abhängig von eingehenden Meldungen) oder auch zeitgesteuert Folgeaktionen auszulösen (z.B. »20:00 Uhr → Rollo Küche schließen«).
- **Bedienoberfläche:**
  Der Controller bietet den Bewohnern des eHome auch eine geeignete Bedienoberfläche. Diese visualisiert den aktuellen Status des eHome und ermöglicht es den Bewohnern, Befehle (z.B. »Licht Küche aus«) per »Mausklick« an die Hauselektrik zu senden.

»eHome-Tools« steht mit seinen Produkten im harten Wettbewerb zu einer Vielzahl von Anbietern. Um in diesem Wettbewerb zu bestehen, wird beschlossen, eine neue Controller-Software zu entwickeln. Allen Beteiligten ist klar, dass Schnelligkeit ein wesentlicher Erfolgsfaktor für

das Vorhaben ist. Denn immer mehr Interessenten und Kunden fragen »eHome-Tools« nach einer Bediensoftware, die auf Smartphones und anderen mobilen Geräten läuft. Auch die Offenheit und Erweiterbarkeit des Systems für Geräte von Fremdherstellern ist enorm wichtig, um Marktanteile hinzuzugewinnen. Wenn das neue System Geräte konkurrierender Hersteller steuern kann, rechnet man sich Chancen aus, auch Kunden dieser Hersteller z. B. beim Ausbau ihrer Systeme für eigene Geräte begeistern zu können. Dazu muss man nicht nur möglichst schnell eine möglichst breite Palette von Wettbewerbs-Hardware unterstützen, sondern auch künftig in der Lage sein, neu am Markt auftauchende Gerätemodelle kurzfristig zu unterstützen.

Daher wird entschieden, den Controller »agil« zu entwickeln und monatlich eine verbesserte, neue Version des Controllers herauszubringen, die mehr Geräte und weitere Protokolle unterstützt.

1.4 Webseite

Die im Buch enthaltenen Codebeispiele sind auf der Webseite zum Buch unter [URL: SWT-knowledge] veröffentlicht und herunterladbar. Der Leser kann diese Beispiele so auf seinem eigenen Rechner nachvollziehen und mit eigenen Testfällen experimentieren.

Auch die Übungsfragen sind dort veröffentlicht und kommentierbar. Über eine rege Onlinediskussion im Leserkreis über mögliche Lösungsalternativen freue ich mich.

Trotz der hervorragenden Arbeit des Verlags und der Reviewer und mehrerer Korrekturläufe sind Fehler im Text nicht auszuschließen. Notwendige Korrekturhinweise zum Buchtext werden ebenfalls auf der Webseite veröffentlicht werden.

3 Planung im agilen Projekt

Kapitel 3 beschreibt, welche leichtgewichtigen Planungs- und Steuerungsinstrumente in Scrum anstelle des klassischen Projektplans zum Einsatz kommen. Denn »agil« Arbeiten bedeutet keineswegs »planlos« zu arbeiten. Das Kapitel richtet sich primär an Leser, die auf agile Entwicklung umsteigen. Die Erläuterungen und Hinweise, welchen Beitrag die jeweiligen Planungsinstrumente zur konstruktiven Qualitätssicherung und damit zur Fehlervermeidung liefern, sind jedoch auch für Leser mit agiler Projekterfahrung wertvoll.

Agiles Projektmanagement setzt darauf, dass das Team von Sprint zu Sprint dazulernt. Entscheidungen, die sich als suboptimal oder falsch erweisen, können im folgenden Sprint revidiert werden. Auf veränderte Rahmenbedingungen kann von Sprint zu Sprint reagiert werden. Jedes neue Produktinkrement, das Sprint für Sprint entsteht, liefert dem Team, aber auch dem Kunden neue Einsichten und tieferes Verständnis der Produktanforderungen. Und natürlich generiert jedes Inkrement neue und eventuell bessere Ideen, was das Produkt leisten sollte, aber auch neue Ideen und zusätzliche Erfahrungen dazu, wie das Produkt am besten und elegantesten implementiert werden kann.

All diese Ideen und Erfahrungen kann und muss das Team regelmäßig auswerten. Punkte, die einen Nutzen versprechen, werden in das Product Backlog aufgenommen. In jeder Sprint-Planung besteht dann die Möglichkeit, die aktuell am nützlichsten bewerteten Ideen zu realisieren, indem man entsprechende Entwicklungsaufgaben (Tasks) formuliert und diese in den Sprint aufnimmt.

