O'REILLY®

3. Auflage

Scrum

kurz & gut

O'Reillys
Taschenbibliothek

Rolf Dräther
Holger Koschek
Carsten Sahling

Inhalt

KAPITEL 4

Scrum im Einsatz

Alle vorangegangenen Kapitel haben Scrum vorwiegend aus der Sicht der »reinen Lehre« betrachtet. Das nun folgende Kapitel beleuchtet Themen, die für den konkreten Einsatz von Scrum in der Produktentwicklung von Bedeutung sind. Dabei wird der Bogen von der Vision über Anforderungen, iterativ-inkrementelles Vorgehen und Release-Management bis hin zur Wartung (Maintenance) gespannt. Der Bezug zur praktischen Erfahrung soll den Leserinnen und Lesern helfen, ihren eigenen erfolgreichen Weg mit Scrum zu finden. Das Kapitel schließt mit einem kurzen Ausblick auf verschiedene Bereiche agilen Vorgehens, darunter Agile Software Engineering, Software Craftsmanship und das Arbeiten mit mehreren oder mit verteilten Scrum-Teams.

Produktentwicklung mit Scrum

Wer Scrum richtig anwenden und die Arbeitsweise eines selbststeuernden Teams erleben will, muss die Prinzipien hinter der Mechanik kennen und verstehen. In diesem Kapitel beschreiben wir den Werdegang eines Produkts. Wir beginnen mit der ersten Idee und der Vision sowie den ersten konkreten Anforderungen, die priorisiert und geschätzt werden. Anschließend gehen diese Anforderungen in die Entwicklung und durchlaufen dort den Scrum-Zyklus. Iterativ-inkrementell entsteht das Produkt. Es wird in Releases veröffentlicht, die in einem oder mehreren Sprints entwickelt werden. Der Übergang von der Entwicklungs- zur Wartungsphase (engl. *Maintenance*) ist dabei oft fließend und manchmal überhaupt nicht wahrnehmbar, weil sich das Scrum-Team um den gesamten Lebenszyklus »seines« Produkts kümmert.

Vision und Ziele

An dieser Stelle wollen wir noch einmal darauf hinweisen, dass der Begriff »Vision« seit 2020 im Scrum Guide nicht mehr auftaucht und die Vision damit im engeren Sinn nicht zu Scrum gehört. Trotzdem hat es unserer Erfahrung nach eine sinnvolle Berechtigung, in Richtung einer Vision zu denken und zu agieren, da Ziele (auch Produkt-Ziele) erreichbar sind und sich ändern können, eine Vision hingegen stabil bleibt. Sie dient als den Zielen übergeordnete Instanz und bildet die Grundlage der Produktentwicklung.

Visionen und Ziele sind zwei grundsätzlich verschiedene Dinge, die häufig miteinander verwechselt werden. In diesem Abschnitt werden wir zunächst beide Begriffe definieren, die Unterschiede herausstellen und beschreiben, wer für Vision und Ziele zuständig ist und wie man in der Praxis mit ihnen umgehen sollte.

Vision

Die Vision ist der Nordstern, an dem sich ein Produkt orientiert. Fehlt diese Orientierung, ist eine sinnvolle Priorisierung der Anforderungen nicht möglich.

Aber was genau ist eine Vision?

Eine Vision ist eine emotionale, mitreißende Formulierung, die den Rahmen für das Produkt absteckt. Sie muss nicht unbedingt realistisch erreichbar sein, aber die Richtung vorgeben. Wenn man zweifelt, wie ein bestimmtes Feature umzusetzen ist, sollte man die Vision anschauen und dort eine Antwort finden können.

Die Vision beschreibt die Zielgruppe(n), für die das Produkt gedacht ist, deren Bedürfnisse und einige Schlüsselfunktionen, die diese Bedürfnisse befriedigen – vielleicht drei bis fünf Funktionen, jedoch nur die wichtigsten, damit die grundsätzliche Produktidee verständlich wird.

Ein Beispiel, das in diesem Zusammenhang helfen kann, stammt aus dem Jahr 2007, hat aber trotz des Alters nicht an Strahlkraft verloren. Steve Jobs hatte bereits Anfang der 2000er-Jahre die Idee, ein revolutionäres Mobiltelefon zu bauen. Die Zielgruppe war in dem Fall klar (alle, die ein Telefon nutzen), und die Bedürfnisse hat er ebenfalls sehr gut beschrieben: Ein Multi-Touch-Bildschirm als neues Eingabemedium, bessere Bedienbarkeit und ein höherer Funktionsumfang, denn die übliche Tastatur der Smartphones passt sich nicht an die unterschiedlichen ausführbaren Programme an, die Bedienung ist alles andere als einfach und der Funktionsumfang recht mager. Die Schlüsselfunktionen sind ebenfalls schnell aufgezählt: ein iPod mit Touch

Control, ein Mobiltelefon mit revolutionärer Benutzeroberfläche und ein mächtiger Internetclient, dazu ein ausgereiftes Betriebssystem und ein stilvolles Aussehen. Wer die Emotionalität und das Mitreißende dieser Vision erleben möchte, sollte bei YouTube nachschauen (Suchwörter »Steve Jobs Vision iPhone«) und sich am Szenenapplaus erfreuen.

Ziele

Ziele hingegen sind etwas völlig anderes: Ein Ziel ist die Beschreibung eines zu erreichenden Zustands im Sinne einer Vorgabe, die es zu erfüllen gilt. Selbstverständlich möchte man mit einem Produkt bestimmte Ziele erreichen: Umsätze, Marktanteile oder Ähnliches. Ziele sind also messbar und erreichbar. Eine Vision hingegen gibt nur die Richtung vor und schafft damit Gestaltungsfreiraum.

Auf Produktebene (Produkt-Ziel) sind Ziele ohne Frage sinnvoll. Sie beschreiben den nächsten großen Schritt in Richtung der Vision. Oder wie es der Scrum Guide formuliert: »Das Produkt-Ziel ist das langfristige Ziel für das Scrum-Team. Das Scrum-Team muss eine Zielvorgabe erfüllen (oder aufgeben), bevor es die nächste angeht.«

Auch auf Sprint-Ebene (Sprint-Ziel) ist es sinnvoll, an einem Ziel zu arbeiten. »Das Sprint-Ziel ist die einzige Zielsetzung für den Sprint«, sagt der Scrum Guide. Mit anderen Worten: Die Auswahl der einzelnen Product Backlog Items orientiert sich an einem Sprint-Ziel, das es zu erreichen gilt, ohne die eigene Kreativität dadurch zu beschränken.

Denn das ist die Gefahr bei der Vorgabe von konkreten, klein gefassten Zielen:

Wenn wir von Teamleistung sprechen und davon, wie man ein Team motiviert, wirklich Herausragendes zu leisten, darf man keine kleinteiligen Ziele vorgeben. Dies wäre Mikromanagement, das die Kreativität der Developer tötet und sie zu Befehlsempfängern degradiert, die lediglich Vorgaben umsetzen. Wirklich gute Produkte entstehen immer dann, wenn Menschen mitdenken, eigene Ideen einbringen dürfen und deshalb Spaß an ihrer Arbeit haben. Das funktioniert aber nur, wenn man mit einer guten Vision die Leitplanken definiert, innerhalb derer die Teammitglieder ihrer Kreativität freien Lauf lassen können.

Zu kleinteilige Ziele engen den Horizont ein, denn man sucht nicht mehr nach Alternativen, sondern möchte einfach das vorgegebene Ziel erreichen. Dadurch verbaut man sich möglicherweise die »bessere Lösung«. Als vergleichbare Situation mag der vom Lebenspartner geschriebene Einkaufszettel dienen, den man »abarbeiten« muss. Man konzentriert sich nur auf die Dinge auf dem Zettel (die Zieldefinition) und schaut nicht, was es an sinnvollen Ergänzungen und

Alternativen gäbe. So verpasst man Sonderangebote, besonders gut aussehende frische Ware oder eben einfach passende Alternativen oder Ergänzungen zur Zieldefinition.

Anforderungen (Features)

Die Anforderungen, die wir im nächsten Abschnitt etwas genauer beleuchten wollen, dienen dazu, die Vision eines Produkts zu erfüllen. Während Ziele wie oben beschrieben messbar und erreichbar, aber eben abstrakt sind, stellen Anforderungen (im Folgenden auch als »Features« bezeichnet) konkrete Eigenschaften dar, die vom Produkt gefordert werden. Immer wenn man überlegt, wie man eine Anforderung konkret umsetzen soll, kann ein Blick auf die Vision helfen. Im Beispiel des iPhone hat sich Steve Jobs vielleicht gefragt, welches Eingabemedium er verwenden will. Eine Tastatur kam auf keinen Fall infrage, denn die hatten die gängigen Smartphones damals alle. Die nächste Überlegung war vielleicht, ob man eine Art Stift verwenden sollte, aber auch das erfüllte nicht den Anspruch des »Revolutionären«. Und so ist Jobs schlussendlich auf das natürlichste Eingabemedium der Welt gekommen, nämlich die eigenen Finger auf einem Multi-Touch-Bildschirm. Revolutionär, weil noch nie da gewesen, und wirklich cool, wenn man die zusätzlichen Features betrachtet, wie z.B. die Multi-Touch-Gesten, die automatisch erkennen, wie viele Finger gleichzeitig in welche Richtung streichen.

Zuständigkeiten

Wer ist in Scrum zuständig für die Vision eines Produkts? Eindeutig der Product Owner! Seine Aufgabe ist es, die Zielgruppe und deren Bedürfnisse zu identifizieren, allererste Schlüsselfunktionen zu definieren und die Vision zu formulieren. Möglicherweise macht er das nicht allein, sondern mithilfe von Stakeholdern und anderen Fachleuten, beispielsweise aus der Vertriebs- und Marketingabteilung seines Unternehmens. Diese Menschen kennen den Markt und die Zielgruppen und können wertvolle Hinweise darauf geben, was dort als wirklich wichtig erachtet wird. Auch das Scrum-Team ist oft ein guter Sparringspartner für die Vision. Eine gemeinsam erarbeitete Vision ist eine gute Basis für gemeinsam getragene Verantwortung und Identifikation mit dem Produkt. Ob miteinander erarbeitet oder nicht: Der Product Owner muss sicherstellen, dass die Vision bekannt und verstanden ist sowie von allen Beteiligten mitgetragen wird.

Die Vision ist nichts für die Schublade. Unter den Events (siehe Abschnitt »Events« auf Seite 74) findet sich eines mit dem Namen »Produktvision teilen«.

Doch die Vision ist nicht nur für die Öffentlichkeit bestimmt. Sie ist in erster Linie für alle an der Produktentwicklung Beteiligten wichtig. Alle müssen die Vision kennen und verstehen, sonst können sie nicht mit ihrer Arbeit dazu beitragen, dass diese erfüllt wird. Andernfalls gehen möglicherweise sämtliche Anstrengungen an der Vision vorbei und sind damit fehlgerichtet.

Deshalb ist es nicht nur eine gute Idee, dass der Product Owner die Vision vorstellt und regelmäßig wieder ins Gedächtnis ruft, sondern auch, sie in ausgedruckter Form in jeden Teamraum zu hängen. So hat jeder und jede die Vision im wahrsten Sinne des Wortes vor Augen und kann sie in der täglichen Arbeit als Leitfaden verwenden.

Anforderungen

Anforderungen sind die Wünsche des Product Owner bzw. der durch ihn vertretenen Stakeholder an das zu entwickelnde Produkt. Die Anforderungsbeschreibung wird im agilen Kontext erst dann konkretisiert, wenn sicher ist, dass sie zeitnah umgesetzt werden soll. Das spart unter Umständen Geld, denn das detaillierte Beschreiben von Anforderungen ist eine Investition, die sich nur dann lohnt, wenn die Anforderung tatsächlich im Produkt realisiert wird. Eigenschaften des fertigen Produkts, die solche Anforderungen erfüllen, nennt man Features.

Im Abschnitt »Vision« auf Seite 126 wurde erläutert, wie man die grobe Richtung bei der Produktentwicklung vorgibt. Bei den Anforderungen müssen wir nun konkreter werden.

Anforderungen genügen üblicherweise einer bestimmten äußeren Form, die wir uns gleich näher anschauen werden. In erster Linie sollten sie jedoch bestimmten Kriterien genügen.

INVEST-Kriterien

Die INVEST-Kriterien wurden ab Seite 108 bereits ausführlich beschrieben und werden hier zur Erinnerung noch einmal kurz erwähnt:

- I = Independent (unabhängig)
- N = Negotiable (verhandelbar)
- V = Valuable (wertvoll)
- E = Estimable (schätzbar)
- S = Sized appropriately bzw. Small (angemessen groß bzw. klein)

- T = Testable (testbar)

User Stories

In diesem Buch verwenden wir für eine (fachliche) Anforderung im Scrum-Kontext grundsätzlich den Begriff *Product Backlog Item* aus dem (englischen) Scrum Guide. Anforderungen werden in Scrum oft in sogenannten *User Stories* formuliert. Sie wurden erstmals von Mike Cohn in [Cohn 2004] beschrieben.

User Stories sind kurze, prägnante Beschreibungen zu einer Produkteigenschaft, die einen Wert entweder für die Anwenderinnen und Anwender oder für diejenigen haben, die das Produkt gekauft haben.

Als äußere Form hat sich eine feste Formulierungsschablone etabliert:

»Als *Rolle* möchte ich *Funktion*, um *Nutzen/Wert* zu erreichen.«

- *Rolle*: Viele Produkte sollen die Bedürfnisse unterschiedlicher Anwender befriedigen. Die Gemeinsamkeiten solcher Anwendergruppen lassen sich zu Rollen abstrahieren. In Softwaresystemen gibt es für diese Rollen sogar technische Entsprechungen: Aus unterschiedlichen Arbeitsweisen und Berechtigungsanforderungen der künftigen Nutzenden wird das Rollen- und Rechtesystem des Softwaresystems abgeleitet. In einer Auftragserfassung mag es beispielsweise eine Rolle »Kunde« geben, der nur lesenden Zugriff auf die Auftragsdaten hat, eine Rolle »Sachbearbeiter«, der die Erfassung vornimmt und deshalb Schreibrechte auf die meisten Felder besitzt, und eine Rolle »Genehmiger«, der nach einer kurzen Qualitätssicherung den Auftrag wirklich erteilen kann. Außerdem kann die Unterscheidung nach Rollen dem Product Owner dabei helfen, User Stories zu finden, die sonst übersehen würden: Indem man sich in eine Rolle hineinversetzt, übernimmt man die Perspektive des Rolleninhabers bzw. der Rolleninhaberin und entdeckt so Details und versteht Bedürfnisse, die »von außen« nur schwer zu erkennen sind.

  Wichtig an dieser Stelle ist, eine real existierende Anwenderrolle einzunehmen. Metarollen wie Product Owner, Projektleiter oder gar Systemteile wie Datenbank oder Microservice haben als Rollen in User Stories nichts zu suchen!

- *Funktion*: Hier steht die eigentliche Funktionsbeschreibung. Es geht in diesem Teil nicht um vollständige Details, sondern um eine kurze Idee dazu, welche Produkteigenschaft gewünscht wird. Klarer Fokus ist hier das »Was« und nicht das »Wie«, deshalb wird die Funktion aus fachlicher Sicht

beschrieben (»… möchte ich die Liste der offenen Bestellungen sortieren können …«). Technische Details, die den Lösungsraum zu stark einschränken (»… möchte ich einen Button, mit dem ich die Liste sortieren kann …«), sind hier wenig hilfreich.

- *Nutzen/Wert*: Dieser leider oft vergessene Teil ist enorm wichtig. Denn nur wer weiß, welchen Nutzen die Funktion stiftet bzw. welchen Mehrwert sie schafft, kann zielgerichtet eine sinnvolle und angemessene Implementierung vornehmen und über Alternativen nachdenken, die dem Product Owner bisher entgangen sind. Außerdem ist der Nutzen bzw. Geschäftswert ein wichtiges Kriterium für die Priorisierung der fachlichen Anforderungen.

Für eine User Story sind drei Aspekte relevant, die Ron Jeffries in [Jeffries 2001] als die »3C« bekannt gemacht hat:

1. *Card* – die Story Card: üblicherweise eine Karteikarte, ein Stück Karton oder ein Blatt Papier, auf dem die Story im oben genannten Format beschrieben ist – oder dessen digitales Pendant.
2. *Conversation* – das Gespräch über die Story: Die oben genannte Formel beschreibt die Story zu knapp, als dass ein Developer weiß, was er implementieren soll. Deshalb muss es zwingend ein Gespräch über den Inhalt der Story geben, in dem die Details diskutiert werden, bis ein gemeinsames Verständnis sichergestellt ist. Dieses Gespräch findet im Refinement (siehe Abschnitt »Events« auf Seite 74) statt und wird im Sprint Planning kurz wiederholt. Wenn die Developer es wünschen, werden die diskutierten Anforderungen und Erkenntnisse auf der Story Card schriftlich festgehalten.
3. *Confirmation* – die Definition der Akzeptanzkriterien: Diese werden aus praktischen Gründen gern auf die Rückseite der Story Card (bzw. in das entsprechende Feld einer digitalen User Story) geschrieben. Spätestens im Sprint Review am Ende des Sprints überprüft der Product Owner für jede Story, inwieweit die Akzeptanzkriterien erfüllt sind. Nur wenn alle Akzeptanzkriterien erfüllt sind, gilt die Story als fertig (*Done*). Damit die Developer von Beginn an wissen, woran eine Story später gemessen wird, definiert der Product Owner bereits beim Schreiben der Story genau diese Kriterien. Im Dialog mit den Developern werden sie oft geschärft oder um neue Akzeptanzkriterien ergänzt. Beschreibt beispielsweise eine Story die Validierung von Bankdaten in einem Webshop, sollte hier definiert werden, ob IBAN und BIC eingegeben werden sollen, ob Leerzeichen in der IBAN erlaubt sind usw. Dies erleichtert den Entwicklern die fokussierte Arbeit

und schafft die nötige Transparenz, damit das Ergebnis den Erwartungen des Product Owner entspricht und somit »das Richtige« entsteht. Bei der Formulierung hilft es, sich zu überlegen, wie die Akzeptanzkriterien für »die Rolle« sicherstellen, dass »die Funktion« den gewünschten »(Mehr-)Wert« erreicht.

Alle drei Aspekte sind unverzichtbar und gehören zusammen, insbesondere sind User Stories ohne Akzeptanzkriterien eben keine User Stories.

Granularität der Anforderungen

Die Anforderungen an das Produkt werden in einer Liste verwaltet, dem *Product Backlog*. Da Scrum ein schlanker (lean) Prozess ist, werden Informationen erst dann zur Verfügung gestellt, wenn sie benötigt werden. Jede bekannte Anforderung muss zwar im Backlog vorhanden sein, allerdings variiert die Granularität, wie wir im Abschnitt »Product Backlog mit Produkt-Ziel« auf Seite 104 beschrieben und dort anhand der Backlog-Pyramide illustriert haben.

Priorisierung

Einer der fünf Scrum-Werte ist Fokus. Fokus bedeutet, sich jederzeit auf nur eine Aufgabe zu konzentrieren und sich insbesondere nicht von der schieren Masse aller Anforderungen eines großen Product Backlog erschlagen zu lassen.

Wenn immer nur eine Aufgabe zur gleichen Zeit erledigt werden darf, ist eine Priorisierung aller aktuell in der Betrachtung befindlichen Anforderungen erforderlich, damit die Developer wissen, in welcher Reihenfolge sie die Aufgaben erledigen sollen.

Priorisierung ist das in der Praxis meistverwendete Wort, wenn man von der Ordnung nach Wichtigkeit spricht. Leider ist es jedoch ein wenig hilfreiches Wort, denn ein Kunde, nach der Priorisierung seiner Anforderungen gefragt, vergibt für 80% von ihnen die Priorität 1, für die restlichen Priorität 2, eine Priorität 3 in der Regel nicht mehr.

So kann man als Developer aber nicht fokussiert arbeiten. Welche von all den Priorität-1-Anforderungen soll als nächste umgesetzt werden? Welche ist noch wichtiger als die anderen? Um solche Diskussionen mit dem Auftraggeber zu vermeiden, wird in Scrum der Begriff »Priorität« durch »Reihenfolge« oder »Ordnung« ersetzt. Bei einer Reihenfolge ist stets gewährleistet, dass keine zwei Anforderungen denselben Rang haben, und die Developer wissen immer zuverlässig, welche die nächste Anforderung ist, die umgesetzt werden soll.

