# Agile Softwareentwicklung
## erfolgreich managen

Dr. Christoph Steindl
Senior IT Architect, Method Exponent
Certified ScrumMaster



# Agenda

Teil 1 (30 Minuten)
- Rückblick auf die 90er Jahre
- On Demand
- Agilität
- Traditionelle Vorgehensweise und ihre Probleme
- Vorteile agiler Softwareentwicklung

Teil 2 (75 Minuten)
- Feedback
- Aufwandschätzung
- Planung
- Fortschrittskontrolle
- Retrospektiven
- Neuproduktentwicklung
- Kulturwechsel
- Reise mit LSD und TOC

**WARUM** **WIE**

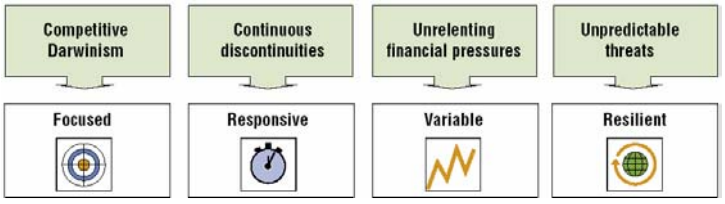
# Am Anfang der 90er Jahre...

- Der kalte Krieg war gerade vorbei, die Berliner Mauer gefallen.
- Das Internet gab es noch nicht (wie wir es jetzt kennen), auch kein Intranet.
- PCs und Handys waren noch nicht weit verbreitet, geschweige denn Handhelds und iPods.
- E-Commerce war noch kein Thema, kaum jemand schrieb E-Mails.
- „Lean manufacturing" klang noch verdächtig, keiner sprach von „Agilität".
- Outsourcing war die Ausnahme, Offshoring erst recht.
- „Knowledge Worker" waren noch keine eigenständige Kategorie.
- Lernende oder virtuelle Organisationen kannten wir noch nicht.
- „Community of practice" gab's an der Kegelbahn.
- Computer Security war was für das Militär und die Nachrichtendienste.
- WWW, ERP und Y2K gab es in der Buchstabensuppe noch nicht.



# On Demand ist jetzt und in der Zukunft essenziell

- Das Umfeld wird dynamischer.
- Nichts ist mehr fix, alles kann sich ändern.
- Die Konkurrenz wird härter.
- Die Zyklen werden kürzer.




Aus: Hancock et. al.: *On demand business: The new agenda for value creation,* IBM Institute for Business Value Study, 2004.

# Definitionen von Agilität

- „Agility is the ability to both create and respond to change in order to profit in a turbulent business environment.“*
- „Agility is the ability to balance flexibility and stability.“*
- „Agility is the physical ability to act (*response ability*) and the intellectual ability to find appropriate things to act on (*knowledge management*).“**

* J. Highsmith: *Agile Software Development Ecosystems*, 2002.
** R. Dove: *Response Ability*, 2001.



# Traditionellen Vorgehensweise (bei Projekten)

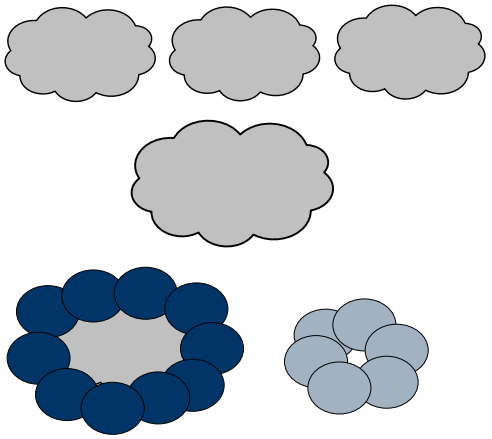
AG ... Auftraggeber
AN ... Auftragnehmer

- Zuerst wird monatelang der Funktionsumfang diskutiert und zu einem Pflichtenheft zusammengeschrieben. — AG
- Das Pflichtenheft ist Grundlage der Ausschreibung. — AG
- Anbieter müssen bei sehr hohen Unsicherheiten ein Fixpreisangebot legen. — AN
- Kaufmännische Überlegungen führen zu drastisch reduzierten Aufwänden und komprimierten Zeitplänen. — AN
- Der Preis entscheidet, der Billigste gewinnt. — AG
- Im Projekt wird jede kleine Änderung an den Anforderungen zusätzlich verrechnet. — AN
- Unwichtige Funktionen werden umgesetzt, wichtige in spätere Phasen verschoben. — AN
- Das Budget wird überzogen, der Zeitplan wird überschritten. — AG