Adaptive, empirische Planung

Diese adaptive, empirische Planung schafft eine sehr hohe Flexibilität. »Ständiges Dazulernen« ist im Prozess verankert. Ebenso die Bereitschaft, aber eben auch die Fähigkeit, kurzfristig auf veränderte Kundenwünsche und neue Rahmenbedingungen zu reagieren.

Die Fähigkeit, in jedem Sprint die Richtung ändern zu können, bedeutet aber nicht, dass dem Team keine Richtung vorgegeben werden muss. Wer nur reagiert, wird auch mit Scrum kein erfolgreiches

Projekt zustande bringen. Auch ein Scrum-Team benötigt Zielvorgaben, die über den aktuellen Sprint hinausreichen und auf die es über mehrere Sprints hinweg hinarbeiten kann. Die Instrumente, die helfen, eine Richtung vorzugeben, werden in den folgenden Abschnitten vorgestellt.

3.1 Produktvision

Die Produktvision ist ein Bild, das möglichst prägnant zusammenfasst, wie das Produkt einmal »aussehen« und was es einmal in Summe leisten soll. Je knapper und anschaulicher diese Botschaft formuliert wird, umso einprägsamer und besser. Eine Skizze am Flipchart kann ausreichen, eine Liste der Top-10-Features, die der Kunde sich wünscht, oder ein Bild des Wettbewerbsprodukts, das man übertreffen möchte.

Product Backlog als »Vision« ungeeignet

Das Product Backlog (s. Abschnitt 3.3) ist als »Vision« in der Regel ungeeignet. Dazu ist es zu detailliert und enthält zu viele Einträge. Im Gegenteil: Ein umfangreiches Backlog kann dazu führen, dass das Projektziel in der Masse der Backlog Items verloren geht. Außerdem muss die Produktvision ja gerade dabei helfen, die Backlog Items zu priorisieren und zielführende von weniger relevanten Items unterscheiden zu können. Das eHome-Team formuliert Folgendes:

Fallbeispiel eHome-Controller 3–1: Produktvision

Der Product Owner hat das Entwicklungsteam zu einem Workshop eingeladen, um den Projektauftrag zu besprechen und gemeinsam eine Produktvision zu entwickeln. Das Team geht dabei von den Vorgaben der Geschäftsleitung und wichtigen Kundenwünschen aus.

Das System soll auf Smartphones und anderen mobilen Geräten laufen und es muss Geräte von Fremdherstellern steuern und jederzeit neu am Markt erscheinende Geräte einbinden können. Darauf basierend formuliert der Product Owner in Zusammenarbeit mit seinem Team im Laufe des Workshops folgende Entwicklungsziele:

- Webbrowser-basierte Bedienoberfläche:
  Das heißt, der Bewohner des eHome kann seine Wohnung oder sein Haus per Browser vom PC aus steuern oder auch per Smartphone oder Tablet. Da die Software im Browser läuft, entfällt für den Endanwender eine aufwendige und komplizierte Softwareinstallation.
- Einfache, anschauliche Bedienung:
  Alle im Haus vorhandenen Zimmer und die dort steuerbaren Geräte sollen durch Icons oder (vom Anwender hochladbare) Fotos dargestellt werden. Die Bedienoberfläche soll nicht »technisch« aussehen. Auch IT-Laien sollen das System bedienen können.

→

5 Integrationstests und Continuous Integration

In diesem Kapitel wird erklärt, wie sich Integrationstests von Unit Tests unterscheiden, wie das Team Integrationstestfälle entwirft und diese zusammen mit den Unit Tests in einer Continuous-Integration-Umgebung vollautomatisiert und kontinuierlich durchführt.

5.1 Integrationstests

Ein Softwaresystem besteht aus vielen Einzelbausteinen. Damit es in Betrieb gehen kann, muss nicht nur jeder Einzelne dieser Bausteine korrekt und zuverlässig funktionieren, sondern alle Bausteine müssen in der vorgesehenen Weise korrekt miteinander zusammenarbeiten. Dies zu prüfen ist die Aufgabe von Integrationstests.

Integrationstests prüfen das Zusammenspiel von Bausteinen.

