Einleitung

Dieses Buch ist keine Forschungsarbeit. Ich habe die vorhandene Literatur nicht gewissenhaft gesichtet. Was Sie lesen werden, sind meine persönlichen Rückbesinnungen, Beobachtungen und Ansichten basierend auf meiner zwanzigjährigen Beschäftigung mit agiler Softwareentwicklung – nicht mehr und nicht weniger.

Der Schreibstil ist erzählerisch und umgangssprachlich. Meine Wortwahl ist manchmal etwas ungeschliffen. Auch wenn ich eigentlich nicht zum Fluchen neige, findet sich hier und da vielleicht ein (mildes) Schimpfwort, weil mir kein besserer Weg eingefallen ist, meine Absichten zu vermitteln.

Zudem ist dieses Buch keine reine Lobhudelei. Wenn es mir notwendig erschien, habe ich Verweise auf Referenzen hinzugefügt, denen Sie folgen können. Ich habe meine Erkenntnisse mit denen anderer verglichen, die der Community der agilen Softwareentwicklung ebenso lange angehören wie ich. Einige habe ich sogar darum gebeten, für eigene Kapitel und Abschnitte ergänzende und entgegengesetzte Sichtweisen bereitzustellen. Dennoch sollten Sie das Buch nicht für eine akademische Arbeit halten. Es ist wohl besser, es als persönliche Aufzeichnung zu betrachten – als das Grummeln eines Griesgrams, der die nächste Generation agiler Softwareentwickler von seinem Land vertreiben will.

Das Buch richtet sich sowohl an Programmierer als auch an Nicht-Programmierer. Es ist nicht technisch; es gibt keinen Code. Es soll einen Überblick über die ursprüngliche Absicht agiler Softwareentwicklung verschaffen, ohne sich in technischen Details der Programmierung, des Testens und des Managens zu verlieren.

Das Buch ist ziemlich kurz, weil das Thema nicht sehr umfassend ist. Agile Softwareentwicklung ist ein einfaches Konzept für einfache Aufgaben, die kleine Teams von Programmierern lösen. Agile Softwareentwicklung ist *kein* großes Konzept zum Lösen umfassender Aufgaben durch große Teams. Es ist schon etwas paradox, dass diese einfache Lösung für ein kleines Problem einen eigenen Namen trägt. Das fragliche kleine Problem wurde schließlich schon in den 1950er- und 1960er-Jahren gelöst, fast unmittelbar, nachdem Software erfunden wurde. Damals lernten kleine Softwareteams, einfache Aufgaben ziemlich gut zu lösen. Das änderte sich in den 1970er-Jahren, als kleine Softwareteams, die einfache Aufgaben lösten, in eine Ideologie verwickelt wurden, die den Standpunkt vertrat, dass große Teams umfassende Aufgaben lösen sollten.

Sollten wir umfassende Aufgaben mit großen Teams lösen? Ach du lieber Himmel, nein! Umfassende Aufgaben lassen sich nicht von großen Teams lösen. Umfassende Aufgaben werden durch die Zusammenarbeit vieler kleiner Teams gelöst, die einfache Aufgaben lösen. Das war den Programmierern in den 1950er- und 1960er-Jahren instinktiv klar. Und das wurde in den 1970er-Jahren vergessen.

Weshalb wurde es vergessen? Ich hege den Verdacht, dass es zu einem Bruch kam. Die Anzahl der Programmierer nahm in den 1970er-Jahren explosionsartig zu. Vorher gab es weltweit nur ein paar Tausend Programmierer, nachher waren es mehrere Hunderttausend. Mittlerweile nähert sich diese Zahl hundert Millionen.

Die ersten Programmierer in den 1950er- und 1960er-Jahren waren keine jungen Leute. Sie lernten mit 30, 40 oder 50 zu programmieren. Aber in den 1970er-Jah-

ren, als die Zahl der Programmierer drastisch zunahm, gingen die alten Hasen allmählich in den Ruhestand. Die erforderliche Ausbildung des Nachwuchses fand also nie statt. Ein Heer junger Leute in den Zwanzigern betrat den Arbeitsmarkt genau in dem Moment, in dem die erfahrenen Mitarbeiter in den Ruhezustand gingen. Ihre Erfahrung wurde also faktisch nicht weitergegeben.

Manche Leute behaupten, dass dadurch eine Art »dunkles Zeitalter« der Programmierung eingeläutet wurde. 30 Jahre lang kämpften wir mit der Vorstellung, dass umfassende Aufgaben von großen Teams erledigt werden müssen, ohne zu wissen, dass der Trick darin besteht, dass viele kleine Teams viele kleine Aufgaben erledigen.

Mitte der 1990er-Jahre wurde uns allmählich klar, dass wir den Kampf verloren hatten. Das Konzept kleiner Teams keimte auf und gedieh. Es verbreitete sich in der Community der Softwareentwickler und nahm Fahrt auf. Im Jahr 2000 stellten wir fest, dass wir einen branchenweiten Neustart brauchten. Wir mussten daran erinnert werden, was unsere Vorgänger instinktiv wussten. Wir mussten abermals zur Kenntnis nehmen, dass umfassende Aufgaben von vielen zusammenarbeitenden kleinen Teams erledigt werden, die kleine Aufgaben erledigen.

Um dieser Vorstellung zum Durchbruch zu verhelfen, gaben wir dem Konzept einen Namen. Wir nannten es »agile Softwareentwicklung«.

Ich habe dieses Vorwort Anfang 2019 verfasst. Der »Neustart« im Jahre 2000 ist nun fast zwei Jahrzehnte her und ich habe den Eindruck, dass ein weiterer Neustart erforderlich ist. Warum? Weil die schlichte und einfache Botschaft, die agile Softwareentwicklung vermitteln soll, in all den Jahren durcheinandergeraten ist. Sie wurde mit Konzepten wie Lean, Kanban, LeSS, SAFe, Modern, Skilled und vielen anderen vermengt. Diese anderen Konzepte sind auch nicht schlecht, haben aber nichts mit der ursprünglichen Botschaft agiler Softwareentwicklung zu tun.

Es ist also wieder an der Zeit, dass wir uns daran erinnern, was unsere Vorgänger in den 1950er- und 1960er-Jahren wussten und was wir schon im Jahr 2000 abermals lernen mussten. Wir müssen uns daran erinnern, was agile Softwareentwicklung tatsächlich ist.

In diesem Buch werden Sie nichts finden, was besonders neu, überraschend oder verblüffend ist, und nichts Revolutionäres, das den üblichen Rahmen sprengt, sondern eine Neuformulierung der agilen Softwareentwicklung, wie man sie aus dem Jahr 2000 kennt. Sie wird allerdings aus einer anderen Perspektive betrachtet, und wir haben in den letzten zwanzig Jahren ein paar Dinge gelernt, die ich ebenfalls erörtern werde. Im Großen und Ganzen entspricht die Botschaft dieses Buchs aber denjenigen aus den Jahren 2001 und 1950.