Aus diesem Grund ist Microsoft Excel (oder eine beliebige andere Tabellendarstellung) ein geeignetes Werkzeug, um Anforderungen zu ordnen (aber nicht, um sie zu verwalten – dafür gibt es bessere Werkzeuge). Zeilennummern sind unbestechlich, es gibt keine zwei Zeilen mit derselben Nummer!

Aber warum ist es so wichtig, alle Anforderungen in eine eindeutige Reihenfolge zu bringen?

Betrachten wir zunächst die Sprint-Ebene, also die Arbeitsebene der Scrum-Teams: Wie im folgenden Abschnitt »Schätzen« auf Seite 136 ausführlich erklärt, plant das Scrum-Team den kommenden Sprint selbst. Am Ende des Sprint Planning geben die Developer eine Prognose dazu ab, welche der Product Backlog Items sie bearbeiten werden, um das gemeinsam formulierte Sprint-Ziel zu erreichen. Dennoch sind dies alles immer nur Schätzungen. Unvorhersehbare Dinge können geschehen (z.B. Krankheit eines Teammitglieds oder eine vorab nicht erkennbare technische Komplexität) – mit dem Ergebnis, dass einige der prognostizierten Anforderungen nicht umgesetzt werden können. In einem solchen Fall soll aber sichergestellt sein, dass die wichtigsten Features, die deshalb in der Reihenfolge weit oben stehen, fertig werden, um den Verlust möglichst gering zu halten. Deshalb ist es auf Sprint-Ebene enorm wichtig, die Features in der richtigen Reihenfolge vorliegen zu haben (und in dieser Reihenfolge zu bearbeiten!).

Auf Release-Ebene sieht es ähnlich aus. Auch hier muss der Product Owner ein Gefühl dafür entwickeln, welche Anforderungen in welchem Release umgesetzt werden sollen. Die Marketingstrategie oder ein Kundenauftrag kann davon abhängen, dass die richtigen Dinge zur richtigen Zeit geliefert werden.

Noch ein weiterer Aspekt kommt auf Release-Ebene hinzu: In Scrum wird so vorgegangen, dass nicht nur die wichtigsten, sondern auch die riskantesten Dinge so früh wie möglich in Arbeit genommen werden. Was nützt es einem Produkt, wenn 90% aller Anforderungen umgesetzt sind, der technische Durchstich jedoch noch nicht erfolgt ist? In der agilen Produktentwicklung sollten schon in einer sehr frühen Phase alle wesentlichen technischen und fachlichen Probleme zumindest im Ansatz so gelöst werden, dass die grundsätzliche Machbarkeit gewährleistet ist.

Aus diesem Grund wird man bei der Priorisierung Wert darauf legen, alle wichtigen Anforderungsbereiche in einer frühen Phase einzuplanen, um dieses Risiko schnell ausschalten zu können. So entsteht sehr früh ein Produkt, das die fachlichen Grundanforderungen erfüllt und seine Fähigkeiten im Sinne eines frühen Feedbacks möglichst schnell im praktischen Einsatz beweisen soll. Ein

solches *Minimum Viable Product* (MVP) hilft dem Scrum-Team, fokussiert das Richtige richtig zu machen.

Wie priorisiert man eigentlich – und wer tut es?

Die Frage der Verantwortlichkeit ist schnell geklärt: Der Product Owner ist derjenige, der die Priorisierung vornehmen muss. Schließlich trägt er die Verantwortung dafür, was in welcher Reihenfolge umgesetzt wird. In der Release-Planung (auch eine Aufgabe des Product Owner, siehe Abschnitt »Release-Management« auf Seite 146) muss ebenfalls abzusehen sein, wann welches Feature geliefert werden kann. Die Marketingabteilung muss beispielsweise wissen, wann welche Werbeaktionen gestartet werden können, um beispielsweise maximalen Umsatz zu erzielen.

Wonach entscheidet der Product Owner nun aber, welches Feature wann an der Reihe ist? Hier kommt der Geschäftswert (engl. *Business Value*) ins Spiel (das »V« aus den INVEST-Kriterien). Die Anforderungen mit dem höchsten Geschäftswert sollten zuerst umgesetzt werden. Dabei müssen selbstverständlich Abhängigkeiten zwischen den Anforderungen und weitere Priorisierungskriterien wie Risiken oder gesetzliche Vorgaben berücksichtigt werden. Falls das Geld ausgeht oder die Entwicklung länger dauert als ursprünglich geplant, sind so wenigstens die wichtigsten Features schon realisiert, da diese als Erstes umgesetzt wurden.

Schätzen

Bevor man mit der Entwicklung eines Produkts beginnt, muss man üblicherweise eine Vorhersage darüber treffen, wie aufwendig die geplante Entwicklung sein wird. Wie das geschieht, ist sehr unterschiedlich. Bei innovativen Neuentwicklungen mag das wichtigste Kriterium der Zeitfaktor sein, um gegenüber der Konkurrenz einen Marktvorteil zu erlangen. Bei der Ablösung von Altsystemen muss man häufig eine Wirtschaftlichkeitsrechnung erstellen, um zu belegen, dass sich die Entwicklung überhaupt lohnt. In jedem Fall wird man eine Kombination aus Zeit- und Kostenvorhersage treffen müssen.

Noch offensichtlicher wird die Bedeutung des Schätzens, wenn das Umsetzungsteam von einem externen Dienstleister gestellt wird. In diesem Fall muss der Auftragnehmer in der Regel ein Angebot dazu abgeben, zu welchem Preis und in welcher Zeit die Entwicklung machbar ist. Dies berührt zwar den Scrum-Prozess nicht, sorgt aber dafür, dass man sich vor der Umsetzung Gedanken darüber machen muss.

So sehr sich viele Menschen Gewissheit über Größe, Komplexität und somit die Entwicklungsdauer eines Produkts wünschen: Da genau dieses Produkt noch nie von genau diesem Team hergestellt worden ist, kann das Team nur eine Schätzung abgeben. Mit zunehmender Erfahrung wird diese Prognose in der Regel genauer – sie bleibt aber immer eine Prognose.

Agile Teams schätzen gemeinsam – relativ und abstrakt.

Gemeinsam schätzen

Es schätzen immer alle Developer, und das hat mehrere Gründe:

- Die Developer sind die Fachleute. Sie haben Erfahrung in der Fachdomäne, mit den verwendeten Werkzeugen und mit den Kolleginnen und Kollegen, da sie schon ähnliche Features gebaut haben. Man sollte ohnehin immer diejenigen fragen, die am besten wissen, wie es geht.
- Teamschätzungen sind immer besser als Einzelschätzungen. Ein einzelner Mensch kann mit seiner Schätzung immer mal danebenliegen. Dass aber ein Team von etwa sieben Personen danebenliegt, ist statistisch extrem unwahrscheinlich.
- Die Developer müssen es auch »ausbaden«. Sie treffen am Ende des Sprint Planning eine Aussage darüber, wie viel sie im kommenden Sprint schaffen werden. Diese Aussage sollte schon so verbindlich sein, dass die Developer alles in ihren Kräften Stehende dafür tun werden, das Prognostizierte im Sprint tatsächlich zu liefern. Wenn sie also in ihrer Schätzung danebengelegen haben, wird die Zeit knapp und der Stress größer. Auch wenn das in Scrum keine gravierenden Folgen (etwa in Form von Überstunden oder Wochenendarbeit) hat und deshalb das Sprint-Ziel vermutlich nicht erreicht wird, kann ein gutes Team daraus lernen und bei der nächsten Schätzung besser werden.
- Der »innere Antrieb« führt zu höherer Motivation. Da die Developer die Schätzung selbst abgegeben haben, können sie niemandem die Schuld geben, wenn etwas nicht wie erwartet funktioniert. Vorgaben anderer erfüllen zu müssen, ist ein Motivationskiller, seine eigenen Vorgaben zu erfüllen, fördert die Motivation.
- Last, but not least geht es in Schätzrunden immer auch um den Aufbau eines gemeinsamen Verständnisses zur jeweiligen Anforderung. Bei einer fachlichen Beschreibung geht es ja immer nur um das »Was«. Für deren

> Umsetzung kommen meist unterschiedlich aufwendige technische Lösungen infrage. In einer Schätzrunde können die Developer ein gutes Gefühl dafür entwickeln, welche dieser Lösungen dem Mehrwert angemessen ist.

Es gibt unterschiedliche agile Schätzverfahren. Wir werden hier das Planning Poker vorstellen. Weitere sind z.B. [Team Estimation] oder [Magic Estimation]. Auch eine Kombination dieser Verfahren ist gut denkbar. So bietet sich für die Schätzung einzelner Product Backlog Items Planning Poker an, bei groberen Schätzungen wie z.B. einer Release-Prognose kann Team Estimation die bessere Wahl sein.

Relativ schätzen

Allen agilen Schätzmethoden liegt die Tatsache zugrunde, dass die meisten Menschen ziemlich schlecht absolute Zahlen schätzen (»Ich brauche für diese Website 3,5 Tage«), aber erstaunlich gut Relationen erfassen können (»Für dieses Feature brauche ich sicher doppelt so lange wie für das andere.«). Absolute durch relative Werte zu ersetzen, führt zu Schätzungen mit einer erstaunlich hohen Genauigkeit.

Ein weiterer Unterschied zu klassischen Schätzverfahren besteht in der Trennung von Größe und Aufwand. Man denke sich einen Steinhaufen aus großen, schweren Steinen, die von einem Punkt A zu einem Punkt B transportiert werden sollen. Wird ein durchschnittlich gebauter Mann diese Arbeit verrichten, braucht er vielleicht drei Tage, bis der Stapel auf der anderen Seite ist. Bittet er noch drei Nachbarn um Unterstützung, ist der Haufen möglicherweise in einem halben Tag drüben. Nimmt er einen Bagger zu Hilfe, ist er vielleicht sogar in einer Stunde fertig. Der Haufen hat aber in jedem Fall die gleiche Größe. Der Aufwand für die Erledigung einer Aufgabe ist also offenbar abhängig von der Anzahl der beteiligten Personen und/oder der Werkzeugauswahl. Das ist der Grund dafür, dass nur Größen und nicht Aufwände geschätzt werden: Man weiß noch nicht, wer (mit welchen Fähigkeiten und welchen Werkzeugen) die Aufgabe später umsetzen wird.

Abstrakt schätzen

Beim agilen Schätzen wird eine künstliche, bewusst abstrakte Maßeinheit verwendet: die Story Points. Diese lassen sich nicht ohne Weiteres in eine »echte« Einheit umrechnen. Dadurch fällt es vielen Teams leichter, sich auf die Größe und Komplexität der Aufgabe zu konzentrieren, anstatt zu überlegen, wie lange man selbst für die Erledigung dieser Aufgabe brauchen würde.

Um der relativen Schätzweise Rechnung zu tragen, sind auch nicht alle möglichen Werte zulässig, sondern nur die Werte 1, 2, 3, 5, 8, 13, 20, 40 und 100 (vereinzelt findet man auch 0, ½ und ∞, das ändert aber nichts an der grundsätzlichen Vorgehensweise). Grund dafür ist die Tatsache, dass es einfach nicht wichtig ist, ob ein Feature z.B. den Wert 42 oder 43 bekommt, schließlich geht es nur um die Größenordnung. Auch sind die einzelnen Werte nicht mathematisch exakt zu verstehen. So bedeutet eine 8 eigentlich einen Wert zwischen 5 und 13. Dieser pragmatische Ansatz erleichtert das Schätzen, da sich eine Gruppe von Personen leichter auf einen solchen Bereich einigen kann als auf eine exakte Zahl. Und außerdem ist und bleibt es nun mal eine Schätzung, sodass eine genauere Skalierung eine Exaktheit suggerieren würde, die tatsächlich nicht gegeben ist. Man erkennt die Ähnlichkeit zur Fibonacci-Folge, in der die Abstände zwischen den einzelnen Folgengliedern mit zunehmendem Wert größer werden. Auch dies hat seinen Sinn: Je größer der geschätzte Wert, desto größer ist die Ungenauigkeit, daher ist eine genauere Skalierung nicht sinnvoll. Überhaupt ist die Größe einer Schätzung nicht nur ein Maß für die Größe bzw. Komplexität der Anforderung, sondern bei den Werten 40 und 100 auch oft ein Zeichen dafür, dass das Team unsicher ist. Das wiederum deutet darauf hin, dass das zu schätzende Item aus dem Product Backlog entweder zu groß ist oder noch nicht vollständig verstanden wurde und deshalb ein Risiko birgt. Aus diesem Grund wird ein gutes Scrum-Team (idealerweise sogar der Product Owner selbst) das Product Backlog Item bei einer solch hohen Schätzung wieder zurücknehmen und neu schneiden oder beim nächsten Mal besser erklären. Ab welchem Wert eine Schätzung nicht mehr akkurat ist, ist so unterschiedlich wie die Teams mit ihren Arbeitsweisen und Gewohnheiten selbst und muss von jedem Team individuell erarbeitet werden.

Refinement

Immer dann, wenn hinreichend viele neue Features zu schätzen sind, kann der Product Owner ein *Refinement* einberufen (siehe Abschnitt »Events« auf Seite 74). Wichtig ist, dass dieses Event zeitlich begrenzt (*timeboxed*) ist, weil nach einer gewissen Zeit die Konzentration nachlässt und das Team den Fokus verliert. Die einzig wichtige Voraussetzung zur Schätzung ist, dass der Product Owner das Product Backlog Item selbst gut genug verstanden hat, um alle Fragen des Teams beantworten zu können, die unweigerlich während der Schätzung aufkommen werden.

Auf folgende drei Aspekte muss sich der Product Owner vor dem Schätzen jedes Product Backlog Item vorbereiten:

- Fragen zum Sinn und Zweck der Anforderung: Manche Product Owner tendieren dazu, schon in Lösungen zu denken. So wünschen sie sich beispielsweise eine Auswahlliste (Drop-down-Liste) zur Auswahl eines Werts. Aus technischer Sicht kann es aber einfacher, ergonomischer oder günstiger sein, Radiobuttons, Tabulatoren oder ein Freitexteingabefeld vorzusehen. Erst durch die Beantwortung der Frage, was der Product Owner fachlich erreichen möchte, kann das Team beurteilen, welche der technischen Möglichkeiten die sinnvollste ist. Ein guter Product Owner vertraut seinen Developern. Er wird ihnen deshalb nur die fachliche Anforderung erläutern, und ihnen im Rahmen der Vorgaben (z.B. Style Guide) die freie Wahl bei der Implementierung lassen.
- Fragen zu Details, die in einem Product Backlog Item möglicherweise noch nicht beschrieben sind: Nichts ist schlimmer als vom Product Owner getroffene implizite Annahmen, die in dem Product Backlog Item nicht enthalten sind. Dann nämlich werden die Developer unweigerlich das Falsche bauen. Um einem solchen Missverständnis vorzubeugen, muss im Gespräch alles auf den Tisch, was im Kopf des Product Owner herumschwirrt. Oft entstehen daraus neue Akzeptanzkriterien.
- Fragen zu vergleichbaren Features: Dies ist nicht zwingend notwendig, kann aber unglaublich helfen, wenn der Product Owner sagen kann: »Es soll so ähnlich funktionieren wie bei der Artikelerfassung.« Dann wissen die Developer, was sie bauen sollen, und haben im Idealfall sogar schon eine recht genaue Vorstellung von der Komplexität.

Planning Poker®

Planning Poker® (ein eingetragenes Warenzeichen von Mountain Goat Software) ist eine agile Schätzmethode, die im Grunde genommen eine Variante der althergebrachten Delphi-Methode darstellt, bei der getrennte Expertenbefragungen vorgenommen werden. Gespielt wird es mit speziellen Spielkarten, die alle Story-Point-Werte (1, 2, 3, 5, 8, 13, 20, 40 und 100) und eine Kaffeetasse als Kartenwert umfassen.

Vor dem ersten Planning Poker muss ein gemeinsames Verständnis der Schätzgrößen erreicht werden. Dazu wird eine Aufgabe, die die Developer bereits umgesetzt oder die zumindest alle gut genug verstanden haben, als Referenz ausgewählt. Es sollte eine möglichst einfache, kleine Anforderung sein, die dann mit dem Wert 2 Story Points versehen wird. Ist eine solche Aufgabe

gefunden, können alle weiteren Schätzungen relativ zur Größe dieser Referenzanforderung erfolgen.

Die Spielregeln sind einfach:

1. Die zu schätzende Aufgabe wird von einer oder einem Mitwirkenden kurz vorgestellt. Idealerweise ist dies der Product Owner, da er die Aufgaben am besten kennen sollte, aber in der Praxis kann es von jedem geleistet werden, der hinreichend gut Bescheid weiß. Selbstverständlich können hier Verständnisfragen gestellt werden, bis alle anwesenden Developer sich in der Lage sehen, eine Schätzung abzugeben.
2. Jeder Mitwirkende sucht aus seinem Kartendeck die Karte heraus, von der er glaubt, dass sie die passende Größe der Aufgabe darstellt, und legt sie verdeckt vor sich.
3. Haben alle Mitwirkenden gewählt, werden sämtliche Karten gleichzeitig aufgedeckt. Nachträgliche Korrekturen sind nicht mehr möglich – gelegt ist gelegt. Jetzt muss man für seine Meinung eintreten.
4. Die Mitwirkende mit dem höchsten und der mit dem niedrigsten Schätzwert diskutieren nun kurz darüber, warum sie glauben, dass diese Aufgabe so klein bzw. groß ist. Alle anderen Teilnehmenden haben bei dieser Diskussion Redeverbot und hören aufmerksam zu.
5. Sind die wichtigsten Argumente ausgetauscht, wird erneut von allen geheim abgestimmt. Diese Schleife kann theoretisch so lange durchlaufen werden, bis ein Konsens gefunden ist. Normalerweise ist nach spätestens drei Runden eine Einigung erreicht. Sollte das einmal nicht der Fall sein, muss man sich auf ein Verfahren einigen. Oft wird bei benachbarten Schätzwerten der größere gewählt. Hier muss man ein wenig experimentieren, welches Verfahren im jeweiligen Kontext am besten funktioniert. Dabei sollte nicht vergessen werden: Eine Schätzung ist immer ein ungefährer Wert bzw. Wertebereich mit einer gewissen Unsicherheit. Eine Diskussion um den einen oder anderen Größenpunkt ist daher nicht angebracht.
6. Dieses Verfahren wird nun auf jede zu schätzende Aufgabe angewendet. Wichtig ist dabei, die vorher festgelegte Timebox nicht zu überziehen, sonst sind sinkende Konzentriertheit und daraus resultierende schlechte Schätzungen die Folge. Ein gutes Indiz für zu lange Sitzungen ist, wenn die Karte mit der Kaffeetasse gezogen wird, die besagt, dass (mindestens) einer der Teilnehmenden eine Pause benötigt.

Der wohl wichtigste Aspekt beim Planning Poker ist die Diskussion über die fachlichen Anforderungen. Durch die Vorstellung der Product Backlog Items und das anschließende »Feilschen« um Punkte wird das für die Realisierung notwendige einheitliche Verständnis und Wissen bei allen Mitwirkenden hergestellt.

Durch die Einigung auf eine geschätzte Größe wird eine geteilte Verantwortung erzeugt. Ein späteres Herausreden (»Ich habe doch gleich gesagt, dass das viel länger dauert …«) ist nicht möglich. Das fördert die Gemeinschaft und zwingt alle dazu, das gemeinsam definierte Ziel auch erreichen zu wollen.

Eher zurückhaltende Personen, aber auch Drückeberger, werden gezwungen, sich zu entscheiden. Durch das gleichzeitige Aufdecken der Karten haben sie keine Chance, sich einfach der Meinung anderer anzuschließen. Das hat zwei positive Effekte: Auf der einen Seite fühlt man sich besser, wenn man sich getraut hat, seine Meinung zu äußern (und eventuell auch in der Diskussion zu vertreten). Auf der anderen Seite haben auch die »stillen Vertreter« oft viel Interessantes beizutragen, was normalerweise leider verloren geht, da sie ihre Meinung nicht so gern offen kundtun. Neben den sozialen Aspekten wird das Produkt also auch qualitativ von Planning Poker profitieren.

#NoEstimates

In den vergangenen Jahren hat sich eine Bewegung in der Scrum-Welt mehr und mehr Gehör verschafft, die unter dem Hashtag #NoEstimates firmiert. Die Grundaussage ist, dass Schätzungen mindestens Verschwendung, wenn nicht sogar schädlich sind.