AG Win : Lose AN → AG Lose : Lose AN

# Kernprobleme

- Lange Wunschliste ohne Priorisierung
- Lange Vorlaufzeit
- Hohe Aufwände wegen hoher Unsicherheiten
- „Schönung" der Aufwände, preisliche Zugeständnisse
- Auffetten durch Verrechnung von Zusatzaufwänden
- Umsetzung unwichtiger Funktionen und Verschieben wichtiger Funktionen
- Belastete Beziehung zwischen Beteiligten

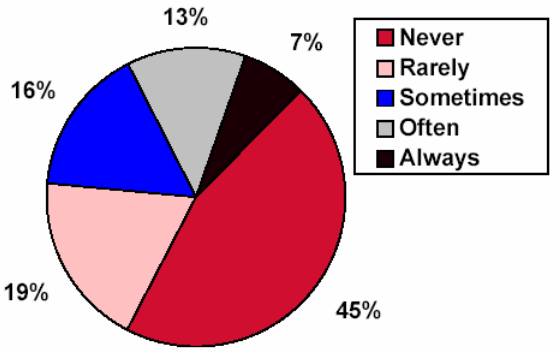


Version 1 Version 2



# Features & Function Usage

THE STANDISH GROUP




Von: http://www.xp2003.org/xp2002/talksinfo/johnson.pdf

# Agile Softwareentwicklung bringt viele Vorteile

- Änderungen stellen kein Problem dar.
- Ergebnisse werden rascher geliefert.
- Die Ergebnisse entsprechen den Erwartungen.
- Risiken werden früher erkannt.
- Die Produktivität steigt.
- Die Qualität steigt.
- Die Mitarbeiterfluktuation nimmt ab.




# Agenda

Teil 1 (30 Minuten)

- Rückblick auf die 90er Jahre
- On Demand
- Agilität
- Traditionelle Vorgehensweise und ihre Probleme
- Vorteile agiler Softwareentwicklung

Teil 2 (75 Minuten)

- Feedback
- Aufwandschätzung
- Planung
- Fortschrittskontrolle
- Retrospektiven
- Neuproduktentwicklung
- Kulturwechsel
- Reise mit LSD und TOC

**WARUM** **WIE**

# Wie erzielt man die Vorteile?

Aus Management-Sicht:

- Priorisierung
- Rascheres Feedback
- Häufige Inspektion und Anpassung
- Offenheit und Vertrauen
- Einbindung der Stakeholder, Teaming

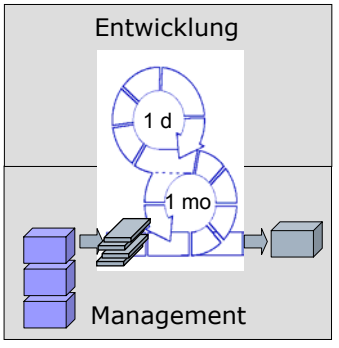
Aus Entwickler-Sicht:

- Pair Programming
- Test-Driven Development
- Refactoring
- Continuous Integration
- Coaching
- Emergenz



# Agile Softwareentwicklung

- Häufiges Feedback, Inspektion und Anpassung
- Kollaboration, enge Kommunikation
- Häufige Auslieferung
- Verbesserung durch Reflexion
- (Inkrementelle) Emergenz von Anforderungen, Technologie und Team-Fähigkeiten
- Ermächtigung und Selbst-Organisation
- Befassung mit Realität, nicht mit Artifakten



# Von agilen Werte über Prinzipien zu Techniken

**Werte**

- Häufige Inspektion und Adaption
- Enge Kollaboration
- Ermächtigung zur Selbst-organisation
- Emergenz von Anforderungen u. Fähigkeiten
- Mut und Respekt

**Prinzipien**

- Frühe und regelmäßige Auslieferung
- Begrüßen von notwendigen Änderungen
- Enge Zusammenarbeit von Business und IT
- Direkte Kommunikation
- Funktionierende Software als Maß für Fortschritt
- Streben nach technischer Exzellenz
- Einfachheit
- Reflexive Verbesserung

