# Clean Agile. Back to Basics

**von Robert C. Martin (Buchzusammenfassung)**

Patrick Bucher

05.09.2021

## Inhaltsverzeichnis



**Hinweis zur Übersetzung:** Hierbei handelt es sich um die deutschsprachige Übersetzung einer englischsprachigen Buchzusammenfassung, die über das englischsprachige Original geschrieben worden ist. Manche Begriffe wurden frei auf Deutsch übersetzt, andere im englischen Original belassen. Auf eigene Übersetzungen folgt beim ersten Auftreten jeweils der Originalbegriff in Klammern.

Im Englischen wurde der Begriff "agile software development" mit "Agile" abgekürzt. Deshalb steht auch in dieser Übersetzung der substantivierte Begriff "Agile" geschrieben, der englisch auszusprechen ist, wo "agile Softwareentwicklung" gemeint ist.

## 1 Einführung in Agile

Das *Manifest für Agile Softwareentwicklung* (*Agile Manifesto*) ist das Ergebnis eines Treffens von 17 Experten für Software anfangs 2001 als Reaktion auf schwergewichtige Prozesse wie Wasserfall (*Waterfall*). Seither erfreute sich Agile weiter Verbreitung und wurde auf verschiedene Arten erweitert – leider nicht immer im Sinne der ursprünglichen Idee.

## 1.1 Geschichte von Agile

Die grundlegende Idee von Agile – die Arbeit mit kleinen Zwischenzielen, wobei der Fortschritt gemessen wird – könnte so alt sein wie unsere Zivilisation. Es ist auch möglich, dass agile Praktiken in den Anfängen der Softwareentwicklung verwendet worden sind. Die Idee des wissenschaftlichen Managements (*Scientific Management*), welche auf dem Taylorismus basiert, von oben herab organisiert ist und auf eine detaillierte Planung setzt, war zu dieser Zeit weit verbreitet in der Industrie, wodurch sie in Konflikt zu den vor-agilen (*Pre-Agile*) Praktiken war, die zu dieser Zeit in der Softwareentwicklung so weit verbreitet waren.

Wissenschaftliches Management war für Projekte geeignet, bei denen Änderungen teuer waren und zu denen es eine genau definierte Problemdefinition mit extrem spezifischen Zielen gab. Vor-agile Praktiken andererseits eigneten sich gut für Projekte, bei denen Änderungen günstig, das Problem nur teilweise definiert und die Ziele informell spezifiziert waren.

Leider gab es zu dieser Zeit keine Diskussion darüber, welcher Ansatz für Softwareprojekte der bessere war. Stattdessen fand das Wasserfallmodell weite Verbreitung, das ursprünglich von Winston Royce in seinem Fachartikel *Managing the Development of Large Software Systems* als Strohmannargument aufgebaut worden war, um dessen Unzulänglichkeit zu demonstrieren. Das Wasserfallmodell mit seinem Fokus auf Analyse, Planung und genaues Einhalten von Plänen war ein Abkömmling des wissenschaftlichen Managements, nicht von vor-agilen Praktiken.