Warum schädlich? In der Tat ist es oft so, dass die Developer »Schätzung« sagen und das Management »Commitment« hört. Eine Schätzung ist eine Vermutung, die naturgemäß Unsicherheiten beinhaltet. Beispielsweise können Ereignisse eintreten, die dazu führen, dass die Dauer der Umsetzung länger ist als ursprünglich gedacht. Dies ist der Grund, warum mit den Story Points eine abstrakte Maßeinheit gewählt wird, um keinen direkten Rückschluss auf die zu erwartende Dauer zuzulassen.

Gerade in großen Firmen, die oft eine eher klassische Sicht auf Projektplanung haben, wird aber nur gesehen, dass eine Zahl an einer Story hängt, die dann wunderbar in einen Projektplan eingebaut werden kann – ganz ohne Zutun des Scrum-Teams. Geschieht dies, werden eine Sicherheit und eine Vorhersagbarkeit suggeriert, die dem Wesen der agilen Planung entgegensteht. So kann tatsächlich ein Schaden entstehen, weil es so aussieht, als hätte das Scrum-

Team seine »Zusage« nicht eingehalten, sollte diese Erwartungshaltung nicht erfüllt werden.

Warum Verschwendung? Untersuchungen haben ergeben, dass das bloße Zählen von erledigten Stories im Sprint keine schlechteren Ergebnisse für die Vorhersage im Sinne der Velocity liefert als das Aufsummieren der Story Points der erledigten Anforderungen. Voraussetzung dafür ist allerdings, dass die Stories im Großen und Ganzen eine ähnliche Größe haben.

Beide Aspekte haben aus unserer Sicht ihre Berechtigung. Allerdings empfehlen wir insbesondere Scrum-Teams mit wenig Erfahrung, trotzdem Schätzungen durchzuführen. Einer der wesentlichen Effekte bei Schätzrunden ist der Wissensaustausch über die Anforderung. Unterschiedliche Zahlen beim Planning Poker beruhen häufig nicht nur auf unterschiedlichen Einschätzungen derselben Tatsachen, sondern offenbaren auch dahinterliegende Gedanken zu möglichen Umsetzungen, die dann direkt im Schätzmeeting besprochen werden können. Diesen Effekt würde man sich ohne das Schätzen nehmen. Die eventuell aufkommenden Missverständnisse im agilen Kontext beim Unterschied zwischen Schätzung und Commitment sollte man aktiv im Unternehmen angehen, anstatt durch das Weglassen von Schätzungen in eine Vermeidungsstrategie zu geraten.

Iterativ-inkrementelles Vorgehen

Scrum ist eine Vorgehensweise, die man als iterativ-inkrementell bezeichnet. Was bedeutet das?

Iterativ

Bei Scrum wird in Iterationen gearbeitet, die Sprints genannt werden. Iterationen sind Zeitabschnitte, die alle nach dem gleichen Muster ablaufen und die gleiche Länge haben. Dadurch entsteht ein Rhythmus, der dafür sorgt, dass die Mitglieder des Scrum-Teams nicht mehr darüber nachdenken müssen, wann z.B. ein Planungsevent oder ein Review stattfinden wird, weil es in jedem Sprint zum gleichen Zeitpunkt passiert. Damit ist man vor terminlichen Überraschungen einigermaßen geschützt und kann seinen Fokus vollkommen auf die vorliegenden Anforderungen legen, man entwickelt sozusagen ein Bauchgefühl für die Sprint-Länge und muss nicht mehr auf den Kalender schauen.

Iterativ bedeutet aber auch, dass man sich dem gewünschten Endzustand schrittweise nähert. Bereits in der ersten Iteration soll die spätere Lösung in

ihren Grundzügen enthalten sein, um dann in jeder weiteren Iteration schrittweise verfeinert zu werden.

Entwickelt man beispielsweise eine Auftragsverwaltung, in der die einzelnen Aufträge laut Anforderung einen Zustand mit sich führen, kann in einer ersten Iteration ein Zustand per Hand aus einer Auswahlliste ausgewählt werden, in einer zweiten Iteration kommen vielleicht noch ein Kommentarfeld und eine Benutzerkennung des letzten Bearbeiters hinzu. In der endgültigen Version existiert dann möglicherweise ein ausgefeilter Algorithmus, bei dem je nach zeitlichem Verlauf, dem Inhalt bestimmter Schlüsselfelder und der Rolle der letzten Bearbeiterin der Auftragsstatus automatisch gesetzt wird und Folgeaktionen ausgelöst werden. Die Grundforderung des Zustands je Auftrag ist aber schon in der allerersten Version vorhanden gewesen und anschließend nach und nach verfeinert worden.

Diese Vorgehensweise hat mehrere Vorteile: Zum einen sieht man schon sehr früh im Entwicklungsprozess, ob die Anforderung den Praxistest bestehen kann. Nach jeder Iteration wird ja der dann fertige Stand dem Kunden übergeben, der ihn auf Herz und Nieren testen kann und soll. So erkennt man bereits frühzeitig, an welchen Stellen noch konzeptionell nachgearbeitet werden muss.

Der Kunde sowie spätere Anwenderinnen und Anwender können gut den Fortschritt der Entwicklung erkennen und haben nach jedem Sprint die Möglichkeit, Feedback zu geben, wenn die Entwicklung aus ihrer Sicht in die falsche Richtung läuft oder sich neue Anforderungen ergeben haben.

Ein weiterer Vorteil, den man nicht unterschätzen sollte, ist die Tatsache, dass der Kunde die Entwicklung bei einer bestimmten Iterationstiefe auch einfach stoppen kann. Wenn die eigentlich noch detaillierter geplante Anforderung den momentanen Bedürfnissen bereits entspricht, kann die Weiterentwicklung gestoppt werden, was letztendlich Zeit und Geld spart.

Inkrementell

Ein Inkrement ist der nächste Teil des Ganzen. Inkrementell zu arbeiten, heißt also, das Gesamtprodukt in einzelnen Teilen zu liefern. Im Gegensatz zum iterativen Vorgehen, wo ein bestehender Teil weiter verfeinert wird, kommt beim inkrementellen Vorgehen ein inhaltlich unabhängiger Teil hinzu.

Die Inkremente haben eine wichtige Funktion: Der Scrum-Wert »Fokus« besagt, dass sich jeder und jede auf die Umsetzung einer einzigen Anforderung konzentriert. Im Umkehrschluss heißt das auch, dass nicht alle funktionalen Teile eines Softwaresystems gleichzeitig entworfen werden, sondern dass das Softwaredesign Schritt für Schritt entsteht. Das birgt die Gefahr, dass Fehler in

der Konzeption erst spät gefunden und korrigiert werden. Deshalb muss die Architektur agil entwickelter Software möglichst einfach, flexibel und modular sein.

Der Begriff *Inkrement* bezeichnet übrigens ebenfalls ein Scrum-Artefakt (siehe Abschnitt »Inkrement mit Definition of Done« auf Seite 111), hier im Sinne eines abgeschlossenen (fertigen) Teils des Ganzen. In Scrum muss spätestens am Ende jedes Sprints immer etwas potenziell Auslieferbares entstehen, also ein Ergebnis, das man in Produktion geben könnte, wenn man denn wollte. Es gibt durchaus gute Gründe dafür, dass eine neue Produktversion nicht am Ende eines jeden Sprints ausgeliefert wird. Die Developer sollten trotzdem den Anspruch an sich stellen, das Sprint-Ergebnis so abzuliefern, dass man es in Produktion geben könnte. Bei der Beschreibung der Definition of Done (siehe auch den Abschnitt »Inkrement mit Definition of Done« auf Seite 111) wird genauer erläutert, was mit diesem »fertig« gemeint ist. Eine griffige Erklärung ist, dass es fertig ist, wenn man den zugehörigen Code nicht mehr anfassen muss – weder zum Einchecken noch zum Testen, Refaktorisieren, Dokumentieren oder aus sonstigen Gründen.

Release-Management

Vordergründig möchte man denken, Release-Planung in Scrum sei ohne Weiteres gar nicht möglich. Schließlich bestimmen die Developer selbst, wie viel sie im jeweiligen Sprint umsetzen werden – und das auch noch von Sprint zu Sprint neu. Man hat also nur einen verlässlichen Planungshorizont von einer Sprint-Länge, beispielsweise zwei Wochen. Wie soll man mit diesen spärlichen Informationen einen Release-Plan erstellen, der mehrere Monate umfasst?

Auf der anderen Seite wird sich kein Kunde darauf einlassen, einen Auftrag für die Erstellung einer Software zu erteilen, wenn er nicht weiß, wann der Auftrag beendet sein wird und wann er mit welchen Features rechnen kann.

Scrum bietet einige Hilfsmittel, die eine Release-Planung ermöglichen. Dazu zählen die Velocity, rechtzeitige Schätzungen in Refinements und bewusste Ungenauigkeiten in der Planung. Leider hilft – gerade in traditionell denkenden Organisationen – keines dieser Hilfsmittel, um einen Auftrag für eine Produktentwicklung zu bekommen. »Wann wird das Produkt fertig, was wird es leisten, und was wird es kosten?« – das sind die Fragen, die vielen Produktteams gestellt werden. Dabei müssten die Stakeholder wissen, dass gemäß dem magischen Dreieck des Projektmanagements (siehe Abschnitt »Reagieren auf Veränderung mehr als das Befolgen eines Plans« auf Seite 26) mindestens eine dieser Fragen unbeantwortet bleiben muss. Ob agil oder nicht agil: Eine

Produktentwicklung beginnt mit einem Vertrauensvorschuss des Kunden gegenüber dem Auftragnehmer. Dieses Vertrauen kann der Auftragnehmer durch die Vermittlung technologischer, fachlicher und methodischer Kompetenz rechtfertigen, weniger durch eine konkrete Zeit- und Aufwandplanung. Der Kunde muss das Vertrauen haben, dass der Auftragnehmer in der ihm gegebenen Zeit im Sinne des Kunden handeln und das bestmögliche Ergebnis erzielen wird.

Ein Scrum-Team arbeitet gemittelt über die Zeit mit einer weitgehend konstanten Geschwindigkeit. Diese Geschwindigkeit wird in Scrum Velocity genannt und in der Einheit Story Points pro Sprint gemessen. Wenn also in einem Sprint eine Velocity von 45 erreicht wurde, waren alle fertigen Stories in Summe auf 45 Story Points geschätzt worden.

Studien von gut funktionierenden Scrum-Teams haben ergeben, dass diese Velocity nach einer Einschwingphase von drei bis fünf Sprints erstaunlich gut vorhersagbar ist. Selbstverständlich werden in den einzelnen Sprints unterschiedlich viele und unterschiedlich komplexe Stories fertiggestellt, aber die Summe der Story Points, die die Developer je Sprint fertigstellen können, bleibt in einem engen Korridor, sodass die Velocity gut als Maß dafür dienen kann, wie viel ein Scrum-Team je Sprint schafft, um daraus eine Release-Planung aufzubauen.

Hier zeigt sich auch, warum das Refinement so wichtig ist. Natürlich kann man sagen, dass man erst genau weiß, wie komplex ein Product Backlog Item ist, nachdem man es fertiggestellt hat, und man kann bezweifeln, dass es wichtig ist, ob dieses Item mit dem Wert 3 oder 5 geschätzt worden ist. Wenn man aber weiß, dass auf Basis dieser Zahlen weitere Planungen erstellt werden, sieht es schon etwas anders aus mit der Forderung nach möglichst akkuraten Zahlen.

Der zweite Aspekt ist die Vorausschau der Schätzungen. Die Stories, die im kommenden Sprint anstehen, werden nicht erst im Sprint Planning geschätzt. Das wäre viel zu spät. Die Refinements sollten so oft stattfinden, dass der Product Owner immer eine gute Vorschau auf etwa die nächsten drei Sprints bekommt. Je kürzer die Sprint-Länge gewählt wird, desto mehr Sprints muss der Product Owner im Voraus betrachten. Allzu weit in die Zukunft sollte man jedoch auch nicht gehen, da ja eine Grundannahme aller agilen Methoden darin besteht, dass sich im Laufe der Zeit Änderungen ergeben werden. Hat das Scrum-Team Zeit in das Verstehen und Schätzen von Features investiert, die sich aufgrund von Änderungen am Ende doch anders darstellen als bei der Schätzung, wäre dies verlorene Zeit und damit Verschwendung, die wir in Scrum möglichst vermeiden wollen.

Der letzte Aspekt, der bei der agilen Release-Planung eine Rolle spielt, ist die Genauigkeit der Planung. Der größte Unterschied zwischen klassischem und agilem Vorgehen besteht darin, dass im klassischen Projektmanagement die Meinung besteht, man müsse nur hinreichend viele möglichst genaue Informationen sammeln, um einen verlässlichen Plan erstellen zu können. Agile Planung geht hingegen von der Annahme aus, dass es nur einen beschränkten Planungshorizont geben kann, weil langfristige Pläne recht schnell von der Wirklichkeit eingeholt werden. Deshalb ist man auf diese Änderungen vorbereitet und kann sehr schnell auf alle Eventualitäten reagieren, ohne bestehende Pläne anpassen zu müssen.

Wie verträgt sich das mit der Forderung nach einem Release-Plan, der einige Monate in die Zukunft schaut? Das ist nur eine Frage der Granularität. Wenn man sich die Backlog-Pyramide im Abschnitt »Product Backlog mit Produkt-Ziel« auf Seite 104 noch einmal anschaut, stellt man fest, dass die Features, die einige Monate in der Zukunft liegen, sehr wohl schon im Product Backlog enthalten sind, allerdings nur sehr grob als Epics beschrieben. Die Developer kennen diese Epics idealerweise schon, die einzelnen Features, die dieses Epic ausmachen, liegen allerdings noch nicht in einer Detailtiefe vor, auf der eine Schätzung abgegeben werden könnte. Aber auch Epics können relativ zueinander geschätzt werden. Wird als Referenz ein Epic gewählt, das schon (weitgehend) komplett umgesetzt worden ist, lässt sich aus der Summe der Story Points dieses Epics und dem angenommenen Größenfaktor für das zu schätzende Epic eine Story-Point-Summe prognostizieren. Damit weiß der Product Owner ungefähr, ob das Bauen der Features in diesem Epic einen Monat oder ein halbes Jahr dauern wird.

Ein Product Owner, der einen Release-Plan abliefern soll, tut also gut daran, diesen sehr allgemein zu halten und nur Schlüsselfunktionen darin zu benennen, damit noch Platz für Unvorhergesehenes bleibt und den Schätzungen des Teams nicht vorgegriffen wird. Mit einer gut vorhersagbaren Velocity und Schätzungen, die einige Sprints im Voraus bereits vorliegen, ist es zwar möglich, verlässliche Aussagen zum Release-Plan vorlegen zu können, allerdings immer mit dem Hinweis, dass es sich um eine Schätzung handelt, die sich durch neue Erkenntnisse und unvorhersehbare Ereignisse ändern kann und deshalb regelmäßig angepasst werden muss.

Wartung (Maintenance)

Unter Wartung (engl. *Maintenance*) versteht man in der Softwareentwicklung üblicherweise die Anpassungen an einer Software nach deren Inbetriebnahme.

In erster Linie sind hier Fehlerbehebungen, Performancesteigerungen und kleinere Weiterentwicklungen gemeint.

In Scrum, zumindest in seiner ersten Definition, taucht diese Phase des Softwarelebenszyklus allerdings überhaupt nicht auf. Das hat folgenden Grund: Kleinere Weiterentwicklungen können einfach in Form von neuen User Stories ins Backlog eingebracht werden. Sie werden den Developern vom Product Owner im Rahmen der regulären Priorisierung zur Realisierung vorgelegt.

In einer idealen Scrum-Welt sind Fehler bzw. mangelnde Performance Aspekte, die eigentlich nicht auftreten bzw. ein gesundes Maß nicht überschreiten sollten, da sie schon während der Entwicklungsphase vom Scrum-Team berücksichtigt werden – nicht erst nach Auslieferung des Produkts. Hierzu gibt es in Scrum die Definition of Done (siehe Abschnitt »Inkrement mit Definition of Done« auf Seite 111), die definiert, wann ein Feature wirklich fertig ist. Dazu zählt insbesondere eine möglichst hohe Testabdeckung, die Fehlerarmut sicherstellen soll. Allerdings gilt auch in Scrum das erste Grundprinzip des Testens: Tests können nur die Anwesenheit von Fehlern aufzeigen, nicht aber deren Abwesenheit. Tests reduzieren die Wahrscheinlichkeit unentdeckter Fehler in der Software, liefern aber keinen Beweis für die Korrektheit der Software – selbst dann nicht, wenn keine Fehler gefunden werden [ISTQB 2011].

Basierend auf Kent Becks Ideen aus dem eXtreme Programming [Beck 2004] wird in Scrum jedes Feature idealerweise mehrfach getestet. Neben Pair Programming und Code Reviews muss jedes Feature vom Entwickler mit Unit Tests versehen sein oder idealerweise sogar testgetrieben entwickelt werden. Darauf aufbauend, sind Integrations- und Kundenakzeptanztests gute agile Praxis, sodass ein Großteil der Fehler in diesem Sicherheitsnetz hängen bleibt und deshalb bei der späteren Nutzung nie auftritt.

Sollte der Fehleranteil dennoch ein nennenswertes Maß erreichen, ist das Mittel der Wahl nicht etwa, ein weiteres Team aufzustellen, das sich um diese Fehler im produktiv genutzten System kümmert. Stattdessen sollte das Problem wieder an die ursprünglichen Developer zurückgespielt werden. Hier wird nun Ursachenforschung betrieben und das Problem an der Wurzel angegangen. Tritt ein Fehler im produktiv genutzten System auf, ist er offensichtlich nicht vorher erkannt worden, was bedeutet, dass die Testabdeckung oder -qualität des verantwortlichen Teams mangelhaft ist.

Schon seit Langem ist bekannt, dass die Behebung eines Fehlers umso teurer ist, je später er entdeckt wird. Was liegt also näher, als dafür zu sorgen, dass die Fehler bereits sehr früh im Entwicklungsprozess gefunden und behoben werden?

Ein weiterer Grund dafür, dass es keine gute Idee ist, ein separates Scrum-Team zur Fehlerbehebung einzusetzen, ist die Tatsache, dass sich die Dauer der einzelnen Behebungen praktisch nicht schätzen lässt und diese Behebungen nicht bis zum Ende des laufenden Sprints (möglicherweise also mehrere Wochen) warten sollen. An dieser Stelle böte sich ein Team an, das nicht nach Scrum arbeitet. Wer auch Maintenance agil betreiben möchte, ist vielleicht mit Kanban besser bedient.

Vom Scrum-Lehrling zum Scrum-Meister

Abhängig davon, wie reif der Softwareentwicklungsprozess im eigenen Unternehmen heute ist, kann es einige Zeit dauern, bis man die in der Literatur oft als »Scrum by the book« bezeichnete »reine Lehre« umgesetzt hat. So einfach das Scrum-Regelwerk auch ist: Die Umsetzung erfordert viel Disziplin, ein hohes Maß an Professionalität und in vielen Bereichen auch ein Umdenken und einen Kulturwandel im Unternehmen. Das alles braucht Zeit und Durchhaltevermögen. Wie man den Weg vom Status quo zur ernsthaften Anwendung von Scrum bewältigt, wollen wir in diesem Abschnitt konkret für einige Aspekte und Scrum-Prinzipien beschreiben.

Unsere erste und eindringlichste Empfehlung lautet:

Nutze die Sprint-Retrospektive!

Scrum ohne Sprint-Retrospektive ist unserer Meinung nach kein Scrum. Wer auf die Sprint-Retrospektive verzichtet, beraubt sich des wichtigsten Mittels zur kontinuierlichen Verbesserung – und wird demzufolge nie oder erst sehr spät das volle Potenzial von Scrum ausschöpfen.

In der Sprint-Retrospektive wird der Fokus aller Teilnehmenden auf den Entwicklungsprozess und die Gruppendynamiken gelenkt. Das ist notwendig, weil im Tagesgeschäft die inhaltlichen Aspekte rund um das Produkt den größten Raum einnehmen. Die Metapher vom Waldarbeiter und seiner Säge [Dräther 2014] macht deutlich, wie wichtig es ist, regelmäßig innezuhalten und »seine Säge zu schärfen«. Eine Sprint-Retrospektive ist dann wirksam, wenn sie von allen Teilnehmenden ernst genommen wird. Das bedeutet, auch dahin zu schauen, wo es wehtut, anstatt sich mit einfachen Maßnahmen zufriedenzugeben, die nur einen geringen Effekt auf die Zusammenarbeit haben.