Integrationstests haben das Ziel, Fehlerzustände im Zusammenspiel und damit »in den Schnittstellen« der beteiligten Bausteine aufzufinden. Um einen Integrationstest durchzuführen, werden jeweils zwei Bausteine miteinander verbunden (integriert) und dann durch geeignete Integrationstestfälle »angesteuert«. Abbildung 5–1 zeigt dies schematisch am Beispiel zweier Klassen.

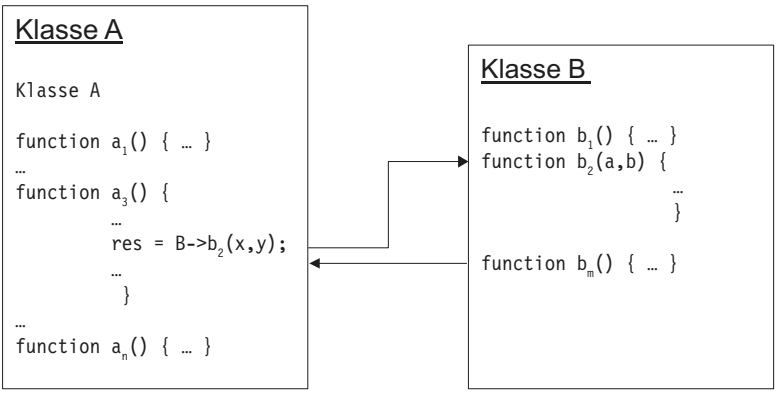

Abb. 5–1
Integrationstest am Beispiel zweier Klassen

Die Schnittstelle (Interface) von Klasse A besteht aus den Methoden `a1,...,an` und das Interface von Klasse B aus den Methoden `b1,...,bm`. Beide Klassen haben ihre vorangegangenen Unit Tests erfolgreich »bestanden«. Die Integrationstestfälle müssen jetzt zusätzlich noch prüfen, ob A und B korrekt zusammenarbeiten. Diese Wechselwirkung erfolgt (im o.g. Beispiel) über die Methode `b2`, die B bereitstellt und die A innerhalb seiner Methode `a3` aufruft[1]. Zu prüfen ist im Beispiel also, ob `b2` richtig aufgerufen wird und ob `b2` dann die gewünschten Ergebnisse an A zurückliefert.

5.1.1 Typische Integrationsfehler und Ursachen

Obwohl A und B jeder für sich korrekt funktionieren, kann im Zusammenspiel beider Bausteine eine Reihe von Fehlerwirkungen auftreten. Die erste Gruppe sind **Schnittstellenfehler**:

- **A ruft eine falsche Methode von B auf:**

  Angenommen, A ist eine Klasse innerhalb der Bedienoberfläche des eHome-Controllers und B die Klasse, die Geräte repräsentiert. Um das Küchenlicht auszuschalten, müsste A beispielsweise die Methode `switch('Kueche', 'Licht', 'off')` in B aufrufen; im Programmtext von A ist aber fälschlicherweise `dim('Kueche', 'Licht', '50')` codiert.

- **A ruft die richtige Methode von B auf, aber mit anderen Parameterwerten, als es für das korrekte Verhalten nötig ist:**

  Beispiel: A soll das Küchenlicht anschalten. Aufgerufen wird aber `switch('Wohnzimmer', 'Licht')`. Der erste Parameter ist semantisch falsch. Der dritte Parameter fehlt.

- **Die beiden Bausteine codieren die übergebenen Daten unterschiedlich:**

  So bedeutet in A der Aufruf `dim('Kueche', 'Licht', '50')`, dass das Licht auf 50 % Helligkeit gedimmt werden soll. Die Helligkeit wird als Wert zwischen `0` und `100` codiert. Die `dim()`-Methode in B erwartet den Helligkeitswert aber als Gleitkommazahl zwischen `0` und `1` codiert, hier also als Wert `0.5`.

