Vorwort

Ein Team, *ein* Softwareprodukt – das ist der Kontext, in dem agile Softwareentwicklung entstanden ist. So weit die Folklore und so weit auch die Beschreibungen von z. B. Scrum oder Extreme Programming (XP).

Und so habe ich es auch lange dargestellt: »Lasst uns erst mal verstehen, wie das Ganze mit einem Team funktioniert. Danach sehen wir uns an, wie man es auf mehrere Teams skalieren kann.«

Mit der Zeit wurde es aber immer unüblicher, dass ein Team allein ein Produkt entwickelte. Die Frage, ob man immer mehr Leute brauchte, weil es so kompliziert wurde, oder ob es so kompliziert wurde, weil man so viele Leute hatte, wird sich vermutlich nie beantworten lassen.

Festzuhalten ist, dass die Menschen in den Unternehmen immer weniger mit der Unterscheidung »nicht skaliert« und »skaliert« etwas anfangen konnten. Sie hatten nicht erlebt, dass etwas Sinnvolles von nur einem Team entwickelt werden konnte.

Da passt es gut ins Bild, dass vor allem in Konzernen das Scaled Agile Framework® (SAFe®) gerne verwendet wird. Schließlich liefert es Antworten auf alle möglichen Fragen rund um die Organisation mehrerer Teams. Doch sind diese Antworten immer nützlich?

Ich nehme jedenfalls wahr, dass sich viele »agile« Implementierungen eher wie die »Behörde für Agilität« (Danke an Henning Wolf für dieses geflügelte Wort) anfühlen. Ist das eine unausweichliche Konsequenz, die quasi der Trägheit der Masse geschuldet ist?

Dazu lohnt sich ein Blick zurück in die Anfänge der Agilität. So beschränkt auf das Ein-Team-Szenario, wie es erscheint, war Agilität selbst in den Anfängen nicht. Feature Driven Development und Crystal haben bereits sehr früh die Frage der Zusammenarbeit vieler Menschen adressiert. Und auch viele der frühen Scrum- und XP-Entwicklungen fanden mit mehreren Teams statt. Die dort verwendeten Koordinationsansätze sind nur nicht in die

Beschreibungen von Scrum und XP eingeflossen. Vermutlich erschien das Feld der Möglichkeiten zu groß, als dass man bestimmte Techniken vorschreiben wollte. Aus demselben Grund finden sich in Scrum auch keine Angaben über zu verwendende Entwicklungspraktiken wie z. B. testgetriebene Entwicklung. Stattdessen sollte jedes Team selbst herausfinden, was am besten für den jeweiligen Kontext geeignet ist.

Dieser Ansatz hat nach meiner Erfahrung gut funktioniert. Wir haben bereits mit einer zweistelligen Anzahl von Teams an einem Produkt gearbeitet, bevor es Skalierungsframeworks wie LeSS oder SAFe® gab. Mit Inspect & Adapt haben wir verschiedene Techniken zur Koordination ausprobiert und für den Kontext passende Ansätze gefunden und kontinuierlich weiterentwickelt. Vor diesem Hintergrund möchte ich einen Beitrag dazu leisten, dass wir wieder zu dieser Haltung zurückfinden, statt Blaupausen zu implementieren und in Verwaltungstätigkeiten unterzugehen. In den letzten Jahren gab es eine Reihe spannender Erkenntnisse und Ansätze, die dabei – insbesondere im Multiteam-Szenario – helfen können.

Ich hatte angefangen, diese Ansätze rein sachlich zu beschreiben und hoffte, daraus ein 15-seitiges PDF machen zu können. Damit bin ich krachend gescheitert. Zum einen reichen 15 Seiten vorne und hinten nicht aus. Zum anderen sind einige der Ideen noch nicht so häufig verwendet worden, dass sie sich genau beschreiben ließen. Außerdem habe ich bemerkt, dass eine genaue Beschreibung der Techniken dazu einladen könnte, diese als Blaupause zu implementieren. Ich will aber keine neuen Frameworks in die Welt setzen, sondern das Nachdenken über die Zusammenarbeit von Teams verändern.

Daher habe ich meine Strategie geändert und entschieden, die Inhalte als Geschichte zu verpacken. Sie ist zu lang für eine Kurzgeschichte und zu kurz für einen Roman. Die Geschichte ist in vier Episoden unterteilt. In der ersten Episode lernen wir den Kontext und wichtige Akteure kennen. Die zweite Episode fokussiert auf das Abhängigkeitsmanagement und wie dieses effizienter gestaltet werden kann. Die dritte Episode thematisiert, wie man Abhängigkeiten reduzieren kann, indem man über gänzlich andere Teamstrukturen nachdenkt. Die vierte Episode ist der Epilog. Hier findet die Geschichte ihren vorläufigen Abschluss.

In der Geschichte kommen Begriffe aus dem Scrum-Framework vor, obwohl die Geschichte und die dargestellten Konzepte nicht Scrum-spezifisch sind. Einige der Konzepte passen tatsächlich gar nicht ins Scrum-Framework. Ich habe trotzdem Begriffe wie Product Owner, Product Backlog, Sprint und

Scrum of Scrums verwendet, weil sie so bekannt und üblich sind. Allgemeinere Begriffe zu verwenden, fühlte sich in der Geschichte unnatürlich und umständlich an.

In der Geschichte kommen einige wenige Fachtermini vor, die dort nicht erklärt werden (z. B. Domain-Driven Design, Team Topologies, Branches) – es erschien mir unpassend, dass die Protagonisten sich Konzepte erklären, die in ihrem Kontext klar sind. Erklärungen finden sich in Kapitel 5 sowie im Glossar.

Die Inhalte der Geschichte funktionieren auch dann, wenn man sich an Kanban oder Extreme Programming orientiert oder einen komplett eigenen Ansatz verwendet. Wichtig ist lediglich, dass iterativ-inkrementell gearbeitet wird.

Im Anschluss an die Geschichte finden sich in Kapitel 5 Anmerkungen zu den einzelnen Episoden. Ich habe dort versucht, darzustellen, woher die ganzen Ideen und Impulse stammen, die sich in der Geschichte wiederfinden. Ich wünschte, es wäre anders, aber das meiste habe ich mir nicht selbst ausgedacht.

Den Abschluss bildet eine Zusammenfassung der Konzepte und Denkansätze der Geschichte. Diese Zusammenfassung kann unabhängig von der Geschichte gelesen werden. Dadurch ergeben sich notwendigerweise Wiederholungen von Konzepten, die in der Geschichte vorkommen.

Danksagung

Ich danke Dr. Adam Melski, Jens Himmelreich und meinem Bruder Arne Roock für das Feedback zu ersten Fassungen der Geschichte. Christa Preisendanz vom dpunkt.verlag danke ich für ihr Feedback und das wie immer großartige Lektorat des Buches.

2 Episode 2: Abhängigkeiten managen

2.1 Abhängigkeitsgraphen

Florence hat sich für den nächsten Tag mit Rob verabredet, um ein klareres Bild von den Abhängigkeiten zwischen den Teams zu bekommen. Rob zeigt ihr einen Abhängigkeitsgraphen, der visualisiert, wann welches Team an welcher Funktionalität arbeitet. Die einzelnen Funktionalitäten sind unterschiedlich eingefärbt, um den Graphen übersichtlicher zu gestalten.

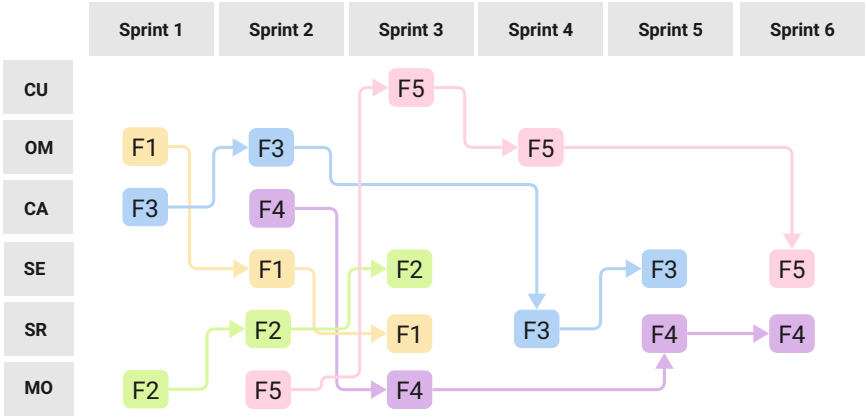


Abb. 2–1 *Abhängigkeitsgraph*

»Wir sind gerade frisch in eine Drei-Monats-Etappe gestartet und der Abhängigkeitsgraph ist das Ergebnis der gemeinsamen Planung«, führt Rob aus.

»Ich nehme mal an, dass die Teams in den Lücken durchaus produktiv arbeiten, aber eben nicht an übergreifenden Funktionalitäten, sondern an Einträgen aus ihren Team-Backlogs?«, vermutet Florence.

»Korrekt«, bestätigt Rob.

»Und den Abhängigkeitsgraphen hast nicht du oder die Product Owner für die Teams erstellt, sondern das haben die Teams selbst gemacht?«, erkundigt sich Florence.

»Genau. Wir führen dazu alle drei Monate eine Etappenplanung durch. Dafür holen wir alle Beteiligten zusammen in einen Raum. Während Corona haben wir die Etappenplanungen natürlich remote durchgeführt. Zunächst hatten wir das sehr rudimentär mit einer Tabellenkalkulation gemacht. Dabei habe ich Blut und Wasser geschwitzt, dass niemand versehentlich wichtige Einträge löscht. Später haben wir es mit den Planungsfunktionen unseres Ticketsystems versucht, aber das führte zu einem Top-down-Vorgehen. Wir haben schließlich festgestellt, dass es für uns am besten mit dem digitalen Whiteboard funktioniert. Damit bekommen wir die für uns am besten geeignete visuelle Darstellung hin, und die Beteiligung aller Teammitglieder gelingt uns am besten. Letztlich erscheint mir der Graph gar nicht so wichtig. Viel wichtiger ist, dass die Teams durch den gemeinsamen Planungsprozess ein gemeinsames Verständnis der Abhängigkeiten und des Arbeitsflusses schaffen. So werden sie selbst aktiv, wenn etwas nicht wie geplant läuft. Sie wissen dann, mit wem sie sprechen müssen.«

»Perfekt«, freut sich Florence. Anscheinend ist sie diesmal in einem Unternehmen gelandet, in dem grundlegende Prinzipien selbstorganisierten Arbeitens verstanden werden.

2.2 Verlässlich unverlässliche Planung

»Du sagtest, die Teams werden selbst aktiv, wenn Dinge anders laufen als geplant«, sagt Florence. »Passiert das oft?«

»Ja, das passiert leider immer mal wieder«, gesteht Rob ein. »Unangenehm häufig, wenn wir ehrlich sind. Wir schaffen oft nur 40 Prozent der geschäftsrelevanten Funktionalität, die wir uns vorgenommen haben.«

Florence ist schockiert. Sind die Teams doch nicht so weit, wie sie dachte? Fehlen ihnen die grundlegenden Prinzipien der Planung?

»Und dabei«, fährt Rob fort, »planen die Teams genau so, wie es überall beschrieben ist. Sie kennen ihre Geschwindigkeit aus der Vergangenheit und benutzen diese für die Vorhersage, wie viel sie in der Zukunft schaffen.«

»Aber diese Vorhersage stimmt dann nicht«, meint Florence.

»Doch! Das ist ja das Verrückte«, erwidert Rob und schüttelt sich. »Meistens schaffen die Teams die Menge an Arbeit, die sie sich vornehmen. Es kommt natürlich immer mal wieder vor, dass ein Team etwas schneller oder

langsamer ist als vorhergesagt. Aber das sollte sich ja eigentlich ausgleichen und erklärt auf keinen Fall, dass wir dann doch so wenig geschäftsrelevante Funktionalität liefern.«

»Vielleicht doch«, murmelt Florence. Irgendetwas dämmert ihr, auch wenn sie es noch nicht richtig greifen kann. Sie mag es nicht, unausgegorene Vermutungen zu diskutieren und schlägt daher vor: »Lass mich mal mit den Teams sprechen und ein paar Daten recherchieren. Ich habe eine vage Vermutung, die ich gerne überprüfen möchte. Ich denke, dass ich bis zu unserem Jour fixe nächste Woche mehr Klarheit habe.«

Rob stimmt zu: »Alles klar. Ich bin gespannt auf deine Erkenntnisse.«

2.3 Die Sache mit den Bussen

In den nächsten Tagen spricht Florence mit Mitgliedern der verschiedenen Teams und wertet aus, wie viel die Teams in der Vergangenheit geplant und tatsächlich geliefert hatten. Tatsächlich liegt die Vorhersagegenauigkeit der Teams bei über 80 Prozent, bezogen auf die Menge gelieferter Funktionalität. In diesem Punkt hat Rob recht behalten. Das sind aber nur die Daten, die sich auf die einzelnen Teams beziehen.