Shu-Ha-Ri

Shu-Ha-Ri ist ein Konzept, das den Lernprozess in den japanischen Kampfkünsten beschreibt. Auf die Softwareentwicklung wurde es von Alistair Cockburn übertragen. Gunter Dueck hat es in einer seiner Kolumnen einmal sehr schön auf den Punkt gebracht:

»*In der japanischen Kampfkunst gibt es drei Stufen des Lernens. Shu ist wie ›gehorche‹ – alles wird genau nach Rezept oder ›Prozess‹ ausgeführt und bis zu absoluter Fehlerfreiheit eingeübt. Ha steht für ›probiere‹ – man weicht von der Lehre ab, sammelt Erfahrungen durch Variation und versteht langsam die Kunst an sich. Ri bedeutet ›verlasse‹ – der Meister löst sich von den Formen, den Lehren und Stilen seiner Vorbilder und vollendet sich*.« [Dueck 2012]

Denkt man an das Erlernen asiatischer Kampfkunst, sieht man oft Filmszenen vor seinem inneren Auge, in denen Karateschülerinnen und -schüler synchron wieder und wieder unter lautem Rufen einzelne Bewegungsabläufe üben. Lässt sich da eine Verbindung zu Scrum herstellen?

Das Bild der trainierenden Schüler ist gar nicht schlecht. In diesen Szenen üben sie stets einen isolierten Bewegungsablauf streng nach den Regeln ihres Meisters, bis er perfekt sitzt. Erst dann dürfen sie eigene Variationen versuchen, und einige dieser Schülerinnen und Schüler werden es am Ende zu wahrer Meisterschaft bringen. Im Handwerk gibt es ein dem Shu-Ha-Ri vergleichbares dreistufiges Rollenmodell: Lehrling – Geselle/Gesellin – Meister/Meisterin. Diese drei Stufen können auf jeden Lernprozess angewendet werden, auch auf die Einführung agiler Vorgehensmodelle wie Scrum.

Shu-Ha-Ri ist kein linearer Prozess. Es wird oft in Form von konzentrischen Kreisen symbolisiert (siehe Abbildung 4-1). Damit soll zum Ausdruck gebracht werden, dass die auf der Shu-Stufe erlernten Fähigkeiten und Fertigkeiten auch auf den höheren Stufen ihre Bedeutung und Berechtigung behalten und dort variiert und ergänzt werden.

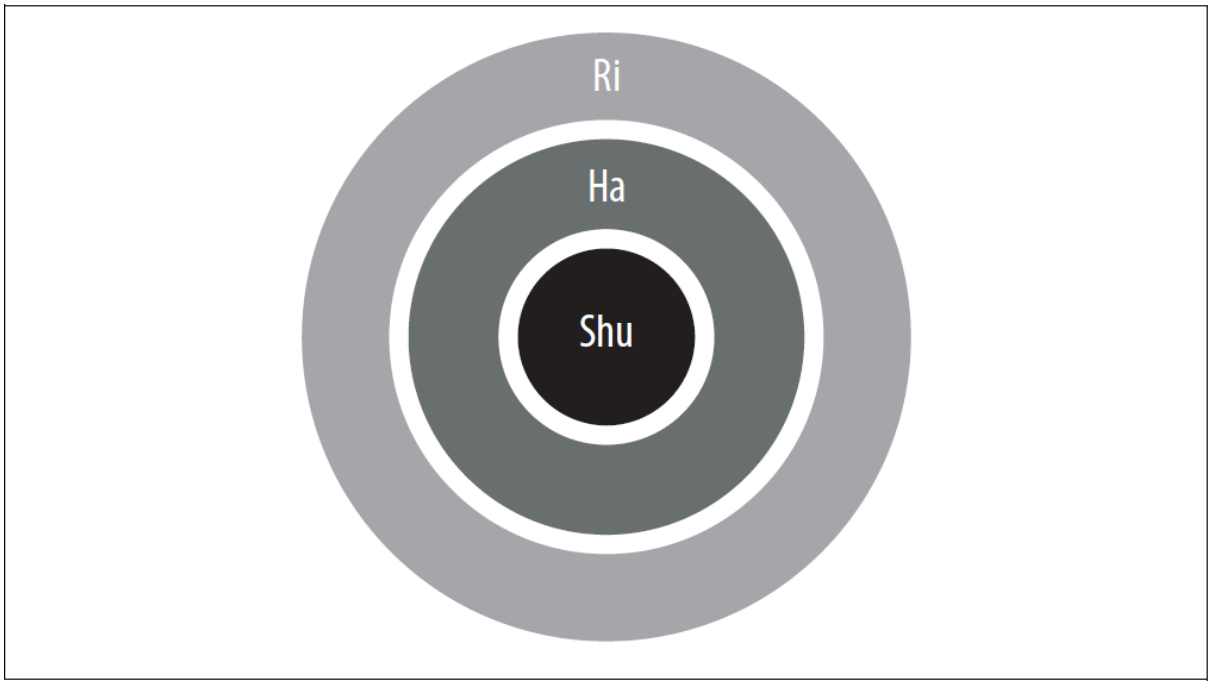

Abbildung 4-1: Drei konzentrische Kreise: das Konzept des Shu-Ha-Ri

Um das Shu-Ha-Ri-Prinzip zu illustrieren, hat Gunter Dueck einmal beschrieben, wie man das Kochen von Tomatensuppe erlernt (die Quelle ist im Internet leider nicht mehr verfügbar). Alles beginnt auf Tütensuppen-Niveau. Das ist Shu. Man hält sich streng an die Regeln, die auf der Rückseite der Tüte stehen, misst sorgsam das Wasser ab, achtet auf die richtige Temperatur, rührt regelmäßig um, kocht kurz auf … und löffelt am Ende das aus, was die Regeln hergeben. Im nächsten Schritt beginnt man zu variieren. Das ist Ha. Da soll es am Ende schmecken. Man verfeinert mit Sahne. Gibt vielleicht klein gewürfelte frische Tomaten und frisches Basilikum hinzu oder überstreut die Suppe mit fein gehackter Petersilie. Voraussetzung ist, dass man vorher das Rezept verstanden und seine Regeln erlernt und befolgt hat. Hat man wahre Meisterschaft, das Ri, erlangt, interessieren weder Rezepte noch Regeln. Man hat sie verinnerlicht und löst sich von ihnen. Die Suppe wird zur eigenen Kreation, sie trägt eine unverwechselbare Handschrift, sie ist ein wahres Meisterwerk.

Auch bei der Einführung von Scrum kommt es darauf an, dass man als Lehrling erst einmal streng nach den Regeln trainiert (Shu). Erst wenn man verstanden hat, wie das Vorgehen gemäß den Regeln funktioniert, und man diese kennt und anerkennt, kann man beginnen, sie zu variieren (Ha). Dann entsteht Stück für Stück ein Prozess, der auf die konkrete Situation, auf den konkreten Kontext zugeschnitten ist. Am Ende hat man als Scrum-Team das neue Vorgehen und seine Regeln derart verinnerlicht, dass man sie lebt, ohne sich ihrer bewusst zu werden. Man hat zu seinem eigenen Stil gefunden und kann sich ändernden Kontexten mühelos anpassen (Ri).

Am Anfang, auf der Shu-Stufe, braucht das Scrum-Team einen erfahrenen Fachmann oder eine Expertin für das neue Vorgehen, der oder die über die Einhaltung der Regeln wacht. Mit ihrer ganzen Autorität lehren diese, wie man dem neuen Prozess folgen kann – zum Vorteil und Nutzen für das Scrum-Team.

Je weiter das Scrum-Team in seiner Entwicklung voranschreitet, desto seltener braucht es konkrete Handlungsanweisungen. Es hat gelernt, die Prinzipien und Praktiken an die Erfordernisse des Teams und seines Umfelds anzupassen und gegebenenfalls die Regeln zu brechen, und bewegt sich abhängig von den Erfordernissen zwischen den drei Stufen des Shu-Ha-Ri hin und her. In Retrospektiven werden die Regeln überprüft und variiert (Ha), das angestrebte neue Verhalten wird trainiert (Shu), und am Ende geht es in das Selbstverständnis über – man braucht nicht mehr darüber nachzudenken (Ri).

Schritt 1: Den Standort bestimmen

Teams, die mit Scrum beginnen wollen, müssen sich oft erst einmal darüber klar werden, wie sie derzeit zusammenarbeiten – wenn sie denn überhaupt schon zusammengearbeitet haben. Wer seinen Standort nicht kennt, kann auch nicht zu einem gewünschten Ziel navigieren. Die zentrale Frage bei der Standortbestimmung lautet: »Welche Aspekte unseres derzeitigen Prozesses sollten wir verbessern, um effektiver und zufriedener miteinander arbeiten zu können?« Die Kenntnis der eigenen Verbesserungspotenziale ist wichtig, um bei der Einführung von Scrum ein besonderes Augenmerk auf sie zu richten. Gelingt es, in diesen Bereichen nachhaltige Verbesserungen zu erzielen, wird die Scrum-Einführung mit hoher Wahrscheinlichkeit von den Beteiligten und den Beobachtenden als Erfolg gewertet werden.

Um den Standort zu bestimmen, kann das Team eine klassische Retrospektive durchführen, wie sie im Abschnitt »Sprint-Retrospektive« auf Seite 93 beschrieben ist. Die Fragen »Was lief gut?« und »Was können wir in Zukunft besser machen?« kann man sich bereits vor der Einführung von Scrum stellen und bezieht sich dabei auf den bislang gelebten Entwicklungsprozess.

Schritt 2: Scrum lernen

Scrum lernt man am besten in der Praxis – aber nur, wenn man zunächst die Theorie beherrscht und verstanden hat. Die kann man sich entweder anlesen oder antrainieren.

Ausbildungswege

Aus der deutschsprachigen Literatur zum Erlernen von Scrum können wir zwei Werke empfehlen: [Wolf 2021] und [Gloger 2016]. Wer anschließend einem Scrum-Team über die Schulter schauen und es bei dessen ersten Schritten mit Scrum begleiten möchte, der werfe einen Blick in [Koschek 2014], [Wolf 2015] und [Röpstorff 2022]. Für die Verantwortlichkeiten in Scrum gibt es darüber hinaus spezifische Literatur, beispielsweise [Schiller 2022] für Scrum Master, [Düsterbeck 2022] und [Schuurman 2020] für Product Owner sowie [Shore 2023] für Developer.

Lebendiger und nachhaltiger lernt man Scrum unserer Meinung nach in einem Training. Der Vorteil einer Schulung gegenüber dem Buch ist, dass man einige Prinzipien und Praktiken von Scrum tatsächlich erleben kann – üblicherweise im Rahmen einer Simulation. Neben den Zertifizierungskursen der Scrum Alliance, in denen man unter anderem den Titel »Certified ScrumMaster®« erwerben kann, gibt es die Kurse von Ken Schwabers Scrum.org (hier lautet der Titel »Professional Scrum Master™«) und viele weitere Schulungsangebote – Tendenz steigend. Sowohl die Scrum Alliance als auch Scrum.org bieten Zertifizierungen an. Die Zertifikate beider Organisationen finden weltweit Anerkennung. Ob man als Scrum-Lehrling Wert auf dieses Zertifikat legt, ist eine individuelle Entscheidung, für oder gegen die wir keine Empfehlung aussprechen wollen.

Wir empfehlen, dass alle Mitglieder eines Scrum-Teams geschult werden und nicht nur Scrum Master und Product Owner. Die Developer werden oft vergessen, wenn es um die methodische Ausbildung geht. Wer hier spart, wird während der Sprints Aufwand treiben müssen, um immer wieder die Methodik und das Rollenverständnis zu erklären. Das geht zulasten der Zeit, die für die Umsetzung der Product Backlog Items zur Verfügung steht. Nicht so recht zu wissen, wie Scrum funktioniert, erzeugt ein Gefühl der Unsicherheit. Auch wird es den Developern schwerfallen, Eigenverantwortung für etwas zu übernehmen, das sie nicht genau kennen. Nur ein gut ausgebildetes Scrum-Team wird schnell agil arbeiten können.

Um Scrum schneller zu verankern, kann ein erfahrener Scrum-Coach das neue Scrum-Team in den ersten Sprints begleiten. Er oder sie hat einen Blick und ein Gespür dafür, wo der Scrum-Prozess noch unrund läuft, und kann dem Scrum Master geeignete Maßnahmen vorschlagen, um den Prozess und das Miteinander zu verbessern. Auch der Product Owner muss erst in seine neue Rolle hineinfinden und ist oft froh über methodische Unterstützung. Die Developer wiederum wenden sich insbesondere in der ersten Zeit gern an eine aus Team-Sicht neutrale Person, um Fragen zu stellen und Hilfe bei der

Konfliktbewältigung zu bekommen. Scrum-Coaches können diese Rolle einnehmen, damit sich das Scrum-Team schneller findet.

Aller Anfang ist schwer

Jetzt geht es los mit Scrum – und zwar auf der Shu-Stufe. Dessen müssen sich alle bewusst sein: das Management und die Stakeholder, die Scrum als Vorgehensmodell ausgewählt haben, aber auch das Scrum-Team selbst, das sich nicht überschätzen sollte. Üblicherweise dauert es mindestens drei Sprints, bis sich ein Scrum-Team auf die neue Vorgehensweise eingespielt hat. In dieser Zeit sollte niemand zu viel erwarten, denn vieles will gelernt und gelebt sein: das Miteinander im Scrum-Team, das Kennenlernen der Fähigkeiten der anderen Teammitglieder, das Schätzen von Product Backlog Items und vor allem die eigenverantwortliche Arbeitsweise. Und hinter all dem stehen die agilen Werte, die nicht für jeden selbstverständlich sind. Mithilfe von Simulationen und Spielen kann der Scrum Master einen Beitrag zum Verinnerlichen der Werte leisten [Wiechmann 2020]. Noch hilfreicher ist es, wenn die Werte von Scrum Master, Product Owner und dem Management vorgelebt werden.

»Machen wir wirklich Scrum?«

Wir machen immer wieder die Erfahrung, dass sich Teams zwar auf die Scrum-Definition im Scrum Guide beziehen, aber gar nicht wissen, welche Verantwortlichkeiten, Events und Artefakte dort beschrieben sind. Das, was viele agile Praktiker unter Scrum verstehen, geht nämlich oft über den definierten Rahmen (drei Verantwortlichkeiten, fünf Events, drei Artefakte und fünf Werte) hinaus.

Zur Erinnerung: Scrum besteht aus

- den drei Verantwortlichkeiten *Product Owner, Developer* und *Scrum Master*, die gemeinsam das *Scrum-Team* bilden,
- dem *Sprint* als erstem Event mit festem Zeitrahmen,
- den vier weiteren Events *Sprint Planning, Daily Scrum, Sprint Review* und *Sprint-Retrospektive*,
- den drei Artefakten *Product Backlog* (mit dem *Produkt-Ziel*), *Sprint Backlog* (mit dem *Sprint-Ziel*) und *Inkrement* (mit der *Definition of Done*),
- den fünf Werten *Commitment, Fokus, Offenheit, Respekt* und *Mut*.

Und hier eine kleine Sammlung der Dinge, die nicht zum Kern von Scrum gehören:

- Das *Refinement*, in dem der Product Owner neue oder modifizierte User Stories mit den Developern bespricht und von diesen schätzen lässt (im Scrum Guide werden Refinement-Tätigkeiten erwähnt, aber kein dediziertes Event dafür).
- *Burnup oder Burndown Charts*, mit denen der Fortschritt im Sprint tagesgenau ermittelt und visualisiert wird.
- Das *Impediment Backlog*, auf dem der Scrum Master alle Hindernisse aufführt, die aus dem Weg geräumt werden müssen, um den Developern ein zügiges und zielgerichtetes Arbeiten zu ermöglichen.
- Das *Team Backlog*, in dem die Developer Aufgaben festhalten, die sie in kommenden Sprints zur Prozessverbesserung umsetzen wollen.
- *User Stories* als formales Beschreibungsmittel für Anforderungen (Product Backlog Items).
- *Planning Poker* als Methode, um gemeinsam die Größe von Product Backlog Items zu schätzen.
- *Story Points* als Maß für die Größe von Product Backlog Items.

Alle diese Elemente wurden auf der Grundlage praktischer Scrum-Erfahrung entwickelt, publiziert, von anderen Scrum-Teams übernommen, gegebenenfalls angepasst und weitergegeben, sodass sie irgendwann Teil des kollektiven Scrum-Verständnisses wurden. Die Entstehungsgeschichte dieser Elemente (von Praktikern für Praktiker) belegt deren Relevanz im Scrum-Kontext – trotzdem sind sie kein Bestandteil der »reinen Lehre«.

Was genau ist dann Scrum?

Das Scrum-Rahmenwerk, wie es hier beschrieben wird, ist unveränderlich. Es ist zwar möglich, nur Teile von Scrum zu implementieren, aber das Ergebnis ist nicht Scrum. Scrum existiert nur in seiner Gesamtheit und funktioniert gut als Container für andere Techniken, Methodiken und Praktiken.

So streng definiert der Scrum Guide, was Scrum ist. Und auf diese Definition beziehen sich viele Scrum-Teams, wenn sie die eigene Vorgehensweise bewerten. Oft kommen sie dann zu dem Ergebnis, dass sie kein Scrum im eigentlichen Sinne machen.

Der Scrum Guide ist für die Shu-Stufe geschrieben

Betrachten Sie den Scrum Guide als Anleitung für die Shu-Stufe. Er soll den »Lehrlingen« helfen, sich um die Anwendung des Scrum-Frameworks zu kümmern, anstatt sich von Beginn an Gedanken über den kontextspezifischen Zuschnitt des Regelwerks machen zu müssen. Deshalb enthält er beispielsweise die Formeln für die Länge von Events: damit ein unerfahrener Scrum Master ohne langes Grübeln einen Termin für das Sprint Review versenden kann.

Für fortgeschrittene Scrum-Teams hingegen kann der Scrum Guide immer wieder als Referenz dienen, um herauszufinden, ob sie sich mit ihren produkt-, domänen- und unternehmensspezifischen Anpassungen noch innerhalb des Scrum-Rahmens bewegen.

Der Scrum Guide fordert Transparenz, Überprüfung und Anpassung

Ein Scrum-Team wird an verschiedenen Stellen im Scrum Guide explizit dazu aufgefordert, den Prozess für seinen Kontext zu optimieren – aber erst, wenn es den Scrum-Prozess verstanden und verinnerlicht hat:

Die Scrum-Artefakte und der Fortschritt in Richtung der vereinbarten Ziele müssen häufig und sorgfältig überprüft werden, um potenziell unerwünschte Abweichungen oder Probleme aufzudecken. Um bei der Überprüfung zu helfen, bietet Scrum einen Rhythmus in Form seiner fünf Events.

Überprüfung ermöglicht Anpassung. Überprüfung ohne Anpassung wird als unsinnig betrachtet. Scrum-Events sind darauf ausgerichtet, Veränderungen zu bewirken.

Wenn einzelne Aspekte eines Prozesses von akzeptablen Grenzen abweichen oder wenn das resultierende Produkt nicht akzeptabel ist, müssen der angewandte Prozess oder die produzierten Ergebnisse angepasst werden. Die Anpassung muss so schnell wie möglich erfolgen, um weitere Abweichungen zu minimieren.

Die Anpassung wird schwieriger, wenn die beteiligten Personen nicht bevollmächtigt (empowered) sind oder sich nicht selbst managen können. Von einem Scrum-Team wird erwartet, dass es sich in dem Moment anpasst, in dem es durch Überprüfung etwas Neues lernt.

Damit legt der Scrum Guide den Grundstein für eine kontinuierliche Weiterentwicklung des Vorgehens, getrieben durch alle Teammitglieder. Mit diesem Rüstzeug kann jedes Scrum-Team die Ri-Stufe erklimmen.

Schritt 3: Scrum-Teams entwickeln

Ein Scrum-Team entsteht nicht von allein. Hier ist die aktive Mitarbeit aller Teammitglieder und des Managements gefordert.