1. A benötigt B (genauer ein Objekt der Klasse B), um lauffähig zu sein. Oder anders ausgedrückt: A hängt von B ab.

Fazit

Insgesamt zieht Dr. Stephan Albrecht ein positives Fazit: »Ein absoluter Vorteil von Scrum ist das sehr enge Zusammenspiel von Entwicklung und Test und die dadurch möglichen kurzen Zyklen. Für den Tester wächst damit allerdings die Herausforderung, sich seine Objektivität und Distanz gegenüber dem Testobjekt zu bewahren.«

8.2 Systemtest nonstop – Scrum in der Test*Bench*-Toolentwicklung

Interview (2012) mit Joachim Hofer, Entwicklungsleiter TestBench, und Dierk Engelhardt, Produktmanager TestBench, imbus AG, Möhrendorf

imbus ist ein spezialisierter Lösungsanbieter für die Qualitätssicherung und das Testen von Software. imbus ist mit über 200 Mitarbeitern (Stand 2012) an den Standorten Möhrendorf bei Erlangen, München, Köln, Hofheim bei Frankfurt und Shanghai/China vertreten. Das Angebot umfasst die Beratung, Softwaretest-Services, Testoutsourcing, Testwerkzeuge und Training. Kunden sind Softwarehersteller, Softwarehäuser und mit Softwareentwicklung befasste Abteilungen aus Behörden und Unternehmen aller Branchen.

Die Test*Bench* ist ein von imbus entwickeltes, leistungsstarkes Testmanagementwerkzeug. Test*Bench* deckt von der Testplanung, dem Testdesign und der Testautomatisierung bis zur Testdurchführung und dem Reporting alle Aufgaben ab, die im Softwaretest anfallen. Eingesetzt wird Test*Bench* von Kunden aus der Medizintechnik, der Verkehrstechnik, der Automobilindustrie, aber auch in Banken und Versicherungen.

Das Werkzeug ist in Java entwickelt worden. Das Test*Bench*-Team bei imbus besteht aus 12–16 Mitarbeitern. Die Einführung und Integration des Produkts beim Kunden leisten weitere Test*Bench*-Produktberater. Bis 2010 erfolgte die Entwicklung nach einem iterativen, aber phasenorientierten Entwicklungsprozess. Programmierer und Tester arbeiteten in zwei Gruppen getrennt. Das Team produzierte ein bis zwei Major-Releases jährlich.

Verbesserungsziele

Dieser Entwicklungsprozess führte zu den aus phasenorientierten Vorgehensmodellen bekannten typischen Schwierigkeiten: Der Systemtest startete nach Abschluss der Implementierung, als die Programmierer sich bereits freuten, dass die Iteration abgeschlossen war. Die Systemtester dämpften diese Freude dann regelmäßig mit einer Serie von Feh-

lermeldungen. Aus Sicht der Programmierer kamen diese Fehlermeldungen unnötig spät und adressierten leider auch Probleme, die Eingriffe nötig machten, für die in dieser Iteration eigentlich keine Zeit mehr war. Jede Iteration zerfiel dadurch in zwei Hälften: die Implementierung und das anschließende mühsame Bugfixing unter einem enormen Zeit- und Erfolgsdruck. Am Ende hatte man ein gutes, stabiles Produkt. Aber der Weg dorthin war mühsam.

Das Ziel war daher, Programmierung und Test zu parallelisieren, um so ein schnelleres Feedback von den Testern an die Programmierer zu ermöglichen. Durch Ausbau der Testautomatisierung sollten darüber hinaus mehr Möglichkeiten und mehr Sicherheit für Code-Refactoring erreicht werden, als Grundlage für den weiteren Ausbau des Produkts.

Einführung agiler Entwicklungstechniken

Um diese Ziele zu erreichen, führte Entwicklungsleiter Joachim Hofer ab 2010 eine Reihe agiler Techniken ein:

- **Anforderungsmanagement:**
  Bisher war eine Anforderung eine Headline im Anforderungsmanagementtool »Caliber«, verknüpft mit einem detaillierten Anforderungsdokument in Word. Erst wenn das Anforderungsdokument als Ganzes freigegeben war, konnte die Implementierung beginnen. Hier wurde entschieden, von den monolithischen Anforderungsdokumenten abzugehen und stattdessen kleinere User Stories in »Jira«, einem Produktverfolgungstool für Teams, zu erfassen, die sukzessive entstehen konnten.
- **Nightly Build:**
  Verschärft wurden auch die Spielregeln für den »Nightly Build«. Eine entsprechende Umgebung, in der jede Nacht das Produkt compiliert und integriert wurde, gab es bereits seit einigen Jahren. Die Programmierer stellten ihren Code dort ein, sobald sie glaubten, ihr Arbeitspaket fertig programmiert zu haben (im Schnitt alle 2–3 Tage). Deshalb gab es immer eine gewisse Menge an unfertigem Code außerhalb des Build. Ab sofort hatte jeder Programmierer die Pflicht, seinen Code jeden Abend einzuchecken. Die vorhandenen automatisierten Unit Tests wurden damit ab sofort jede Nacht auf dem vollständigen Code ausgeführt. Zunächst krachte es dabei natürlich an allen Ecken. Das Ergebnis war aber der erhoffte Lernprozess: Man arbeitete sorgfältiger auf fertigen, lauffähigen Code an jedem Abend hin. Die »Code-Päckchen«, die man morgens anpackte, wurden kleiner, und abends war die jeweilige Änderung fertig.