»Ist dir schon mal aufgefallen, dass Busse oft in Rudeln auftauchen?«, fragt Florence ganz am Anfang ihres Jour fixe mit Rob.

»Was?« Rob wirkt verwirrt.

»Das Beispiel habe ich von Don Reinertsen, der sich intensiv mit Flow in der Produktentwicklung auseinandergesetzt hat. Er weist darauf hin, dass Busse oft in Rudeln auftauchen. Wenn man an der Haltestelle wartet, kann es passieren, dass unverhältnismäßig lange kein Bus kommt. Und dann kommen plötzlich mehrere Busse in sehr kurzen Abständen.«

»Ah, das habe ich auch schon mal erlebt. Ich wohne in einem Vorort und da ist es schon sehr nervig, wenn der Bus nicht wie geplant alle 15 Minuten kommt, sondern man 30 Minuten oder länger warten muss«, berichtet Rob. »Und wenn er dann endlich kommt, ist er auch noch irre voll. Kaum ist man eingestiegen und der Bus fährt los, sieht man durch das Rückfenster schon den nächsten Bus derselben Linie. Und dann frage ich mich immer, ob ich nicht hätte warten und den nächsten Bus nehmen sollen. Dann hätte ich nur zwei Minuten verloren, aber vielleicht hätte ich dafür einen Sitzplatz bekommen.«

Florence lächelt. »Ja, vermutlich wäre Warten tatsächlich die bessere Option gewesen und die Wahrscheinlichkeit für einen Sitzplatz definitiv größer.« Sie macht eine kurze Pause. »Dazu müssen wir verstehen, wie es zu der Rudelbildung der Busse gekommen ist. Auf der Straße fahren ja nicht nur Busse, sondern auch andere Fahrzeuge. Außerdem gibt es Kreuzungen und Ampeln. Die Geschwindigkeit, mit der der Bus vorankommt, ist also nicht vollständig vorhersagbar. Sie unterliegt Schwankungen. Und wenn hohes Verkehrsaufkommen herrscht, kann der Bus die verlorene Zeit auch nicht wieder aufholen. Er kommt also verspätet bei der nächsten Haltestelle an.«

»Logisch«, sagt Rob.

»Und wenn der Bus verspätet kommt, was findet er an der nächsten Haltestelle vor?«, fragt Florence.

»Verärgerte Fahrgäste«, erwidert Rob sofort.

»Bestimmt.« Florence schmunzelt. Aber darauf wollte sie nicht hinaus. »Und was noch?«

Rob findet Florence' Frage manipulativ. Sie könnte ihm doch einfach sagen, worauf sie hinauswill. Aber er denkt sich *Wenn es der Wahrheitsfindung dient* und spielt mit. Er überlegt eine Weile und kommt dann darauf: »Viele Fahrgäste.«

»Genau«, bestätigt Florence. »Durch die Verspätung konnten sich mehr Fahrgäste an der Haltestelle versammeln, als durchschnittlich zu erwarten gewesen wäre. Dadurch dauert das Einsteigen länger als im Schnitt zu erwarten wäre.«

»Die Verspätung des Busses nimmt also noch weiter zu«, führt Rob den Gedanken weiter. »Dadurch verringert sich der zeitliche Abstand zum nächsten Bus. Der nächste Bus findet also weniger Fahrgäste an der Haltestelle vor, als zu erwarten gewesen wäre, und das Einsteigen geht schneller. Dadurch schrumpft der Abstand zwischen dem ersten und dem zweiten Bus weiter. Wenn das mehrmals passiert, entsteht eine große Verspätung beim ersten Bus, und dann kommt das Busrudel geballt.«

»Das Beispiel veranschaulicht sehr schön, dass sich Varianzen aufschaukeln können. Der einzelne Bus schafft die Strecke zwischen zwei Haltestellen in der vorgegebenen Zeit mit ein paar kleinen Abweichungen in die eine oder andere Richtung. In Summe entsteht aber eine sehr große Varianz«, schließt Florence die Argumentation ab.

2.4 Verhalten sich Teams wie Busse?

»Du meinst also, unsere Teams sind wie die Busse?«, fragt Rob.

»Exakt«, bestätigt Florence. »Jedes Team hat in seiner Geschwindigkeit und Vorhersagegenauigkeit nur eine geringe Varianz, genauso wie jeder Bus für seine Fahrzeit zur nächsten Haltestelle nur eine geringe Varianz hat. In Summe kann sich das aber dramatisch auswirken, wenn es Abhängigkeiten gibt. Die geringe Varianz der Fahrzeit beim Bus wäre kein Problem, wenn der Bus immer nur eine Haltestelle anfahren würde und dann eine lange Pause macht. Und die geringe Varianz der Teams wäre kein Problem, wenn es keine teamübergreifenden Funktionalitäten gäbe. Und das ist nicht nur ein Gedankenkonstrukt. Die Daten der Teams bestätigen, dass es bei euch genau so gelaufen ist.«

Florence legt Rob ein Blatt Papier hin.

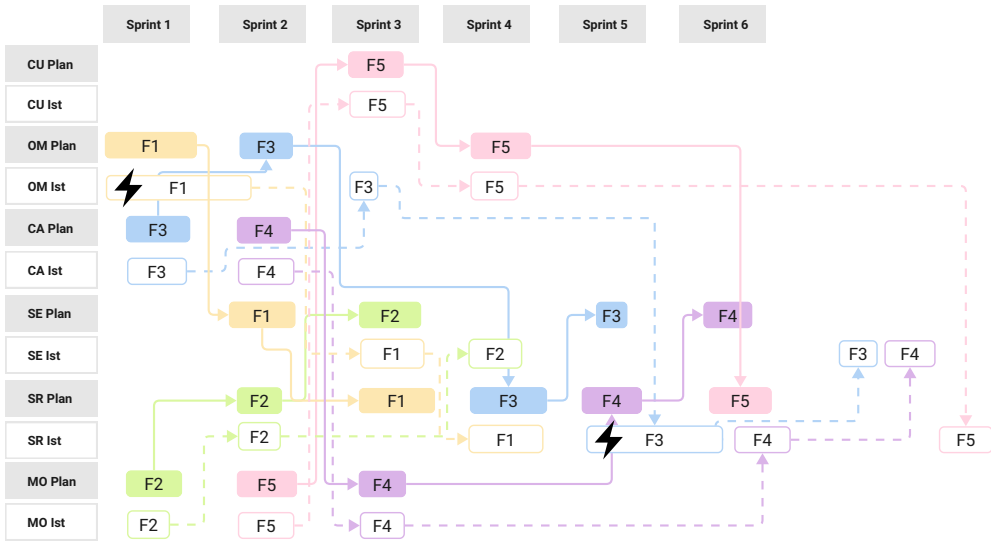


Abb. 2–2 *Entwicklung Plan vs. Realität*

»Oh, ein Mikroprozessor-Schaltplan«, frotzelt Rob.

»Es wirkt unübersichtlicher, als es ist«, beschwichtigt Florence.

»Dann erleuchte mich mal«, fordert Rob sie auf.

»Wir sehen einen Abhängigkeitsgraphen. Nur habe ich neben dem Plan auch die Realität visualisiert. Die ausgefüllten Rechtecke bilden den Plan, die unausgefüllten Rechtecke zeigen, was tatsächlich passiert ist«, erläutert Florence.

»Okay, dann lass mal sehen«, murmelt Rob und vertieft sich in die Abbildung. »Zwei der unausgefüllten Kästchen sind mit Blitzen markiert. Ah, ja. Da hat die Entwicklung länger gedauert als geplant. Das ist nur zweimal passiert und trotzdem wurden drei von fünf Features nicht fertiggestellt. Das ist krass.« Nach einer kurzen Pause fährt Rob fort: »Ich bin beeindruckt. Damit haben wir eine erste harte Nuss geknackt und verstanden, wie es zu der schlechten Gesamtvorhersage kommt. Das ist gut, weil wir dem Phänomen jetzt nicht mehr hilflos ausgeliefert sind. Hast du auch schon eine Idee, wie wir mit dieser neuen Erkenntnis konstruktiv arbeiten können?«

»Ehrlich gesagt habe ich noch keine Lösung und bin immer noch fasziniert von dem Problem«, entgegnet Florence. »Vielleicht können wir Puffer einbauen. Dann sollten sich die Varianzen nicht mehr aufschaukeln.«

»Hm«, brummt Rob. »Das würde die Verlässlichkeit bestimmt erhöhen. Aber so wie ich die Sache verstehe, würden Puffer uns langsamer machen. Schließlich stimmen die Teamvorhersagen in 80 Prozent der Fälle. Die Puffer würden nur in den restlichen 20 Prozent der Fälle benötigt werden. In den 80 Prozent der regulären Fälle wären Zulieferungen der Teams früher fertig als geplant. Folglich könnten sie nicht direkt weiterverarbeitet werden und würden eine Weile herumliegen. Nach ›just in time‹ hört sich das nicht an und nach ›frühestmöglicher Fertigstellung‹ auch nicht.«

»Verdammt«, murmelt Florence, »du hast recht. Das ergäbe nur Sinn, wenn unsere höchste Priorität die plangemäße Fertigstellung wäre und wir bereit wären, dafür Geschwindigkeit zu opfern. Aber das Gegenteil ist der Fall. Wir müssen schneller werden. Dass wir nicht fertigstellen, was wir uns vorgenommen haben, ist für sich genommen erst mal kein allzu großes Problem. Die Diskrepanz zwischen Plan und Ist hat uns nur geholfen, ein paar Dinge besser zu verstehen.«

»Immerhin«, meint Rob. »Mir raucht ganz schön der Kopf. Vielleicht sind das für heute genug Erkenntnisse und wir schlafen mal eine Nacht drüber. Vielleicht kommen wir morgen weiter.«

»Ja, Rob. Du hast recht. Lass uns das Thema morgen weiter diskutieren«, stimmt Florence zu.

2.5 Übergreifende Priorisierung

Am nächsten Morgen treffen Florence und Rob an der Kaffeemaschine aufeinander. Rob beginnt: »Mir ging unsere Diskussion von gestern noch eine Weile durch den Kopf. In gewisser Weise arbeiten wir bereits mit Puffern. Die einzelnen Beiträge der Teams zu übergreifenden Funktionalitäten benötigen selten die gesamte Teamkapazität. Der Rest wird mit Einträgen aus dem Team-Backlog aufgefüllt. Eigentlich sollte es Puffer geben. Die Teams könnten bei Problemen ihren Beitrag zur übergreifenden Funktionalität meist trotzdem zum Sprint-Ende fertig haben. Sie würden dafür dann weniger Einträge aus ihrem Team-Backlog erledigen. Dafür können sich Varianzen aber nicht mehr so leicht aufschaukeln.«

»Klingt logisch«, stimmt Florence zu. »Und warum passiert das nicht?«

»Das ist eine gute Frage. Ich vermute, dass es mit der Priorisierung zusammenhängt. Wir sollten mit ein paar Product Ownern sprechen, um das besser zu verstehen«, schlägt Rob vor. »Nachher findet der wöchentliche Austausch der Product Owner statt. Ich bin sowieso dabei. Komm doch auch einfach mit in die Videokonferenz. Am Ende können wir mit dem einen oder anderen noch kurz sprechen.«

»Perfekt«, sagt Florence.

In dem Austausch-Meeting der Product Owner geht es um Abhängigkeiten zwischen einzelnen Funktionalitäten und wie bei dezentraler Priorisierung durch die Product Owner die Produktkonsistenz sichergestellt werden kann. Rob bringt einen Wunsch von Catherine ein, für dessen Umsetzung mehrere Teams benötigt werden. Sie möchte, dass für Kater Stammbäume hinterlegt und in den Suchergebnissen angezeigt werden. Die Product Owner versprechen zu klären, welches Team was dafür tun muss, um die Arbeit in ihre Product Backlogs aufzunehmen.

Als sich das Meeting dem Ende zuneigt, fragt Rob, wer für Fragen zur Priorisierung noch ein paar Minuten länger bleiben kann. Josephine und Marty können es einrichten.