Das Tuckman-Modell der Teamentwicklung

Damit aus einer Gruppe von Individuen ein Team wird, müssen diese Personen nach einem Modell von Bruce Tuckman [Tuckman 1965] vier Phasen durchlaufen:

- *Forming* – Das Team entsteht. Jeder muss seine Rolle finden, will die anderen kennenlernen, hält sich zurück (»Man«-Orientierung – »man müsste mal …«).
- *Storming* – Die Nagelprobe für das in der Entstehung begriffene Team. Konflikte werden offen ausgetragen, Machtkämpfe ausgefochten (»Ich«-Orientierung).
- *Norming* – Hat das Team die Storming-Phase überstanden, werden gemeinsame Normen festgelegt und Vereinbarungen für die Zusammenarbeit getroffen. Durch vertrauensvolle Kooperation entsteht ein »Wir«-Gefühl.
- *Performing* – Das Team hat sich und seinen Rhythmus gefunden. Es ist jetzt produktiv und arbeitet weitestgehend selbstbestimmt.

Tuckman hat später noch eine fünfte Phase (*Adjourning*) definiert, in der sich das Team auflöst. Ändert sich die Teamzusammensetzung, durchläuft das neu formierte Team alle Phasen von vorn.

Dieses Modell ist eine Erklärung dafür, dass Scrum-Teams ca. drei Sprints brauchen, um effektiv und effizient zusammenzuarbeiten (die andere Erklärung ist, dass Scrum zwar ein einfaches Regelwerk hat, dessen praktische Umsetzung aber nicht trivial ist und einige Zeit benötigt). Das Tuckman-Modell hilft dem Scrum Master bei der Einschätzung, wie viel Unterstützung das Team von ihm braucht.

[Koschek 2022] enthält eine kritische Betrachtung des Tuckman-Modells und beschreibt ein alternatives Modell, das der Komplexität von Teamentwicklung besser gerecht wird.

Scrum-Teams sind selbststeuernd

Scrum-Teams sind interdisziplinär, d.h., die Mitglieder verfügen über alle Fähigkeiten, die erforderlich sind, um in jedem Sprint Wert zu schaffen. Sie managen sich außerdem selbst, d.h., sie entscheiden intern, wer was wann und wie macht.

Entsprechend dieser Definition aus dem Scrum Guide überlassen selbststeuernde Teams die Entscheidung, wie sie gemeinsam die anstehenden Aufgaben erledigen wollen, niemand Geringerem als sich selbst. Niemand außerhalb des Scrum-Teams ist befugt, sich in die Arbeitsweisen der Developer, des Scrum Master oder des Product Owner einzumischen. Sie dürfen allerdings beobachten und ihre Beobachtungen mit dem Scrum-Team teilen. Darüber hinaus stellt der Scrum Master sicher, dass das Scrum-Team die Scrum-Werte und -Regeln nicht verletzt. Diese Verlagerung der Verantwortung von einem Team- oder Projektleiter hin zum Scrum-Team löst bei den Teammitgliedern unterschiedliche Gefühle aus. Was für den einen ein revolutionärer Befreiungsschlag ist, erscheint der anderen als große Last. Um unter dieser Last nicht zusammenzubrechen, wünschen sich viele Softwareentwickler, die sich mit Selbststeuerung nicht sofort anfreunden können, eine Schulung zu diesem Thema.

Kann man Selbststeuerung lehren und lernen? Auf diese Fragen gibt es unserer Meinung nach keine einfache Antwort. Eines aber ist sicher: Der Weg zum selbststeuernden Team braucht Zeit, Vertrauen und einen geschützten Rahmen, in dem das Team ausprobieren und lernen kann.

Warum wir in diesem Buch den Begriff »selbststeuernd« anstelle von » selbstmanagend« verwenden, haben wir im Abschnitt »Das Zusammenspiel von Werten, Prinzipien und Praktiken« auf Seite 42 beschrieben.

Selbststeuerung vermitteln. Lehren im klassischen Sinne kann man Selbststeuerung nicht. Allerdings lassen sich die Gründe, die dafür sprechen, sowohl erzählerisch als auch spielerisch (in Form von Gruppenübungen oder Simulationen) vermitteln. Eine Übung, die den Vorteil von Selbststeuerung im Vergleich zu Command-and-Control-Strukturen erlebbar macht, ist der menschliche Knoten [Human Knot]. Am besten fördert man Selbststeuerung jedoch durch Vorleben. Jedes Teammitglied trifft wesentliche Entscheidungen gemeinsam mit den Teamkolleginnen und -kollegen. Es fragt sie um Rat, wenn es nicht weiterweiß. Es weist auf Probleme und offene Punkte hin und bietet Lösungsmöglichkeiten an (konstruktive Kritik). Und es fordert die Unterstützung anderer, beispielsweise des Product Owner, aktiv ein. Damit macht das

Teammitglied deutlich, dass es Verantwortung übernimmt und Entscheidungen herbeiführen will, um Ergebnisse erzielen zu können.

Das Management und die Führungskräfte, in deren Verantwortungsbereich ein Scrum-Team arbeitet, müssen den Rahmen schaffen, der den Teammitgliedern das oben geschilderte Verhalten ermöglicht. Dazu müssen sie zunächst anerkennen, dass Organisationen, Teams und Projekte komplexe adaptive Systeme (im Sinne der Komplexitätsforschung) sind. Wesentliches Merkmal solcher Systeme ist, dass sich ihr Verhalten nicht steuern lässt, weil es aus verschiedenen meist zyklischen Ursache-Wirkung-Ketten entsteht. In der Folge ist die Wirkung einer Maßnahme nicht absehbar und deshalb auch nicht kontrollierbar. Hier versagt also das Command-and-Control-Muster. Deshalb ist es besser, einen Handlungsrahmen abzustecken und einen Raum zu schaffen, in dem Zusammenarbeit, Interaktion und Innovation stattfinden können. Nicht alles, was in diesem Raum geschieht, ist auf den ersten Blick zielführend. Anstatt vermeintliche Fehlentwicklungen sofort zu unterbinden, sollte man abwarten und den weiteren Weg der Entwicklung beobachten. Das bedeutet, darauf zu vertrauen, dass das Team ein gutes Gespür dafür hat (oder entwickelt), welche Maßnahmen und Arbeitsweisen progressiv sind. Dem Team zu vertrauen, fällt vielen Führungskräften nicht leicht – nicht etwa, weil sie die Reife des Teams in Zweifel stellen, sondern vielmehr, weil sie die Kontrolle und damit ein Stück Macht abgeben. Die durch das Loslassen gewonnene Zeit lässt sich wunderbar darauf verwenden, das Team zu unterstützen und zu schützen. Mit diesem Führungsstil wird die Selbststeuerung des Teams bestmöglich gefördert. Dies steht dann auch wunderbar im Einklang mit dem fünften Prinzip des Agilen Manifests, das da lautet: *Errichte Projekte rund um motivierte Individuen. Gib ihnen das Umfeld und die Unterstützung, die sie benötigen, und vertraue darauf, dass sie die Aufgabe erledigen*.

Ein für die Gestaltung des organisatorischen Rahmens agiler Teams wichtiges Konzept ist das Pull-Prinzip. Anstatt Aufträge aktiv an Teammitglieder zu verteilen, sollen sich die Teammitglieder diese Aufträge selbstständig abholen (»pullen«) und bearbeiten.

Das Vermitteln von Selbststeuerung durch Vorleben und das Schaffen geeigneter Rahmenbedingungen auf der Basis eines Wertesystems unter Nutzung von Feedback-Zyklen und dem Pull-Prinzip ist eine wesentliche Voraussetzung, um ein komplexes adaptives System erfolgreich leben zu lassen. Die Wirklichkeit sieht in vielen Unternehmen heute anders aus, und die Transformation zu einer Organisationsform mit den oben genannten Charakteristika ist die größte Herausforderung, wenn die agile Denkweise aus einzelnen Teams in das gesamte Unternehmen auszustrahlen beginnt.

Selbststeuerung lernen. Der Frage, ob man Selbststeuerung lernen kann, nähern wir uns über die Erkenntnis, dass man Selbststeuerung nicht befehlen kann. Damit ist für uns klar, dass eine intrinsische Motivation gegeben sein muss. Die allein reicht jedoch nicht aus. Sie muss gepaart sein mit einer hohen fachlichen Qualifikation, großer Flexibilität im Denken und Handeln sowie einer hohen Leistungsfähigkeit, weil ein selbststeuerndes Teammitglied das »Was« verstehen muss und über das »Wie« diskutieren und entscheiden können muss. Neben der fachlichen Kompetenz muss vor allem die Prozess- und Sozialkompetenz gestärkt werden, beispielsweise durch entsprechende Weiterbildungsangebote. Flankierend sollten die Führungskräfte sowie jene Teammitglieder, die bereits über diese Kompetenzen verfügen, in ihrem Verhalten klassische Werte wie Orientierung, Geborgenheit und Sicherheit erkennen lassen, um die anderen Kolleginnen und Kollegen in ihrem Wandel bestmöglich zu unterstützen [Vigenschow 2016].

Mentoring. Ein in der Praxis bewährtes Mittel, um Teammitglieder zu einer selbststeuernden Arbeitsweise zu befähigen, ist die Unterstützung durch einen Mentor oder eine Mentorin. Eine Mentorin ist jemand, die schon einmal dort war, wo der Mentee (bewusst oder unbewusst) hinwill. Für das Ausüben dieser Rolle ist vor allem Methodenkompetenz und Vertrautheit mit den impliziten und expliziten Regeln der Organisation vonnöten, idealerweise gepaart mit Integrität und Seniorität. Solche Menschen sind schwer zu finden, aber die Suche nach ihnen lohnt sich doppelt. Nicht nur, dass sie ihren Mentee auf dem Weg in die Selbststeuerung begleiten – sie führen ihn durch die Begleitung unter Umständen auch an die Mentorenrolle heran. So wird vielleicht aus dem Mentee selbst der nächste Mentor, der die erlernten und erlebten Werte und Fähigkeiten an andere Kolleginnen und Kollegen weitergibt. Auf diese Weise entsteht eine Unternehmenskultur, die auf gegenseitigem Respekt, Vertrauen und Verantwortung fußt.

Warum braucht ein selbststeuerndes Team einen Scrum Master? »Hat das Team die Prinzipien und Praktiken von Scrum verinnerlicht und weiß diese zu nutzen, muss niemand mehr über die Einhaltung der Spielregeln wachen.« Mit diesem Argument versuchen viele unerfahrene Scrum-Teams, ihren Scrum Master wegzurationalisieren. Dabei ist der Scrum Master nicht allein ein Regelwächter. Er ist Moderator, Mentorin, Coach, Schlichterin, Tröster, Motivatorin, kurz: die gute Seele und das Gewissen des Teams. Er sorgt dafür, dass Hindernisse aus dem Weg geräumt werden, steht den anderen Mitgliedern des Scrum-Teams mit Rat und Tat zur Seite und hält ihnen den Rücken frei. Und er verhindert, dass sie im täglichen Trott zu erlahmen drohen.

Man darf außerdem nicht vergessen, dass Scrum-Teams meistens dann wieder in alte Verhaltensmuster zurückfallen, wenn sie zum ersten Mal das Gefühl haben, so richtig Scrum-konform zu arbeiten. Das ist vergleichbar mit einem Autofahrer, der in einem Land mit Rechtsverkehr wohnt und zum ersten Mal in ein Land mit Linksverkehr reist (siehe [Koschek 2012]): Zunächst konzentriert er sich stark auf seine Fahrweise. Irgendwann kommt der Punkt, an dem er glaubt, den Linksverkehr verinnerlicht zu haben. Die Konzentration lässt nach, und schon steuert er beim Abbiegen die rechte Fahrspur an. Offensichtlich arbeitet das Unterbewusstsein immer noch im Rechtsverkehr-Modus. Ähnlich geht es den Mitgliedern eines Scrum-Teams. Viele der über lange Jahre praktizierten Vorgehensweisen werden in Scrum durch neue Prinzipien und Praktiken ersetzt – genauer: überdeckt, denn im Unterbewusstsein sind die alten Muster noch vorhanden. Sie kommen zum Vorschein, sobald die Konzentration nachlässt oder der Druck steigt. Dann ist der Scrum Master gefragt.

Leider nimmt man das Fehlen eines Scrum Master nicht sofort wahr. Der durch die Abwesenheit eines Scrum Master ausgelöste Prozess ist schleichend. Deshalb merken oft nicht einmal die Betroffenen selbst, dass sich die Qualität der Zusammenarbeit immer weiter verschlechtert. Ein selbststeuerndes Team ohne Scrum Master sollte deshalb regelmäßig eine unabhängige Beobachtung von außen einfordern – beispielsweise durch den Scrum Master eines anderen Teams.

Scrum-Teams sind interdisziplinär

Ein solch autarkes Team zusammenzustellen, ist für viele Unternehmen eine große Herausforderung: Die Expertin, die man gern im Team hätte, hat nur bedingt Zeit oder gehört zu einer anderen Abteilung. Die Tester arbeiten in einer separaten Qualitätssicherungsabteilung. Software kann nur vom Betriebsteam auf den produktiven Systemen installiert werden. Diese und andere Hürden gilt es zu meistern, wenn man Scrum-Teams zusammenstellt – von den zwischenmenschlichen Herausforderungen ganz zu schweigen. Für viele dieser Hürden gibt es kein Patentrezept – wohl aber ein Grundprinzip, das die Verantwortlichkeiten klar regelt:

Collective Ownership. In Scrum-Teams sind alle für alles verantwortlich. Wann immer eine Aufgabe ansteht, wird sie von derjenigen erledigt, die Zeit hat und über die erforderliche Kompetenz verfügt. Fehlt ihr die Kompetenz, bittet sie die anderen Teammitglieder um Rat und Unterstützung. Das Prinzip der Collective Ownership sorgt dafür, dass Product Backlog Items tatsächlich fertiggestellt werden, denn niemand kann sagen: »Das ist keine Aufgabe für mich!« Dieses Prinzip gilt nicht nur für die Tasks im Rahmen der Bearbeitung

eines Product Backlog Item, sondern auch für den Programmcode. Wann immer ein Entwickler einen Fehler irgendwo im Code feststellt, muss er diesen beheben – auch dann, wenn der fehlerhafte Code von einem anderen Entwickler stammt. Diese von den Pfadfindern bekannte Grundeinstellung trägt stark zu einem ausgeglichenen Qualitätsniveau bei, erfordert aber eine sehr offene und ergebnisorientierte Kultur. Nicht jedem fällt es auf Anhieb leicht, zu akzeptieren, dass andere einen Fehler im »eigenen« Code gefunden und behoben haben. Eitelkeit und Versagensängste spielen eine große Rolle und lassen sich nur durch den Aufbau und die Pflege des Vertrauens im Team lindern.

Wenn alle für alles verantwortlich sind, stellt sich die Frage, wie das Qualifikations- und Präferenzprofil des idealen Scrum-Teammitglieds auszusehen hat.

Alle sind »Developer« – aber nicht zwingend Generalisten. Die Developer haben keine spezifischen Titel, alle sind »Developer«. Aber sie dürfen durchaus unterschiedliche Rollen haben und sich auf unterschiedliche Gebiete spezialisiert haben – genau das macht ja ein interdisziplinäres Team aus. Zusätzlich zu ihrem Spezialwissen zeichnen sich die Developer eines Scrum-Teams durch ein breites Allgemeinwissen und ein ernsthaftes Interesse an den Spezialgebieten ihrer Teammitglieder aus. Dieses Allgemeinwissen vertiefen sie, indem sie paarweise mit einer Teamkollegin oder einem Teamkollegen an einer Aufgabe arbeiten und somit einen praktischen Einblick in deren Aufgabengebiet sowie deren Denk- und Arbeitsweisen bekommen. Das Kompetenzprofil solcher Menschen nennt man T-förmig (*T-shaped*). Der horizontale T-Balken steht für das breite Allgemeinwissen, der senkrechte Balken für das tief gehende Spezialwissen. Zunehmend finden sich auch Pi-förmige (*Pi-shaped*) bis Kamm-förmige (*Comb-shaped*) Persönlichkeiten, die sich durch mehr als nur ein Spezialgebiet bei dennoch breitem Basiswissen auszeichnen.

Vielfach herrscht immer noch die Annahme, dass der Verzicht auf Titel in einem Scrum-Team zur Konsequenz hat, dass es in Scrum keine Spezialistinnen bzw. Spezialisten gibt oder geben darf. Diese Diskussion entzündet sich oft an der Rolle des Softwarearchitekten. Dessen Aufgaben, so die landläufige Meinung, werden in Scrum von allen Developern wahrgenommen. Tatsächlich ist es oft so, dass ein erfahrener Softwarearchitekt als Mitglied eines Scrum-Teams diese Rolle exklusiv einnimmt. In Scrum gibt es für die anderen Teammitglieder viele Gelegenheiten, mit dem erfahrenen Softwarearchitekten zusammenzuarbeiten. Dadurch entwickeln sie ein besseres Verständnis für die Softwarearchitektur, sodass sie diese Rolle zunächst als Urlaubsvertretung übernehmen und später komplett ausfüllen können. Das setzt nicht nur den Willen zu lernen voraus,

sondern aufseiten des Softwarearchitekten auch die Bereitschaft, die Fähigkeit und die Geduld, sein Wissen zu teilen.

Zusammenarbeit – intern und extern

Die kollaborative Prägung von Scrum stützt die erste Gegenüberstellung des Agilen Manifests: »Individuen und Interaktionen mehr als Prozesse und Werkzeuge«. Miteinander reden und einander zuhören, so die Meinung überzeugter Scrum-Anwender, ist zielführender als der bloße Austausch von Dokumenten. Das liegt unter anderem darin begründet, dass das geschriebene Wort für viele Menschen eine gewisse Endgültigkeit suggeriert, wohingegen ein Dialog zu neuen Erkenntnissen und Ergebnissen führen kann. Deshalb sieht Scrum im Laufe eines Sprints eine Reihe von Events vor, beginnend mit dem Sprint Planning über die Daily Scrums bis hin zu Sprint Review und Sprint-Retrospektive. Auch außerhalb dieser Events wird die Zusammenarbeit gefördert, z.B. durch die interdisziplinäre Teamkonstellation und Praktiken wie Pair Programming.

Arbeit am Taskboard. Das Daily Scrum dient der täglichen Synchronisation und findet am Taskboard statt. Auf diese Weise hat das gesamte Scrum-Team die anstehenden und die erledigten Aufgaben sowie gegebenenfalls die Burndown Charts im Blick. Dieser für alle transparente Status ist wichtig, um Hindernisse und Verzögerungen frühzeitig zu erkennen und Gegenmaßnahmen einzuleiten.

Um im Sprint zielgerichtet und erfolgsorientiert voranschreiten und einen tagesaktuellen Status fortschreiben zu können, gibt es zwei Prinzipien, die für die Arbeit am Taskboard von Bedeutung sind:

- Product Backlog Items werden sequenziell abgearbeitet.
- Jeder Developer nimmt sich genau eine Aufgabe vor.

Durch die sequenzielle Bearbeitung der Product Backlog Items wird verhindert, dass am Ende des Sprints alle Items »fast fertig« sind. Natürlich müssen sich die Developer nicht künstlich auf ein Item beschränken. Wenn zur Fertigstellung eines Item nicht mehr alle Entwicklerinnen und Entwickler benötigt werden, können einige davon bereits mit der Bearbeitung des nächsten Product Backlog Item beginnen. Das Ziel soll aber immer sein, möglichst wenige Product Backlog Items gleichzeitig in Arbeit zu haben.

Die Beschränkung eines Entwicklers auf eine einzige Aufgabe unterstützt das Ziel, ein Product Backlog Item tatsächlich fertig zu bekommen. Fokussierung (einer der Werte im Scrum) ist die Voraussetzung für eine ergebnisorientierte

Arbeitsweise. Scrum ist ergebnisorientiert, indem es für jeden Sprint ein Sprint-Ziel fordert und mit jedem Sprint mindestens ein Inkrement liefert.

Zusammenarbeit mit dem Product Owner. Zur Erinnerung: Der Product Owner ist für das »Was« verantwortlich, die Developer für das »Wie«. Nur gemeinsam entsteht aus »Was« und »Wie« ein Inkrement. Deshalb müssen Product Owner und Developer gut zusammenarbeiten – sie sind schließlich Mitglieder desselben Scrum-Teams. Je respektvoller der Umgang miteinander und je größer das Interesse an der Perspektive des anderen, desto reibungsloser wird die Zusammenarbeit vonstattengehen. Leicht gesagt, nicht ganz so leicht getan.