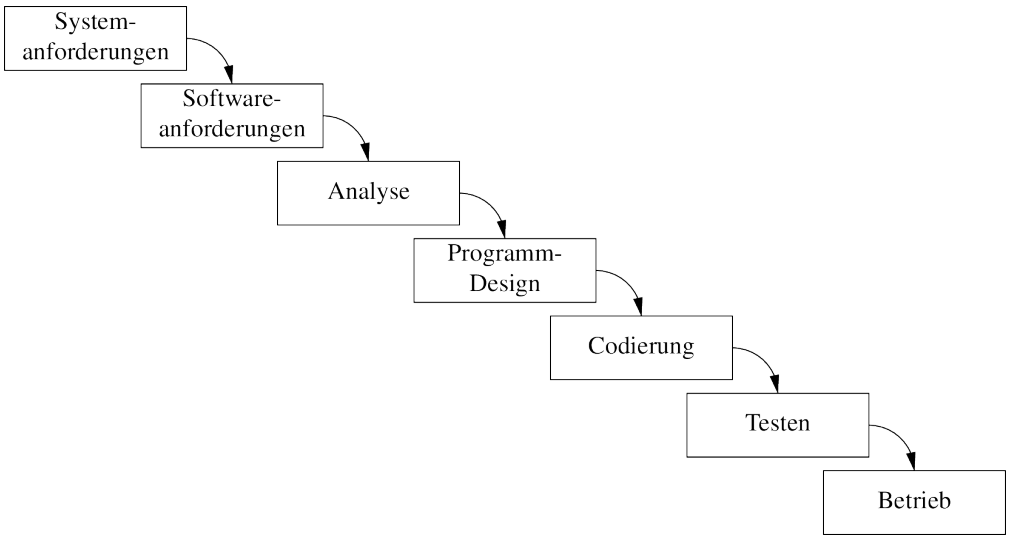


Abbildung 1: Das Wasserfallmodell

Das Wasserfallmodell dominierte die Industrie ab den 1970er-Jahren für fast 30 Jahre. Seine aufeinanderfolgenden Phasen von Analyse, Design und Umsetzung sah vielversprechend aus für Entwickler, welche in endlosen "Programmieren und Korrigieren"-Zyklen (*"code and fix" cycles*) arbeiteten, und dabei nicht einmal die vor-agile Disziplin aufbrachten.

Was auf dem Papier gut aussah – und zu vielversprechenden Ergebnissen nach der Analyse- und Design-Phase führte – scheiterte oft kläglich in der Umsetzungsphase. Diese Probleme wurden jedoch auf eine schlechte Ausführung geschoben, und der Wasserfall-Ansatz selber wurde nicht kritisiert. Stattdessen wurde dieser Ansatz so dominant, dass auf neue Entwicklungen in der Software-Industrie wie strukturierte oder objektorientierte Programmierung bald die Disziplinen der strukturierten und objektorientierten Analyse und des strukturierten und objektorientierten Designs folgten – und so perfekt zur Wasserfall-Denkweise passten.

Einige Befürworter dieser Ideen begannen jedoch das Wasserfallmodell mitte der 1990er-Jahre in Frage zu stellen, wie z.B. Grady Booch mit seiner Methode des objektorientierten Designs (OOD), die Entwurfsmuster-Bewegung (*Design Pattern movement*), und die Autoren des *Scrum*-Papers. Kent Becks Ansätze des *Extreme Programming* (XP) und der testgetriebenen Entwicklung (*Test-Driven Development, TDD*) der späten 1990er-Jahre waren eine klare Abkehr vom Wasserfallmodell hin zu einem agilen Ansatz. Martin Fowlers Gedanken zum *Refactoring* mit dessen Betonung von kontinuierlicher Verbesserung passt sicherlich schlecht zum Wasserfallmodell.

## 1.2 Das Manifest für Agile Softwareentwicklung

17 Vertreter verschiedener agiler Ideen – Kent Beck, Robert C. Martin, Ward Cunningham (XP), Ken Schwaber, Mike Beedle, Jeff Sutherland (Scrum), Andrew Hunt, David Thomas ("Pragmatic Programmers") und weitere – trafen sich anfangs 2001 in Snowbird, Utah, um ein Manifest zu erarbeiten, dass die gemeinsame Essenz all dieser leichtgewichtigen Ideen erfassen sollte. Nach zwei Tagen konnte ein breiter Konsens erreicht werden:

> Wir erschließen bessere Wege, Software zu entwickeln, indem wir es selbst tun und anderen dabei helfen. Durch diese Tätigkeit haben wir diese Werte zu schätzen gelernt:
>
> - **Individuen und Interaktionen** mehr als Prozesse und Werkzeuge
> - **Funktionierende Software** mehr als umfassende Dokumentation
> - **Zusammenarbeit mit dem Kunden** mehr als Vertragsverhandlung
> - **Reagieren auf Veränderung** mehr als das Befolgen eines Plans
>
> Das heißt, obwohl wir die Werte auf der rechten Seite wichtig finden, schätzen wir die Werte auf der linken Seite höher ein.

Das *Manifest für Agile Softwareentwicklung* wurde nach dem Treffen auf agilemanifesto.org veröffentlicht, wo es noch immer unterschrieben werden kann. Die 12 Prinzipien hinter dem Agilen Manifest wurden nach den beiden Wochen, die auf das Treffen folgten, in gemeinsamer Arbeit verfasst. Dieses Dokument erläutert die vier Werte, die im Manifest aufgeführt sind, und verleiht ihnen eine Richtung; es legt dar, dass diese Werte wirkliche Konsequenzen haben.

## 1.3 Überblick über Agile

Viele Softwareprojekte werden mit einem Ansatz basierend auf Zuversicht und Motivationstechniken geführt. Das Ergebnis ist, dass solche Projekte chronisch verspätet sind, obwohl die Entwickler Überstunden leisten.