**Techniken**

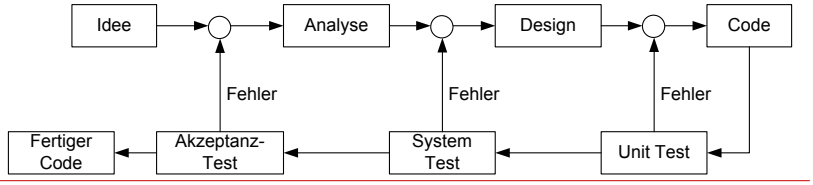
- Arbeiten in Paaren
- Test-getriebene Entwicklung
- Refactoring
- Laufende Integration
- Gemeinsame Verantwortung
- Programmier-richtlinien
- 40h-Woche
- Eigene Projekträum
- Priorisierte Anforderungen
- Einfaches Design
- Kunde vor Ort
- Tägliche Steh-Besprechungen
- Visuelle Fortschritts-berichte
- Demos
- Prototyping
- Verständliche Sprache
- Timeboxing
- Gemeinsame Anforderugns-definition
- Gemeinsamer Anwendungs-entwurf
- Unmittelbares Feedback
- Kurze Iterationen
- Häufige Synchronisation
- Optionen-Denken

# Feedback

- Früher
- Öfter – durch kürzere Zyklen, engere Zusammenarbeit
- Offener, direkter – verbal statt schriftlich
- Ohne Grenzen – durch ein großes Team

Kürzere Zyklen

- Von der Idee bis zur Umsetzung
- Vom Auftauchen einer Frage in beim Entwickler bis zur Antwort durch Kunden
- Vom Einführen eines Fehlers bis zur Erkennung
- Vom Definieren einer Architektur bis zum Realitätstest

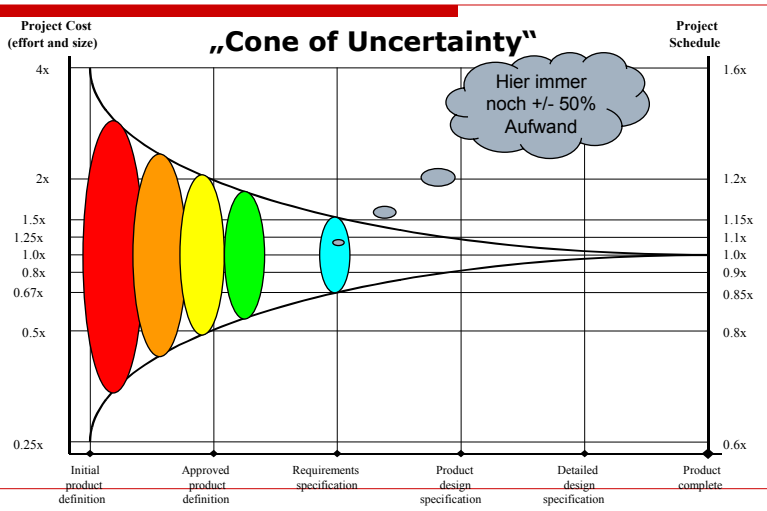


# Aufwandschätzung – traditionelle Vorgehensweise

- Ein paar Experten / Gurus schätzen den Aufwand.
  - Aber wer ist schon ein Experte für Aufwandschätzungen?
  - Wie gut kann ein Experte abschätzen, wie lange die Nicht-Experten später für die Umsetzung brauchen werden?
- Das passiert ganz am Anfang, z.B. auf Basis eines Pflichtenhefts.
  - Ist sichergestellt, dass die Schätzung auch während der Umsetzung aktualisiert wird?
  - Gibt es eine retrospektive Schätzung am Ende mit perfektem Wissen?
- Je besser die Erfahrung oder die Vergleichsdaten, desto besser ist die Schätzung.
  - Wie schaut es mit der Erfahrung aller Beteiligten aus, ist die berücksichtigt?
  - Wie genau passen die bisherige Erfahrung bzw. Vergleichsdaten auf die neue Situation (Team, Fachgebiet, Projektumstände, Technologie,...)?