- **Automatisierte Systemtests als »Nightly Tests«:**
  Zusätzlich zum Ausbau der automatisierten Unit Tests wurde der automatisierte Ablauf von Systemtests vorangetrieben. Hierzu wurde die Testumgebung so weiterentwickelt, dass (im Anschluss an die Unit Tests und Integrationstests) die automatisierten Systemtests aus der Build-Umgebung heraus jeden Abend als Nightly Tests gestartet wurden. Jeder neue Systemtest wurde ab sofort so entworfen und mit dem Tool »QF-Test« implementiert, dass er in diese Nightly-Tests-Umgebung eingefügt werden konnte.
- **Continuous Integration:**
  Um einen vollständigen Build zu erzeugen, wurden damals 4 Stunden benötigt. Das war gerade noch nicht zu langsam für einen regelmäßigen Nightly Build. Aber um die anschließenden Nightly Tests vor dem »Schichtbeginn« am nächsten Morgen sicher abzuspulen, wurde es schon knapp. Durch Zerlegung des Build in unabhängige Teil-Builds und den weiteren Umbau der Build-Umgebung (u.a. Aufsetzen einer Jenkins/Hudson-Umgebung) konnte die Build-Zeit auf 15 Minuten reduziert werden. Alle automatisierten Unit Tests und Integrationstests sowie die Systemtests aus dem Nightly Build wurden neu paketiert und in diese verbesserte Build-Umgebung eingebunden. Je nachdem, welche Testpakete man ausführen ließ, konnte damit eine Feedbackzeit je Build von minimal 15 Minuten erreicht werden. Nicht mehr als eine Kaffeepause!
- **Statische Codeanalyse und Coverage-Messung:**
  In der Continuous-Integration-Umgebung wurde den dynamischen Unit Tests eine statische Codeanalyse nachgeschaltet, die dann parallel zu den Integrationstestfällen läuft. Hier wird u.a. mit dem Werkzeug »FindBugs« der Java-Code auf Java-typische Probleme untersucht (z.B. nicht korrekt genutzte API-Aufrufe).
- **Taskorientiertes Arbeiten:**
  Um innerhalb jeder Iteration eine feingranulare Aufgabensteuerung zu erreichen, wurde – ausgehend von den User Stories – taskorientiertes Arbeiten eingeführt. Zur Priorisierung der Tasks wurde ein Scoring-System eingeführt. In den Score gehen u.a. die Priorität der Anforderung aus Kundensicht, die Anzahl entsprechender Kundenwünsche und ein teaminternes Voting ein.
- **Daily Standup:**
  Das Team trifft sich ab sofort jeden Morgen für 15 Minuten. Jeder berichtet, woran er gerade arbeitet, wie er vorankommt und ob es Probleme gibt. Jeder entscheidet selbst, über welche Tasks er berichtet. Eine echte Sprint-Planung, gegen die man die Tasks spie-

geln kann, gibt es noch nicht. Die Kultur des »Daily Scrum« kann aber schon hervorragend eingeübt werden.

Alle diese Techniken wurden unter Beibehaltung des früheren iterativen Entwicklungsmodells eingeführt. Durch User Stories und Continuous Integration hatten sich die ursprünglichen Entwicklungsphasen allerdings schon weitgehend aufgelöst bzw. wie angestrebt miteinander verschränkt und parallelisiert. Was jetzt noch fehlte, war die Einführung agiler Techniken im Produkt- und Projektmanagement und in der Teamführung.