Florence wendet sich an die beiden: »Rob hat Catherines Wunsch nach der Stammbaum-Funktionalität eingebracht. Was passiert jetzt damit?«

»Na, was schon«, beginnt Marty. »Wir schauen, welches Team welche Funktionen dafür umsetzen muss. Der jeweilige Product Owner erstellt dann die zugehörigen User Stories und nimmt sie in sein Backlog auf.«

»Klar«, antwortet Florence. »Aber wie priorisiert ihr die so entstandenen User Stories im Verhältnis zu den anderen User Stories im Backlog?«

»Autsch«, rutscht es Josephine raus. »Da sprichst du einen wunden Punkt an. Eigentlich können wir das gar nicht. Die Stammbaum-Anforderung führt vermutlich zu einer oder mehreren User Stories im Team *Offer Management*, die dafür sorgen, dass Besitzer von Katern bei der Anlage von Angeboten den Stammbaum hinterlegen können. Aus Sicht des Product Owners dieses Teams hat dies jedoch keinen Wert. Schließlich wird dadurch nichts für die Kunden besser. Aus lokaler Sicht würde er die User Stories superniedrig priorisieren und faktisch nie umsetzen.«

»Logisch«, sagt Florence.

»Aber wir sind ja auch nicht bekloppt«, wirft Marty ein. »Obwohl es lokal betrachtet logisch wäre, machen wir das natürlich nicht so. Wir wissen ja, dass die User Stories Teil einer größeren Anforderung sind, die sehr wohl einen Wert hat.«

»Wäre es dann nicht logisch, die übergreifende Anforderung gegen die anderen User Stories zu priorisieren?«, erkundigt sich Florence.

»Logisch, aber sehr anspruchsvoll«, antwortet Josephine. »Dann müssten ja im Grunde alle User Stories in einem Gesamt-Backlog stehen. Das stelle ich mir sehr unübersichtlich vor.«

»Außerdem ist es superschwierig, sehr große Dinge mit sehr kleinen Dingen zu vergleichen«, ergänzt Rob. »Wir haben das früher mal versucht, aber wieder verworfen. Man hätte dann eine übergreifende Anforderung, die vielleicht zwei Monate zur Umsetzung braucht und einen großen Wert hat. Dagegen haben wir eine User Story, die in zwei Tagen implementiert ist und dafür ›nur‹ die Usability leicht verbessert. Was kriegt die höhere Priorität?«

»Vielen Dank für die Informationen«, sagt Florence. »Jetzt verstehe ich die Problematik besser. Aber wie macht ihr das jetzt tatsächlich? Irgendwie kommt ihr Product Owner ja zu einer Reihenfolge, oder?«

»Das passiert in der gemeinsamen Etappenplanung. Da haben wir die Backlogs der Teams und die übergreifenden Anforderungen als Input und versuchen gemeinsam, das Beste daraus zu machen«, führt Josephine aus. »Immerhin sind die übergreifenden Anforderungen untereinander priorisiert, sodass wir in der Regel eine stimmige Gesamtplanung bekommen.«

Die aber meistens nicht aufgeht, denkt sich Florence. »Und wie behandeln die Teams dann ihre User Stories im Verhältnis zu den User Stories für die übergreifenden Anforderungen?«

»Total unterschiedlich«, sagt Marty. »Das variiert von Team zu Team und hängt auch davon ab, wie dringlich dem Team die übergreifenden Anforderungen erscheinen. Manchmal arbeiten die Teams zuerst an den User Stories für die übergreifenden Anforderungen und manchmal beschäftigen sie sich erst gegen Sprint-Ende damit. Da die Teams meist schaffen, was sie sich vorgenommen haben, spielt das letztlich aber keine große Rolle.«

Denkste!, schießt es Rob durch den Kopf.

Josephine ergreift wieder das Wort: »Es kann also passieren, dass die Teams die User Stories der übergreifenden Anforderungen eher als Hobby nebenbei betreiben.«

»Das ist etwas extrem formuliert, aber in einigen Fällen trifft es zu«, gesteht Marty ein. »Das passiert vor allem dann, wenn der Product Owner des Teams selbst ein wichtiges und motivierendes Ziel verfolgt; dann steht dieses Ziel natürlich im Vordergrund und der Beitrag zur übergreifenden Anforderung eher im Hintergrund.«

»Verständlich«, sagt Florence. »Ich danke euch für die wertvollen Einblicke. Hast du noch Fragen, Rob?«

Rob verneint und sie entlassen die beiden Product Owner aus dem Meeting. Rob fasst zusammen: »Wenn die User Stories der übergreifenden Anforderung erst am Ende des Sprints erledigt werden, ist der theoretisch vorhandene Puffer natürlich zum Teufel.«

»Und damit haben wir das Puzzleteil, das noch zum Verständnis der Situation fehlte«, sagt Florence. »Lass uns morgen weiter am Thema arbeiten. Ich bin für heute ziemlich erledigt.«

2.6 Weniger Stop and Go

»Also, wie kriegen wir jetzt die Priorisierung besser hin, um die theoretischen Puffer auch praktisch nutzen zu können?«, beginnt Rob das Gespräch am Morgen des nächsten Tages.

»Das Priorisierungsthema ist schwer in den Griff zu kriegen. Das haben uns Josephine und Marty gestern ja klargemacht. Ich schlage daher vor, dass wir uns zunächst um ein anderes Thema kümmern«, wendet Florence ein. »Wie wäre es, wenn die Teams die implementierten User Stories der übergreifenden Anforderungen nicht erst am Ende des Sprints dem nächsten Team zur Verfügung stellen, sondern direkt, wenn sie fertig sind?«

»Ist das denn erlaubt?«, fragt Rob.

»Die Idee beim Sprint ist, durch die Timebox Fokus herzustellen. Wenn wir während des Sprints bereits Arbeitsergebnisse an andere Teams übergeben oder sogar Funktionalitäten in Produktion geben, schadet das dem Fokus nicht«, erwidert Florence. »Also spricht nichts dagegen, schon während des Sprints Arbeitsergebnisse an andere Teams zu übergeben.«

»Aber so einfach, wie du dir das vorstellst, geht das nicht«, antwortet Rob abwehrend.

»Lass uns die Frage nach der Machbarkeit mal kurz aufschieben«, besänftigt ihn Florence. »Was wäre, wenn wir es doch hinkriegen würden?«

Rob überlegt. »Dann würden sich die User Stories der übergreifenden Anforderungen bei leichten Verzögerungen nicht gleich um einen ganzen Sprint verschieben, sondern nur um ein oder zwei Tage.« Einen Moment später ergänzt er: »Und es ist dadurch auch weniger wichtig, ob diese User Stories für den Anfang oder das Ende eines Sprints eingeplant werden.«

»Genau«, bestätigt Florence. »Das Priorisierungsproblem ist dadurch zwar nicht gelöst, aber es wird in seinen Auswirkungen entschärft.«

»Gut, gut«, erwidert Rob. »Können wir jetzt über die Umsetzungsschwierigkeiten sprechen?«

»Ja, leg los«, ermuntert Florence ihn.

»Wie gesagt: So einfach ist das nicht. Die technischen Voraussetzungen dafür sind anspruchsvoll. Ich bin aber ehrlich gesagt schon eine Weile aus der Technik raus. Wir sollten mit einem erfahrenen Entwickler sprechen. Alex ist schon lange bei uns und kennt den Code und die Entwicklungsumgebung sehr gut. Ich habe ihn vorhin hier herumlaufen sehen. Er scheint heute im Büro zu sein. Vielleicht hat er ja spontan Zeit für uns. Ich rufe ihn mal an.«

Rob telefoniert mit Alex und verkündet anschließend, dass sie sich zum Mittagessen mit ihm treffen können. Treffpunkt ist ein Restaurant in der Nähe. Nachdem sie bestellt haben, kommt Rob direkt zur Sache. Er schildert kurz die Situation und Problematik und bittet Alex dann um seine Ideen zur Machbarkeit: »Alex, wir haben uns gefragt, ob wir implementierte User Stories übergreifender Anforderungen direkt an das nächste Team weitergeben können, ohne das Sprint-Ende abzuwarten.«

Es dauert einen Moment, bis Alex antwortet. Er scheint nachzudenken, bevor er etwas sagt. Das gefällt Florence. Alex erläutert: »Im Moment entwickelt jedes Team in einem eigenen Branch und integriert diesen zum Sprint-Ende. Dadurch stehen erst zum Sprint-Wechsel allen Teams die implementierten User Stories der anderen Teams zur Verfügung. Wir könnten für jede

übergreifende Anforderung einen eigenen Branch erstellen und den von Team zu Team weiterreichen. Dann bleibt der Branch aber lange aktiv und je länger ein Branch aktiv bleibt, desto größer werden die Integrationsaufwände. Das könnte ein aufwendiges Vergnügen werden.«

»Ich habe ja gesagt, dass es nichts wird«, sagt Rob.

»Leider«, stimmt Florence zu.

Alex unterbricht die beiden: »Aber das ist natürlich nicht die einzige Möglichkeit, mit dem Thema umzugehen. Wir könnten auch komplett auf alle Branches verzichten und immer mit allen Teams direkt am gemeinsamen Codebestand arbeiten. Dann stehen jegliche Codeänderungen immer sofort allen Teams zur Verfügung.«

»Leider auch der fehlerhafte Code«, sagt Rob. »Wir haben die Branches damals eingeführt, weil wir mit dem Wachstum – mit den neuen Teams – wahnsinnige Qualitätsprobleme im Code bekommen haben. Irgendjemand hat versehentlich etwas kaputt gemacht, und das ist erst mit etwas Verzögerung entdeckt worden. Bis dahin gab es aber noch eine ganze Menge weiterer Codeänderungen, und schließlich wusste niemand, wer das Problem verursacht hatte. Alle haben gehofft, dass der Verursacher sich selbst um das Problem kümmert. Wegen der Unklarheiten ist das natürlich nicht passiert, und als kurz vor Lieferzeitpunkt klar wurde, dass sich das Problem nicht von selbst beheben würde, hatte es bereits so viele Änderungen am Code gegeben, dass die Fehlerbehebung irrsinnig aufwendig wurde.«

»Ja, ich erinnere mich noch gut daran«, berichtet Alex mit schmerzverzerrtem Gesicht. Doch dann hellt sich seine Miene auf: »Das läuft inzwischen deutlich entspannter. Ich vermute, dass euer Anliegen einen guten Grund hat und es sich lohnen könnte, das Thema noch einmal anzugehen. Schließlich ist das Integrieren der Branches durchaus auch mit relevantem Aufwand verbunden. Es wäre schon schön, wenn wir den nicht mehr bräuchten. Uns ist damals auf die Füße gefallen, dass wir den fehlerhaften Code so spät entdeckt haben. Seitdem haben wir in automatisierte Tests investiert und inzwischen eine ganz gute Testabdeckung erreicht. Ich denke, dass wir fehlerhaften Code heute schon vor dem Einchecken entdecken würden oder kurz danach. Wir müssen das auch nicht direkt mit allen Teams auf einmal ausprobieren. Wir können mit einem Team beginnen und wenn das funktioniert, nehmen wir das zweite dazu. Und so fügen wir immer ein zusätzliches Team hinzu, das ohne Branch arbeitet. Das machen wir so lange, bis alle Teams ohne Branches arbeiten oder wir feststellen, dass es doch zu viele Probleme gibt.«

»Es scheint also möglich zu sein«, sagt Rob. »Aber lohnt es sich auch? Ich meine, da steckt ja doch Aufwand und Risiko drin. Im schlimmsten Fall stellen wir kurz vor Ende fest, dass wir es nicht schaffen, und dann war alles für die Katz.«

»Ja, das sollten wir auf jeden Fall abwägen. Lasst uns überlegen, wie groß der Vorteil wirklich sein kann«, stimmt Florence zu. »Sehen wir uns deinen Abhängigkeitsgraphen noch einmal an.« Florence fingert das Blatt mit dem Diagramm, das sie vorsichtshalber zum Treffen mitgenommen hat, aus ihrer Tasche und breitet es auf dem inzwischen abgeräumten Tisch aus.