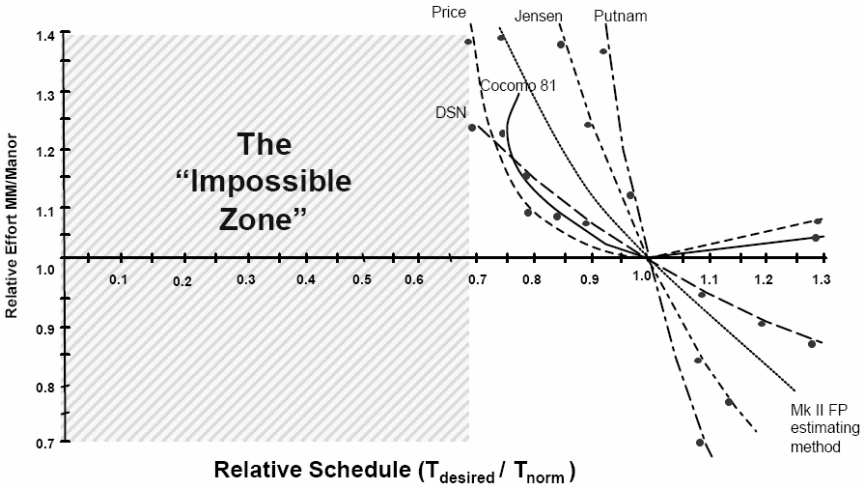
# Wann ist es zu früh, sich auf eine Schätzung festzulegen?






Adapted from Rapid Development: Taming Wild Software Schedules (McConnell 1996).

# Wie weit kann man den Zeitplan komprimieren?




Source: Adapted from Charles Simons, *Software Sizing and Estimating: Mk II*, John Wiley & Sons, 1991.

# Aufwandschätzung – die agile Antwort

- Das Projektteam schätzt selbst den Aufwand.
  - Jeder schätzt den Aufwand für seine Aufgaben, jeder fühlt sich verantwortlich für die Einhaltung der Aufwände.
  - Jeder kann seine Bedenken / Sicht einbringen.
  - Dadurch reduziert man das Fingerzeigen und Wegschieben der Verantwortung.
- Das passiert am Anfang einer jeden Iteration (2-4 Wochen) für kleine Projektteile.
  - Sobald man eine Anforderung detailliert, überlegt man auch schon die Aufwände für die Umsetzung.
  - Was man oft übt, kann man bald besser.
  - Kleine Teile kann man leichter schätzen.
- Basierend auf der Teamerfahrung aus früheren Projekten und aus den bisherigen Iterationen desselben Projekts.
  - Man verwendet die Regel „Yesterday's Weather".
  - Man lernt mit dem Projekt, passt sich automatisch an.

# Planung – traditionelle Vorgehensweise

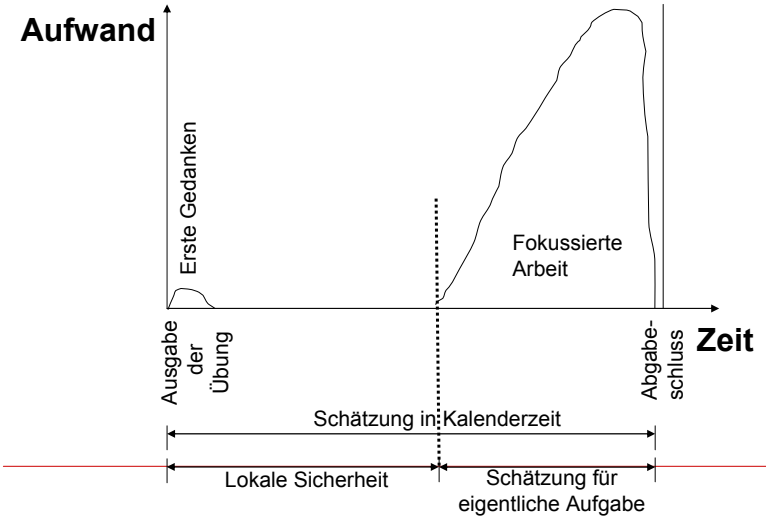
- Der Projektleiter erstellt den Projektplan am Projektbeginn mit vielen kleinen Arbeitsschritten und Abhängigkeiten.
  - Aber wie kann man am Anfang schon alles vorhersehen?
  - Aber wie kann der Auftragnehmer dann verstehen, welche Schritte wichtiger als andere sind?
  - Aber wie relevant sind die Abhängigkeiten?
- Der Projektleiter teilt die Arbeitsschritte den Projektmitarbeitern zu.
  - Aber was ist, wenn der Mitarbeiter etwas anders vorgehen möchte?
  - Wird der Projektleiter die Arbeitsschritte optimal verteilen können?
- Der Projektleiter ist froh, wenn er den Projektplan nicht oft ändern muss.
  - Aber wie aktuell kann dann der Projektplan sein?
  - Wie wertvoll ist ein Projektplan, wenn er nicht aktuell ist?