Diese Botschaft ist schon alt, aber sie ist wahr. Sie liefert uns eine einfache Lösung für das kleine Problem, dass kleine Teams kleine Aufgaben erledigen.

Kapitel 1

Einführung in agile Softwareentwicklung

Im Februar 2001 versammelten sich in Snowbird, Utah, 17 Softwareexperten, um über den beklagenswerten Zustand der Softwareentwicklung zu diskutieren. Damals wurde die meiste Software mithilfe von ineffektiven, schwergewichtigen und immer gleichbleibenden Verfahren erstellt, wie etwa nach dem Wasserfallmodell oder gemäß RUP (*Rational Unified Process*). Die 17 Experten hatten zum Ziel, ein Manifest zu erstellen, das einen effektiveren, leichtgewichtigeren Ansatz beschreibt.

Das war kein leichtes Unterfangen. Die 17 Experten hatten ganz unterschiedliche Erfahrungen gesammelt und vertraten sehr verschiedene Meinungen. Es war sehr unwahrscheinlich, dass solch eine Gruppe zu einem Konsens kommen würde.

Und doch wurde entgegen allen Erwartungen ein Konsens erzielt, das agile Manifest. Damit wurde eine der einflussreichsten und langlebigsten Bewegungen auf dem Gebiet der Softwareentwicklung geboren.

Solche Bewegungen in der Softwareentwicklung verlaufen immer vorhersehbar. Am Anfang gibt es eine Minderheit begeisterter Unterstützer, eine weitere Minderheit enthusiastischer Kritiker und eine große Mehrheit, der das Ganze egal ist. Viele dieser Bewegungen verlaufen in dieser Phase im Sande oder kommen nie über sie hinaus. Denken Sie nur an aspektorientierte Programmierung, logische Programmierung oder CRC-Karten (Class-Responsibility-Collaboration-Karten). Einige jedoch schaffen den Sprung, erfreuen sich außerordentlicher Beliebtheit und werden kontrovers diskutiert. In manchen Fällen wird auch diese Kontroverse überwunden, und die Bewegung wird zum allgemeinen Gedankengut. Für Letzteres ist Objektorientierung (OO) ein gutes Beispiel. Ebenso wie agile Softwareentwicklung.

Ist eine solche Bewegung erst einmal groß geworden, kommt es durch Missverständnisse und Usurpation leider schnell dazu, dass ihre Bezeichnung unscharf wird. Produkte und Verfahren, die nichts mit der Bewegung zu tun haben, verwenden den Begriff, um aus seiner Bekanntheit und Signifikanz Profit zu schlagen. So war es auch bei der agilen Softwareentwicklung.

Der Zweck dieses Buchs, das fast zwei Jahrzehnte nach dem Treffen in Snowbird entstand, ist es, das richtigzustellen. Es ist der Versuch, so pragmatisch wie möglich zu sein und agile Softwareentwicklung ohne Nonsens und ohne unklare Begriffe zu beschreiben.

In diesem Buch werden die Grundlagen der agilen Softwareentwicklung vorgestellt. Diese Konzepte wurden in der Vergangenheit oftmals erweitert und ausgeschmückt – und daran ist auch nichts verwerflich. Allerdings sind solche Erweiterungen und Ausschmückungen keine agile Softwareentwicklung. Es handelt sich vielmehr um agile Softwareentwicklung plus irgendetwas anderes. In diesem Buch werden Sie lesen, was agile Softwareentwicklung ist, was sie war und was sie zwangsläufig immer sein wird.

1.1 Ursprung der agilen Softwareentwicklung

Wann nahm die agile Entwicklung ihren Anfang? Vermutlich vor mehr als 50.000 Jahren, als Menschen sich erstmals entschlossen, zusammenzuarbeiten, um ein gemeinsames Ziel zu erreichen. Die Vorstellung, einfachere Zwischenziele zu verfolgen und nach ihrem Erreichen den Fortschritt zu beurteilen, ist schlicht und einfach zu naheliegend und zu menschlich, um als irgendeine Art von Revolution betrachtet zu werden.

Und wann nahm die agile Softwareentwicklung in der modernen Wirtschaft und Gesellschaft ihren Anfang? Das ist schwer zu sagen. Ich denke, die erste Dampfmaschine, die erste Mühle, der erste Verbrennungsmotor und das erste Flugzeug sind durch Verfahren zustande gekommen, die wir heute als agil bezeichnen würden. Der Grund dafür ist, dass es einfach zu naheliegend und menschlich ist, in kleinen, messbaren Schritten vorzugehen, sodass es gar nicht auf andere Weise hätte geschehen können.

Wann also nahm die agile Softwareentwicklung ihren Anfang? Ich hätte gerne Mäuschen gespielt, als Alan Turing 1936 seine Arbeit verfasst hat.[1] Ich nehme an, dass er die vielen »Programme« in dieser Arbeit in kleinen Schritten mit jeder Menge informeller Tests, den sogenannten *Desk Checks* (auch: *Schreibtischtests*) entwickelt hat. Den ersten Code, den er 1946 für die *Automatic Computing Engine* geschrieben hat, wird er wohl auch in kleinen Schritten entwickelt haben, wieder mit vielen Desk Checks und vielleicht sogar mit einigen richtigen Tests.

In der Anfangszeit der Softwareentwicklungen finden sich viele Beispiele für ein Verhalten, das wir heute als agile Softwareentwicklung bezeichnen würden. So haben beispielsweise die Programmierer, die die Steuerungssoftware für die Mercury-Raumkapsel schrieben, in halbtäglichen Schritten gearbeitet, die durch Unit-Tests unterbrochen wurden.

Über diese Zeit wurde an anderen Stellen schon viel veröffentlicht. Craig Larmann und Vic Basili haben eine Historie geschrieben, die in Ward Cunninghams Wiki[2] und auch in Larmans Buch *Agile & Iterative Development: A Manager's Guide* zusammengefasst ist.[3]

Aber die agile Softwareentwicklung war nicht die einzige Hochzeit, auf der man tanzen konnte. Tatsächlich gab es ein konkurrierendes Verfahren, das im verarbeitenden Gewerbe und in der Großindustrie beträchtlichen Erfolg hatte: *Scientific Management* (zu Deutsch etwa: wissenschaftliche Betriebsführung).

Scientific Management ist ein Top-down-Ansatz zur Führung. Manager verwenden solche Verfahren, um zu gewährleisten, dass zum Erreichen eines Ziels die besten verfügbaren Methoden eingesetzt werden, und weisen die Mitarbeiter an, dem Plan exakt zu folgen. Oder anders ausgedrückt: Es gibt eine umfassende vorherige Planung, der eine sorgfältige und detaillierte Umsetzung folgt.

---

1 Turing, A. M. 1936. On computable numbers, with an application to the Entscheidungsproblem [proof]. *Proceedings of the London Mathematical Society*, 2 (1937 veröffentlicht), 42(1):230–65. Am besten lässt sich diese Arbeit verstehen, wenn man Charles Petzolds Meisterwerk liest: Petzold, C. 2008. *The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine*. Indianapolis, IN: Wiley.