Einführung von Scrum

Während die Einführung der o.g. Entwicklungstechniken hauptsächlich vom Entwicklungsleiter Joachim Hofer getrieben wurde, ging es nun darum, das ganze Team einzubeziehen. Joachim Hofer und Produktmanager Dierk Engelhardt entschieden sich für ein Umsteigen in mehreren Schritten:

- **»Informationen sammeln und informieren« stand am Anfang:**
  Zur Debatte standen Kanban, Scrum, XP und andere agile Methoden. Relativ schnell fokussierten sich die Diskussion und die weitere Planung aber auf Scrum. Es wurde ein internes Diskussionsforum eingerichtet, in dem das Team seine Überlegungen und Vorschläge für den neuen Entwicklungsprozess nach Scrum austauschte. Über Literaturstudium und Konsultation einschlägiger Internetforen (z.B. scrum.org) informierte sich jeder im Team selbstständig – geleitet vom eigenen Interesse. Dieser Prozess wurde begleitet durch interne Workshops und regelmäßige Teamdiskussionen.
- **Neue Aufstellung des Teams:**
  Die Umstellung auf Scrum erforderte natürlich auch die Veränderung von Rollen, Verantwortlichkeiten und Aufgaben. Joachim Hofer tauschte seine Rolle in die des Scrum Masters. Die bisherige Testmanagerin wurde seine Stellvertreterin und der Produktmanager übernahm die Position des Product Owners. Die Trennung zwischen Testteam und den Programmierern wurde aufgehoben. Stattdessen wurde »Pairing« (i.d.R. bei Unit-Test-/Integrationstestaufgaben: 1 Tester, 1 Entwickler; bei Entwicklungsaufgaben: 2 Entwickler; bei Systemtestaufgaben: 2 Tester) als Arbeitsmodell eingeführt.
- **»Sprinten«:**
  Es gibt keinen idealen oder »leichten« Zeitpunkt, an dem ein Team den alten Entwicklungsprozess hinter sich lässt und seinen ersten Sprint startet. Man muss einfach beginnen. Und so fand an einem

Montag Anfang 2011 das erste Sprint-Planungsmeeting statt. Der Product Owner hatte im Vorfeld die aus seiner Sicht wichtigsten Anforderungen aus seiner alten Planung in ein initiales, überschaubares Product Backlog übertragen. Dieses Backlog wurde in einer eintägigen Sprint-Planung vom Team bearbeitet, und als Resultat hatte das Team abends die Taskkarten für den ersten 4-Wochen-Sprint am Whiteboard angebracht (später wechselte das Team auf 3 Wochen).

Wesentliche Veränderungen

- Die Auflösung der klassischen Rollen »Tester« und »Entwickler« hin zu Teammitgliedern mit Know-how-Schwerpunkt wurde von den Beteiligten als gravierendste Veränderung empfunden. Die Einführung von »Pairing« ist ein wesentlicher Baustein, um diese Rollenänderung erfolgreich umzusetzen.
- Durch das Pairing werden Codereviews gängige Praxis: Jeder Programmierer gibt neuen Code ganz selbstverständlich an seinen jeweiligen Partner zum Review. Diese Praxis sorgt auch für laufenden Know-how-Austausch im Team.
- Die Meilenstein-/Workpackage-orientierte Aufgabenplanung wurde durch eine taskorientierte Aufgabensteuerung ersetzt. In diesem Zuge trat das Anforderungsmanagementtool »Caliber« in den Hintergrund und die Taskverwaltung mit »Jira« und Plug-in »Greenhopper« in den Vordergrund für Backlog-Verwaltung, Task-Ranking, Taskboard-Verwaltung und Metriken/Charts.
- Die wesentlichen Scrum-Techniken (Backlog, Sprint Planning mit Planning Poker, Timeboxing, Retrospektiven) wurden erfolgreich eingeführt und werden diszipliniert und nachhaltig angewendet.
- Test-Driven Development: Das Format der User Story erlaubte, eine verschärfte Spielregel einzuführen. Zu jeder Codeänderung und zu jeder User Story in »Jira« muss das Team Testfälle entwerfen – und zwar vor Beginn der Programmierung. Bisher war es erlaubt, über Testfälle irgendwann im Anschluss an eine Codeänderung nachzudenken und diese zu schreiben. Dabei sind für eine User Story jeweils 2 Personen, ein Programmierer und ein Tester, gemeinsam zuständig. Sie analysieren die User Story und entwerfen die aus ihrer Erfahrung heraus notwendigen Tests und entscheiden, auf welcher Ebene (Unit Test, Integrations-, Systemtest) bzw. mit welchen Werkzeugen diese Tests zu realisieren sind. Der Programmierer setzt dann die Tests auf Unit-Test- und Integrationstestebene im Unit Test Tool »TestNG« um. Der Tester automatisiert die Systemtestfälle in »QF-Test«.