Alle Projekte sind eingeschränkt von einem Kompromiss, den man als das *eiserne Kreuz des Projektmanagements* (*Iron Cross of Project Management*) bezeichnet: gut, schnell, günstig, fertig – wähle drei! Gute Projektmanager verstehen diesen Kompromiss und streben nach Ergebnissen die gut genug sind, in einem akzeptablen Zeitrahmen und Budget erreicht werden können und die wesentlichen Features bieten.

Agile produziert Daten, welche Managern dabei helfen gute Entscheidungen zu treffen. Die *Velocity* zeit die Menge der Punkte, die ein Entwicklungsteam innerhalb einer Iteration abarbeitet. Ein *Burn-Down Chart* zeigt verbleibenden Punkte zur Erreichung des nächsten Meilensteins. Dieses schrumpft nicht notwendigerweise mit der Geschwindigkeit der Velocity, weil Anforderungen und deren Schätzung sich ändern können. Trotzdem kann das Gefälle des Burn-Down Charts dazu verwendet werden, um ein wahrscheinliches Release-Datum für den nächsten Meilenstein voerherzusagen.

Agile ist ein Ansatz, der auf Rückkoppelung basiert (*feedback-driven approach*). Auch wenn im Agile-Manifest weder Velocity noch Burn-Down Charts erwähnt werden, ist das Sammeln solcher Daten und das Treffen von Entscheidungen auf dieser Grundlage entscheidend. Solche Daten sollen öffentlich, offensichtlich und transparent gemacht werden.

Das Enddatum eines Projekts ist normalerweise gegeben und kan nicht verhandelt werden, oft aus guten Gründen des Geschäftsinteresses. Die Anforderungen ähndern sich hingegen häufig, weil Kunden nur ein grobes Ziel haben, aber nicht die genauen Schritte kennen, um dieses zu erreichen.

## 1.4 Ein Wasserfall-Projekt

Zu Zeiten des Wasserfallmodells wurde ein Projekt oft in drei Phasen gleicher Länge aufgeteilt: Analyse, Design und Umsetzung. In der Analysephase wurden Anforderungen gesammelt und die Planung wurde durchgeführt. In der Designphase wurde eine Lösung skizziert und die Planung verfeinert. Keine der beiden Phasen haben harte und greifbare Ziele; sie waren abgeschlossen, wenn das Enddatum der Phase erreicht worden war.

Die Umsetzungsphase muss jedoch funktionierende Software hervorbringen – ein hartes und greifbares Ziel, dessen Erreichung einfach zu beurteilen ist. Verspätungen sind oft erst in dieser Phase zu erkennen, und Anspruchsgruppen (*stakeholders*) erfahren erst von solchen Problemen, wenn das Projekt eigentlich schon beinahe fertig sein sollte.

Solche Projekte enden häufig in einem *Todesmarsch* (*Death March*): eine kaum funktionierende Lösung wird nach vielen Überstunden herausgebracht, obwohl die Abgabefrist (*deadline*) mehrmals verschoben worden ist. Die “Lösung” für das nächste Projekt besteht normalerweise

darin, dass noch mehr Analyse und Design gemacht wird – mehr von dem, was schon vorher nicht funktioniert hat (*Runaway Process Inflation*).

## 1.5 Der agile Ansatz

Wie beim Wasserfallmodell beginnt auch ein agiles Projekt mit der Analyse – doch die Analyse ist nie fertig. Die Zeit wird in Iterationen oder *Sprints* von normalerweise zwei Wochen eingeteilt. Die *Iteration null* (*Iteration Zero*) wird dazu verwendet, die anfänglichen Stories zu schreiben und zu schätzen, sowie um die Entwicklungsumgebung aufzusetzen, ein vorläufiges Design zu entwerfen, und einen groben Plan zu machen. Analyse, Design und Umsetzung finden in jeder Iteration statt.

Nach Abschluss der ersten Iteration sind normalerweise weniger Stories abgeschlossen als ursprünglich geschätzt. Das ist kein Misserfolg, sondern bietet eine erste Messung, die zur Anpassung des ursprünglichen Plans verwendet werden kann. Nach ein paar Iterationen kann eine realistische Durchschnittsvelocity berechnet und eine Schätzung des Releasedatums abgegeben werden. Das mag oft enttäuschend ausfallen, ist aber wenigstens realistisch. Hoffnung wird schon früh durch echte Daten ersetzt.