2 Wards wiki, `c2.com`, ist das allererste Wiki, das im Internet auftauchte. Möge es noch lange erreichbar sein.

3 Larman, C. 2004. *Agile & Iterative Development: A Manager's Guide*. Boston, MA: Addison-Wesley.

Scientific Management ist wohl schon so alt wie die Pyramiden, Stonehenge oder die anderen großen Werke der Antike, weil es einfach unmöglich ist, zu glauben, dass sie ohne es entstanden sein könnten. Ich wiederhole es noch einmal: Das Konzept, einen erfolgreichen Vorgang zu wiederholen, ist zu naheliegend und zu menschlich, um es als revolutionär zu betrachten.

Die Bezeichnung *Scientific Management* geht auf die Arbeiten von Frederick Winslow Taylor in den 1880er-Jahren zurück. Er formalisierte und kommerzialisierte diesen Ansatz und machte als Management-Berater ein Vermögen. Das Verfahren war äußerst erfolgreich und führte in den nachfolgenden Jahrzehnten zu einer deutlichen Erhöhung der Effizienz und der Produktivität.

Und so kam es dazu, dass die Softwarewelt 1970 vor dem Scheideweg dieser beiden gegensätzlichen Verfahren stand. Beim prä-agilen Verfahren (agile Softwareentwicklung, bevor sie so bezeichnet wurde) werden kleine, reagierende Schritte unternommen, die bewertet und verfeinert werden, um durch zufällige Bewegungen in Richtung eines guten Ergebnisses vorwärtszukommen. Beim Scientific Management hingegen werden Aktionen verschoben, bis durch gründliche Analyse ein detaillierter Plan erstellt worden ist. Prä-agile Verfahren sind gut für Projekte geeignet, bei denen Änderungen nicht aufwendig sind und die teilweise definierte Aufgaben mit zwanglos festgelegten Zielen lösen. Scientific Management funktioniert am besten, wenn Projekte, bei denen Änderungen sehr aufwendig sind, genau definierte Aufgaben mit äußerst präzisen Zielen lösen.

Nun stellte sich die Frage: Zu welcher Kategorie gehörten Softwareprojekte zu diesem Zeitpunkt? Waren Änderungen sehr aufwendig und die Ziele präzise festgelegt oder waren Änderungen nicht aufwendig und die Ziele nur teilweise und zwanglos definiert?

Messen Sie dem letzten Absatz keine allzu große Bedeutung bei. Nach meinem Wissen stellt tatsächlich niemand diese Frage. Der Weg, den wir 1970 einschlugen, wurde ironischerweise offenbar eher zufällig als absichtlich ausgewählt.

1970 veröffentlichte Winston Royce eine Arbeit[4], die sein Konzept zur Verwaltung umfangreicher Softwareprojekte beschreibt. Die Arbeit enthielt ein Diagramm (Abbildung 1.1), das seinen Plan veranschaulichte. Royce hatte dieses Diagramm nicht selbst erstellt und er befürwortete das dargestellte Vorgehen auch nicht. Tatsächlich diente ihm das Diagramm nur als Vorwand, um es auf den nachfolgenden Seiten seiner Arbeit auseinandernehmen zu können.

4 Royce, W. W. 1970. Managing the development of large software systems. *Proceedings, IEEE WESCON*. August: 1–9. `http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf`.

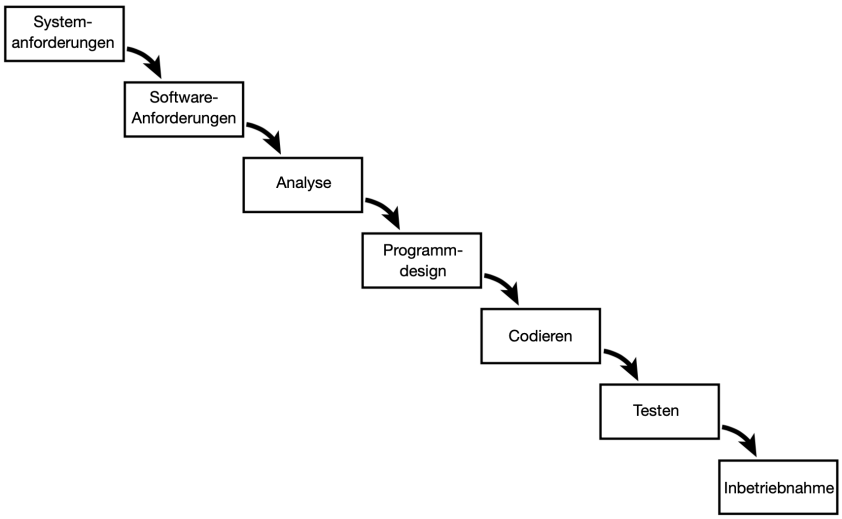

Abb. 1.1: Das Diagramm von Winston Royce, auf dem das Wasserfallmodell beruht

Dessen ungeachtet führten die Platzierung an prominenter Stelle und die Neigung der Leser, aus einem Diagramm auf der ersten oder zweiten Seite einer Arbeit auf ihren Inhalt zu schließen, zu einer drastischen Umwälzung in der Softwarebranche.

Das Diagramm hat große Ähnlichkeit mit Wasser, das eine Reihe von Steinen herabfließt, deshalb wurde das Verfahren unter der Bezeichnung *Wasserfallmodell* bekannt.

Das Wasserfallmodell war der logische Nachfolger des Scientific Managements. Es geht vor allem darum, eine gründliche Analyse vorzunehmen, einen detaillierten Plan auszuarbeiten und diesen Plan in die Tat umzusetzen.

Royce hatte das Modell überhaupt nicht empfohlen, aber dieses Konzept war es, was die Leser seiner Arbeit entnahmen. Und es sollte die nächsten drei Jahrzehnte dominieren.[5]

Und hier fängt die eigentliche Geschichte an. 1970 war ich 18 Jahre alt und bei einer Firma namens A.S.C. Tabulating in Lake Bluff, Illinois, als Programmierer tätig. Das Unternehmen verfügte über eine IBM 360/30 mit 16K Kernspeicher, eine IBM 360/40 mit 64K Kernspeicher und einen Minicomputer des Typs Varian 620/f mit 64K Kernspeicher. Ich programmierte die 360er in COBOL, PL/1, Fortran und Assembler. Auf dem 620/f programmierte ich nur Assembler.

5 Hier ist anzumerken, dass meine Interpretation dieses Zeitraums infrage gestellt wurde, und zwar in Kapitel 7 des Buchs von Bossavit, L. 2012. *The Leprechauns of Software Engineering: How Folklore Turns into Fact and What to Do About It.* Leanpub.