- Durch Einführung von Continuous Integration wurde die Parallelisierung von Programmierung und Test wie gewünscht erreicht. Jede Codeänderung, die eingecheckt wird, löst sofort einen Build-Lauf aus – im Minimum bestehend aus Kompilierung und Unit Tests. Die Laufzeit beträgt etwa 3 Minuten bis zum Feedback an den Programmierer. Danach folgen die Integrationstests mit ca. 15–30 Minuten Laufzeit.
- Zusätzlich zur Continuous Integration (die von den Programmierern durch Code-Check-in mehrmals täglich ausgeführt wird) startet jeden Abend ein »Nightly Build«. Hier wird vollautomatisch eine Virtual Machine mit frischem Betriebssystem gestartet, und der Test beginnt mit der Installation des Produkts aus dem letzten erfolgreichen Stand der Continuous Integration. Dann laufen die automatisierten Systemtests ab. Der Umfang dieser Tests liegt hier derzeit bei ca. 15.000 Testschritten (datengetrieben aus der Test*Bench* heraus) mit einer Laufzeit von ca. 10 Stunden. Auch hier wird der Code vorher automatisch instrumentiert. Die erreichte Coverage beträgt derzeit ca. 40 % Line Coverage. Von diesen nächtlichen Systemtests wird außerdem eine automatische Videoaufzeichnung erstellt. Fehlschlagende Tests können so am nächsten Morgen am Bildschirm in ihrem Verlauf nachverfolgt werden. Die durch die Continuous-Integration-Umgebung erreichte Coverage liegt insgesamt bei ca. 60 %. Für neu implementierte Features bzw. User Stories erreicht man hier nahezu 100 %. Aber älterer Code, für den noch keine umfassende Testautomatisierung existiert, drückt leider nach wie vor die Gesamt-Coverage nach unten.
- Die früheren Testspezifikationen wurden weitgehend abgelöst durch den kommentierten Testcode und im Systemtest durch schlüsselwortbasierte, maschinell ausführbare Testspezifikationen. Im Systemtest wird sehr erfolgreich Test*Bench* eingesetzt, über die alle Systemtests verwaltet und gemanagt werden. Test*Bench* wurde dazu eng mit Jira und Jenkins verknüpft.
- Da die Systemtests nicht zu 100 % automatisiert sind und auch künftig manuelle Tests nötig sein werden (z.B. zur Überprüfung von Reportgrafiken, Usability etc.), findet am Sprint-Ende noch zusätzlich ein halbtägiger sessionbasierter explorativer Systemtest durch in der Regel zwei Tester statt. Der manuelle Testaufwand hat sich allerdings drastisch reduziert, von mehreren Personenwochen im früheren iterativen Vorgehen auf heute einen Personentag alle 3 Wochen.
- Die Sprints laufen im 3-wöchigen Rhythmus und haben ein getestetes, internes Produktrelease zum Ergebnis. Externe Releases erfolgen weiterhin 2-mal jährlich. Dabei kann auch im letzten Sprint,

also bis 3 Wochen vor Auslieferung, auf geänderte Kundenwünsche reagiert werden.