# Kernprobleme

- Projektmitarbeiter identifizieren sich nicht mit dem Projektplan des Projektleiters.
- Projektpläne sind nie aktuell, der Projektleiter verbringt viel zu viel Zeit mit dem Aktualisieren bzw. mit Werkzeugen wie Microsoft Project.
- Auftragnehmer verstehen Projektpläne nicht, können nicht zwischen wichtigen und unwichtigen Arbeitsschritten unterscheiden.
- Projektmitarbeiter arbeiten die vorgegebenen Arbeitsschritte einfach ab
  - Wenn sie länger brauchen, ist der Projektleiter schuld.
  - Wenn sie schneller wären, verbrauchen sie die Zeit.
  - Sie wissen wenig über die Arbeit ihrer Kollegen.
  - Sie wissen wenig über die Dringlichkeit ihrer Aufgaben.
- Projektleiter verwenden den Projektplan, um die Projektmitarbeiter zu kontrollieren.

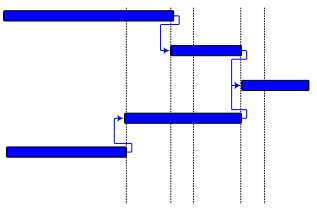
# Das “Studenten-Syndrom”






# Lokale Puffer verpuffen

- Entwickler lassen den lokalen Puffer ohne Stress verstreichen.
- Nachfolgende Arbeiten verzögern sich, sobald sich einer der Vorgänger verzögert.
- Eingesparte Zeit wird nicht weitergereicht, sondern verbraten.



# Planung – agile Antwort

- Das gesamte Team erstellt die Pläne für Releases mit groben Funktionsblöcken und für Iterationen mit feingranularen Funktionen bzw. Arbeitsschritten.
  - Jeder kann seine Bedenken / Sicht einbringen.
  - Dadurch reduziert sich das Fingerzeigen und Wegschieben der Verantwortung.
  - Der Auftraggeber kann die Pläne verstehen und Funktionen priorisieren.
- Die Teammitglieder wählen sich die Funktionen aus und sagen deren Umsetzung zu.
  - Jeder entscheidet selbst, wie er arbeitet.
  - Jeder weiß, was er am besten kann.
- Das Team plant nur die nahe Zukunft (aktuelle und nächste Release bzw. Iteration).
- Spätere Änderungen können leicht eingebaut werden.
  - Während der Iteration gibt es kein Umplanen.
  - Alles außerhalb des Planungshorizonts ist unproblematisch.