Es ist wichtig, sich vor Augen zu führen, was es damals bedeutete, als Programmierer tätig zu sein. Wir schrieben unseren Code mit Bleistift auf Formulare, anhand derer Kartenstanzer-Operatoren Lochkarten für uns erstellten. Wir übergaben unsere sorgfältig überprüften Lochkarten an Computer-Operatoren, die unsere Programme und Tests während der dritten Schicht ausführten, weil die Computer tagsüber viel zu sehr damit beschäftigt waren, richtige Arbeit zu erledigen. Es dauerte oft mehrere Tage, um vom ursprünglichen Schreiben des Codes zur ersten Kompilierung zu gelangen. Und jede nachfolgende Aktualisierung dauerte für gewöhnlich einen Tag.

Bei der 620/f verhielt es sich für mich etwas anders. Die Maschine stand unserem Team zur Verfügung, sodass wir rund um die Uhr auf sie zugreifen konnten. Wir schafften zwei, drei, vielleicht auch mal vier Aktualisierungen oder Tests pro Tag. Das Team, zu dem ich gehörte, bestand aus Leuten, die – im Gegensatz zu heutigen Programmierern – in der Lage waren, Lochkarten zu stanzen. Wir konnten unsere Lochkartenstapel also selbst stanzen und waren nicht auf die Launen der Kartenstanzer-Operatoren angewiesen.

Welches Verfahren haben wir damals genutzt? Jedenfalls kein Wasserfallmodell. Wir hatten kein Konzept, einem detaillierten Plan zu folgen. Wir programmierten einfach so vor uns hin, kompilierten und testeten unseren Code und behoben Fehler. Es handelte sich um eine strukturlose Endlosschleife. Es war auch keine agile Softwareentwicklung, noch nicht einmal eine prä-agile. Unsere Arbeitsweise war undiszipliniert. Es gab keine Testsuiten oder vorgegebene Zeitintervalle. Wir programmierten und behoben Fehler, Tag für Tag, Monat für Monat.

Um 1972 habe ich zum ersten Mal in einer Fachzeitschrift etwas über das Wasserfallmodell gelesen. Es erschien mir als ein Glücksfall. Sollte es tatsächlich möglich sein, ein Problem vorab zu analysieren, eine Lösung dafür zu entwickeln und diese dann zu implementieren? Konnten wir wirklich anhand dieser drei Phasen einen Plan erstellen? Hätten wir nach Abschluss der Analyse tatsächlich ein Drittel des Projekts erledigt? Mir wurde die Leistungsfähigkeit dieses Konzepts bewusst. Ich wollte daran glauben. Denn wenn es funktionierte, würde das bedeuten, dass ein Traum wahr wird.

Ich war offenbar nicht der Einzige, denn viele andere Programmierer waren ebenfalls darauf aufmerksam geworden. Und wie gesagt dominierte das Wasserfallmodell allmählich unsere Denkweise.

Es war zwar vorherrschend, funktionierte aber nicht. Die nächsten 30 Jahre sollten ich, meine Kollegen und Programmierer rund um den Globus es immer wieder versuchen, die Analyse und die Fehlerbehebung richtig hinzubekommen. Aber immer wenn wir dachten, das Ziel erreicht zu haben, entglitt uns die Lösung in der Implementierungsphase wieder. Monate sorgfältiger Planung wurden durch irgendeinen unvermeidlichen kleinen Fehler hinfällig – vor den Augen der Manager und Kunden, nachdem die Abgabefrist schon krass überzogen worden war.

Doch trotz der praktisch nicht enden wollenden Reihe von Fehlschlägen blieben wir dem Wasserfallmodell treu. Was könnte schon falsch daran sein? Wie konnte eine gründliche Analyse des Problems, eine sorgfältige Entwicklung einer Lösung und die anschließende Implementierung immer wieder so spektakulär scheitern? Es war undenkbar, dass die Problematik etwas mit der Strategie zu tun hatte. Wir mussten das Problem sein. Irgendetwas machten wir falsch.

Das Ausmaß, in dem das Wasserfallmodell unsere Denkweise dominierte, zeigt sich in der damaligen Sprache. Nachdem Dijkstra 1968 strukturierte Programmierung vorgestellt hatte, folgten schon bald strukturierte Analyse[6] und strukturiertes Design[7]. Nachdem 1988 objektorientierte Programmierung (OOP) Verbreitung fand, folgten ebenfalls schon bald objektorientierte Analyse[8] und objektorientiertes Design[9] (OOD). Dieses Dreigestirn der Phasen hatte uns völlig in seinen Bann gezogen. Wir konnten uns schlicht und einfach keine andere Arbeitsweise vorstellen.

Und plötzlich konnten wir es doch.

Die Umstellung auf agile Softwareentwicklung nahm Ende der 1980er- oder Anfang der 1990er-Jahre ihren Anfang. In der Smalltalk-Community gab es Ende der 1980er-Jahre die ersten Anzeichen. Im 1991 erschienenen Buch von Booch gibt es ebenfalls Hinweise. Weitere Anzeichen finden sich in Cockburns Crystal Methods und die Design-Pattern-Community begann 1994, angespornt durch eine Arbeit von James Coplien[10], agile Softwareentwicklung zu diskutieren.

1995 veröffentlichten Beedle[11], Devos Sgaron, Schwaber und Sutherland ihre berühmte Arbeit über Scrum.[12] Nun gab es kein Halten mehr. Die Bastion des Wasserfallmodells war gefallen und es gab kein Zurück.

Jetzt komme ich wieder zur eigentlichen Geschichte. Das Folgende entstammt meiner Erinnerung und ich habe keinen Versuch unternommen, mich mit anderen Beteiligten abzustimmen. Sie sollten deshalb davon ausgehen, dass diese

6 DeMarco, T. 1979. *Structured Analysis and System Specification*. Upper Saddle River, NJ: Yourdon Press.

7 Page-Jones, M. 1980. *The Practical Guide to Structured Systems Design*. Englewood Cliffs, NJ: Yourdon Press.

8 P. Coad und E. Yourdon. 1990. *Object-Oriented Analysis*. Englewood Cliffs, NJ: Yourdon Press.

9 Booch, G. 1991. *Object Oriented Design with Applications*. Redwood City, CA: Benjamin-Cummings Publishing Co.

10 Coplien, J. O. 1995. A generative development-process pattern language. *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, S. 183.

11 Mike Beedle wurde am 23. März 2018 in Chicago von einem geistesgestörten Obdachlosen ermordet, der bereits 99 Mal inhaftiert worden war. Er hätte in einem Heim untergebracht werden sollen. Mike Beedle war mein Freund.

12 Beedle, M., M. Devos, Y. Sharon, K. Schwaber und J. Sutherland. SCRUM: An extension pattern language for hyperproductive software development. `http://jeffsutherland.org/scrum/scrum_plop.pdf`.

Rückbesinnungen lückenhaft sein können und Zweifelhaftes oder zumindest Ungenaues enthalten. Aber keine Sorge, ich habe mich immerhin bemüht, sie zumindest ein wenig unterhaltsam zu gestalten.

Kent Beck bin ich zum ersten Mal auf der PLOP-Konferenz[13] 1994 begegnet, als Copliens Arbeit vorgestellt wurde. Es war ein zwangloses Treffen und es hat sich nicht viel daraus ergeben. Das nächste Mal begegnete ich ihm im Februar 1999 auf der OOP-Konferenz in München. Aber zu diesem Zeitpunkt wusste ich schon beträchtlich mehr über ihn.