Als fachlich und wirtschaftlich Verantwortliche haben Product Owner ein großes Interesse daran, ein Maximum an Funktionalität in minimaler Zeit zu erhalten. Das verträgt sich nicht immer mit der nachhaltigen Geschwindigkeit (*Sustainable Pace*), mit der die Developer über einen langen Zeitraum gute Ergebnisse mit hoher Qualität unter Berücksichtigung der Gesundheit aller Teammitglieder erzielen wollen. Die Developer können aber einiges zu einem verträglichen Miteinander mit dem Product Owner beitragen:

Developer, die sich zu viele Product Backlog Items in den Sprint ziehen, haben am Ende entweder nicht alles geschafft oder viele Überstunden investiert, um das Sprint-Ziel zu erreichen. Beides ist zu vermeiden. Deshalb empfehlen wir eine defensive Planung, die Reserven einkalkuliert. Aber Vorsicht: Die Grenze zwischen defensiver und bequemer Planung ist fließend. Developer, die sich bewusst zu wenig vornehmen, um im Sprint ganz gemütlich arbeiten zu können, laufen Gefahr, am Ende gar nichts zu schaffen (weil der sanfte Druck fehlt) und den Product Owner zu enttäuschen (weil er zu wenig geliefert bekommt).

Fordert der Product Owner zu viel oder fügt er im laufenden Sprint weitere Anforderungen hinzu (z.B. in Form zusätzlicher Akzeptanzkriterien oder gar neuer Product Backlog Items), müssen ihn die Developer daran erinnern (unter Umständen mithilfe des Scrum Master), dass der Sprint geschützt ist und das Sprint Backlog nur von den Developern nachträglich verändert werden darf. Ähnliches gilt für spontane Anfragen und Aufträge von außen, die nicht eingeplant wurden und nichts mit dem Sprint-Ziel zu tun haben (»Kannste mal eben …?«). Mit Hinweis auf das Sprint-Ziel, das vom Scrum-Team gemeinsam akzeptiert wurde, sollte sich jedes Teammitglied solcher Anfragen erwehren können – entweder allein oder mithilfe des Scrum Master.

Einige Scrum-Teams haben ein schlechtes Gewissen, wenn nach der Bearbeitung des kompletten Sprint Backlog noch ein wenig Sprint übrig ist. Passiert das bei jedem Sprint, plant es eventuell zu defensiv. Geschieht es hingegen nur ab und zu, sollten die Developer dem ersten Impuls widerstehen,

mit der Bearbeitung eines neuen Product Backlog Item zu beginnen – vor allem dann, wenn es dieses Item in der verbleibenden Zeit nicht fertigstellen kann. Stattdessen sollten sie überlegen, wie sie die verbleibende Zeit am sinnvollsten nutzen – etwa um den Code noch einmal zu refaktorisieren, technische Schulden abzubauen, den Automatisierungsgrad zu erhöhen oder den Prozess zu verbessern. Die nachhaltige Wirkung dieser Aktivitäten ist Rechtfertigung genug, um der kontinuierlichen Verbesserung den Vorzug gegenüber einem Mehr an Fachlichkeit zu geben.

Was mache ich mit Individualisten? Ob Diva oder Held: Es genügt ein Exemplar dieser egozentrischen Spezies, um ein ganzes Team zu sprengen. Während Individualisten in anderen Umgebungen ihre Bühne oder Nische finden, sorgen sie in einem Scrum-Team für Unruhe, denn hier steht der Teamgedanke im Mittelpunkt – und nicht ein einzelnes Teammitglied.

Selbststeuernde Teams sollten in der Lage sein, Individualisten zu einer teamorientierten Arbeitsweise zu bewegen, indem sie beispielsweise im Sprint Review und in der Sprint-Retrospektive deutlich betonen, dass alle Ergebnisse vom Scrum-Team gemeinsam erarbeitet wurden und somit dem Selbstdarsteller keine Bühne bieten. Der Scrum Master kann unterstützen, indem er beispielsweise konsequentes Pair Programming empfiehlt und damit keine Einzelleistungen zulässt. Im Rahmen von Vieraugengesprächen, die er mit allen Teammitgliedern regelmäßig führt, kann der Scrum Master einen Individualisten auf dessen Verhalten ansprechen und ihm die Wirkung dieses Verhaltens auf andere verdeutlichen. Wichtig ist nur, dass man diesen Personen nicht die Erfolgserlebnisse nimmt, sondern versucht, ihnen klarzumachen, dass ein gemeinsam errungener Erfolg viel mehr wert ist als ein Einzelerfolg.

Schritt 4: Scrum in den organisatorischen Kontext einbetten

Scrum-Teams sind selten isoliert von der Außenwelt. Somit ergeben sich zwangsläufig Schnittstellen – zum Gesamtprojekt, zur Organisation, zu Kunden oder Lieferanten. Diese Schnittstellenpartner sind nicht immer agil. Es muss deshalb ein Weg der Zusammenarbeit gefunden werden, der den Bedürfnissen der agilen und der nicht agilen Kommunikationspartner gerecht wird.

Wie hole ich Hilfe aus der Welt jenseits von Scrum?

Da viele Unternehmen ihre ersten agilen Gehversuche in ausgewählten Projekten stattfinden lassen, bevor sie (wenn überhaupt) diese Arbeitsweise auf

das gesamte Unternehmen ausdehnen, tritt unweigerlich die Situation ein, dass ein Scrum-Team auf die Unterstützung aus Organisationseinheiten angewiesen ist, die nicht nach Scrum arbeiten. Dabei kann es sich zum Beispiel um die Unterstützung durch einen Spezialisten handeln, dessen Erfahrung für ein spezielles Thema erforderlich ist. Eine solche Person dauerhaft im Scrum-Team zu halten, wäre einerseits kontraproduktiv und ist andererseits unrealistisch. Die Integration beispielsweise einer Expertin erfolgt idealerweise für einen gesamten Sprint. Ist das nicht möglich, sollte die Expertin zumindest am Sprint Planning teilnehmen. Auf diese Weise ist sie in die Sprint-Planung involviert und weiß, wo sich ihr Spezialwissen im Ergebnis wiederfinden wird – ein wichtiger Motivator. Anstatt immer nur tage- oder stundenweise an ihren Aufgaben zu arbeiten, empfehlen wir Experten eine kontinuierliche und fokussierte Arbeitsweise – vorausgesetzt, sie können ihre Arbeit entsprechend organisieren. Am Ende bekommt die Expertin im Sprint Review die Chance, ihren Beitrag zum Sprint-Ziel zu präsentieren. Und ganz nebenbei hat sie erste praktische Scrum-Erfahrungen gesammelt.

Synchronisierung von Sprints und nicht agilen Projektplänen

Es sind nicht nur Experten aus der »Außenwelt«, zu denen eine Abhängigkeit besteht. Oft ist ein Scrum-Team auf Artefakte angewiesen, die außerhalb des Teams entstehen. Wenn das Sprint-Ziel von solchen Zulieferungen abhängt, erschwert das die Planung. Schließlich arbeiten die Zulieferer nicht zwingend im selben Sprint-Takt, oft sogar ganz ohne Takt. Solange ein Lieferzeitpunkt verbindlich zugesagt (bzw. im Scrum-Sinne prognostiziert) werden kann, lässt sich das zugelieferte Artefakt in dem auf die Lieferung folgenden Sprint verwenden oder verarbeiten. Schwierig wird es nur, wenn kein Liefertermin genannt wird – etwa weil der Zulieferer keine prioritätsgesteuerte Planung nutzt. In diesem Fall hilft nur ein (er-)klärendes Gespräch, an dessen Ende ein Liefertermin steht.

Einbettung in ein nicht agiles Projekt

Neben der oben beschriebenen Unterstützung durch Personal und Artefakte und dem Synchronisierungsproblem gibt es bei der Einbettung eines agilen Teams in ein nicht agiles Gesamtprojekt weitere Aspekte zu beachten.

Rollenverständnis. Nicht allen Beteiligten in traditionellen Projekten sind die Scrum-Verantwortlichkeiten geläufig. Das kann zu Missverständnissen führen – etwa wenn der Projektleiter eines anderen Teilprojekts den

Projektleiter des Scrum-Teilprojekts sucht. Eine kurze Scrum-Schulung, mindestens aber eine Beschreibung von Scrum und insbesondere der Scrum-Verantwortlichkeiten sowie ein Mapping auf die traditionellen Projektrollen wird den anderen Teilprojektteilnehmenden helfen, die Arbeitsweise des Scrum-Teams besser zu verstehen. Besonders empfehlenswert ist es, die Beteiligten der traditionellen Teilprojekte zu den Daily Scrums und zum Sprint Review einzuladen, um ihnen ein Gefühl für die Scrum-Arbeitsweise zu geben. Oft helfen auch Mitglieder des Scrum-Teams, die als »Außenminister« die Schnittstelle zur klassischen Organisation bilden. Sie leisten Aufklärung, werben um Verständnis und schützen damit den agilen Raum.

Bedeutung von Anforderungsbeschreibungen. Viele klassisch ausgebildete Businessanalysten und Requirements Engineers empfinden die Dynamik, mit der sich das Product Backlog entwickelt, und den unterschiedlichen Detaillierungsgrad der im Product Backlog beschriebenen Anforderungen als unkalkulierbares Risiko. Sie sind es gewohnt, vor Beginn der Softwareentwicklung alle Anforderungen nach bestem Wissen und Gewissen möglichst genau beschrieben und den für die Implementierung benötigten Aufwand möglichst genau geschätzt zu haben. Bei der Aufwandschätzung projizieren sie die Anforderungen bzw. Aufgaben oft implizit oder explizit auf die am geeignetsten erscheinenden Teammitglieder. All das passiert in Scrum nicht – und das aus gutem Grund, wie wir in diesem Buch erläutert haben. Anstatt nun von den Anforderungsermittlern zu erwarten, dass diese ihr Wissen und ihre Erfahrung über Bord werfen und fortan die Anforderungen Scrum-konform erheben, sollte ein Scrum-Team einen kooperativen Ansatz wählen, in dem die Stärken der klassischen Anforderungsermittlung mit den Prinzipien agiler Anforderungsermittlung in Einklang gebracht werden. Unserer Erfahrung nach liegt der Unterschied nämlich hauptsächlich in der Frage, *wann* und *wie detailliert* die Anforderungen erhoben werden, und nicht so sehr in der Art, *wie* Anforderungen erhoben und beschrieben werden. Das Instrumentarium des klassischen Anforderungsmanagements ist sehr umfangreich und fundiert. Es lohnt sich, als Product Owner (und auch als Developer) einen Blick zu riskieren und geeignete Werkzeuge für die eigene agile Arbeit zu identifizieren – und zu nutzen.

Dass die Dynamik und Detailvarianz der Items im Product Backlog eine Stärke von Scrum ist, lässt sich besser erleben als beschreiben. Deshalb sollte der Product Owner seine Kolleginnen und Kollegen aus den traditionell arbeitenden Teilprojekten dazu einladen, der Backlog-Pflege beizuwohnen, beispielsweise in einem Refinement.

Vertikaler Durchstich vs. schichtenorientierte Entwicklung. Der Fokus auf ein funktionierendes (Teil-)Produkt führt bei agiler Softwareentwicklung dazu, dass zunächst ein vertikaler Durchstich durch alle Ebenen der Produktarchitektur (bei Softwareprodukten durch alle Ebenen der IT-Schichtenarchitektur) geschaffen wird, bevor man Schritt für Schritt und Iteration für Iteration weitere Funktionen (wieder als Durchstich) ergänzt. In einigen traditionellen Vorgehensweisen wird hingegen schichtenorientiert vorgegangen: Zunächst wird das Datenmodell entwickelt, darauf aufbauend die Datenzugriffsschicht, dann folgen die Funktionsmodule bzw. die Serviceschicht und gegebenenfalls weitere Zwischenschichten, bis schließlich die Benutzeroberfläche (das Frontend) entsteht. Neben der Tatsache, dass bei einer schichtenorientierten Entwicklung erst ganz am Ende etwas Funktionsfähiges entsteht, kann diese Methode dazu führen, dass die vom Scrum-Team benötigten Zulieferungen aus einem schichtenorientiert arbeitenden Team erst sehr spät nutzbar sind (nämlich erst dann, wenn alle Schichten implementiert sind). Das kann die Synchronisierung der Zulieferungen stark beeinträchtigen. Deshalb muss der Product Owner die benötigten Zulieferungen so früh wie möglich identifizieren und mit der liefernden Einheit einen Liefertermin verhandeln. Der noch viel bessere Weg wäre natürlich, Möglichkeiten zur direkten Zusammenarbeit zu finden. Das fördert das Lernen des Scrum-Teams in der bis dahin externen Domäne und gleichzeitig das Kennenlernen von Scrum außerhalb des Scrum-Teams.

Einbettung in eine nicht agile Organisation

Während eine Einbettung in ein nicht agiles Gesamtprojekt gelingen kann, wird dagegen eine Einbettung in eine nicht agile Organisation auf Dauer problematisch werden. Eine Organisation ist nach unserem Verständnis dann nicht agil, wenn sie die im Agilen Manifest beschriebene Denkweise und Grundhaltung und die darauf basierenden Prinzipien ablehnt oder höchstens duldet. Ohne eine agile Grundhaltung bis hinauf ins Management wird es ein agiles Team immer schwer haben. Das beginnt bei der Forderung des Scrum-Teams nach maximaler Transparenz und dem Anspruch der Teammitglieder, unabhängig von der hierarchischen Einordnung eine Meinung vertreten und das »Wie« bestimmen zu dürfen. Und es endet beim Personalmanagement, weil die Qualifikationsprofile für Mitglieder agiler Teams neben technischer und fachlicher Exzellenz auch Methodenkompetenz und Teamfähigkeit fordern. Oft haben agile Teams sogar ein Mitspracherecht bei der Besetzung offener Stellen im Team. Aus diesen Gründen empfehlen wir, die ersten Erfolge agiler Teams zu nutzen, um die agile Denkweise in die Organisation zu tragen. Wie das geht, skizzieren wir im folgenden Abschnitt.

Schritt 5: Die agile Organisation

Um agile Erfahrungen auf andere Teile der Organisation zu übertragen, muss man sich zunächst darüber klar werden, dass man damit im Begriff ist, eine neue Denkweise zu etablieren.

Der Effekt bei der Einführung agiler Methoden ist vergleichbar mit dem Übergang von der prozeduralen zur objektorientierten Programmierung: Sobald man die Prinzipien verstanden und verinnerlicht hat, kann man sich kaum noch vorstellen, je wieder anders zu denken und zu arbeiten. Das liegt daran, dass es sich bei beiden Konzepten nicht allein um eine neue Arbeitsweise handelt, sondern vor allem um eine neue Denkweise oder Sicht auf die Welt. Wo der objektorientierte Programmierer eine Welt voller Objekte mit Eigenschaften sieht, betrachtet der agile Entwickler die Welt als komplexes adaptives System und Softwareentwicklung als kollaborative Tätigkeit in interdisziplinären Teams, die von einem gemeinsamen Wertesystem zusammengehalten werden. Um aus einer Organisation eine agile Organisation zu machen, genügt es folglich nicht, das Handeln der Akteurinnen und Akteure in dieser Organisation zu beeinflussen. Vielmehr muss man die Handelnden dazu bewegen, neu zu denken – und das ist ungleich schwieriger. Wer hier tiefer einsteigen will, dem bietet das Buch »Agile Transformation« [Lieshout 2020] einen guten Leitfaden auf dem Weg zur agilen Organisation.

Nachfolgend einige Handlungsempfehlungen, um das Umdenken zu initiieren:

Die eigene Geschichte erzählen

Zunächst sollte man die Geschichte des eigenen Scrum-Teams erzählen und erlebbar machen. Warum hat das Team den agilen Weg gewählt? Welche Hindernisse mussten überwunden werden? Welche Fehler wurden gemacht? Was hat das Team daraus gelernt? Welche Erfolge konnten gefeiert werden? Wie wurde der Scrum-Prozess unter Berücksichtigung der gegebenen Rahmenbedingungen ausgestaltet? Zur Illustration dieser Fragen können Artefakte verwendet werden: ausgewählte Product Backlog Items, die zeigen, wie in Scrum Anforderungen beschrieben werden, ein Foto vom Taskboard oder ein Burndown Chart, der belegt, dass Scrum ein planvoller Prozess ist. Zitate von Stakeholdern und Kunden geben der Geschichte mehr Authentizität und Bedeutung. Klingt nach Marketing? Genau das ist es. Denn nur wer Gutes tut und darüber redet, der wird andere davon überzeugen können, es ebenfalls zu versuchen. Wo und wie Sie diese Geschichte erzählen, hängt ganz von Ihren Möglichkeiten und Ihrer Kreativität ab: Ein Beitrag zur hausinternen Mitarbeiterzeitung, ein Aushang am Schwarzen Brett, ein Flyer, ein Comic oder

ein Audio- oder Video-Podcast – wählen Sie das Format, das bei den gewünschten Empfängern am besten ankommt.

Einblicke bieten und erklären

Laden Sie Gäste zu den Scrum-Events ein, um Scrum erlebbar zu machen. Eine Stunde »mittendrin« ist viel intensiver als ein Bericht über Scrum. Planen Sie nach dem Event eine halbe bis eine Stunde Zeit für Fragen an das Scrum-Team ein. Hier werden Verständnisfragen beantwortet und Vorurteile aus dem Weg geräumt. Das erhöht die Chance, dass sich die Gäste immer mehr für Scrum und agile Vorgehensweisen interessieren. Empfehlen Sie Literatur (z.B. dieses Buch), Weblogs und andere Internetseiten, auf denen die interessierten Gäste mehr Informationen zum Thema finden. Sie haben das nächste Zwischenziel erreicht, wenn Sie die erste Anfrage nach Unterstützung beim Aufbau eines weiteren agilen Teams bekommen.

Unterstützen und begleiten

Wenn Sie Glück haben, dürfen Sie jetzt als hausinterne Beraterin und als Coach ein anderes Scrum-Team begleiten. Dabei werden Ihre eigenen Erfahrungen mit Scrum eine wertvolle Hilfe sein. Vergessen Sie bitte bei aller Begeisterung nicht, dass Ihr Erfahrungsschatz zum Teil nur in Ihrem bisherigen Kontext eine Bedeutung hat. So wie Sie den optimalen Prozess für Ihr »altes« Scrum-Team iterativ entwickelt haben, so wird sich auch das neue Team, das Sie begleiten dürfen, iterativ entwickeln. Anstatt Patentrezepte zum Besten zu geben, sollten Sie deshalb besser Anekdoten aus Ihrem Team erzählen, die verdeutlichen, wie wertvoll das Prinzip von Transparenz, Überprüfung und Anpassung ist. Versetzen Sie sich immer wieder in die Lage der Teammitglieder und bewerten Sie die eigenen Handlungen aus deren Sicht. Mit dieser Sichtweise werden Sie für das Team ein Partner sein, dessen Unterstützung man schätzt. Ein Modell und viele Impulse für nachhaltige Teamentwicklung finden Sie in [Koschek 2022].

Gemeinsam lernen

Sobald es mehr als ein Scrum-Team in der Organisation gibt, kann es sinnvoll sein, sich rollenspezifisch auszutauschen. Es gibt Fragen, die sich jeder Product Owner oder Scrum Master stellt – warum also nicht gemeinsam nach den Antworten suchen? Zu diesem Zweck kann man regelmäßige und gegebenenfalls rollenspezifische Treffen für Scrum-Praktiker (*Communities of Practice*) einrichten, die der Klärung konkreter Fragen aus dem Scrum-Alltag,

aber auch der Weiterbildung dienen. Auflockernde Spiele für die nächste Retrospektive können dort ebenso vorgestellt werden wie neue Schätzverfahren, man diskutiert aktuelle Themen und Thesen der agilen Community oder den neuesten Scrum Guide. Auch die lokale Meetup-Szene sowie Barcamps und andere (Un-)Konferenzen bieten gute Gelegenheiten zum Lernen und Netzwerken. Egal auf welchem Weg – die Hauptsache ist, dass sich die agilen Praktiker kontinuierlich mit Scrum und anderen agilen Themen beschäftigen, bevor sie mit ihrem Team nur noch auf der Stelle zu treten. Schließlich lebt Scrum vom ständigen Lernen und von Transparenz, Überprüfung und Anpassung. Mit anderen Worten:

Scrum ist kontinuierliche Verbesserung

Auf die Frage, wann eine Scrum-Einführung beendet ist, antworten wir: »Nie!« Da sich die Welt um uns herum ständig verändert und auch wir immerzu lernen und Erfahrungen sammeln, wird sich die Art und Weise, wie wir Scrum in unserer Organisation verwenden, ebenfalls immer wieder ändern müssen. Scrum hat dieses System der kontinuierlichen Verbesserung fest eingebaut – wir müssen es nur nutzen, das heißt:

- kurz innehalten,
- unsere Schlüsselaktivitäten (auch bekannt als »Tagesgeschäft«) beiseitelegen und eine neue Perspektive einnehmen,
- die Verbesserungsmöglichkeiten erkennen,
- die Veränderung (oftmals ein Experiment) einleiten und
- den Erfolg der Veränderung bewerten.