# Fortschrittskontrolle – traditionelle Vorgehensweise

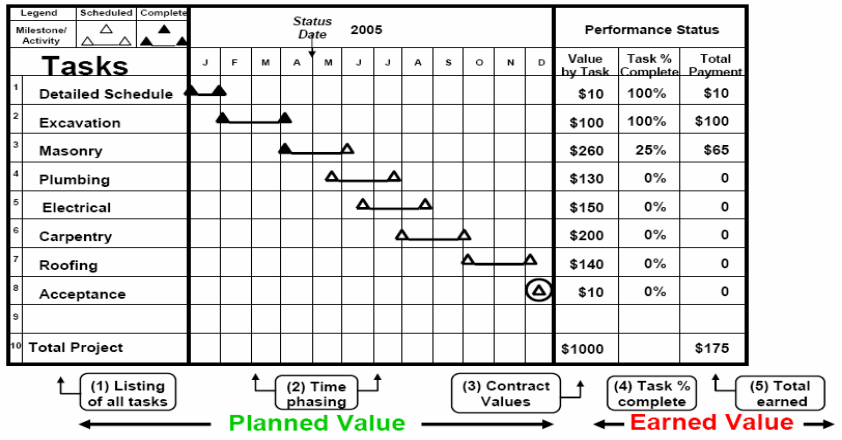
- Der Projektleiter berechnet den (scheinbaren) Fortschritt basierend auf „Earned Value". Dazu vergleicht er die geplanten Aufwände für die Arbeitsschritte mit den tatsächlichen.
  - Aber wie sehr kann man dieser berechneten Zahl vertrauen?
  - Wie aussagekräftig sind die ursprünglich geplanten Aufwände noch?
- Arbeitsschritte, die noch nicht zur Gänze erledigt sind, zählen prozentuell.
  - Warum schreiten Arbeiten anfangs schnell voran und dauern zum Schluss dann doch alle viel länger?
  - Wie objektiv und vergleichbar sind die Aussagen der Projektmitarbeiter („ich bin zu 80% fertig")?
  - Warum dauern die letzten 20% immer viel länger als die ersten 20% Prozent?
- Der Projektleiter möchte die Umsetzung kontrollieren und Abweichungen erkennen.
  - Aber was hat man von der Einhaltung des ursprünglichen Plans?
  - Wer lässt sich schon gerne kontrollieren?

# Kernprobleme

- Anfänglich geht alles nach Plan, doch das Projekt verzögert sich immer mehr.
- Wenn sich kritische Arbeiten verzögern, kann man den Fortschritt durch unkritische Arbeiten (über)kompensieren.
- Die Projektmitarbeiter haben kein Bewusstsein dafür, ob sich ihre Arbeiten auf dem kritischen Pfad befinden.
- Projektmitarbeiter geben die erwarteten Antworten: „Tell me how you will measure me and I will tell you how I will behave.“ (Eli Goldratt)



# Beispiel für „Earned Value“



# Fortschrittskontrolle – agile Antwort

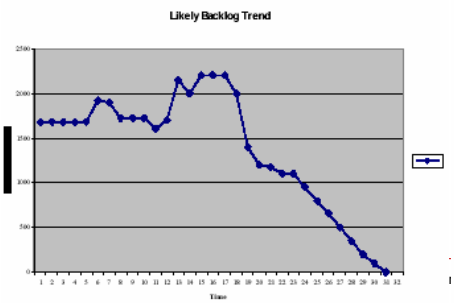
- Der Fortschritt im Gesamtprojekt basiert auf der tatsächlich ausgelieferten Funktionalität, nicht auf durchgeführten Arbeitsschritten.
  - Der Fortschritt bedeutet für den Auftraggeber somit unmittelbaren Geschäftsnutzen.
  - Wenn das Projekt vorzeitig beendet wird, ist der ausgewiesene Fortschritt tatsächlich schon realisiert.
- Der Fortschritt innerhalb einer Iteration stellt nur die noch geplanten Aufwände dar.
  - Der Blick ist nach vorne gerichtet.
  - Dazu werden die Funktionen, die gerade bearbeitet werden oder noch offen sind, regelmäßig neu geschätzt und bewertet.
- Das Team möchte aus der Fortschrittskontrolle lernen. Es möchte verstehen, welche Funktionalität gefährdet ist.
  - Das Team kann dadurch zusammenhelfen.
  - Arbeiten am kritischen Pfad bleiben nicht liegen, sondern werden gemeinsam angegangen

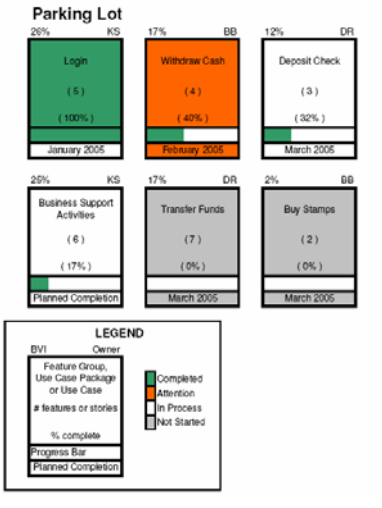


# Agile Beispiele

Burndown Diagramm (unten) stellt den noch offenen Aufwand dar.