Damals war ich als C++- und OOD-Berater tätig und viel mit dem Flugzeug unterwegs, um Kunden bei der Implementierung von C++-Anwendungen mithilfe von OOD-Verfahren zu helfen. Sie erkundigten sich bei mir nach Verfahren. Sie hatten gehört, dass das Wasserfallmodell nicht zur OO passt, und baten mich um Rat. Ich bestätigte das, [14] denn ich hatte mir selbst schon viele Gedanken darüber gemacht. Ich hatte sogar erwogen, ein eigenes OO-Verfahren zu entwickeln. Glücklicherweise habe ich das schnell wieder aufgegeben, weil ich auf Kent Becks Artikel über Extreme Programming (XP) gestoßen war.

Je mehr ich über XP las, desto faszinierter war ich. Die Ideen waren revolutionär (zumindest dachte ich das damals). Sie ergaben einen Sinn, insbesondere in einem OO-Kontext (wie ich damals annahm). Ich wollte unbedingt mehr darüber erfahren.

Zu meiner Überraschung lief mir bei der OO-Konferenz in München in einer Pause Kent Beck über den Weg und ich schlug ein gemeinsames Essen vor, um uns über XP zu unterhalten. Dieses Essen war der Ausgangspunkt für eine bedeutende Partnerschaft. Die Diskussion mit ihm führte dazu, dass ich ihn zu Hause in Medford, Oregon, besuchte, um mit ihm zusammen einen Kurs über XP zu entwickeln. Während dieses Besuchs kam ich zum ersten Mal mit testgetriebener Entwicklung (Test-Driven Development, TDD) in Berührung und war gefesselt.

Damals leitete ich eine Firma namens Object Mentor. Wir gingen eine Partnerschaft mit Kent Beck ein und boten einen fünftägigen Kurs über XP an, den wir *XP Immersion* nannten. Die Kurse fanden von Ende 1999 bis zum 11. September 2001[15] statt und waren ein voller Erfolg. Wir bildeten mehrere Hundert Leute aus.

Im Sommer 2000 lud Kent eine Reihe von Leuten aus der XP- und der Design-Patterns-Community zu einem Treffen in seiner Nachbarschaft ein. Er bezeichnete es als »XP Leadership«-Treffen. Wir fuhren auf dem nahegelegenen Fluss Rogue

13 Pattern Languages of Programming war eine Konferenz, die in den 1990er-Jahren in der Nähe der University of Illinois abgehalten wurde.

14 Hierbei handelt es sich um einen dieser seltsamen Zufälle, die manchmal vorkommen. Es gibt eigentlich nichts Besonderes an OO, das es unwahrscheinlicher macht, es zusammen mit einem Wasserfallmodell zu verwenden. Dennoch war diese Vorstellung damals weit verbreitet.

15 Die Bedeutung dieses Datums sollte nicht übersehen werden.

River Boot und wanderten am Ufer entlang. Und wir trafen uns, um zu entscheiden, wie es mit XP weitergehen soll.

Eine Idee war, eine gemeinnützige Organisation zu gründen. Ich war dafür, viele andere aber nicht. Sie hatten offenbar schlechte Erfahrungen mit einer ähnlichen Organisation gemacht, die sich mit Design Patterns befasst. Ich verließ diese Sitzung frustriert, aber Martin Fowler folgte mir nach draußen und schlug vor, dass wir uns später in Chicago treffen, um das auszudiskutieren. Ich stimmte zu.

Martin und ich trafen uns also im Herbst 2000 in einem Coffee-Shop in der Nähe des Büros der Firma Thought Works, bei der er tätig war. Ich stellte ihm meine Idee vor, alle Befürworter konkurrierender leichtgewichtiger Prozesse zusammenzubringen, um ein Manifest zu verfassen. Martin gab mehrere Empfehlungen für die Einladungsliste und wir arbeiteten zusammen, um die Einladung zu schreiben. Ich verschickte die Einladungen noch am selben Tag. Der Titel lautete »Light Weight Process Summit«.

Einer der Eingeladenen war Alistair Cockburn. Er rief mich an, um mir mitzuteilen, dass er gerade ein ähnliches Treffen einberufen wollte, unsere Einladungsliste ihm aber besser gefiel als seine eigene. Er bot an, dass wir seine Liste mit der unsrigen kombinieren und dass er sich um die Organisation des Treffens kümmern würde, wenn wir zustimmen, dass es im Wintersportort Snowbird in der Nähe von Salt Lake City stattfindet.

Jetzt war also ein Treffen in Snowbird geplant.

1.2 Snowbird

Ich war ziemlich überrascht, wie viele Teilnehmer zusagten. Wer will schon zu einem Treffen kommen, das den Titel »Light Weight Process Summit« trägt? Aber hier waren wir alle, versammelt im Besprechungsraum »Aspen« in der Unterkunft in Snowbird.

Wir waren insgesamt 17. Wir sind seitdem dafür kritisiert worden, dass wir 17 weiße Männer mittleren Alters waren. Die Kritik ist in gewissem Maße gerechtfertigt. Wir hatten immerhin eine Frau eingeladen, Agneta Jacobson, sie konnte aber nicht teilnehmen. Und letzten Endes bestand die große Mehrheit erfahrener Programmierer zum damaligen Zeitpunkt aus weißen Männern mittleren Alters. Die Gründe dafür, warum das der Fall war, würden ein eigenes Buch füllen.

Die 17 Teilnehmer vertraten allerdings einige sehr unterschiedliche Standpunkte. So gab es beispielsweise fünf verschiedene bevorzugte leichtgewichtige Verfahren. Die größte Gruppierung war das XP-Team: Kent Beck, ich, James Grenning, Ward Cunningham und Ron Jeffries. Dann folgte das Scrum-Team: Ken Schwaber, Mike Beedle und Jeff Sutherland. Jon Kern repräsentierte Feature-getriebene Entwick-

lung und Arie van Bennekum die DSDM (*Dynamic Systems Development Method*). Und Alistair Cockburn schließlich vertrat die Gruppe seiner Crystal-Verfahren.

Die übrigen Teilnehmer lassen sich nicht so eindeutig zuordnen. Andy Hunt und Dave Thomas waren Pragmatic Programmers, Brian Marick war Berater für Tests. Jim Highsmith war Berater für Softwaremanagement. Steve Mellor war dabei, um dafür zu sorgen, dass wir aufrichtig blieben, denn er repräsentierte die Modellgetriebene Philosophie, die viele von uns argwöhnisch betrachteten. Und schließlich war Martin Fowler anwesend, der zwar enge persönliche Kontakte zu den Mitgliedern des XP-Teams pflegte, aber trotzdem allen Verfahren, die eine Art Marke aufbauen wollen, skeptisch gegenüberstand. Er war allen sympathisch.