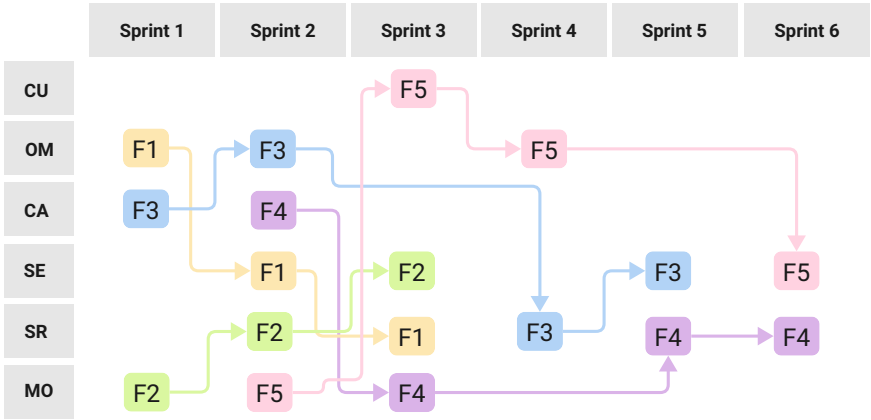


Abb. 2–3 *Abhängigkeitsgraph mit Übergabe zum Sprint-Ende*

Sie fährt fort: »Lasst uns mal über den Daumen ein paar Zahlen rechnen. Die erste übergreifende Funktionalität ist nach drei Sprints fertig. Das trifft sowohl auf F1 wie auch F2 zu. Das ist auch gleichzeitig die kürzeste Durchlaufzeit: drei Sprints. Die längste Durchlaufzeit liegt bei fünf Sprints; das gilt für F3, F4 und F5. Die durchschnittliche Durchlaufzeit liegt also bei 4,2 Sprints. Alle übergreifenden Funktionalitäten zusammen sind nach sechs Sprints fertig.«

Florence schreibt die Daten in eine Tabelle auf einem weiteren Blatt:

	Sprints
Durchlaufzeit min.	3
Durchlaufzeit max.	5
Durchlaufzeit ∅	4,2
Durchlaufzeit alle	6

Florence fährt fort: »Und jetzt machen wir dieselbe Übung mit der Annahme, dass fertiggestellte User Stories noch im selben Sprint von anderen Teams weiterbearbeitet werden können.«

Rob holt sein Notebook hervor. »Ich suche die Schätzungen für die einzelnen User Stories der übergreifenden Funktionalitäten heraus.«

»Und ich recherchiere die Geschwindigkeiten der Teams«, ergänzt Florence. »Dann wissen wir grob, wie lange die Teams für die User Stories brauchen und wann das nächste Team an einer übergreifenden Funktionalität weiterarbeiten kann.«

»Ich fürchte, das ist etwas zu einfach gedacht«, wirft Alex ein. »Unser Abhängigkeitsgraph würde dann davon ausgehen, dass immer das komplette Team an einer User Story arbeitet, und zwar ohne Effizienzverlust.«

»Oh, Mist«, flucht Rob.

Alex beginnt zu lächeln: »Wir können das leicht lösen. Wir brauchen die Geschwindigkeit der Teams gar nicht. Wir können bei bereits fertiggestellten User Stories nachsehen, wie lange die Erledigung gedauert hat. Die Dauer können wir in Beziehung zur Schätzung setzen und so auch für noch nicht erledigte User Stories die Dauer prognostizieren.«

Florence runzelt die Stirn: »Das war zu abstrakt für mich. Kannst du mir das noch mal konkreter erklären?«

»Klar«, antwortet Alex. Lasst uns mal irgendeinen Sprint ansehen, zum Beispiel Sprint 35 von Team SR. In diesem Sprint hat das Team User Stories im Umfang von 23 Story Points erledigt. Eine User Story hatte acht Story Points, zwei hatten fünf Story Points, eine hatte drei Story Points und nochmal zwei jeweils einen Story Point. Wir können auch nachsehen, wann die Arbeit an einer User Story begonnen wurde und wann sie beendet wurde. Daraus können wir die Durchlaufzeit der User Story berechnen und diese in Bezug zur Schätzung setzen.« Alex erstellt eine Tabelle.

Schätzung in Story Points	Durchlaufzeit in Werktagen
8	6
5	3
5	6
3	3
1	4
1	1

»So ganz eindeutig ist die Beziehung zwischen Schätzung und Dauer ja nicht«, merkt Rob an. »Warum dauert eine User Story mit einem Story Point länger als eine mit drei Story Points?«

Florence antwortet darauf: »Schätzungen stimmen nicht immer mit der Realität überein – das sagt der Begriff Schätzung ja bereits. Außerdem schätzen die Teams mit Story Points den Umfang der Arbeit und nicht die Dauer. Vielleicht hat ein Junior-Entwickler die langsame Ein-Punkt-User Story umgesetzt und zwei Seniors im Pair die schnelle Drei-Punkt-Story.«

»Ich verstehe«, sagt Rob. »Aber wie erhalten wir jetzt aus diesen Zahlen eine Dauer pro Story Point, mit der wir rechnen können?«

»Dazu schauen wir uns eine ganze Reihe von Sprints pro Team an und nehmen dann für die Dauer die Mittelwerte«, antwortet Alex. »Für unseren Kontext sollte das genau genug sein. Ich erstelle eben ein Skript, das das berechnet.«

Während Alex in die Tasten seines Laptops haut, wendet sich Rob an Florence. »Eigentlich ein genialer Ansatz. Wenn man das weiterdenkt, brauchen wir am Ende vielleicht gar keine Schätzungen mehr.«

Florence unterbricht seine Gedanken: »Das ist sicher interessant, aber wir wollen gerade ein anderes Problem lösen. Ich glaube, Alex ist gleich so weit. Lass uns mit Alex' Ansatz den Abhängigkeitsgraphen erstellen.«

Zu dritt haben sie schnell den neuen Graphen erstellt. Die unterschiedlichen Dauern der einzelnen User Stories visualisieren sie durch ihre Breite. Florence, Rob und Alex sehen sich das Ergebnis gemeinsam an.

»Wow«, staunt Rob, »das ist aber unübersichtlich. Der alte Abhängigkeitsgraph hat mir besser gefallen.«

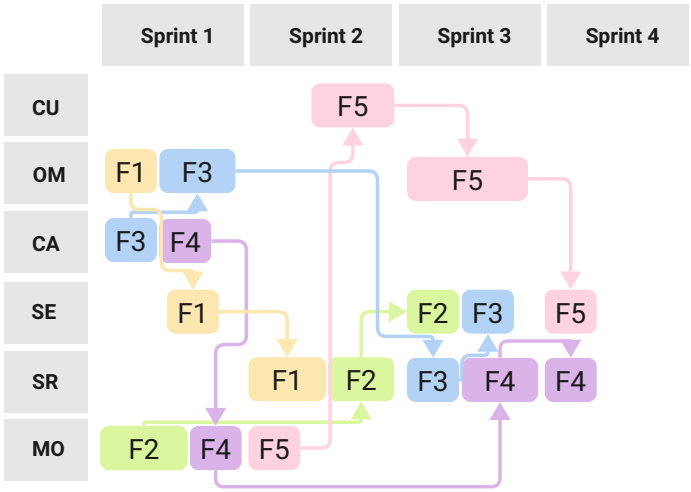


Abb. 2–4 *Abhängigkeitsgraph mit Übergabe im Sprint*

»Stimmt, das ist unübersichtlicher. Das dürfte daran liegen, dass jetzt alles gedrängter ist. Wir werden also schneller«, erläutert Florence.

Alex nimmt sich die Tabelle vor, die Florence vorhin erstellt hat, und ergänzt sie um eine weitere Spalte.

	Sprints mit Warten auf das Sprint-Ende	Sprints ohne Warten auf das Sprint-Ende
Durchlaufzeit min.	3	2
Durchlaufzeit max.	5	4
Durchlaufzeit ∅	4,2	3
Durchlaufzeit alle	6	4

Er stellt fest: »Wir erreichen bei jeder Metrik einen Fortschritt. Über den Daumen gepeilt werden wir um ein Drittel schneller.«

»Das ist doch ein guter Anfang«, freut sich Florence.

»Die Änderung der *Branching*-Strategie birgt immer noch ein Risiko, aber der potenzielle Nutzen scheint den Aufwand wert zu sein«, stimmt Rob zu. »Wer nicht wagt, der nicht gewinnt.«

»Dann machen wir es so«, sagt Alex. »Ich nehme das Branching-Thema mit in die Teams und sorge dafür, dass wir Team für Team ausprobieren, ohne Branches zu arbeiten.« Florence bezahlt für alle und gemeinsam machen sie sich auf den Weg zurück ins Büro.

2.7 Es reicht noch nicht

Die Änderung der Branching-Strategie dauert ein paar Wochen, ist aber, abgesehen von kleineren Problemen, erfolgreich verlaufen. Tatsächlich beschleunigt sich schon in der aktuellen Etappe die Entwicklung der übergreifenden Funktionalitäten.

Florence, Alex und Rob sitzen mit Catherine zusammen und besprechen die Lage. Nachdem sie von den anderen drei auf den aktuellen Stand gebracht wurde, ergreift Catherine das Wort: »Eine Beschleunigung um ein Drittel ist ein toller Fortschritt. Allerdings sind wir für das, was wir in den USA vorhaben, immer noch zu langsam. Ihr müsst weiter an der Beschleunigung arbeiten.«

Florence antwortet ihr: »Wir verstehen, dass die Beschleunigung wichtig ist. Um das zu erreichen, müssen wir unsere Arbeitsweisen ändern, und das bedeutet auch, dass wir Risiken eingehen. Es wird sicher auch mal Rückschläge geben, und wir werden temporär auch mal langsamer werden.«

»Das verstehe ich«, sagt Catherine. »Als die Hochspringer vom Straddle zum Fosbury-Flop übergingen, sprangen sie zunächst nicht so hoch wie mit dem Straddle. Sie mussten die alte Sprungtechnik verlernen und die neue erlernen. Das gilt genauso auch für unsere Arbeitsweise. Wir müssen nur sicherstellen, dass wir uns nicht nur mit uns selbst beschäftigen, sondern auch liefern und irgendwann tatsächlich auch schneller werden.«

»Garantieren können wir den Erfolg leider nicht«, wirft Rob ein.

»Wir sollten bei unseren regelmäßigen Meetings dieses Thema diskutieren, um ein gemeinsames Verständnis der jeweiligen Situation zu schaffen«, schlägt Florence vor.

Der Vorschlag passt für alle Anwesenden. Es ist ein guter Abschluss für das Meeting.

2.8 Blockaden lösen

Als sie das Meeting verlassen, schlägt Rob vor, sich in der Cafeteria noch zusammenzusetzen. Nachdem die drei sich mit dem Kaffee ihrer Wahl versorgt haben, sagt Rob freudig: »Ich habe eine Idee, wie wir bei den übergreifenden Funktionalitäten noch schneller werden können.«

»Schieß los«, sagen Alex und Florence gleichzeitig.

»Florence' Vorschlag, von den festen Sprint-Zyklen wegzugehen und einen fließenden Übergang zwischen den Teams zu ermöglichen, hat letztlich den Stein ins Rollen gebracht«, beginnt Rob. »Dadurch haben wir die Abhängigkeiten zusammengestaucht und den Graphen unübersichtlicher gemacht.« Rob fährt grinsend fort: »Ich habe mich gefragt, wie wir den Graphen noch unübersichtlicher machen können. Oder anders ausgedrückt: Wie können wir ihn noch weiter stauchen?«

»Wie soll das denn gehen? Im Abhängigkeitsgraphen sieht man doch, dass alles voll ist«, widerspricht Florence.

»Schön, dass ich nicht der Einzige bin, der manchmal Denkblockaden hat«, sagt Rob lächelnd.

»Ich habe eine Idee, worauf Rob hinauswill«, schmunzelt Alex und erläutert: »Wenn wir die User Stories der übergreifenden Funktionalitäten in beliebiger Reihenfolge oder sogar parallel entwickeln könnten, wäre das Bild im Abhängigkeitsgraphen noch gestauchter.«

»Das wäre ja phänomenal, aber geht das überhaupt?«, fragt Florence. »Ich kann das nicht einschätzen. Dazu fehlt mir die technische Expertise.«

»Prinzipiell ist das denkbar und das haben wir dir, Florence, zu verdanken«, antwortet Alex.

»Mir?«, fragt Florence ungläubig.

»Genau. Auf deine Anregung hin haben wir die Entwicklung so umgestellt, dass alle Teams kontinuierlich im zentralen Code ohne Branches arbeiten«, erklärt Alex. »Und das ist auf jeden Fall notwendig, wenn wir die User Stories parallel entwickeln wollen. Bevor wir aber zu den Voraussetzungen kommen, die dafür notwendig wären, würde ich mir gerne den resultierenden Abhängigkeitsgraphen ansehen.« Zum Glück hatte Alex daran gedacht, seine Unterlagen mit in die Cafeteria zu nehmen. Er holt ein Blatt heraus und fängt an, einen neuen Abhängigkeitsgraphen zu erstellen. »Das müsste dann ungefähr so aussehen.«

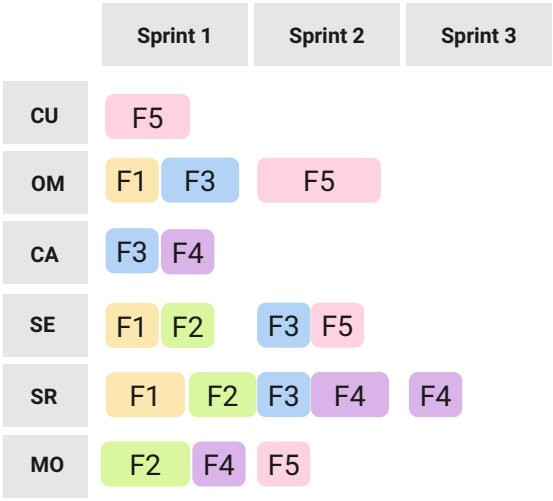


Abb. 2–5 *Abhängigkeitsgraph mit nicht blockierenden Abhängigkeiten*

»Du hast die Pfeile vergessen«, merkt Rob an.