Parking Lot (rechts) stellt Fortschritt nach Funktionsgruppen dar.



# Retrospektiven – traditionelle Vorgehensweise

- Im Prinzip sollte am Projektende eine Retrospektive („Lessons Learned") stattfinden.
  - Aber typischerweise hat man dann dazu keine Zeit mehr.
  - Aber typischerweise ist das Budget schon aufgebraucht.
- Das Projektteam zerstreut sich am Ende des Projekts wieder. Es ist unklar, wer von den Ergebnissen profitiert und ob die Ergebnisse überhaupt verwertet werden.
  - Viele Spezialisten sind beim Projektende schon gar nicht mehr im Team.
  - Werden die Projektmitarbeiter noch einmal an einem ähnlichen Projekt arbeiten? Werden sie noch einmal miteinander arbeiten?
  - An wen richten sich die Erkenntnisse?
  - Oft werden „Lessons Learned" nur deshalb erledigt, weil sie halt auch auf dem Projektplan vorgesehen waren.



# Retrospektiven – agile Antwort

- Retrospektiven werden am Ende jeder Iteration abgehalten.
  - Die Geschehnisse liegen noch nicht so weit zurück, man erinnert sich noch leicht daran.
  - Anfangs drängt die Zeit noch nicht so. Später hat man sich daran gewöhnt und sieht die Zeit dafür fix vor, weil man den Nutzen einschätzen kann.
  - Probleme werden früher erkannt.
  - Verbesserungen können länger wirken.
- Team bleibt noch bis zum Ende zusammen und kann die Ergebnisse verwerten
  - Jeder an der Retrospektive beteiligte profitiert von den Ergebnissen.
- Team lernt und wird selbst-korrigierend und selbst-heilend.
  - Wenn etwas nicht funktioniert, kann man das erkennen und an der Lösung arbeiten.
  - Wenn etwas gut funktioniert, kann man das verstärken und breiter kommunizieren.
  - Man kann mit minimalen Vorgaben starten und diese mit der Zeit ergänzen.
  - Man kann Ballast erkennen und abbauen.

# Softwareentwicklung ist Neuprodukt-Entwicklung

| Vorhersagbare Fertigung | Neuprodukt-Entwicklung |
|---|---|
| Man kann zuerst die **Spezifikation** erstellen und dann das System bauen. | Man kann nur selten am Anfang eine unveränderliche Spezifikation erstellen. |
| Am Anfang kann man verlässliche **Schätzungen** für Aufwand und Kosten abgeben. | Am Anfang kann man nicht verlässlich schätzen. Mit der Zeit bekommt man Erfahrungswerte als Basis für Schätzungen und Pläne. |
| Man kann detaillierte **Arbeitsschritte** identifizieren, definieren und planen. | Am Anfang kann man das nicht. Kurze Feedback-Zyklen sind notwendig, um sich geeignet anpassen zu können. |
| Anpassung an unvorhergesehene **Änderungen** ist nicht die Norm, die Änderungsrate ist gering. | Kreative Anpassung an unvorhergesehene Änderungen ist die Norm, die Änderungsrate ist hoch. |

Eine "Wasserfall"-artige Vorgehensweise mit dicken Anforderungsspezifikationen und Aufwandschätzungen am Anfang, auf denen spekulative Pläne aufbauen, ist lange Zeit fälschlicherweise für Neuprodukt-Entwicklung verwendet worden.

Nach: Craig Larman: *Agile & Iterative Development, 2003*



# Es geht um einen Kulturwechsel

- **Priorisierung der Anforderungen mit häufiger Auslieferung** statt **„alles am Ende"**
- **Iterationen mit fixer Länge und fixen Kosten** statt **überzogener Zeit- und Budgetrahmen**