Ich kann mich nur noch an Weniges von dem zweitägigen Treffen erinnern. Andere Teilnehmer haben manches anders in Erinnerung behalten als ich.[16] Deshalb berichte ich einfach davon, woran ich mich erinnere, und rate Ihnen, es als Rückbesinnungen eines 65 Jahre alten Mannes aufzufassen, die fast zwei Jahrzehnte her sind. Vielleicht fehlen ein paar Details, aber das Wesentliche dürfte richtig sein.

Irgendwie haben wir uns darauf geeinigt, dass ich das Treffen eröffnen sollte. Ich dankte allen Anwesenden für ihr Erscheinen und schlug vor, dass wir versuchen sollten, ein Manifest zu erstellen, das die Dinge beschreibt, die all diese leichtgewichtigen Verfahren und Softwareentwicklung im Allgemeinen unserer Ansicht nach gemeinsam haben. Dann setzte ich mich. Ich glaube, das war mein einziger Beitrag in diesem Meeting.

Dann machten wir das Übliche. Wir notierten verschiedene Themen auf Karten und sortierten sie auf dem Fußboden in zusammengehörige Gruppen. Ich weiß nicht mehr, ob das zu irgendetwas geführt hat, ich kann mich nur daran erinnern, es gemacht zu haben.

Ich kann mich nicht daran erinnern, ob der magische Moment am ersten oder am zweiten Tag stattgefunden hat. Ich meine, es war gegen Ende des ersten Tages. Es könnte die Sortierung nach Gruppenzugehörigkeit gewesen sein, bei der sich die vier Grundwerte herauskristallisierten, nämlich Individuen und Interaktionen, funktionierende Software, kundenorientierte Zusammenarbeit und flexible Anpassungen. Irgendjemand schrieb diese Grundwerte an die Tafel an der Vorderseite des Raums und hatte die brillante Idee zu sagen, dass sie zwar zu bevorzugen seien, aber die ergänzenden Werte Verfahren, Tools, Dokumentation, Vereinbarungen und Pläne nicht ersetzen.

16 Kürzlich wurde ein Bericht über die damaligen Ereignisse veröffentlicht: *The Atlantic:* Mimbs Nyce, C. 2017. The winter getaway that turned the software world upside down. *The Atlantic*. 8. Dezember. `https://www.theatlantic.com/technology/archive/2017/12/agile-manifesto-a-history/547715/`. Ich habe diesen Artikel bisher noch nicht gelesen, weil ich meine Rückbesinnungen, die ich hier wiedergebe, nicht beeinflussen möchte.

Das ist das zentrale Konzept des agilen Manifests, und offenbar kann sich niemand mehr genau daran erinnern, wer es zuerst an die Tafel geschrieben hat. Ich meine mich daran zu erinnern, dass es Ward Cunningham war, aber Ward denkt, es war Martin Fowler.

Sehen Sie sich das Foto auf `http://agilemanifesto.org` an. Ward sagt, er hat es aufgenommen, um diesen Moment festzuhalten. Es zeigt eindeutig, dass Martin an der Tafel steht und wir anderen uns um ihn versammelt haben.[17] Das verleiht Wards Äußerung Glaubwürdigkeit, dass es Martin war, der die Idee hatte.

Aber vielleicht ist es auch besser, dass wir es nie genau wissen werden.

Nach diesem magischen Moment ging die ganze Gruppe eine neue Verbindung ein. Wir feilten an den Formulierungen und änderten und verbesserten sie. Wenn ich mich richtig erinnere, war es Ward, der die Präambel schrieb: »Wir machen bessere Verfahren zur Entwicklung von Software zugänglich, indem wir sie selbst praktizieren und anderen dabei helfen, sie zu praktizieren.« Einige der anderen nahmen kleine Änderungen vor und machten Verbesserungsvorschläge, aber es stand fest, dass wir fertig waren. Im Raum herrschte das Gefühl, einen Schlussstrich gezogen zu haben. Keine Meinungsverschiedenheiten. Keine Debatten. Es wurden noch nicht einmal irgendwelche Alternativen ernsthaft diskutiert. So ergaben sich die folgenden vier Prinzipien:

- Individuen und Interaktionen haben Vorrang vor Prozessen und Tools.
- Funktionierende Software hat Vorrang vor umfassender Dokumentation.
- Kundenorientierte Zusammenarbeit hat Vorrang vor vertraglichen Absprachen.
- Flexible Anpassungen haben Vorrang vor der Einhaltung eines Plans.

Hatte ich gesagt, dass wir fertig waren? So fühlte es sich jedenfalls an. Aber es mussten natürlich noch viele Details ausgearbeitet werden. Wie sollten wir beispielsweise das, was wir herausgearbeitet hatten, benennen?

Die Bezeichnung »agil« stand nicht von vornherein fest. Es gab eine ganze Reihe anderer Kandidaten. Mir gefiel »Light Weight« (leichtgewichtig), aber niemandem sonst. Sie waren der Ansicht, der Begriff impliziert »Belanglosigkeit«. Anderen gefiel »adaptiv«. Als »agil« ins Spiel kam, merkte jemand an, dass der Begriff derzeit in Militärkreisen ein Modewort sei. Letzten Endes war niemand von »agil« wirklich begeistert, aber es war die beste Wahl aus einer Menge schlechter Alternativen.

17 Das Foto zeigt, von links nach rechts im Halbkreis um Martin versammelt, Dave Thomas, Andy Hunt (oder vielleicht Jon Kern), mich (erkennbar an der Jeans und dem Leatherman an meinem Gürtel), Jim Highsmith, sonst jemanden, Ron Jeffries und James Grenning. Hinter Ron sitzt noch jemand und auf dem Fußboden neben seinem Schuh liegt offenbar noch eine der Karten, die wir bei der Sortierung nach Gruppenzugehörigkeit verwendet hatten.

Als sich der zweite Tag dem Ende näherte, bot Ward an, die Website `http://agilemanifesto.org` einzurichten. Ich glaube, er hatte die Idee, dass die Besucher der Website das Manifest unterzeichnen sollten.

1.3 Nach Snowbird

Die folgenden zwei Wochen waren nicht annähernd so stimmungsvoll oder ereignisreich wie die zwei Tage in Snowbird. Sie waren von der harten Arbeit geprägt, die mit der Ausarbeitung eines Dokuments unserer Prinzipien verbunden war, das Ward schließlich auf der Website veröffentlichte.

Wir waren uns alle einig, dass es notwendig war, dieses Dokument zu verfassen, um die vier Grundwerte zu erklären und die Richtung vorzugeben. Schließlich handelt es sich bei den vier Grundwerten um die Art von Aussagen, denen jeder zustimmen kann, ohne die Arbeitsweise auch tatsächlich zu ändern. Die Prinzipien verdeutlichen, dass die vier Grundwerte nicht nur traditionelle Werte versinnbildlichen, sondern auch Konsequenzen nach sich ziehen.