»Das trifft es nicht ganz«, antwortet Alex. »Wir brauchen sie schlicht nicht mehr. Wenn es keine Reihenfolge mehr gibt, die wir einhalten müssen, müssen wir diese nicht mit Pfeilen ausdrücken.«

»Ja, das ergibt Sinn«, sagt Florence. »Lasst uns unsere Tabelle ergänzen. Die ersten beiden Varianten haben blockierende Abhängigkeiten. Das bedeutet, dass die Weiterentwicklung blockiert ist, bis das aktuelle Team seinen Beitrag erledigt hat. Jetzt haben wir nicht blockierende Abhängigkeiten. Die Teams müssen nicht aufeinander warten.«

	Blockierende Abhängigkeiten		Nicht blockierende Abhängigkeiten
	Sprints mit Warten auf das Sprint-Ende	Sprints ohne Warten auf das Sprint-Ende	Sprints
Durchlaufzeit min.	3	2	1
Durchlaufzeit max.	5	4	3
Durchlaufzeit ∅	4,2	3	1,8
Durchlaufzeit alle	6	4	3

»Das ist noch einmal eine spürbare Verbesserung«, stellt Rob fest.

Florence hat noch eine weitere Idee: »Wir könnten die durchschnittliche Durchlaufzeit sogar noch optimieren, wenn die Teams nicht so früh wie möglich mit ihren User Stories für die übergreifende Funktionalität beginnen, sondern just in time. Dann sieht der Graph so aus.« Florence aktualisiert die Grafik.

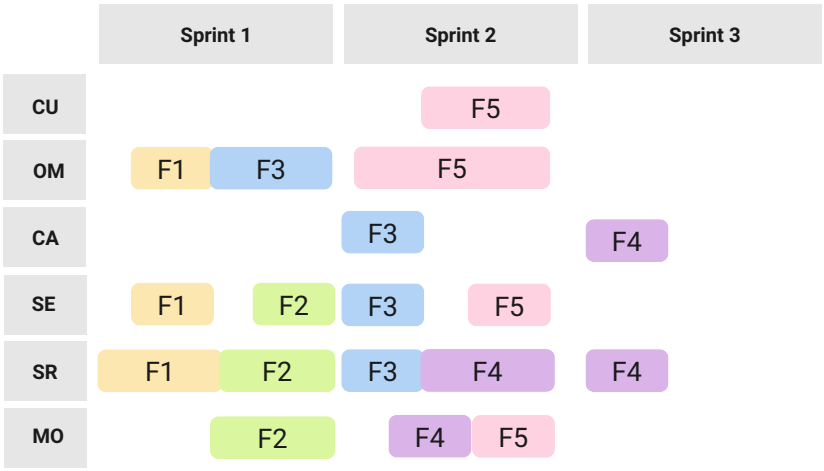


Abb. 2–6 *Abhängigkeitsgraph mit nicht blockierenden Abhängigkeiten – just in time*

Dann stellt sie fest: »Und wir bekämen diese Kennzahlen.«

	Blockierende Abhängigkeiten		Nicht blockierende Abhängigkeiten	
	Sprints mit Warten auf das Sprint-Ende	Sprints ohne Warten auf das Sprint-Ende	Sprints bei möglichst frühem Start	Sprints bei Just-in-time-Start
Durchlaufzeit min.	3	2	1	1
Durchlaufzeit max.	5	4	3	2
Durchlaufzeit ∅	4,2	3	1,8	1,4
Durchlaufzeit alle	6	4	3	3

»Gegenüber unserem Start wären wir damit je nach Kennzahl zwei- bis dreimal schneller«, fährt Florence fort.

»Ich bin sehr beeindruckt«, bemerkt Rob.

»Ja, das wäre eine grandiose Verbesserung. Bisher ist es aber nur eine Idee, die wir noch zum Fliegen bringen müssen«, dämpft Florence die Begeisterung etwas.

Alex hat den Blick auf den letzten Abhängigkeitsgraphen gerichtet und sagt grübelnd: »Der Abhängigkeitsgraph zeigt nach wie vor, welches Team wann an welcher User Story für die übergreifende Funktionalität arbeitet. Aber wir haben ja gar keine sequenzielle Abhängigkeit mehr. Ist es vor diesem Hintergrund wirklich wichtig, vorab zu planen, wann welches Team mit seinem Beitrag zur übergreifenden Funktionalität beginnt?«

»Du hast recht«, sagt Florence nachdenklich. »Da gibt es doch eine alternative Darstellung. Ich glaube, die sieht so aus.« Sie erstellt auf einem neuen Blatt Papier eine Matrix mit den Teams in den Spalten und den übergreifenden Funktionalitäten in den Zeilen.

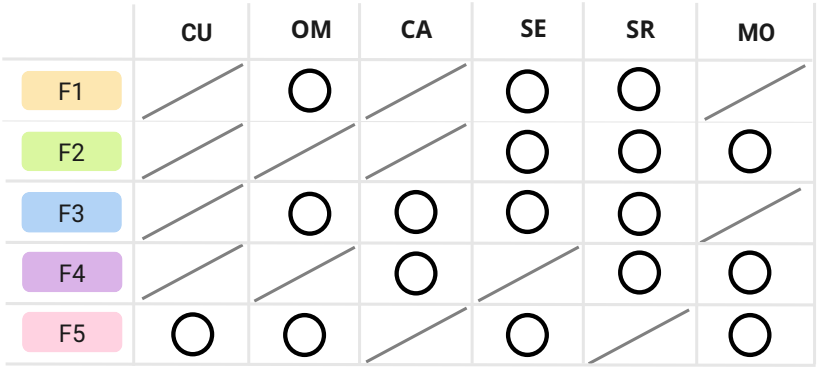


Abb. 2–7 *Abhängigkeitsmatrix*

»Die Kreise zeigen an, welche Teams zu der jeweiligen Funktionalität beitragen müssen. Durchgestrichene Zellen geben an, dass kein Beitrag notwendig ist«, führt Florence aus.

»Und wenn ein Team seinen Beitrag zur übergreifenden Funktionalität geleistet hat, hakt man den Kreis einfach ab«, vermutet Rob.

»Exakt«, bestätigt Florence. »Damit würden wir nicht mehr planen, wann welches Team mit seiner Arbeit dran ist. Die Teams würden in einer gemeinsamen Planung identifizieren, welche Teams was beitragen müssen, und daraus diese Matrix entwickeln. Ah, jetzt fällt mir auch der Name wieder ein«, freut sie sich. »Das ist eine *Abhängigkeitsmatrix*. Natürlich ist es Geschmackssache, ob man die Teams in die Zeilen oder in die Spalten schreibt. Ich finde die Variante mit den Funktionalitäten in den Zeilen etwas eingängiger, weil wir von oben nach unten und von links nach rechts lesen und ich gerne den Fokus auf die Funktionalitäten lege.«

»Wenn wir nicht mehr vorher festlegen, wann welches Team mit seinem Beitrag beginnt und bis wann der Beitrag fertiggestellt sein soll, geht den Teams ein Stück Orientierung verloren«, überlegt Alex laut. »Das müssten wir kompensieren.«

Florence schlägt vor: »Wie so häufig ist auch hier die Antwort: mehr Dialog. Die Teams müssen kontinuierlich miteinander kommunizieren.«

Rob schaltet sich ein: »Das hört sich immer so schön und leicht an. Ich habe aber immer wieder erlebt, dass es dann Unmengen an Meetings gab, alle genervt waren und kaum noch Zeit für die eigentliche Arbeit – das Entwickeln – blieb.«

Alex verzieht das Gesicht, als er sich daran erinnert und nickt zustimmend.

Florence merkt, wie Ärger in ihr aufsteigt. Natürlich muss man mehr miteinander reden. Wenn das dazu geführt hat, dass bei CatDate nichts mehr entwickelt wurde, müssen sie es falsch gemacht haben. Sie will eigentlich zu einer Verteidigung ihrer Position ansetzen, kann sich aber doch beherrschen. Sie gibt sich eine Sekunde, um ihre eigenen Gedanken zu reflektieren, und erinnert sich an das Prinzip der kunstvollen Teilnahme: *Wie muss ich mich verhalten, um die Kooperation der Gruppe optimal zu unterstützen?* Offensichtlich würde das nicht passieren, wenn sie einfach auf ihrer Position beharrt. Sie entscheidet sich, zusätzliche Informationen zu geben: »Wenn man nur noch in Meetings sitzt, kann das ein Indiz dafür sein, dass sich die Organisationsstruktur zu weit von der Wertschöpfungsstruktur entfernt hat. Dann entstehen viele Abhängigkeiten, und man muss ständig darüber sprechen.«

Alex und Rob denken über ihre Worte nach und wirken leicht irritiert. Rob ergreift zuerst das Wort: »Du meinst also, wir sollten die Teamstruktur ändern? Und wieso sprechen wir jetzt über Teamstrukturen und nicht mehr über die Koordination der existierenden Teams?«

Jetzt hatte auch Alex seine Gedanken sortiert: »Ich glaube, Florence meint etwas anderes. Als wir damals das Problem der ausufernden Meetings hatten, hatten wir andere Teamstrukturen. Die waren tatsächlich deutlich weiter weg von der Wertschöpfung als die heutigen Teams. Es kann also gut sein, dass es diesmal besser läuft.«

»Danke, Alex«, sagt Florence. »Genau das habe ich gemeint.«

Rob lenkt ein: »Da mag etwas dran sein. Ich wäre auf jeden Fall gewillt, es noch mal auszuprobieren. Florence, mach bitte weiter.«

»Gerne«, sagt Florence. »Wo war ich noch gleich. Ah ja. Wir hatten diskutiert, dass es Vorteile hätte, wenn die Abhängigkeiten der Teams bei übergreifender Funktionalität keine Reihenfolge mehr implizieren würden und die Teams daher auch parallel an der Funktionalität arbeiten könnten. Dadurch geht den Teams ein Stück Orientierung verloren, und das müssen wir durch mehr Dialog kompensieren. Das erzeugt natürlich mehr Abstimmungsaufwand. Aber wir hoffen, dass sich dieser Zusatzaufwand lohnt, weil wir die Time-to-Market weiter senken.« Florence trinkt einen Schluck ihres nur noch lauwarmen Kaffees. »Wir brauchen teamübergreifend das, was wir *gemeinsames Bewusstsein* nennen«, fährt sie fort. »Dafür braucht es Klarheit über das Spielfeld, die Aufstellung und den Spielplan und während des Spiels Klarheit darüber, was gerade passiert. Über die gemeinsame Planung bekommen wir Klarheit über Spielfeld, Aufstellung und Spielplan. Ein Teil davon bildet sich in der Abhängigkeitsmatrix ab. Und Klarheit über das, was passiert, können die Teams zum Beispiel über *Scrum of Scrums* herstellen: Vertreter der Teams treffen sich täglich, um über den Fortschritt an den übergreifenden Funktionalitäten zu sprechen. Wenn ein Team seinen Beitrag erledigt hat, werden die Kreise in der Abhängigkeitsmatrix abgehakt.«

»Ich erinnere mich, dass wir früher mal mit Scrum of Scrums gearbeitet haben«, wirft Rob ein. »Warum haben wir damit eigentlich aufgehört?«

Alex antwortet: »Weil wir sie nicht mehr gebraucht haben. Mit der Etappenplanung und dem daraus resultierenden Abhängigkeitsgraphen war ja klar, wer wann was tut. Die Teams hatten sich in den Scrum of Scrums nichts mehr zu sagen. Das Meeting war für uns Zeitverschwendung. Mit der veränderten Arbeitsweise könnte das wieder anders aussehen.«

»Damit hätten wir eine geeignete Darstellung der Abhängigkeiten und eine Struktur, um gemeinsam Klarheit über den Stand der Dinge zu bekommen«, fasst Florence zusammen. »Reicht das schon aus, um zu nicht blockierenden Abhängigkeiten zu kommen?«

»Gute Frage«, sagt Alex. »Die Teams könnten bei der Entwicklung ihrer User Stories die Funktionen noch nicht nutzen, die die anderen Teams gerade erst entwickeln. Sie müssten sich also darüber abstimmen, wie die Schnittstellen aussehen werden und diese simulieren – mit *Stubs* oder *Mocks*. Technisch ist das keine Raketenwissenschaft, aber die Teams sind es nicht gewohnt, so zu arbeiten. Wir brauchen sicher etwas Zeit, um diese neue Arbeitsweise zu verinnerlichen. Es entsteht außerdem zunächst ein vermeintlicher Mehraufwand durch die Entwicklung der Stubs und Mocks. Theoretisch sollte das nicht wirklich ins Gewicht fallen, weil sie für automatisierte

Tests sehr nützlich wären und uns am Ende sogar schneller machen könnten. Aber dazu haben wir in den Teams noch einiges zu lernen.«

»Dass wir zwischendurch auch mal langsamer werden, haben wir mit Catherine bereits abgeklärt«, sagt Rob. »Das ist in Ordnung.«

»Können wir diese Änderung der Arbeitsweise wieder schrittweise einführen, so ähnlich wie bei der Branching-Strategie?«, fragt Florence.