Wem dieses Prinzip in Fleisch und Blut übergegangen ist, wer mühelos Potenzial erkennt und innerhalb der von Scrum gesteckten Grenzen Dinge verändern will und kann, der darf sich wahrlich einen Scrum-Meister nennen.

Was noch?

Wir beschließen dieses Kapitel mit einem Ausblick auf Themen, die in vielen Scrum-Teams eine Rolle spielen, wenngleich sie nicht Bestandteil von Scrum sind. Wir können diese Themen im Kontext dieses Buchs nur anreißen und verweisen für die Vertiefung auf die weiterführende Literatur.

Agile Software Engineering

Im Scrum Guide werden keine Softwareentwicklungspraktiken im Zusammenhang mit Scrum erwähnt. Es ist schließlich Aufgabe des Teams, zu entscheiden, welche Methoden und Praktiken verwendet werden, um das bestmögliche Ergebnis zu erzielen. Einige dieser Praktiken sind mittlerweile sehr etabliert, daher können wir sie guten Gewissens für die erfolgreiche Softwareentwicklung empfehlen. Allgemein kann man sagen, dass sich die Praktiken des eXtreme Programming [Beck 2004] und Scrum gut ergänzen. Darüber hinaus gibt es weitere Konzepte und Praktiken, die besonders gut zur Scrum-Denkweise passen. Einige davon stellen wir kurz vor.

Software Craftsmanship

Bei Software Craftsmanship [Martin 2008] geht es im Wesentlichen darum, eine hohe handwerkliche Qualität der abgelieferten Arbeit sicherzustellen und ständig zu verbessern. Auch hier existiert ein Manifest [Craftsmanship], das sich direkt auf das Agile Manifest bezieht und somit die Nähe zur agilen Softwareentwicklung verdeutlicht.

Hinsichtlich der Qualität wird unterschieden zwischen innerer und äußerer Qualität. Die äußere Qualität ist die nach außen sichtbare. Hier geht es in erster Linie um Akzeptanz und die Erfüllung funktionaler Anforderungen, aber durchaus auch um nicht funktionale Anforderungen wie Performanz, Robustheit, Bedienbarkeit und ähnliche »weiche« Faktoren.

Die innere Qualität hingegen ist diejenige, die Nachhaltigkeit erzeugt. Hier geht es um eine gute, nachhaltige Architektur, ständiges Refactoring, lesbaren Code usw.

Dies mag im ersten Moment dem agilen Prinzip der Einfachheit und des iterativ-inkrementellen Anspruchs widersprechen. Bei näherer Betrachtung fällt jedoch auf, dass iterativ-inkrementelle Entwicklung eben auch häufige Änderung und damit leichte Änderbarkeit bedeutet. Dies wiederum setzt »aufgeräumten« Code und zukunftssichere Architekturen voraus, da sonst Änderungen zum Albtraum werden und im Zweifel nicht mehr stattfinden aus Angst, etwas kaputt zu machen.

Wenn man bedenkt, dass im Laufe des Lebenszyklus einer Software nur 20% der Zeit für die Erstentwicklung, aber 80% für Änderungen und Weiterentwicklungen aufgewendet werden, ist es umso wichtiger, dass die innere Qualität stimmt.

Theoretisches Wissen um die notwendigen Praktiken ist wichtig, aber wichtiger noch ist das regelmäßige Üben dieser Praktiken, bis sie einem in Fleisch und Blut

übergegangen sind. Die Software-Craftsmanship-Bewegung empfiehlt Coding Katas (siehe u. a. *codekata.com*), in denen eine bestimmte Aufgabenstellung auf verschiedene Art und Weise immer wieder geübt wird, sodass die dafür notwendigen Techniken wie z.B. Test-Driven Development (TDD) und Refactoring als selbstverständliche Handgriffe in die tägliche Arbeit eingehen und hier zu Qualität und Nachhaltigkeit beitragen.

Pair Programming

Pair Programming oder auch paarweises Programmieren ist eine Technik aus dem eXtreme Programming [Beck 2004]. Hierbei arbeiten zwei Personen gemeinsam an der Umsetzung einer Anforderung. Normalerweise wird dies mit verteilten Rollen getan. Der eine Entwickler tippt und erklärt dabei, was er tut (der »Driver«), der andere sitzt daneben, versucht, die Gedankengänge des Kollegen zu verstehen, und hilft dabei, fokussiert eine adäquate Lösung zu erarbeiten (der »Observer« oder »Navigator«). Adäquat bedeutet in diesem Fall nicht allumfassend oder möglichst ausgefeilt, denn ein Leitsatz des eXtreme Programming lautet: »Baue die einfachste Lösung, die die Anforderung gerade eben erfüllt.«

Die Rollenverteilung kann auf verschiedene Arten interpretiert werden: Arbeiten zwei gleichermaßen erfahrene Personen gemeinsam an einer Aufgabe, dann wechseln sie sich in regelmäßigen Abständen ab, um sicherzustellen, dass die Konzentration hoch bleibt und dass durch die unterschiedlichen Blickwinkel alle Aspekte der Anforderung abgedeckt sind. Bilden hingegen eine Senior-Entwicklerin und ein Junior-Entwickler das Paar, liegt der Fokus eher auf der Wissensvermittlung auf der einen und dem Code-Review auf der anderen Seite. Arbeiten ein Entwickler und eine Testerin gemeinsam, wird die Testerin (insbesondere beim Test-Driven Development) die Tests schreiben (die zunächst fehlschlagen) und der Entwickler durch Schreiben von Programmcode dafür sorgen, dass die Tests erfüllt werden. Das Ganze passiert in einer schnellen Abfolge und in kleinen Schritten, sodass durchaus alle paar Minuten die Tastatur den Besitzer wechselt.

Nun mag man auf den ersten Blick vermuten, dass sich ein solcher Aufwand nicht lohnt, denn wenn immer zwei Personen an einer Aufgabe arbeiten, hat sich vordergründig die Kapazität des Teams halbiert, die Kosten für die Umsetzung einer Anforderung haben sich aber verdoppelt. Dies ist glücklicherweise nicht richtig. Man stellt sehr schnell fest, dass die absolute Zeit, die an einer Aufgabe verbracht wird, deutlich geringer ist, als würde man allein davor sitzen. Durch die begleitende Diskussion und das regelmäßige Hinterfragen läuft man seltener in die falsche Richtung, entwickelt qualitativ bessere Lösungen, die weniger

fehleranfällig sind, und hat außerdem in aller Regel ein besseres Wir-Gefühl im Team. So hat eine Untersuchung von Alistair Cockburn und Laurie Williams [Cockburn 2000] bereits im Jahr 2000 gezeigt, dass der Mehraufwand ca. 15% beträgt, dieser sich aber durch die eben genannten Vorteile mehr als lohnt.

Continuous Integration

Ähnlich wie bei Software Craftsmanship geht es bei Continuous Integration darum, schwierige Tätigkeiten oft zu üben, sie ständig zu verbessern und sie zu automatisieren, wo immer es sinnvoll ist.

Im Rahmen der Softwareentwicklung wird regelmäßig (bei Scrum potenziell immer dann, wenn ein Product Backlog Item fertiggestellt ist) ein Inkrement an die Kunden ausgeliefert. Bevor das geschieht, müssen die einzelnen Module der Software in einer isolierten Umgebung (dem Integrationssystem) ihr erfolgreiches Zusammenspiel unter Beweis stellen. In dieser Umgebung wird also nicht mehr getestet, ob eine Komponente fehlerfrei funktioniert (das hat schon auf dem Testsystem stattgefunden), sondern ob sie fehlerfrei mit den anderen Komponenten zusammenarbeitet. Erst wenn das bewiesen ist, kann eine Auslieferung an das Produktionssystem in Betracht gezogen werden.

Für diese Integration sind mehrere Schritte nötig. Zunächst muss die neue Softwareversion auf die Integrationsumgebung transportiert werden. Neben dem oft umfangreichen Programmcode kommen meistens noch Konfigurationsdateien, Datenbankänderungen und weitere Ressourcen wie beispielsweise Bilder, sprachabhängige Fehlermeldungen usw. hinzu. Alle diese Artefakte müssen in der richtigen Version zusammengebaut und transportiert werden, damit ein fehlerfreies Zusammenspiel gewährleistet ist. Dieser Schritt ist alles andere als trivial und will daher oft geübt sein, damit alles möglichst fehlerfrei und in kurzer Zeit erledigt ist. Der Build-Prozess sollte weitestgehend automatisiert sein, da es sich in der Regel um immer wiederkehrende Tätigkeiten handelt, die gut automatisierbar sind und bei manueller Durchführung eine vermeidbare Fehlerquelle darstellen.

Der mechanische Verteilungsakt ist aber nur ein Aspekt der Continuous Integration. Der andere, wahrscheinlich noch schwierigere, aber auch wichtigere Teil ist das integrative Testen aller Komponenten. Hierzu werden von den Entwicklern sogenannte Integrationstests implementiert, die das Zusammenspiel der einzelnen Komponenten untereinander überprüfen. End-to-End-Tests sind auf der höchsten Ebene über den Integrationstests angesiedelt. Sie prüfen beispielsweise, ob die in der Benutzeroberfläche getätigten Eingaben fehlerfrei und mit den richtigen Ergebnissen durch die Middleware in die Backend-Systeme gelangen und dort in der gewünschten Form für etwaige

nachgelagerte Systeme wie Reporting, Zahlprozesse oder Archivierung zur Verfügung stehen. Solche Tests sind nur dann wirklich sinnvoll, wenn sie vollständig automatisiert ablaufen. Und genau hierin besteht der Nutzen der Continuous Integration: Durch sofortiges automatisiertes Bauen, Transportieren und Testen der beteiligten Komponenten hat man nach jeder Änderung eines Bausteins ein sehr schnelles Feedback darüber, ob die Änderung die neue Funktionalität fehlerfrei bereitstellt oder ob durch diese Änderung andere Programmteile derart in Mitleidenschaft gezogen wurden, dass sie nun fehlerhaft sind.

Dieses sofortige Feedback durch automatisierte Deployment- und Testprozesse trägt erheblich zur Softwarequalität bei, da Folgefehler früh erkannt werden und somit schnell behoben werden können.

Continuous Delivery

Wenn das Paketieren, Integrieren, Testen und Ausliefern eines Softwaresystems vollständig automatisiert ist, können die Developer den nächsten Schritt gehen: Bei der Continuous Delivery wird die neue Softwareversion regelmäßig oder beim Erreichen bestimmter Qualitätskriterien automatisiert ausgeliefert – mindestens auf eine produktionsähnliche Umgebung, oft auch auf die Produktiv-Infrastruktur.

Mehrere Teams

Im Scrum Guide wird nur von »dem Scrum-Team« gesprochen. Ein Scrum-Team besteht üblicherweise aus zehn oder weniger Personen (inklusive Scrum Master und Product Owner), was unweigerlich zu der Frage führt, ob Scrum auch für die Entwicklung größerer Produkte geeignet ist.

In der Tat sind alle Scrum-Prinzipien der Selbststeuerung auf den ersten Blick nur auf ein einzelnes Team ausgerichtet, und auch der Scrum Guide geht von einem Team aus. Und doch können auch Projekte und Produktentwicklungen, an denen mehrere Teams beteiligt sind, von Scrum profitieren. Für ein solches »skaliertes Scrum« haben sich in den vergangenen Jahren verschiedene Rahmenwerke etabliert, von denen wir einige kurz vorstellen wollen, bevor wir auf die Schneidung von Teams und die Beziehungen zwischen den Teams eingehen.

Scrum skalieren

Um Scrum in mehreren auf ein gemeinsames Ziel hinarbeitenden Teams einsetzen zu können, genügt es nicht, die einzelnen Teams parallel zueinander arbeiten zu lassen und das gleiche Verständnis von *Done* zu haben. Sie müssen sich synchronisieren, um auf dasselbe Ziel ausgerichtet zu bleiben und gegenseitige Zuarbeiten so zu organisieren, dass der Entwicklungsfluss nicht ins Stocken gerät. Auch die Erhebung der fachlichen Anforderungen und deren Verteilung auf die Teams muss so organisiert werden, dass der Nutzen der entstehenden Inkremente möglichst groß ist. Um das zu erreichen, ist es notwendig, einen etwas größeren Rahmen um den Scrum-Prozess herum zu bauen. Im Laufe der Zeit sind aus der praktischen Erfahrung größerer agiler Projekte und Produktentwicklungen heraus verschiedene Rahmenwerke entstanden. Die drei prominentesten Vertreter mit der größten Verbreitung sind LeSS, SAFe und Nexus.

Large-Scale Scrum (LeSS)

In [Larman 2017] ist *Large-Scale Scrum* (LeSS) wie folgt definiert: »LeSS ist Scrum, angewandt auf viele Teams, die gemeinsam an einem Produkt arbeiten.« Es besteht aus zwei Frameworks (LeSS und LeSS Huge), mit denen Scrum sehr schlank skaliert werden kann. Während LeSS für zwei bis acht Teams gedacht ist, wird LeSS Huge in Projekten und Produktentwicklungen mit mehr als acht Teams eingesetzt.

Die Entwicklungsteams in LeSS sind sogenannte Feature-Teams: Sie sind interdisziplinär und multifunktional besetzt und arbeiten komponentenübergreifend an abgeschlossenen und auslieferbaren Features. In beiden Frameworks gibt es einen Product Owner und ein gemeinsames Product Backlog für alle Teams, die Teams arbeiten in einem gemeinsamen Sprint, und sie liefern spätestens am Ende des Sprint ein Inkrement für das Gesamtprodukt. In LeSS gibt es genau einen Product Owner, dem wie in Scrum das Produkt gehört und der das Product Backlog verwaltet. In LeSS Huge werden die fachlichen Anforderungen in Bereiche untergliedert, die sogenannten Requirement Areas. Es gibt immer noch ein gemeinsames Product Backlog, das aber nach den Requirement Areas gefiltert werden kann. Area-Feature-Teams arbeiten gemeinsam mit einem Area Product Owner in einer dieser Requirement Areas. Zusätzlich gibt es in LeSS Huge einen Product Owner, der für die Gesamtvision des Produkts verantwortlich ist.

LeSS ist ein agiles Framework im besten Sinn. Es basiert konsequent auf Scrum und ist bewusst unvollständig, um Raum für das Lernen zu geben und

Transparenz, Überprüfung und Anpassung zu fördern. Dabei legt es großen Wert auf Transparenz. Erfahrene Scrum-Teams werden die Mechanik von LeSS sehr schnell verstehen und anwenden können. Dennoch ist das synchronisierte Arbeiten mehrerer Teams an einem gemeinsamen Produkt eine herausfordernde Aufgabe, die erlernt werden will – und das kostet Zeit.

Scaled Agile Framework® (SAFe®)

Das *Scaled Agile Framework* (SAFe) [Mathis 2017] wurde entwickelt, um Lean-Prinzipien und agile Methoden unternehmensweit einzusetzen – auch in großen Organisationen. Um das zu erreichen, skaliert SAFe auf drei Ebenen. Auf der Teamebene arbeiten Teams (möglichst als Feature-Teams) mit Scrum, Kanban oder eXtreme Programming (XP). Das interdisziplinäre *Agile Team* besteht aus zehn oder weniger Personen und nutzt die Rollen *Scrum Master/Team Coach* und *Product Owner*. Auf der Programmebene bilden mehrere dieser Teams (50 bis 125 Personen) einen *Agile Release Train* (ART). In Programminkrementen, die länger sind als ein einzelner Sprint, wird schrittweise ein Produkt oder ein anderes werthaltiges Ergebnis entwickelt. Auf der dritten Ebene, der Portfolioebene, werden mehrere Vorhaben analysiert, priorisiert und an die Agile Release Trains delegiert. Auf dieser Ebene kommt Kanban zu Einsatz, um den Durchsatz der Vorhaben zu optimieren.

SAFe spannt den Bogen von der Produktentwicklung im einzelnen Team bis zum Programmmanagement. Für dieses komplexe Aufgabenspektrum definiert SAFe eine Reihe von Rollen, Artefakten und Abhängigkeiten, die alle in einem Gesamtbild visualisiert sind. Unsere Erfahrung ist, dass dieses Bild die Existenz einer Blaupause für skalierte agile Vorhaben suggeriert – die es naturgemäß nicht gibt und nicht geben kann. Das sagt auch Dean Leffingwell, der geistige Vater von SAFe. Das Gesamtbild ist Fluch und Segen zugleich. Einerseits schafft es Klarheit, andererseits signalisiert es Vollkommenheit. Wenn das Modell vollkommen ist, muss man es nur noch anwenden – so der oft fälschlicherweise gezogene Schluss. Eine mögliche Konsequenz dieses Trugschlusses ist, dass SAFe nicht wie gewünscht funktioniert – was dann oft der Methode angelastet wird.

Nexus™ Framework

Kurt Bittner et. al. bezeichnen Nexus in [Bittner 2018] als »Exoskelett«, das in großen, drei bis neun Scrum-Teams umfassenden Projekten oder Produktentwicklungen die Abhängigkeiten zwischen den Teams vereinfacht und

organisiert und deren Zusammenarbeit transparent gestaltet. Dabei ist Nexus, wie auch LeSS, im Kern immer noch »echtes« Scrum.

Nexus geht von einem einzigen Product Backlog aus, auf dem alle Teams gemeinsam arbeiten. Hinzu kommt das Nexus Sprint Backlog – der Plan für den Sprint, in dem die Aufgaben aller Teams und die Abhängigkeiten zwischen ihnen transparent dargestellt sind. Das im Abschnitt »Refinement« auf Seite 87 beschriebene Refinement hat in Nexus einen festen Platz gefunden. Sprint Planning, Daily Scrum und Sprint-Retrospektive finden in Nexus sowohl teamintern als auch zusätzlich teamübergreifend statt, um die Arbeit bestmöglich zwischen den Teams aufzuteilen und den Wissensaustausch zu fördern. Ein Sprint Review auf Teamebene gibt es nicht mehr, dafür aber das projektweite Nexus Sprint Review.

In Nexus ist – genau wie in Scrum – ein einziger Product Owner vorgesehen. Gemeinsam mit einem Scrum Master und Mitgliedern der Scrum-Teams bildet er das Nexus-Integrationsteam (NIT). Das Aufgabenspektrum dieses Gremiums ähnelt dem eines Scrum Master: Es schult und begleitet die Teams im Umgang mit Nexus und hilft ihnen dabei, Probleme zu identifizieren und Lösungen zu finden. Das NIT unterstützt die Teams, damit diese mit jedem Sprint ein integriertes Inkrement erzeugen.

Nexus wurde so entworfen, dass ein Projekt klein starten und dann kontinuierlich wachsen kann, ohne die Methodik wechseln zu müssen. Mit dieser Fähigkeit adressiert Nexus all jene, die nicht wissen, wie ihre Produktentwicklung mittel- bis langfristig verlaufen wird – was bei komplexen Produkten eher Regel als Ausnahme sein dürfte. Nexus wird übrigens von Scrum.org entwickelt – der Organisation, die Scrum-Mitbegründer Ken Schwaber 2009 ins Leben gerufen hat.

Den Product Owner skalieren?