Lessons Learned

- Die Einführung agiler Entwicklungstechniken (wie z.B. Continuous Integration) schon vor der organisatorischen Umstellung auf Scrum schuf einen stabilen Unterbau, auf dem die Sprints vom ersten Sprint an sicher ablaufen konnten.
- Um die Toolinfrastruktur (im Wesentlichen für Continuous Integration) aufzubauen, ist ein signifikanter Initialaufwand nötig. Das darf nicht unterschätzt werden. Diese Aufbauarbeit mindert die »Feature-Produktivität« in den ersten Sprints.
- »Pairing« ist eine erfolgsentscheidende Maßnahme. Aber nicht jeder kann mit jedem. Die Paare müssen sich selbst finden.
- Test-Driven Development verbessert die Codearchitektur und reduziert die Anzahl der zu verwaltenden Fehlermeldungen erheblich.
- Die regelmäßigen Retrospektiven liefern kontinuierlich Ansätze für weitere Verbesserungen (z.B. »Aufwandsschätzung verbessern«, »was genau ist ein Story Point«).
- Die Menge der Arbeit wird durch Scrum nicht weniger! Scrum erzeugt keine zusätzlichen Ressourcen. Aber: Die Arbeiten, die »done« sind, sind erledigt und abgeschlossen. Es gibt keine großen Fehlerberge mehr, die in Bugfixing-Runden mühsam abgetragen werden müssen.
- Die Produkt-Roadmap wird zur strategischen Planung.
- Das Umstellen auf Scrum geht nicht nebenher, sondern erfordert Zeit für Informationsbeschaffung, Training und Infrastrukturaufbau und das Commitment der Teamführung und des ganzen Teams.

Fazit

»Die Verbesserungsziele, die wir uns gesetzt hatten, konnten wir erreichen«, bestätigen Dierk Engelhardt und Joachim Hofer als Fazit nach einem Jahr Scrum. Das Team wendet heute ein Jahr nach Start des Umstiegs wesentliche agile Techniken, wie zum Beispiel Test-Driven Development und Continuous Integration, diszipliniert und nachhaltig an. Test-Driven Development und Null-Fehler-Strategie wirken spürbar. Durch die Integration von Test*Bench* und Jenkins wird »Test nonstop« realisiert, also kontinuierliches, automatisiertes Testen vom Unit Test bis zum Systemtest. Auch umfangreiche Refactoring-Eingriffe sind so risikoarm machbar. Der Aufwand zur Produktion eines exter-

nen Release ist deutlich gesunken. Im Team herrscht eine hohe Zufriedenheit.

Auf den erreichten Erfolgen will sich das Team nicht ausruhen. Auf dem Programm stehen schon die nächsten Verbesserungsmaßnahmen, u.a. der Einsatz von »Atlassian Confluence« zur Beschreibung der Systemanforderungen und die weitere Beschleunigung der Continuous Integration durch Parallelisierung der Testläufe und durch Ausbau der Build- und Test-Server-Hardware.

8.3 Scrum in der Webshop-Entwicklung

Interview (2012) mit Sabine Herrmann,
Agile Testerin bei der zooplus AG in München

Die zooplus AG ist im Geschäftsfeld E-Commerce im Handel mit Heimtierprodukten für den Privatkundenbereich tätig. Insgesamt bietet die zooplus AG ihren Kunden rund 8.000 Produkte in den Gattungen Hund, Katze, Kleintier, Vogel, Reptil, Aquaristik und Pferd an. Diese umfassen Produkte des täglichen Bedarfs wie fachhandelsübliches Markenfutter, Eigenmarken sowie auch Spezialartikel wie Spielzeug, Pflegeprodukte oder sonstige Accessoires. Auf ihren Webseiten bietet die zooplus AG zudem diverse kostenfreie Informationsangebote, tierärztliche Beratung sowie interaktive Anwendungen wie Diskussionsforen und Blogs an.

Nach Gründung im Jahr 1999 und anfänglicher Fokussierung der Kernaktivitäten auf Deutschland und Österreich steht seit 2005 die europaweite Expansion des Unternehmens im Zentrum des unternehmerischen Handelns. Das Unternehmen wuchs in den vergangenen vier Jahren in Bezug auf den Umsatz um jeweils mehr als 33 % jährlich, das Gleiche gilt für die Mitarbeiteranzahl.

Die zooplus AG hat aktuell 200 Mitarbeiter und Mitarbeiterinnen, davon 51 in der IT und 102 im Bereich Marketing.

Vorher

Mit der schnellen und erfolgreichen Expansion der Firma in den letzten 10 Jahren ist die IT in einem kurzen Zeitraum sehr schnell gewachsen.

Vor der Umstellung auf Scrum in 2008 gab es zwei Entwicklungsteams – ein Shop-Team und ein Backend-Team. Zu diesem Zeitpunkt waren neun Shops bei der zooplus AG online.

Für das Testen neuer Shop-Funktionalitäten (Integrationstests) und der Basisfunktionalität (Regressionstests) gab es ein separates Testteam.