»Team für Team funktioniert es leider nicht«, antwortet Alex. »Wir brauchen schließlich häufig vier Teams für eine übergreifende Funktionalität, und die müssen alle gleichzeitig nach der neuen Arbeitsweise arbeiten. Es ist aber sicher sinnvoll, es erst einmal nur mit einer übergreifenden Funktionalität zu üben und nicht gleich mit allen gleichzeitig.«

»Dann mal los«, sagt Rob.

2.9 Iterationen verkürzen

»Wir haben das neue Vorgehen direkt bei der nächsten übergreifenden Funktionalität ausprobiert. Einige Teammitglieder waren zunächst skeptisch, haben sich dann aber doch auf das Experiment eingelassen. Natürlich klappte nicht alles auf Anhieb. Aber ich bin guter Dinge, dass sich die neue Zusammenarbeit schnell einspielen wird«, führt Alex aus, als er sich mit Florence und Rob das nächste Mal im Büro trifft. »Und uns ist dabei noch etwas aufgefallen. Wir können die meisten übergreifenden Funktionalitäten in einem Sprint umsetzen und brauchen bei keiner Funktionalität mehr als zwei Sprints.«

Rob sieht noch mal in der Tabelle der Durchlaufzeiten nach, die sie jetzt regelmäßig pflegen, und bestätigt: »Ja, stimmt. Das ist ziemlich cool.«

Alex fährt fort: »Und in diesem Zusammenhang haben wir uns in den Teams gefragt: Wozu brauchen wir dann noch die Drei-Monats-Etappenplanungen?«

»Hoppla«, entfährt es Rob. »Das hatte ich ja gar nicht auf dem Schirm.« Nach kurzem Überlegen gibt er zu bedenken: »Aber leider schaffen wir nicht alle übergreifenden Funktionalitäten in einem Sprint; einige brauchen zwei Sprints. Wir bräuchten also schon noch so etwas wie Etappenplanungen. Die müssten aber nur noch zwei Sprints, also vier Wochen, umfassen und nicht mehr drei Monate. Das ist doch schon ein super Fortschritt.«

»Oder wir verlängern die Sprints auf vier Wochen«, schlägt Florence vor.

»Was?«, entgegnet Rob irritiert. »Ist das überhaupt erlaubt?«

»Oh ja«, erwidert Florence. »Das ist sehr wohl erlaubt. Tatsächlich haben wir uns bisher auch nicht strikt an Scrum gehalten. Wir liefern nicht nach jedem Sprint ein lieferbares Produktinkrement – wegen der übergreifenden Funktionalitäten. Mit Vier-Wochen-Sprints könnten wir nach jedem Sprint liefern. Und am Ende ist es ja nicht wichtig, wie sehr wir uns an die Vorgaben eines Frameworks halten. Es muss darum gehen, wie wir schneller Wert liefern.«

»Die Idee gefällt mir«, sagt Alex. »So werden wir ein Meeting los und können uns stärker darauf fokussieren, Wert zu liefern. Und mit der Zeit können wir zusammen mit den Product Ownern lernen, wie wir übergreifende Funktionalitäten so schneiden können, dass sie in einem Zwei-Wochen-Sprint umsetzbar sind, und dann die Sprints wieder verkürzen.«

2.10 Work in Progress

Einige Tage später treffen sich Florence und Rob in der Cafeteria. Rob beginnt: »Wir haben die Durchlaufzeiten der übergreifenden Funktionalitäten drastisch verkürzt, ohne mehr Kapazität aufgebaut zu haben. Das fühlt sich wie ein Taschenspielertrick an.«

Florence stimmt zu: »Ja, das Gefühl kann ich gut nachvollziehen. Es gibt aber eine fundierte Erklärung dafür.« Sie kritzelt eine Formel auf eine Serviette:

Little's Law

$$\text{Durchlaufzeit} = \frac{\text{Work in Progress}}{\text{Durchsatz}}$$

»Die durchschnittliche Durchlaufzeit errechnet sich nach Little's Law, indem man den durchschnittlichen Work in Progress durch den durchschnittlichen Durchsatz teilt«, erklärt Florence die Formel. »Der Work in Progress ist die Menge an Arbeit, die parallel bearbeitet wird. Der Durchsatz ist quasi die Geschwindigkeit, mit der gearbeitet werden kann.«

»Ich erinnere mich dunkel, davon mal gehört zu haben«, sagt Rob und erzählt: »Wenn ich mit meinen Kindern im Freizeitpark Achterbahn fahren möchte, kenne ich das aus eigener Erfahrung. Je mehr Leute anstehen, desto länger dauert es, bis man an der Reihe ist. Die anstehenden Leute sind quasi der Work in Progress. Der Durchsatz ist die Menge der Leute, die die Achterbahn gleichzeitig befördern kann. Wenn 100 Leute anstehen und 20 Personen in die Achterbahn passen und die Achterbahn für eine Runde inklusive

Ein- und Aussteigen zwei Minuten braucht, dann dauert es etwa zehn Minuten, bis die Person am Ende der Schlange gefahren ist. Wenn der Work in Progress geringer wäre und nur 50 Personen anstünden, dann würde es nur etwa fünf Minuten dauern. Wenn wir den Durchsatz erhöhen wollten, bräuchten wir eine zweite Achterbahn. Wenn wir diese zusätzlich hätten, würde die Durchlaufzeit bei 50 Personen sogar nur 2,5 Minuten betragen.«

»Genau«, bestätigt Florence. »Auf dieser Basis werden im Wartebereich ja häufig Markierungen angebracht, dass man ab hier noch 60, 45 oder 30 Minuten warten muss.«

»Das wird heutzutage aber kaum noch gemacht«, sagt Rob grinsend.

»Warum nicht? Ich war schon lange nicht mehr in einem Freizeitpark«, sagt Florence.

»Das ist heute alles digital«, antwortet Rob. »Es gibt Apps und oft auch große Übersichtsmonitore im Park, die anzeigen, bei welcher Attraktion welche Wartezeiten zu erwarten sind. Ich vermute, dass es in den Wartebereichen Sensoren gibt, mit denen der Parkbetreiber ermitteln kann, wie lang die Warteschlangen sind. Mit den Infos über die Wartezeiten kann man dann vorher schon abschätzen, ob man sich irgendwo eine Stunde anstellen will oder ob man erst mal etwas anderes fährt und hofft, dass bei der Hauptattraktion später die Wartezeiten kürzer sind.«

»Das ist ja cool. Vielleicht gehe ich demnächst auch mal wieder Achterbahnfahren«, sagt Florence. »Die Apps und Monitore sind ein pfiffiger Mechanismus, um den Work in Progress zu senken. Man kann den Leuten ja schlecht verbieten, sich irgendwo anzustellen. Stattdessen setzt man darauf, dass die Leute selbstorganisiert gute Entscheidungen treffen, wenn die Wartezeiten transparent sind.«

»Aber ist das wirklich wichtig für den Parkbetreiber? Die Leute haben ja vorher den Eintritt bezahlt. Wie lange sie dann anstehen, kann ihm doch egal sein«, wirft Rob ein.

»Ich könnte mir vorstellen, dass die Kundenzufriedenheit sinkt, wenn man lange warten muss. Dann wäre die Optimierung der Durchlaufzeit eine Maßnahme, um die Kundenzufriedenheit zu erhöhen und dadurch mehr Geschäft durch wiederkehrende Besucher und Weiterempfehlungen zu generieren«, vermutet Florence.

»Klar«, sagt Rob. »Darauf hätte ich auch selbst kommen können.«

»Das Beispiel mit der Achterbahn lässt sich direkt auf unsere Arbeitswelt übertragen: Wenn wir schneller werden wollen, denken wir zuerst daran, den Durchsatz zu erhöhen, und landen dann schnell bei der Idee, mehr Leute zu integrieren.«

»Und das funktioniert regelmäßig nicht gut. Und trotzdem wird es immer wieder versucht«, ergänzt Rob. »Eigentlich müsste man jeden in der Produktentwicklung zwingen, das Buch ›The Mythical Man-Month‹ von Frederick Brooks zu lesen.[1] Brooks's Law ›Adding manpower to a late software project makes it later‹ müsste eigentlich an jeder Tür stehen.«

»Genau«, fährt Florence fort. »Little's Law betrachtet die Situation relativ mechanisch und berücksichtigt nicht, dass neue Leute eingearbeitet werden müssen und mehr Leute zumindest anfänglich den Durchsatz reduzieren. Dabei gibt es noch einen zweiten Hebel: Wir reduzieren den Work in Progress. Dazu brauchen wir niemanden einzustellen und einzuarbeiten. Dieser Hebel ist meist preisgünstiger und wirkt schneller«, sagt Florence.

»Unsere Annahme wäre also, dass wir die Beschleunigung durch Reduktion des Work in Progress bewirkt haben, ohne jemals über Work in Progress gesprochen zu haben«, schlussfolgert Rob.

»Wir können das überprüfen, indem wir unsere Tabelle der Durchlaufzeiten um den Work in Progress ergänzen«, sagt Florence. Sie holt das Blatt mit der Tabelle aus ihrer Tasche, breitet es zwischen ihnen auf dem Tisch aus und ergänzt die Tabelle um drei Zeilen: eine für den minimalen, eine für den maximalen und eine für den durchschnittlichen Work in Progress.

»Aber dazu müssten wir uns auch noch anschauen, welche User Stories die Teams wann parallel in Bearbeitung hatten«, merkt Rob an. »Das geht bestimmt über die Daten, die wir im Ticketsystem haben. Ich kann mir das morgen Vormittag mal ansehen.«

»Ich glaube nicht, dass das notwendig ist. Uns interessieren nur die übergreifenden Funktionalitäten. Dann können wir anhand der Abhängigkeitsgraphen erkennen, wie viele von ihnen gleichzeitig in Arbeit waren«, erklärt Florence.

Rob ergänzt: »Wenn wir uns die Abhängigkeitsgraphen ansehen, analysieren wir, was wir geplant haben, und nicht das, was passiert ist. Die Realität hält sich leider ungern an unsere Pläne. Der Work in Progress wird in der Praxis vermutlich etwas anders gewesen sein, als die Graphen suggerieren. Für eine

1. Siehe [Brooks 1995].

erste Überprüfung unserer Annahme ist die Arbeit mit den Abhängigkeitsgraphen aber vermutlich gut genug. Wir können uns die realen Daten bei Bedarf später noch ansehen.«

»Okay, dann lass uns die drei neuen Tabellenzeilen ausfüllen«, sagt Florence. »In der ersten Variante ist der Durchschnitt leicht zu berechnen. Ich summiere einfach die Work-in-Progress-Werte je Sprint und teile die Summe durch die Anzahl der Sprints, also sechs. Für die dann folgenden Abhängigkeitsgraphen ist das nicht mehr ganz so einfach, weil der Work in Progress je Sprint schwankt. Ich visualisiere also, wann neue Arbeit begonnen wurde, und nehme das als eine Art virtuelle Zeitscheibe.«

Florence holt nun auch die Übersicht aus ihrer Tasche, legt sie vor sich auf den Tisch und zeichnet gestrichelte Linien in den Abhängigkeitsgraphen, um die Zeitscheiben zu visualisieren. Den Work in Progress je Zeitscheibe schreibt sie darunter.