Das Projektmanagement, dass sich mit dem eisernen Kreuz befassen muss – gut, schnell, günstig, fertig: wähle drei! – kann nun die folgenden Anpassungen vornehmen:

- *Planung* (*Schedule*): Das Abschlussdatum ist gewöhnlich nicht verhandelbar, und wenn es das ist, entstehen der Firma bei Verspätungen normalerweise signifikante Kosten.
- *Personal* (*Staff*): *"Durch das Hinzufügen von Arbeitskräften zu einem verspäteten Projekt verspätet sich das Projekt nur noch mehr.* (Brookes Gesetz, *"Adding manpower to a late project makes it later."*) Wenn einem Projekt mehr Personal zugewiesen wird, fällt die Produktivität zunächst stark ab, und verbessert sich erst nach längerer Zeit. Personal kann langfristig aufgestockt werden, sofern man es sich finanziell leisten kann.
- *Qualität* (*Quality*): Die Qualität zu senken mag zwar kurzfristig den Eindruck vermitteln, dass man schneller vorwärts kommt. Langfristig wird aber dadurch das Projekt verzögert, weil mehr Fehler eingebaut werden. *"Die einzige Möglichkeit schnell voranzukommen, ist gut voranzukommen."* (*"The only way to go fast, is to go well."*)
- *Umfang* (*Scope*): Wenn es keine andere Möglichkeit git, können die Anspruchsgruppen oft davon überzeugt werden, ihre Anforderungen auf Features einzuschränken, die unbedingt notwendig sind.

Die Reduktion des Umfangs ist oftmals die einzige vernünftige Wahl. Darum soll man zu Beginn eines jeden Sprints sicherstellen, dass dabei nur Features umgesetzt werden, die wirklich von den Anspruchsgruppen benötigt werden. Andernfalls läuft man Gefahr wertvolle Zeit in optionale (*"nice to have"*) Features zu investieren.

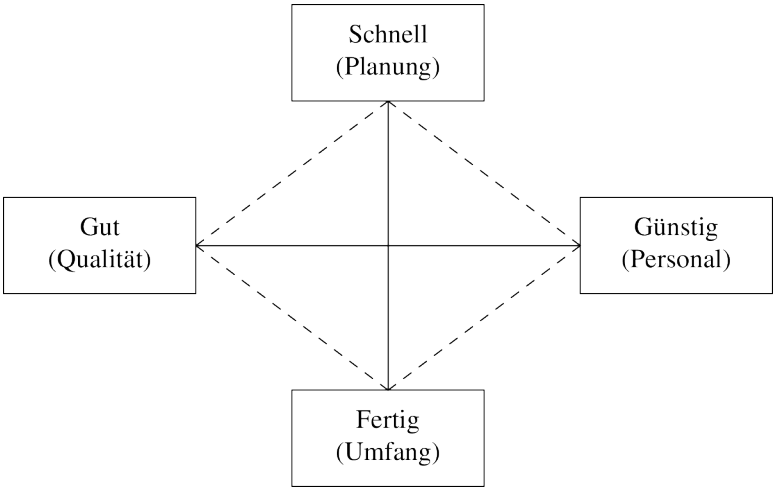


Abbildung 2: Das eiserne Kreuz des Projektmanagements (Interventionen in Klammern)

## 1.6 Der Kreis des Lebens

Extreme Programming (XP), wie es in Kent Becks *Extreme Programming Explained* beschrieben ist, erfasst die Essenz der agilen Softwareentwicklung. Die Praktiken von XP sind im *Kreis des Lebens* (*Circle of Life*) organisert, welcher aus drei Ringen besteht. (Die übersetzten Begriffe werden hier nur ergänzend angegeben. Im weiteren Text werden die Originalbegriffe verwendet, da diese im deutschsprachigen Raum geläufig sind, zumindest in der Softwareentwicklung.)

Der äussere Ring beinhaltet die geschäftsorientierten (*business-facing*) Praktiken, welche ziemlich ähnlich sind wie der Scrum-Prozess:

- **Planning Game** (Planungsspiel): das Projekt in Features, Stories und Aufgaben herunterbrechen
- **Small Releases** (kleine Releases): kleine, aber regelmässige Inkremente ausliefern
- **Acceptance Tests** (Akzeptanztests): unmissverständliche Abschlusskriterien angeben (*definition of "done"*)
- **Whole Team** (Team als Ganzes): in verschiedenen Funktionen (Programmierer, Tester, Management) zusammenarbeiten

Der mittlere Ring beinhaltet die teamorientierten (*team-facing*) Praktiken:

- **Sustainable Pace** (nachhaltiges Tempo): Fortschritt machen und dabei das Ausbrennen (*burnout*) des Entwicklugsteams verhindern
- **Collective Ownership** (gemeinsamer Besitz): Wissen über das Projekt austauschen, um Wissenssilos zu vermeiden
- **Continuous Integration** (kontinuierliche Integration): häufiges Schliessen des *feedback loops* und den Fokus des Teams aufrechterhalten

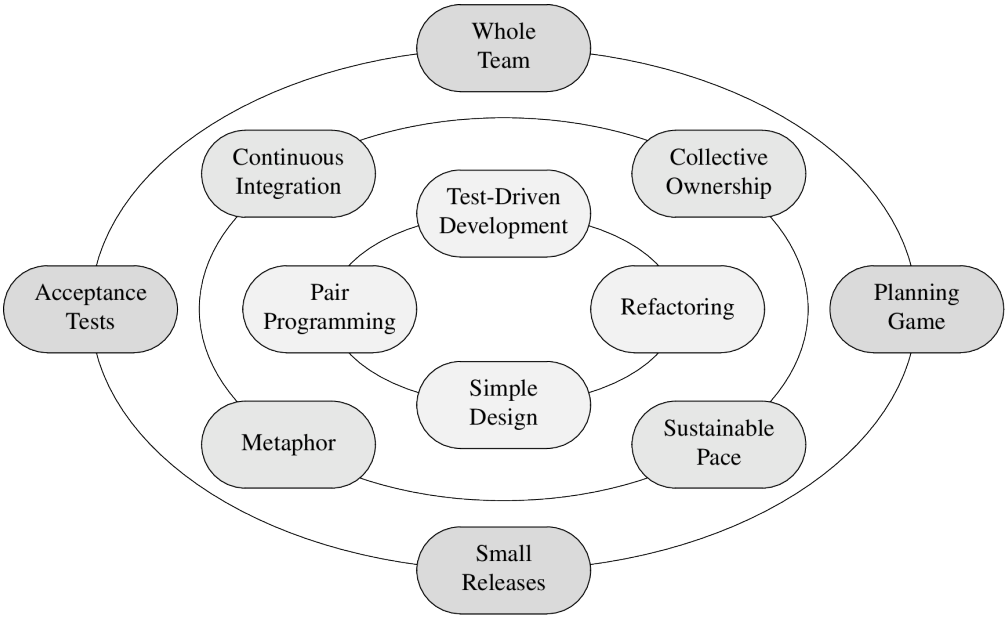


Abbildung 3: Der Kreis des Lebens

- **Metaphor** (Metapher): mit einem gemeinsamen Wortschatz und mit einer gemeinsamen Sprache arbeiten

Der innere Ring beinhaltet die technischen (*technical*) Praktiken:

- **Pair Programming/Pairing** (paarweises Programmieren): Wissen austauschen, Reviews durchführen, zusammenarbeiten
- **Simple Design** (einfaches Design): unnötige Aufwände vermeiden
- **Refactoring** (Überarbeitung): alle Arbeitserzeugnisse kontinuierlich verbessern
- **Test-Driven Development** (testgetriebene Entwicklung): die Qualität beim schnellen Fortschreiten hoch halten

Diese Praktiken haben eine gute Übereinstimmung zu den agilen Werten aus dem Manifest:

- **Individuen und Interaktionen** mehr als Prozesse und Werkzeuge
  - Whole Team (geschäftsorientiert)
  - Metaphor (teamorientiert)
  - Collective Ownership (teamorientiert)
  - Pair Programming/Pairing (technisch)
- **Funktionierende Software** mehr als umfassende Dokumentation
  - Acceptance Tests (geschäftsorientiert)
  - Test-Driven Development (technisch)
  - Simple Design (technisch)

  - Refactoring (technisch)
  - Continuous Integration (technisch)

- **Zusammenarbeit mit dem Kunden** mehr als Vertragsverhandlung
  - Planning Game (geschäftsorientiert)
  - Small Releases (geschäftsorientiert)
  - Acceptance Tests (geschäftsorientiert)
  - Metaphor (team)

- **Reagieren auf Veränderung** mehr als das Befolgen eines Plans
  - Planning Game (geschäftsorientiert)
  - Small Releases (geschäftsorientiert)
  - Acceptance Tests (geschäftsorientiert)
  - Sustainable Pace (teamorientiert)
  - Refactoring (technisch)
  - Test-Driven Development (technisch)

Zusammenfassend:

> Agile ist eine kleine Disziplin, welche kleinen Software-Teams beim Handhaben kleiner Projekte hilft. Grosse Projekte sind aus kleinen Projekten gemacht.