Ich habe keine ausgeprägte Erinnerung an diesen Zeitraum, nur dass wir das Dokument, das die Prinzipien enthielt, ständig hin und her gemailt und immer wieder daran herumgefeilt haben. Es war eine Menge Arbeit, aber ich denke, wir hatten alle den Eindruck, dass es die Mühe wert sei. Nachdem das erledigt war, kehrten wir alle wieder in unseren Alltag zurück. Ich vermute, dass die meisten von uns gedacht haben, die Geschichte habe damit ein Ende gefunden.

Keiner von uns hatte mit der enormen Welle von Unterstützung gerechnet, die dann folgte. Keiner von uns hatte vorhergesehen, wie folgenreich diese zwei Tage sein würden. Damit ich mir nicht den Kopf darüber zerbrechen muss, dass ich ein Teil davon gewesen bin, führe ich mir immer wieder vor Augen, dass Alistair kurz davor gewesen war, ein vergleichbares Treffen einzuberufen. Und deshalb frage ich mich, wie viele andere noch kurz davor gewesen waren. Ich gebe mich also damit zufrieden, dass die Zeit dafür gekommen war. Wenn wir 17 uns nicht auf dem Berg in Utah getroffen hätten, wäre irgendeine andere Gruppe irgendwo anders zusammengekommen und hätte ähnliche Schlussfolgerungen gezogen.

1.4 Überblick über agile Softwareentwicklung

Wie managt man ein Softwareprojekt? Im Laufe der Jahre hat es eine Vielzahl von Ansätzen gegeben – und die meisten waren ziemlich mangelhaft. Unter Managern, die an Götter glauben, die für das Schicksal von Softwareprojekten verantwortlich zeichnen, sind Gottvertrauen und Gebete weit verbreitet. Ungläubige greifen eher auf motivierende Verfahren zurück, wie die Durchsetzung von Abga-

befristen mithilfe von Peitschen, Ketten, siedendem Öl und Bildern von bergsteigenden Menschen und über dem Meer fliegenden Möwen.

Diese Ansätze führen fast immer zu dem charakteristischen Symptom fehlerhaften Softwaremanagements: Entwicklerteams, die trotz zu vieler Überstunden ständig Abgabefristen versäumen. Solche Teams erstellen Produkte von offensichtlich geringer Qualität, die den Anforderungen des Kunden nicht annähernd gerecht werden.

1.4.1 Das Eiserne Kreuz

Der Grund für das spektakuläre Scheitern dieser Verfahren ist, dass die Manager, die sie einsetzen, die physikalischen Grundlagen von Softwareprojekten nicht verstanden haben. Allen Projekten ist ein unanfechtbarer Kompromiss auferlegt, der als das *Eiserne Kreuz* (engl. *Iron Cross*) des Projektmanagements bezeichnet wird. Gut, schnell, preiswert, erledigt: Wählen Sie drei dieser Attribute aus – das vierte ist nicht verfügbar. Ein Projekt kann gut, schnell und preiswert sein, wird dann aber nicht erledigt. Ein Projekt kann auch erledigt, preiswert und schnell sein, dann ist es jedoch nicht gut.

Einem guten Projektmanager ist klar, dass diese vier Attribute verschiedene Beiträge leisten. Ein guter Projektmanager leitet ein Projekt so, dass es den Anforderungen entsprechend gut, schnell, preiswert und erledigt ist. Ein guter Projektmanager passt die Beiträge, die diese Attribute leisten, demgemäß an, anstatt zu verlangen, dass alle 100 Prozent betragen. Diese Art des Managements wird mit agiler Softwareentwicklung angestrebt.

An dieser Stelle möchte ich mich vergewissern, dass Ihnen klar ist, dass agile Softwareentwicklung Rahmenbedingungen schafft, die Entwicklern und Managern dabei *helfen*, diese Art eines pragmatischen Projektmanagements durchzuführen. Ein solches Projektmanagement ergibt sich allerdings nicht automatisch und es gibt keine Garantie, dass Manager angemessene Entscheidungen treffen. Tatsächlich ist es durchaus möglich, vollständig auf agile Softwareentwicklung zu setzen und ein Projekt durch Missmanagement dennoch zum Scheitern zu bringen.

1.4.2 Diagramme an der Wand

Wie fördert agile Softwareentwicklung diese Art des Managements? *Sie stellt Daten bereit*. Ein agiles Entwicklerteam erstellt nur genau die Daten, die Manager benötigen, um die richtigen Entscheidungen zu treffen.

Betrachten Sie Abbildung 1.2. Stellen Sie sich vor, das Diagramm befände sich an der Wand des Besprechungsraums. Wäre das nicht toll?

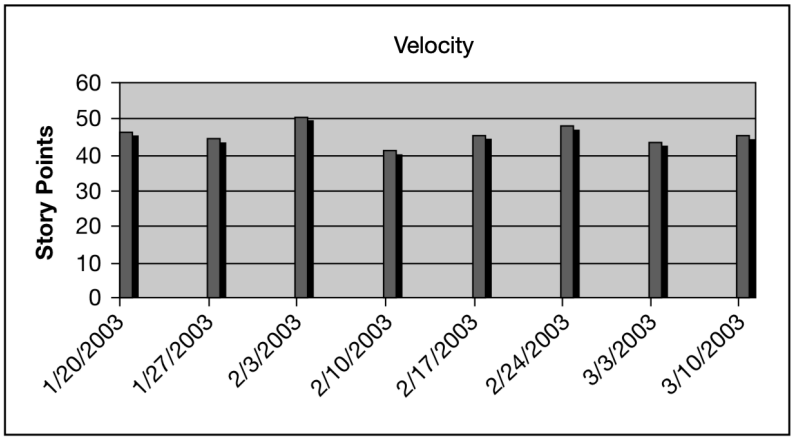

Abb. 1.2: Die Velocity des Teams

Das Diagramm zeigt, wie viel das Entwicklerteam in jeder Woche erledigt hat. Die Maßeinheit ist »Points«. Was genau damit gemeint ist, erörtern wir später. Sehen Sie sich jetzt nur das Diagramm an. Jeder kann auf den ersten Blick erkennen, wie schnell das Team vorankommt. Man braucht weniger als zehn Sekunden, um festzustellen, dass die durchschnittliche Velocity etwa 45 Points pro Woche beträgt.

Wirklich jeder, selbst ein Manager, kann vorhersagen, dass das Team in der nächsten Woche etwa 45 Points erledigen wird. In den nächsten zehn Wochen sollten etwa 450 Points zu schaffen sein. Das nenne ich leistungsstark! Besonders leistungsfähig sind solche Diagramme, wenn die Manager und das Team über ein gutes Gespür für die Anzahl der Points im Projekt verfügen. Tatsächlich erfassen gute agile Teams diese Information in einem weiteren Diagramm.