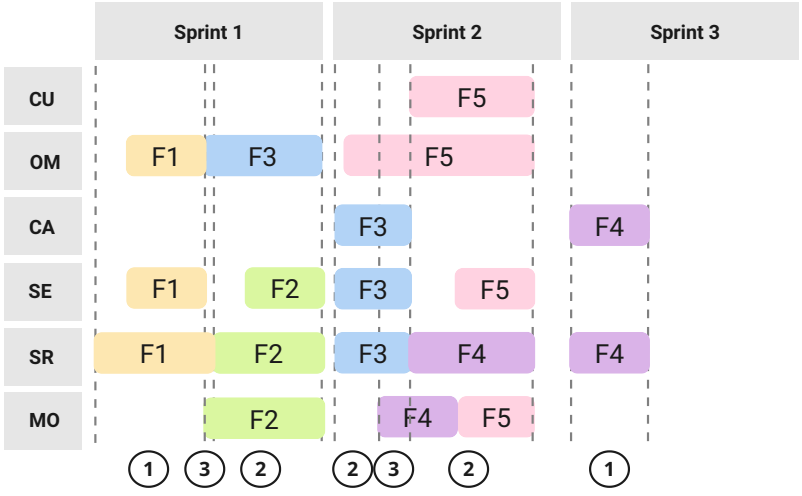


Abb. 2–8 *Zeitscheiben zur Ermittlung des Work in Progress*

Sie erklärt: »So kriegen wir im zweiten Abhängigkeitsgraphen sieben Zeitscheiben, obwohl wir nur drei Sprints benötigen.«

Dann trägt sie die Kennzahlen in die Tabelle ein.

	Blockierende Abhängigkeiten		Nicht blockierende Abhängigkeiten	
	Sprints mit Warten auf das Sprint-Ende	Sprints ohne Warten auf das Sprint-Ende	Sprints bei möglichst frühem Start	Sprints bei Just-in-time-Start
Durchlaufzeit min.	3	2	1	1
Durchlaufzeit max.	5	4	3	2
Durchlaufzeit ∅	4,2	3	1,8	1,4
Durchlaufzeit alle	6	4	3	3
Work in Progress min.	2	2	1	1
Work in Progress max.	5	4	4	3
Work in Progress ∅	3	2,7	2,7	2

»These bestätigt, würde ich sagen«, freut sich Rob. »Der durchschnittliche Work in Progress ist kontinuierlich gesunken.«

»Aber wenn man genau hinschaut, passt es doch nicht richtig«, widerspricht Florence.

»Was ist denn jetzt schon wieder?«, wundert sich Rob.

»Vergleiche doch mal die durchschnittliche Durchlaufzeit mit dem durchschnittlichen Work in Progress!«, fordert Florence ihn auf. »Den größten Sprung in der Durchlaufzeit haben wir von Variante 2 zu Variante 3, beim Work in Progress aber beim Übergang von Variante 3 zu Variante 4.«

»Wie kann das denn sein?«, fragt Rob. »Gibt es einen dritten Hebel, den der gute Herr Little in seiner Formel vergessen hat?«

»Wohl kaum«, antwortet Florence grübelnd.

»Ich hab's«, ruft Rob begeistert aus. »Wir haben den Durchsatz erhöht!«

»Aber wir haben doch gar nicht mehr Leute in die Teams gesteckt. Oder haben die Teams mehr gearbeitet? Überstunden gemacht?«

»Nein, viel einfacher.« Rob grinst. »Wir haben die Arbeit an den übergreifenden Funktionalitäten zeitlich verdichtet. Das ging mit Sicherheit zulasten der übrigen User Stories auf Teamebene. Dadurch haben wir mehr Kapazität für die übergreifenden Funktionalitäten geschaffen.«

Florence ist baff: »Du hast recht. Sehr schlau analysiert. Wir können den Durchsatz also nicht nur durch mehr Leute steigern, sondern auch durch Priorisierung. Damit schaffen wir mehr Kapazität für übergreifende Funktionalitäten.«

2.11 Koordinationsprinzipien

Nach wenigen Wochen hat sich das neue Vorgehen ohne Etappenplanung in den Teams eingeschwungen und Florence, Alex und Rob verabreden sich zu einer Reflexion des bisher Erreichten. Alle drei sind sehr zufrieden mit den Ergebnissen und ihrer Zusammenarbeit.

»Ich frage mich, ob wir aus dem, was wir gemacht haben, allgemeine Prinzipien ableiten können«, sagt Alex. »Darüber könnten wir das Erlernte besser im Unternehmen verbreiten und die Chance erhöhen, dass in Zukunft dezentral Entscheidungen so getroffen werden, dass sie mit den Prinzipien harmonieren.«

»Dann lasst uns mal rekapitulieren«, sagt Florence. »Zuerst haben wir dafür gesorgt, dass der Beitrag der Teams zu einer übergreifenden Funktionalität nicht bis zum Sprint-Ende wartet, bis das nächste Team daran weiterarbeitet. Danach haben wir die Arbeitsweise der Teams so umgestellt, dass die einzelnen Beiträge der Teams zu den übergreifenden Funktionalitäten parallel erstellt werden können. In dem Zuge sind wir vom Abhängigkeitsgraphen zur Abhängigkeitsmatrix übergegangen und haben die Etappenplanung abgeschafft.«

»Unser erster Schritt erinnert mich an die Prinzipien der *Lean Production*. Dort möchte man nicht, dass Zwischenprodukte herumliegen. Sie sollen möglichst schnell weiterverarbeitet werden. Das Ziel ist Flow in der Arbeit«, führt Rob aus.

Alex schreibt auf: `1. + Flow herstellen, - Arbeit warten lassen`

»Was soll das mit dem Plus- und Minuszeichen?«, fragt Florence verwirrt.

»Ah, da ist wohl der Nerd mit mir durchgegangen.« Alex schmunzelt. »Das Pluszeichen bedeutet: mehr. Das Minuszeichen heißt: weniger. Aber ich kann das auch ausschreiben.«

»Ich finde kurze, prägnante Darstellungen gut«, bremst ihn Florence. »Aber vielleicht geht es noch etwas intuitiver. Wie wäre es mit einem Pfeil nach oben und einem Pfeil nach unten?«

»Das gefällt mir«, sagt Alex und ändert das erste Prinzip: *1. Flow herstellen↑, Arbeit warten lassen↓*

»Sehr schön«, sagt Florence.

Rob fährt fort: »Und danach haben wir die Art der Abhängigkeiten geändert. Von blockierend zu nicht blockierend.«

Alex schreibt als zweiten Punkt auf: *2. Nicht blockierende Abhängigkeiten↑, blockierende Abhängigkeiten↓*

»Und schließlich haben wir ein unnötiges Meeting abgeschafft«, kommt Rob zum dritten Punkt.

Alex ergänzt seine Liste: *3. Unnötige Meetings abschaffen*

»Hm, dass man unnötige Meetings abschafft, scheint mir zu offensichtlich. Außerdem passt das Prinzip nicht in die Systematik mit Pfeil nach oben und Pfeil nach unten«, gibt Florence zu bedenken.

»Vielleicht lautet das Prinzip einfach ›weniger ist mehr‹«, wirft Alex ein. »Das könnte man ganz lustig so aufschreiben: 3. Weniger↑, Mehr↓«

»Supernerdig«, bescheinigt ihm Florence grinsend.

Rob merkt an: »Damit haben wir zwar die Pfeilsystematik gerettet, aber inhaltlich geht es nicht über eine Kalenderblattweisheit hinaus.«

»Da hast du leider recht«, bestätigt Alex. »Dann lasst uns noch einmal nachdenken. Wir konnten auf die Etappenplanung verzichten, weil wir uns kontinuierlich ausgetauscht haben. Tatsächlich haben wir ein großes, langes Meeting alle drei Monate durch tägliche, kleine Meetings ersetzt.«

»Also geht es eigentlich darum, den beplanten Zeithorizont zu verkürzen«, schlussfolgert Florence und schlägt sich direkt mit der flachen Hand vor die Stirn. »Wer hätte das gedacht. Darum geht es doch die ganze Zeit bei den ganzen agilen Ansätzen: kürzere Zeithorizonte, um flexibler auf Veränderungen reagieren zu können.«

»Stimmt. Bisher lag unser Fokus nur darauf, dass wir dadurch schneller werden. Als Seiteneffekt werden wir sogar reaktionsfähiger«, bemerkt Rob.

»Dann müsste das Prinzip, das wir suchen, eher so lauten«, ergreift Alex wieder das Wort und schreibt: *3. Kürzere Zeithorizonte↑, längere Zeithorizonte↓*

»Das ist vielleicht noch nicht perfekt, aber für den Moment sollte die Formulierung so reichen«, meint Florence. »Mir fällt gerade noch ein zusätzlicher Aspekt zur Etappenplanung ein. Der Impuls, das Meeting abzuschaffen, kam doch von den Teams, oder?«

»Ganz genau«, bestätigt Alex.

»Daraus können wir ein viertes Prinzip ableiten. Diejenigen, die die Arbeit machen, legen fest, wie sie sich koordinieren«, fährt Florence fort. Sie ergänzt Alex' Liste: *4. Koordinationstechniken im Besitz der Teams↑, Koordinationstechniken von außen vorgegeben↓*

»Die Liste braucht noch eine Überschrift«, sagt Rob. »Es hat alles mit der Koordination zwischen Teams zu tun. Daher schlage ich einfach ›Koordination‹ vor. Und lasst mich die Texte noch etwas umformatieren.«

Alex und Florence nicken zustimmend, Rob macht sich an die Arbeit und dann schauen die drei zufrieden auf die vier gefundenen Prinzipien:

Koordination

1. ↑ *Flow herstellen*
   ↓ *Arbeit warten lassen*

2. ↑ *Nicht blockierende Abhängigkeiten*
   ↓ *Blockierende Abhängigkeiten*

3. ↑ *Kürzere Zeithorizonte*
   ↓ *Längere Zeithorizonte*

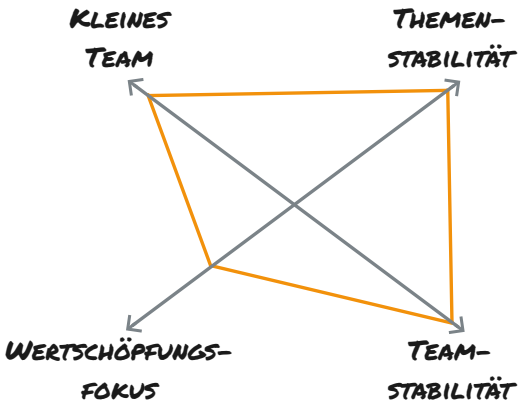
4. ↑ *Koordinationstechniken im Besitz der Teams*
   ↓ *Koordinationstechniken von außen vorgegeben*

6.5 Kohäsion maximieren: Unterschiedliche Teamansätze

Wir haben oben bewährte Techniken beschrieben, mit denen Teams innerhalb von Produkten oder Plattformen geschnitten werden können. Sie sind langfristig stabil und kümmern sich um ein statisches Themengebiet. Ich nenne sie daher statische Teams.[3] Sie sind fast immer ein guter Startpunkt und eine gute Basis für alternative Teamansätze. Neben einer Reflektion über statische Teams beschäftigt sich dieser Abschnitt mit mobilen Teams, Mission Teams und Floating Teams.

6.5.1 Statische Teams

Die oben beschriebenen Verfahren für den statischen Teamschnitt orientieren sich an der Wertschöpfung. Die einzelnen Teams können aber Wertschöpfung oft nicht vollständig verantworten. Dafür können die Teams klein sein, langfristig stabil bleiben und sich in einem Thema spezialisieren. Abbildung 6–13 zeigt die Charakteristik statischer Teams.

Statische Teams	
Statische Teams werden so nach stabilen fachlichen Kontexten gebildet, dass die Abhängigkeiten zwischen den Teams minimiert werden. Statische Teams sind ■ interdisziplinär besetzt, ■ maximal 9 Personen groß, ■ in ihrer Zusammensetzung langfristig stabil und ■ dauerhaft mit einem fachlichen Themenbereich beschäftigt.	

→

3. Das mag abwertend klingen, ist aber nicht so gemeint. Statische Teams haben eine Reihe sehr nützlicher Vorteile.

Vorteile	Herausforderungen
■ Langfristige Teamstabilität erhöht die Wahrscheinlichkeit für hochperformante Teams. ■ Langfristige Beschäftigung mit einem Thema führt zu Professionalisierung und damit zu effizienterer Arbeit in diesem Thema. ■ Langfristige Qualitätsverantwortung ist klar definiert und fördert das Qualitätsbewusstsein bei den Teams.	■ Kapazitäten der Teams werden nicht optimal genutzt, wenn es für einzelne Themenbereiche der Teams zurzeit keine hoch priorisierten Anforderungen gibt. ■ Abhängigkeiten zwischen statischen Teams können die Planbarkeit und Entwicklungsgeschwindigkeit erheblich reduzieren. ■ Um effizient mit den Abhängigkeiten umgehen zu können, müssen die Teams die Entwicklungspraktiken wie testgetriebene Entwicklung, Continuous Integration etc. gemeistert haben.