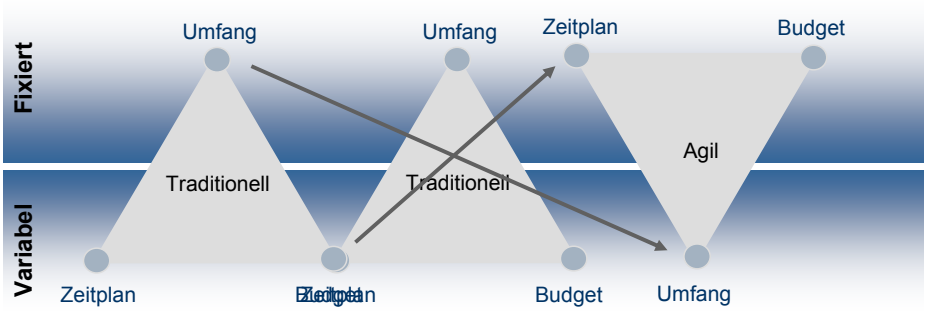
**Traditionelle Vorgehensweise**

Umfang → Budget → Zeitplan

Zeitplan → Budget → Umfang

**Agile Vorgehensweise**

# Beim Umstieg auf agile Methoden ändert sich der Fokus






# Doch dann beginnt die Reise erst...

Durch den Kulturwechsel werden große Produktivitäts-steigerungen möglich

- Von einem Gegeneinander zu einem Miteinander
- Kein Vergeuden von Ressourcen mehr, sondern fokussiertes Arbeiten
- Laufendes Lernen und laufende Verbesserungen

Die Reise setzt auf bewährte Techniken aus:

- Lean Software Development
- Theory of Constraints

# Lean Software Development with 7 Principles and 22 Tools

- Eliminate Waste
  - Seeing Waste, Value Stream Mapping
- Amplify Learning
  - Feedback, Iterations, Synchronization, Set-Based Development
- Decide as Late as Possible
  - Options Thinking, The Last Responsible Moment, Making Decisions
- Deliver as Fast as Possible
  - Pull Systems, Queuing Theory, Cost of Delay
- Empower the Team
  - Self-Determination, Motivation, Leadership, Expertise
- Build Integrity In
  - Perceived Integrity, Conceptual Integrity, Refactoring, Testing
- See the Whole
  - Measurements, Contracts

From: Mary and Tom Poppendieck: *Lean Software Development*, 2003



# Theory of Constraints

- In einem System gibt es zu jeder Zeit genau einen Flaschenhals, genau so wie es bei einer Kette genau ein schwächstes Glied gibt.
- Wenn man am Durchsatz des Systems erhöhen möchte, muss man sich auf den Flaschenhals konzentrieren.

5 Focusing Steps

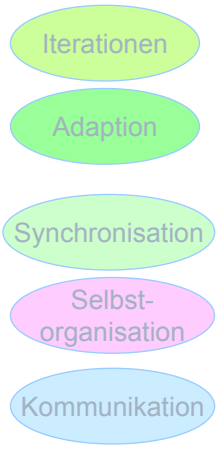
1. Identify the constraint
2. Exploit the constraint
3. Subordinate to the constraint
4. Elevate the constraint
5. Back to step 1

- Know what to change!
- Know to what to change to!
- Know how to cause the change!

http://www.goldratt.com

# Die Ursprünge von agiler Softwareentwicklung gehen weit zurück

- 70er
  - Light Airborne Multipurpose System (1975), 200 Personenjahre, Millionen Zeilen Code, 45 einmonatige Iterationen, alle rechtzeitig und innerhalb des Budgetrahmens
  - Software für das Space Shuttle: 17 inkrementellen Releases innerhalb von 31 Monaten (Okt. 1977 – Apr. 1980)
- 80er:
  - Toyota mit Lean Manufacturing
  - The Knowledge Creating Company
- 90er:
  - MIL-STD-498 u.ä. schreibt iterative und evolutionäre Vorgehensweise vor
  - Boom agiler Methoden als Gegenbewegung zu CMM/CMMI






# Abschlussworte

Lou Gerstner:
„Cultural change must be led from the top if it is to be effective."

(in *Who Says Elephants Can't Dance?*, 2002)

# Fragen?



# Weitere Schritte

- **Informationsveranstaltung zum Thema Agilität**
- **Potenzialanalyse**
  - Wie agil und lean ist Ihre Anwendungsentwicklung?
- **Unterstützung bei Auswahl, Gestaltung und Bewertung von Vorhaben (Business Case Analyse)**
- **Umsetzung konkreter Projekte**
- **Anfragen an Christoph Steindl**

Christoph_Steindl@at.ibm.com
0664/6185186