Bei der Beschreibung der Skalierungs-Frameworks fällt auf, dass es in LeSS und Nexus nur einen einzigen Product Owner je Produkt gibt. In SAFe hat hingegen jedes »Agile Team« einen eigenen Product Owner. Um herauszufinden, welcher Ansatz eher im Scrum-Sinne ist, lohnt der Blick in den Scrum Guide. Dort steht:

- »Der Product Owner ist ergebnisverantwortlich für die Maximierung des Wertes des Produkts.«
- »Der Product Owner ist eine Person, kein Gremium.«
- »Damit der Product Owner Erfolg haben kann, muss die gesamte Organisation seine Entscheidungen respektieren. Diese Entscheidungen

> sind im Inhalt und in der Reihenfolge des Product Backlog sowie durch das überprüfbare Inkrement beim Sprint Review sichtbar.«

Wir leiten daraus ab, dass es je Produkt genau einen Product Owner geben muss. Das ist der Ansatz, den auch LeSS und Nexus verfolgen. Wie aber soll ein einziger Product Owner seiner Verantwortung für ein umfangreiches und facettenreiches Produkt gerecht werden? Auch auf diese Frage hat der Scrum Guide eine gute Antwort parat: Der Product Owner kann einige seiner Aufgaben, beispielsweise das Ausformulieren der Product-Backlog-Einträge, durch die Developer erledigen lassen. Der Schwerpunkt seiner Rolle verlagert sich weg vom Redakteur der User Stories hin zum Visionär, Ideengeber und fachlichen Netzwerker. Nichtsdestotrotz bleibt der Product Owner ergebnisverantwortlich.

Ein Scrum Master je Scrum-Team?

In der Praxis begegnet uns beides: Scrum Master, die in genau einem Scrum-Team arbeiten, und solche, die zeitgleich für mehrere Teams verantwortlich sind. Ein Zitat, das Ken Schwaber zugeschrieben wird, lautet: »Ein mittelmäßiger Scrum Master betreut zwei bis drei Teams, ein guter Scrum Master nur genau eins!« Stimmt das?

Eine pauschale Antwort auf diese Frage gibt es nicht. Teams sind komplexe soziale Gebilde, und kein Team gleicht dem anderen – wie auch, schließlich arbeiten dort Individuen, von deren Verschiedenheit das Team profitiert. Der Entwicklungsstand von Scrum-Teams spielt eine wichtige Rolle, wenn man den Betreuungsbedarf durch einen Scrum Master ermitteln möchte. Teams, die erste gemeinsame Schritte mit Scrum gehen, benötigen sehr viel Unterstützung, um zielgerichtet mit Scrum zu arbeiten und zu lernen. Am anderen Ende der Skala stehen Scrum-Teams, die seit mehreren Jahren erfolgreich und stabil zusammenarbeiten. Diese Teams benötigen eine andere Art von Betreuung – weniger Mechanik, dafür mehr Impulse und Mentoring, um nicht im ewig gleichen Rhythmus der Sprints zu versauern. Wir haben übrigens die Erfahrung gemacht, dass man auch in solch eingespielten Scrum-Teams nicht davon ausgehen darf, dass alles von allein rund läuft. Oft treten beispielsweise Konflikte erst zutage, wenn Menschen über lange Zeit zusammenarbeiten.

Obwohl das alles für eine Eins-zu-eins-Beziehung zwischen Scrum-Team und Scrum Master spricht, kann es durchaus funktionieren, wenn ein Scrum Master zwei oder gar drei Teams zeitgleich betreut. Er wird sich dann aber voll und ganz auf die Arbeit in den Teams fokussieren müssen und damit zwangsläufig eine Aufgabe vernachlässigen, die ihm der Scrum Guide ins Rollenprofil geschrieben hat: der Dienst an der Organisation. In größeren Projekten und Organisationen

gibt es deshalb zusätzlich die Rolle des Agile Coach (siehe Abschnitt »Agile Coach« auf Seite 73). Das Wirken in die Organisation hinein ist der Schwerpunkt seiner Arbeit. Um dabei die Realität in den Scrum-Teams nicht aus den Augen zu verlieren, muss der Agile Coach in den agilen Teams gut vernetzt sein. Auch die Scrum Master, die in einem größeren agilen Kontext arbeiten, profitieren von einem guten Netzwerk untereinander. Indem sie sich über die Herausforderungen in ihren Teams austauschen (unter Wahrung der Vegas-Regel, versteht sich), können sie strukturelle und teamübergreifende Probleme erkennen und beheben.

Schneiden von Teams

Beim Schneiden von Teams für die agile Produktentwicklung, also bei der personellen Zusammensetzung der Scrum-Teams, gibt es im Wesentlichen vier Ansätze:

- Schneiden nach Organisationseinheiten
- Schneiden nach technischen Komponenten
- Schneiden nach fachlichen Themen
- Schneiden fachlich und technisch gleichartiger Teams (Feature-Teams)

Das Schneiden nach Organisationseinheiten ist unserer Meinung nach die schlechteste aller Möglichkeiten. Die Abteilungsstruktur eines Unternehmens hat oft nichts mit dem zu entwickelnden Produkt zu tun. So gibt es möglicherweise eine Marketingabteilung, eine Systemadministration, eine Architekturabteilung und eine Entwicklungsabteilung, um nur einige Beispiele zu nennen. Wir erinnern uns an die Forderung nach interdisziplinären Teams: Hier geht es darum, alle Fähigkeiten, die zur Entwicklung der Produkteigenschaften nötig sind, in einem Team zu vereinigen, um möglichst wenig Zeit zu verlieren mit dem Warten auf Zulieferungen von anderen Teams oder Abteilungen.

Das Schneiden nach technischen Komponenten ist ein durchaus üblicher, aber nicht immer zielführender Weg. Das Argument der fehlenden Interdisziplinarität greift auch hier. Wenn ein SAP-, ein Middleware- und ein Frontend-Team gemeinsam ein Produkt entwickeln, kann jedes Team nur einen für sich genommen nicht funktionalen Bestandteil liefern. Um Produktfunktionalität zu entwickeln, die potenziell auslieferbar ist, bedarf es üblicherweise der Zuarbeit aller Teams. Warum sollte man diese Zusammenarbeit durch das unglückliche Ziehen von Teamgrenzen unnötig erschweren?

Ein aus unserer Sicht erfolgreiches Schneiden von Teams geschieht entlang der fachlichen Komponenten. Entwickelt man beispielsweise ein Internetportal, das Reisen anbietet, könnte sich ein Team um Pauschalreisen kümmern, ein weiteres um Hotelbuchungen und ein drittes alle Themen rund um die Autovermietung behandeln. Der Abstimmungsaufwand zwischen den Teams ist vergleichsweise gering. Jedes Team arbeitet in seiner eigenen Geschwindigkeit und schafft ein unabhängiges, für Kunden wertvolles Teilprodukt. Hier wird alles, was zur jeweiligen Funktionalität gehört, von genau einem Team produziert und geliefert: von Datenbankstrukturen über Geschäftsobjekte und Suchalgorithmen in angeschlossenen Drittportalen bis hin zur grafischen Benutzeroberfläche. Keinerlei Zuarbeit von anderen Teams ist nötig. Als Nebeneffekt steigt die Zufriedenheit der Teammitglieder; sie sind keine Fließbandarbeiter in einem Gesamtprozess, sondern bauen eigenständig ein unabhängiges Teilprodukt. Der Product Owner muss darauf achten, dass die Teilprodukte gut aufeinander abgestimmt sind. Das gelingt am besten, wenn er sich die Frage stellt, wie er den Nutzen für den Endkunden maximieren kann.

Wenn die fachliche Domäne überschaubar groß ist, kann es gelingen, die Verantwortung für das gesamte Produkt in die Hände aller Teams zu legen. Der große Vorteil solcher Feature-Teams ist die gleichmäßige Auslastung. Wenn, um beim obigen Beispiel zu bleiben, für die Autovermietungskomponente viele neue fachliche Anforderungen vorliegen, für die Pauschalreisen hingegen kaum etwas zu tun ist, wird ein Team überfordert sein, wohingegen ein anderes Team nicht ausgelastet ist. Bei Feature-Teams werden die Anforderungen immer gleichmäßig auf alle Teams verteilt, weil jedes Team grundsätzlich jede fachliche Anforderung umsetzen kann.

Verteilt arbeitende Teams

Sind Teams auf mehrere Standorte verteilt, ist es oft schwierig, ein Zusammengehörigkeitsgefühl zu entwickeln. Das spontane Gespräch in der Kaffeeküche, die engagiert geführte Diskussion mit der Möglichkeit, sich anschließend in die Augen zu schauen und die fachliche Meinungsverschiedenheit von der persönlichen Beziehung zu trennen – das fehlt, wenn man in verschiedenen Städten, Ländern oder gar Zeitzonen zu Hause ist. Dann sind Konflikte vorprogrammiert: Das Team in München schimpft auf das Team in Hamburg, weil Zulieferungen nicht rechtzeitig eingetroffen sind. Kulturelle Unterschiede zwischen den verschiedenen Standorten können diesen Effekt noch verstärken. Die Distanz entsteht übrigens nicht nur auf räumlicher, sondern auch auf organisatorischer Ebene: Fachabteilungen und IT, Stammhaus und Filialen, Firmenübernahmen – Gründe für mangelnde Kooperation gibt es

viele. Erst wenn solche Konflikte gelöst sind, wird die interdisziplinäre Zusammensetzung des Teams erneut zu einem Vorteil, und die Diversität ist wieder eine Bereicherung und kein Störfaktor.

Verstreut arbeitende Teams

Verstreute Teams sind ein Sonderfall verteilt arbeitender Teams. Hier ist das Team selbst über mehrere Standorte verteilt. Grundsätzlich gelten die gleichen Aussagen wie oben, allerdings gilt es noch ein paar zusätzliche Aspekte zu berücksichtigen.

Durch die räumliche Trennung nicht nur zwischen den Teams, sondern sogar innerhalb eines Teams können Konflikte nicht länger ignoriert werden, sondern behindern täglich die Zusammenarbeit. Dies kann zur Folge haben, dass die Teamleistung dramatisch sinkt.

Das Verstreuen der Teammitglieder über mehrere Standorte ist nicht ideal, aber gerade in großen Konzernen manchmal nicht zu verhindern. Durch einen Firmenzukauf sitzen beispielsweise die Fachleute, die für ein neues Produkt zusammenarbeiten müssen, an verschiedenen Standorten. Solange diese aufeinander angewiesen sind, ist meist alles in Ordnung. Kann jedoch ein Teilteam so eigenständig arbeiten, dass es den Rest des Teams nicht braucht, wird es schwierig. Die fehlende innere Abhängigkeit wird ein gemeinsames Teamgefühl und damit auch eine Teamleistung verhindern.

Um dem entgegenzuwirken, sollten die Teams ihre Aufgaben so aufteilen, dass Teammitglieder aus verschiedenen Standorten gemeinsam an einem Product Backlog Item arbeiten. Innerhalb dieser Zusammenarbeit hat das Team dann die Möglichkeit, kulturelle, regionale oder sonstige Konflikte zu erkennen, sich durch die Zusammenarbeit den gegenseitigen Respekt zu sichern und so trotz räumlicher Trennung eine echtes Team zu werden.

Ein Wir-Gefühl entwickeln

Ganz sicher ist es bei verteilten Teams aufwendiger, ein Wir-Gefühl zu erzeugen. Es wird nötig sein, in regelmäßigen Abständen gemeinsame Veranstaltungen zu organisieren, um sich besser kennenzulernen. Denn nur wenn man sich von Angesicht zu Angesicht kennt, kann man eine gute persönliche Beziehung aufbauen, die für erfolgreiche Teamarbeit zwingend erforderlich ist. Das ist bei verteilten und verstreuten Teams immer mit Reisekosten verbunden. Die Erfahrung zeigt aber, dass sich diese Investition lohnt, wenn man es dadurch schafft, die Teams so zusammenzuschweißen, dass die räumliche Trennung

keine negativen Auswirkungen auf die Teamarbeit hat. Diese persönlichen Treffen müssen regelmäßig stattfinden, da ansonsten die Gefahr groß ist, allzu leicht wieder in die alten Verhaltensmuster zurückzufallen. Besonders wichtig für eine reibungslose Zusammenarbeit über die Standorte hinweg ist eine stabile und schnelle Internetverbindung.

Die ganz praktischen Probleme der räumlichen Trennung, wie z.B. Daily Scrums in unterschiedlichen Zeitzonen oder Besprechungen via Audio- und Videotechnik, klammern wir hier bewusst aus und verweisen für eine detaillierte Behandlung dieses facettenreichen Themas gern auf [Eckstein 2009].

Remote Scrum

Die Jahre 2020 bis 2022 waren geprägt von COVID-19. Viele Organisationen haben ihre Mitarbeitenden gebeten, im Homeoffice zu bleiben. In der Folge hat sich eine Arbeitsweise durchgesetzt, die oft als das »neue Normal« bezeichnet wird. Scrum-Teams, die an einem Ort möglichst gemeinsam in einem Raum arbeiten – was lange als ideale Form der Zusammenarbeit galt –, gibt es vielfach nicht mehr. Das Homeoffice hat Teams in eine verstreute Arbeitsweise gezwungen.

Tooling

Ein Punkt, der sich in den letzten Jahren immer weiter verbessert hat, ist die Geschwindigkeit der Internetverbindungen. 100 MBit/s ist eher die Regel als die Ausnahme, und auch höhere Geschwindigkeiten wie 1 GBit/s sind keine Seltenheit mehr. Damit fallen technische Beschränkungen für Remote-Server-Zugänge und Videokonferenzen weitgehend weg.

Neben der Geschwindigkeit der Internetverbindung ist sicherlich die Nutzung von Kollaborationstools ein entscheidender Faktor. Glücklicherweise sind die heute üblichen Tools wie beispielsweise Slack, Zoom oder Microsoft Teams inzwischen so performant und gut bedienbar, dass man remote annähernd so gut zusammenarbeiten kann, als wäre man in einem Raum.

Arbeiten im Homeoffice

Das Arbeiten im Homeoffice hat Vor- und Nachteile. Auf der Haben-Seite steht sicherlich die oft gute Ausstattung des Arbeitsplatzes. Gerade gegenüber Offices, die nach dem Shared-Desk-Prinzip arbeiten, bei dem Mitarbeitende jeden Tag ihren Arbeitsplatz neu herrichten müssen, kann das die Produktivität und das

Wohlbefinden deutlich steigern. Dies gilt allerdings nur für Menschen, die zu Hause einen abgetrennten Arbeitsbereich haben. Wer an seinem Esstisch in der Küche oder im Wohnzimmer arbeiten muss, hat häufig Schwierigkeiten, Arbeit und Privatleben auseinanderzuhalten, was zu beträchtlichen Problemen führen kann.

Zu Hause zu sein, heißt nicht nur, sich wohlzufühlen, sondern auch, die Alltagspflichten ständig um sich zu haben. Die Waschmaschine, der Geschirrspüler oder auch die Kinder nach der Schule oder in den Ferien können Konzentration und Fokus auf die Arbeit beeinträchtigen. Deshalb sind Selbstdisziplin und Selbstmanagement erforderlich, um in Summe trotzdem effektiv zu sein. Demgegenüber bietet das Arbeiten im Homeoffice auch einige Vorteile. Den Arbeitsweg zweimal am Tag nicht zu haben, ist oft eine derart große Zeitersparnis, dass größere Pausen, z.B. für Hausarbeit oder Kinderbetreuung, meist mehr als kompensiert werden.

Wie auch immer man sich den Tag im Homeoffice einteilt: Für ein gutes Miteinander im Scrum-Team sollten gemeinsame Pflichtzeiten etabliert werden, zu denen die Teammitglieder im Sinne einer guten Zusammenarbeit erreichbar sind.

Darüber hinaus bietet es sich an, feste Kommunikationskanäle und damit verbunden Erwartungen an Antwortzeiten miteinander zu vereinbaren. Das Scrum-Team kann sich beispielsweise auf die Nutzung von Signal-App, Slack und E-Mail einigen, wobei eine Signal-Nachricht von hoher Dringlichkeit ist, sodass auf sie sofort reagiert werden sollte. Eine Slack-Nachricht hingegen kann im Laufe der nächsten Stunden beantwortet werden, und auf eine E-Mail wird nicht vor dem nächsten Tag eine Antwort erwartet. Das sorgt für einen Zugewinn an Fokus, denn nun brauchen die Teammitglieder nicht mehr alle Kanäle gleichermaßen im Auge zu behalten, sondern können den größten Teil von ihnen entweder zu festen Zeiten (E-Mail zum Beispiel morgens oder abends) oder gelegentlich auf neue Nachrichten prüfen und gegebenenfalls reagieren.

Zusammenarbeit und Zusammenhalt

Das große Credo in Scrum lautet »zusammenarbeiten« anstelle von »zusammen arbeiten«. Auch wenn das Tooling inzwischen so weit ist, dass es kaum noch technische Hürden gibt, ist es nicht selbstverständlich, dass dies bei Remote-Arbeit auch wirklich geschieht. Schließlich ist es wesentlich einfacher, über den Schreibtisch hinweg den Kontakt zu suchen, als einen Call im Kollaborationstool der Wahl zu initiieren. Hier bedarf es der Teamdisziplin, um auch außerhalb der Events und täglichen Meetings den engen Kontakt zu halten.

Einige Scrum-Teams nutzen dafür den ganzen Tag offene Kanäle oder (virtuelle) Räume. Jeder, der mag, kann dem Kanal beitreten. Es gibt keine Verpflichtung, Aufgaben gemeinsam zu bearbeiten, man kann auch einfach nur schweigend für sich die eigene Arbeit erledigen. Doch es besteht die Chance, jederzeit etwas anzusprechen, zu kommentieren oder zu fragen. Dies kommt einem gemeinsamen Teamraum schon recht nahe.

Eine Zusammenarbeit funktioniert remote teilweise sogar besser als vor Ort. So haben beim Pair Programming beide Developer (oder Tester) eine eigene Tastatur und können je nach Tooling zur selben Zeit am gleichen Code arbeiten. Möchte man sich gegenseitig etwas zeigen, bieten moderne Tools wie z.B. Slack die Möglichkeit, dass mehrere Personen gleichzeitig ihren Bildschirm teilen, sodass mehr Informationen parallel sichtbar sind. Insbesondere für Retrospektiven gibt es elektronische Whiteboards (z.B. Conceptboard, Mural und Miro) mit einer Vielzahl an Templates für die gängigen Formate. Selbst anonymes Dot-Voting ist vielfach möglich. Gepaart mit der Möglichkeit von Break-out-Räumen für die Arbeit in Kleingruppen ist das Tooling fast besser als bei Retrospektiven in Präsenz am selben Ort.

Was unserer Erfahrung nach viel zu kurz kommt, ist die informelle Seite der Arbeit im Team. Remote-Arbeit und Remote-Werkzeuge unterstützen vor allem strukturiertes, effizientes Arbeiten. Die Gespräche vor und nach einem Präsenzmeeting und die berühmten Gespräche an der Kaffeemaschine und auf dem Flur finden nicht mehr von selbst statt. Darunter leidet auch die Qualität des Coachings durch den Scrum Master oder andere Personen. Dem Coach ist die Möglichkeit weitgehend verwehrt, die Stimmung im Raum zu spüren und entsprechend darauf zu reagieren. Gerade dieses Hineinhorchen in den Raum, das Mithören von Gesprächen, das Lesen von Körperhaltungen oder auch das Erfahren der Stille liefern Indizien für die Qualität der Zusammenarbeit im Scrum-Team. In Remote-Situationen fehlt diese Sensorik. Oft erweist es sich als Hemmschwelle, einfach so Kolleginnen und Kollegen anzurufen, ohne zu wissen, ob es gerade passt.

Deshalb müssen andere Mechanismen gefunden werden, um diese Kontaktfläche zu schaffen. Für das, was in Präsenz an informeller Begegnung und zwanglosem Austausch einfach so passiert, müssen in Remote-Umgebungen bewusst (Zeit-)Räume geschaffen und eingeplant werden. Vielleicht ein Online-Stammtischmeeting ins Leben rufen, in dem zwanglos über alles außerhalb der Arbeit geplaudert werden kann? Oder warum sich nicht einmal gegenseitig remote zu Hause besuchen und sich per Videorundgang durchs Haus führen lassen? Feste Bürotage oder auch abendliche Freizeitveranstaltungen können eine weitere Möglichkeit sein, den Zusammenhalt zu festigen.

Genau dieser Zusammenhalt ist enorm wichtig. Das Scrum-Team ist gemeinschaftlich für die Produktentwicklung ergebnisverantwortlich, und das funktioniert nur dann exzellent, wenn man sich aufeinander verlassen kann, weil psychologische Sicherheit bzw. großes Vertrauen innerhalb des Teams besteht. Hier muss jedes Team seinen eigenen Weg finden, dieses Vertrauen herzustellen und aufrechtzuerhalten. Das Buch »Die Arbeit hat das Gebäude verlassen« [Kramer 2021] bietet viele Erfahrungen und Anregungen aus der Praxis der vergangenen Jahre dazu, wie Teambuilding und Teamkultur auch unter Remote-Bedingungen funktionieren können.