Abb. 6–13 *Steckbrief für statische Teams*

Bei statischen Teams wird man immer wieder mit Abhängigkeiten zu tun haben, die koordiniert werden müssen. In einigen Kontexten ist das kein Problem, in anderen kann es die Entwicklung sehr träge machen. Dann kann ein alternativer Teamansatz Abhilfe schaffen.

6.5.2 Mobile Teams

Agile Teams arbeiten mit T-Shaped Skill Sets ihrer Mitglieder. Die Teammitglieder sind jeweils in einem Bereich spezialisiert (z.B. Java-Backend-Entwicklung). Wenn allerdings alle Teammitglieder nur diese eine Spezialisierung mitbringen, kann kaum echte Kooperation entstehen. Zumindest wird die Arbeit immer wieder stocken, weil einzelne Spezialisierungen zum Engpass werden (siehe Abb. 6–14).

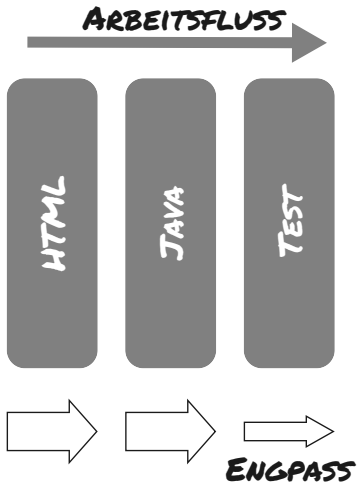


Abb. 6–14 *Engpass bei hochgradiger Spezialisierung*

Um die Kooperation zu verbessern und die Arbeit im Fluss zu halten, arbeiten sich Teammitglieder in zusätzliche Themen ein (z.B. HTML und Qualitätssicherung). Es entstehen T-Shaped Skill Sets (siehe Abb. 6–15). Sie werden in diesen Themen vermutlich keine Spezialisten werden, und die jeweiligen Spezialisten werden entsprechende Aufgaben effizienter durchführen können.

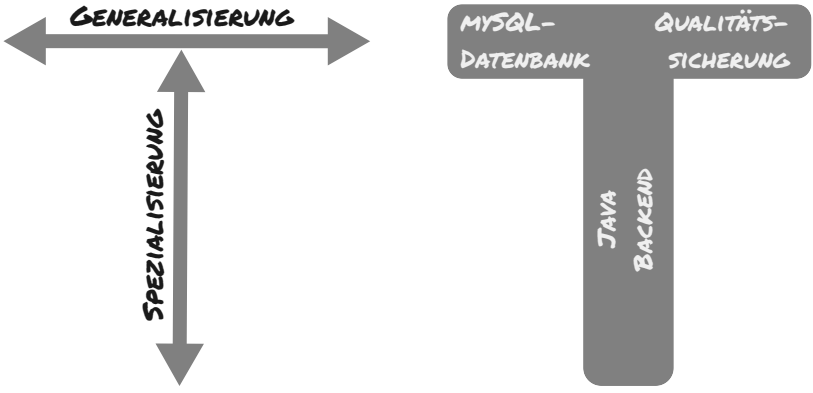


Abb. 6–15 *T-Shaped Skill Set für Teammitglieder*

Allerdings kann die Arbeit durch die T-Shaped Skill Sets im Fluss gehalten werden, und meist ist eine kurze Durchlaufzeit wertvoller als die erhöhten Kosten.

Diese Idee der T-Shaped Skill Sets lässt sich auch auf Teams übertragen. Die Teams haben ihr festes Thema (z.B. Suche) als Spezialisierung, arbeiten sich aber in andere Bereiche des Produkts ein (siehe Abb. 6–16). Dadurch wird es möglich, dass die Teams die übergreifenden Funktionalitäten vollständig autonom umsetzen können, ohne dass Übergaben zwischen den Teams notwendig sind.

Abb. 6–16 *T-Shaped Skill Sets für Teams*

Das LeSS-Framework spricht in diesem Zusammenhang von Feature Teams. Da der Begriff der Feature Teams in anderen Kontexten schlicht für Teams verwendet wird, die einzelne User Stories Ende-zu-Ende umsetzen können, nennen wir diese Teams *mobile Teams*. Sie sind mobil im ganzen Produkt unterwegs. Abbildung 6–17 zeigt die Charakteristik der mobilen Teams im Vergleich zu den statischen Teams.

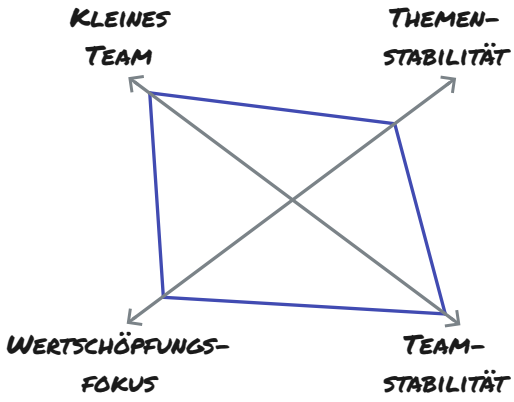
Mobile Teams	
Mobile Teams haben einen fachlichen Schwerpunkt, entwickeln übergreifende Funktionalität aber von Ende zu Ende. Mobile Teams sind ■ interdisziplinär besetzt, ■ maximal 9 Personen groß, ■ in ihrer Zusammensetzung langfristig stabil, ■ mobil im ganzen Produkt tätig und ■ für die Qualität ihres Themenbereichs verantwortlich.	
Vorteile	**Herausforderungen**
■ Teams bringen ihre Kapazität entlang der Prioritäten ein. ■ Langfristige Teamstabilität erhöht die Wahrscheinlichkeit für hochperformante Teams. ■ Es besteht eine wesentlich geringere Notwendigkeit, Abhängigkeiten zu managen, verglichen mit statischen Teams. ■ Langfristige Qualitätsverantwortung ist klar definiert und fördert das Qualitätsbewusstsein bei den Teams.	■ Die Teams müssen sich in mehr Themen einarbeiten. ■ Ein Mindestmaß an Einheitlichkeit im Produkt ist erforderlich. ■ Die Teams müssen sich untereinander so weit vertrauen, dass sie Änderungen am »eigenen« Code durch andere Teams tolerieren können.

Abb. 6–17 *Steckbrief für mobile Teams*

Mobile Teams sind ein wirksamer Mechanismus, um Abhängigkeiten zu vermeiden und Durchlaufzeiten zu minimieren. Sie haben aber anspruchsvollere Voraussetzungen als statische Teams:

- **Vertrauen:** Die Teams müssen akzeptieren, dass andere Teams in »ihrem« Produktbereich Änderungen vornehmen. Das kann nur dann gelingen, wenn die Teams untereinander Vertrauen in die Fähigkeiten und positiven Absichten haben.
- **Testautomatisierung:** Da die Teams Änderungen im Code vornehmen, mit dem sie nicht täglich arbeiten, steigt die Gefahr von Fehlern. Es braucht eine ausreichende Testautomatisierung, damit diese früh entdeckt werden können.
- **Einheitlichkeit:** Die Einarbeitung in »fremde« Produktbereiche ist nur dann realistisch, wenn das Produkt ausreichend einheitlich ist. Wenn jede Produktkomponente in einer anderen Programmiersprache geschrieben ist, stößt der Ansatz der mobilen Teams schnell an seine Grenzen.

Wenn man bereits statische Teams hat und mit mobilen Teams arbeiten möchte, muss man nicht alle Teams in mobile Teams umwandeln. Man kann erst mal mit einem Team anfangen, um Erfahrungen mit dem Ansatz zu machen.

6.5.3 Mission Teams

Mission Teams sind ein alternativer Ansatz zu mobilen Teams. Mission Teams kümmern sich exklusiv um eine Mission, die typischerweise 2-4 Monate dauert. Das Mission Team bleibt nur für diesen Zeitraum zusammen und die Teammitglieder gehen nach Abschluss der Mission zurück in ihre Stammteams.

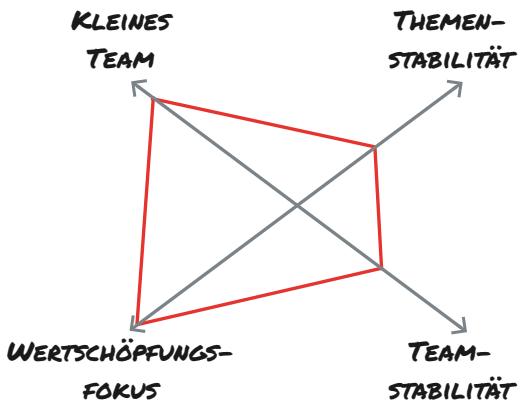
Damit Mission Teams nicht die Dysfunktionen erzeugen, die in einer reinen Projektwelt immer wieder zu beobachten sind, brauchen sie eine stabile Basis. Statische oder mobile Teams können diese Basis bilden (siehe Abb. 6–18).

Abb. 6–18 *Rekrutierung für ein Mission Team*

Mission Teams eignen sich gut, um eine größere Funktionalität oder strategische Initiative fokussiert umzusetzen.

Im Gegensatz zu mobilen Teams sind Mission Teams nicht so sehr auf die Einheitlichkeit des Produkts angewiesen. Durch die Rekrutierung aus statischen Teams können in das Mission Team die Fähigkeiten integriert werden, die für die Mission notwendig sind.

Mission Teams stellen also Wertschöpfung (die Mission) in den Vordergrund und nehmen dafür Abstriche bei der Teamstabilität in Kauf. Daraus leitet sich die größte Herausforderung bei der Arbeit mit Mission Teams ab: Teambuilding. Es muss gelingen, dass das Team schnell ins Performing (siehe [Stahl 2017]) kommt und nicht Wochen im Storming verbringt. Schnelles Teambuilding wird durch ein hohes Maß an Vertrauen unter den Beteiligten gefördert. Abbildung 6–19 zeigt die Charakteristik von Mission Teams.

Mission Teams	
Mission Teams werden für eine Mission gebildet und lösen sich danach wieder auf. Mission Teams sind ■ interdisziplinär besetzt, ■ maximal 9 Personen groß und ■ in der Regel 3-4 Monate stabil.	

→

Vorteile	Herausforderungen
■ Absoluter Fokus im Team auf die Mission. ■ Es ist weniger Einheitlichkeit im Produkt erforderlich, da die notwendigen Spezialisierungen durch die Besetzung des Teams sichergestellt werden können. ■ Es besteht eine wesentlich geringere Notwendigkeit zum Management von Abhängigkeiten im Vergleich zu statischen Teams.	■ Geringere Teamstabilität macht Teambuilding herausfordernd. ■ Nur ein generell hohes Maß an Vertrauen unter allen potenziellen Teammitgliedern liefert eine gute Basis für Teambuilding. ■ Mission Teams können die langfristige Qualitätsverantwortung für ihre Entwicklung nicht übernehmen. Es muss Klarheit geschaffen werden, wo die Verantwortung liegen wird, und ggf. braucht es einen Übergabeprozess vom Mission Team dorthin.

Abb. 6–19 *Steckbrief für Mission Teams*

Die Firma Pipedrive hatte ursprünglich mit statischen Teams gearbeitet und irgendwann festgestellt, dass sie strategisch wichtige Weiterentwicklungen des Produkts nicht mehr schnell umgesetzt bekamen (siehe [Appelo 2022]). Pipedrive hat dann eine neue Struktur gebildet: Tribes – bestehend aus 20–25 Personen – verantworten ein Funktionsbündel wie Statistiken oder das Kampagnenmanagement. Die Basisstruktur im Tribe ist das Launchpad. Es ist für Betrieb und Wartung der Funktionen des Tribes verantwortlich. Aus dem Launchpad werden Missionen gestartet, die relevante neue Funktionalitäten implementieren. Die Mitglieder der Missionen werden aus dem Launchpad rekrutiert und gehen nach Abschluss der Mission zurück ins Launchpad. Im Launchpad verbleiben ca. 20–40 % der Tribe-Mitglieder.

6.5.4 Floating Teams

Floating Teams entstehen, wenn man den Mission-Team-Gedanken ins Extrem treibt. Beim Floating-Team-Ansatz übernimmt ein Team mit einer Anzahl von Mitgliedern, die deutlich oberhalb üblicher Teamgrößen liegt, die Gesamtverantwortung für ein Produkt oder ein Funktionsbündel. Aus diesem Team bilden sich kurzlebige *Story Teams*, die sich um eine User Story kümmern und nur wenige Tage zusammenbleiben. Nach bisherigen Erfahrungen waren die Gesamtteams 20–35 Personen groß; die Story Teams haben typischerweise drei bis vier Mitglieder.

Inhaltsverzeichnis