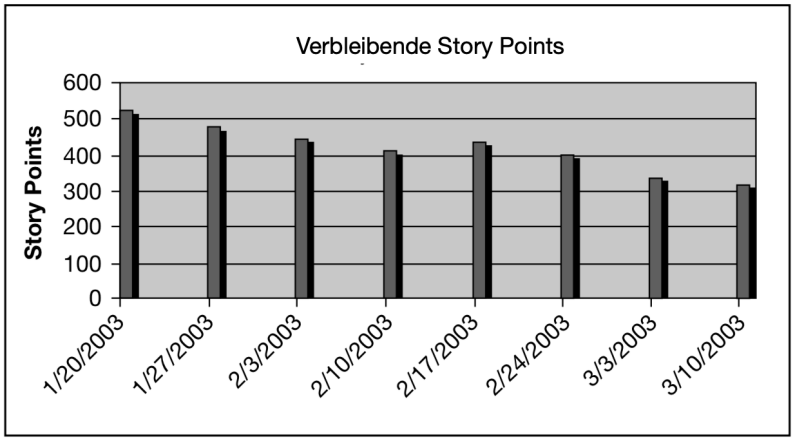

Abb. 1.3: Burn-down-Chart

Das Diagramm in Abbildung 1.3 wird als *Burn-down-Chart* bezeichnet. Es zeigt, wie viele Story Points noch bis zum nächsten größeren Meilenstein verbleiben. Beachten Sie, dass die Anzahl in fast jeder Woche abnimmt und dass die Abnahme geringer ist als die Anzahl der Points im Velocity-Diagramm. Das liegt daran, dass es ständig neue Anforderungen gibt und dass während der Entwicklung Probleme entdeckt werden.

Aus dem Burn-down-Diagramm kann eine Steigung abgeleitet werden, anhand derer sich vorhersagen lässt, wann der nächste Meilenstein vermutlich erreicht wird. Praktisch jeder kann den Raum betreten, sich die beiden Diagramme ansehen und schlussfolgern, dass der nächste Meilenstein mit einer Velocity von 45 Points pro Woche im Juni erreicht wird. Beachten Sie auch, dass es im Burn-down-Diagramm einen Ausrutscher gibt. In der Woche, die am 17. Februar beginnt, ist die Entwicklung irgendwie in Rückstand geraten. Das könnte auf das Hinzufügen eines neuen Features oder auf eine andere größere Änderung der Anforderungen zurückzuführen sein. Es könnte auch daran liegen, dass die Entwickler eine neue Abschätzung der noch verbleibenden Arbeit vorgenommen haben. In jedem Fall möchten wir wissen, wie sich das auf die Planung auswirkt, damit das Projekt richtig gemanagt werden kann.

Diese beiden Diagramme an die Wand zu bringen, ist ein entscheidendes Ziel der agilen Softwareentwicklung. Einer der Gründe für agile Softwareentwicklung ist die Bereitstellung der Daten, die Manager benötigen, um Entscheidungen bezüglich der Beiträge zum Eisernen Kreuz zu treffen, um so das bestmögliche Ergebnis zu erzielen.

Viele würden dem letzten Absatz nicht zustimmen. Im agilen Manifest sind diese Diagramme überhaupt nicht erwähnt, und manche agilen Teams verwenden sie auch gar nicht. Eigentlich sind die Diagramme auch gar nicht von Bedeutung. Das Wichtige sind die Daten.

Agile Softwareentwicklung ist vor allem ein Feedback-getriebener Ansatz. Jede Woche, jeden Tag, jede Stunde und manchmal sogar jede Minute sieht man sich die Ergebnisse der vorangegangenen Woche, des vorhergehenden Tags und der letzten Stunden und Minuten an und nimmt entsprechende Anpassungen vor. Das gilt für einzelne Programmierer, aber auch für das Management des gesamten Teams. Ohne Daten kann man ein Projekt nicht managen.[18]

Selbst wenn Sie die beiden Diagramme nicht an die Wand bringen können, sollten Sie sicherstellen, dass Sie die Daten erhalten und dass die Manager davon wissen. Vergewissern Sie sich, dass den Managern bekannt ist, wie schnell das Team vorankommt und wie viel es noch zu erledigen hat. Und präsentieren Sie diese

18 Hier gibt es eine enge Verbindung zu John Boyds OODA-Loop, die hier zusammengefasst ist: `https://de.wikipedia.org/wiki/OODA-Loop`. Boyd, J. R. 1987. *A Discourse on Winning and Losing*. Maxwell Air Force Base, AL: Air University Library, Document No. M-U 43947.

Informationen für alle zugänglich und auf offensichtliche Weise – ganz so, wie die beiden Diagramme an der Wand.

Warum aber sind diese Daten so wichtig? Ist es möglich, ein Projekt ohne diese Daten effektiv zu managen? Wir haben es 30 Jahre lang versucht. Und so ging es aus ...

1.4.3 Das Erste, was man weiß

Was ist das Erste, was man über ein Projekt weiß? Bevor man den Namen und irgendeine der Anforderungen kennt, ist eine Sache schon bekannt: natürlich *Das Datum*. Sobald *Das Datum* festgelegt wurde, ist es endgültig. Es ist sinnlos, *Das Datum* verhandeln zu wollen, denn es gab gute geschäftliche Gründe, *Das Datum* auszuwählen. Wenn *Das Datum* der September ist, dann liegt das daran, dass im September eine Messe oder eine Aktionärsversammlung stattfindet oder eine Förderungsmaßnahme ausläuft. Was auch immer der Grund sein mag, es gibt einen guten *geschäftlichen* Grund, und *Das Datum* wird nicht geändert, nur weil ein paar Entwickler denken, dass sie nicht in der Lage sind, den Termin einzuhalten.

Gleichzeitig ändern sich ständig die Anforderungen, die nie endgültig festgelegt werden. Das liegt daran, dass die Kunden gar nicht wissen, was sie eigentlich wollen. Sie haben zwar eine ungefähre Vorstellung von der Aufgabe, die sie *lösen* möchten, aber das in Anforderungen zu übersetzen, ist nie trivial. Also werden die Anforderungen immer wieder neu bewertet und überarbeitet. Neue Features kommen hinzu. Alte Features werden gestrichen. Und die Benutzeroberfläche ändert sich wöchentlich oder sogar täglich.

So sieht die Welt von Softwareentwicklerteams aus. Es ist eine Welt, in der Termine festgelegt sind und sich Anforderungen ständig ändern. Und das Entwicklerteam muss irgendwie dafür sorgen, dass es zu einem guten Ergebnis kommt.

1.4.4 Das Meeting

Das Wasserfallmodell verhieß uns eine Möglichkeit, dieses Problem in Angriff zu nehmen. Damit Sie verstehen, wie verführerisch und ineffektiv das war, muss ich Sie zu einem *Meeting* mitnehmen.

Heute ist der 1. Mai. Der Chef ruft uns alle in den Konferenzraum.

»Wir haben ein neues Projekt« sagt er. »Es muss bis zum 1. November erledigt sein. Die Anforderungen liegen uns noch nicht vor. Die bekommt ihr in den nächsten Wochen. Wie lange braucht ihr für die Analyse?«

Wir blicken uns aus den Augenwinkeln an. Keiner will etwas sagen. Wie soll man eine solche Frage auch beantworten? Einer von uns murmelt: »Aber uns liegen doch noch keine Anforderungen vor.«