Agile Inspektionen

Übersicht

1. Wasserfallmodell und klassische Inspektionen
2. Agile Software-Entwicklung
   - Prinzipien
   - Pair Programming und anderes „Reviewartiges“ in der agilen Software-Entwicklung
3. Agile Inspektionen (auch für Wasserfallprojekte)
4. Diskussion: wie agil sind Agile Inspektionen wirklich?

Dipl.-Inform. Peter Rösler
ReviewConsult

1. Wasserfallmodell und klassische Inspektionen

- Die in diesem Beitrag betrachteten „Software-Inspektionen“ werden oft auch als „Software-Reviews“, „Peer Reviews“ oder „Fagan/Gilb style inspections“ bezeichnet.

  Im Folgenden gilt auch: „Artefakte“ = „Dokumente“

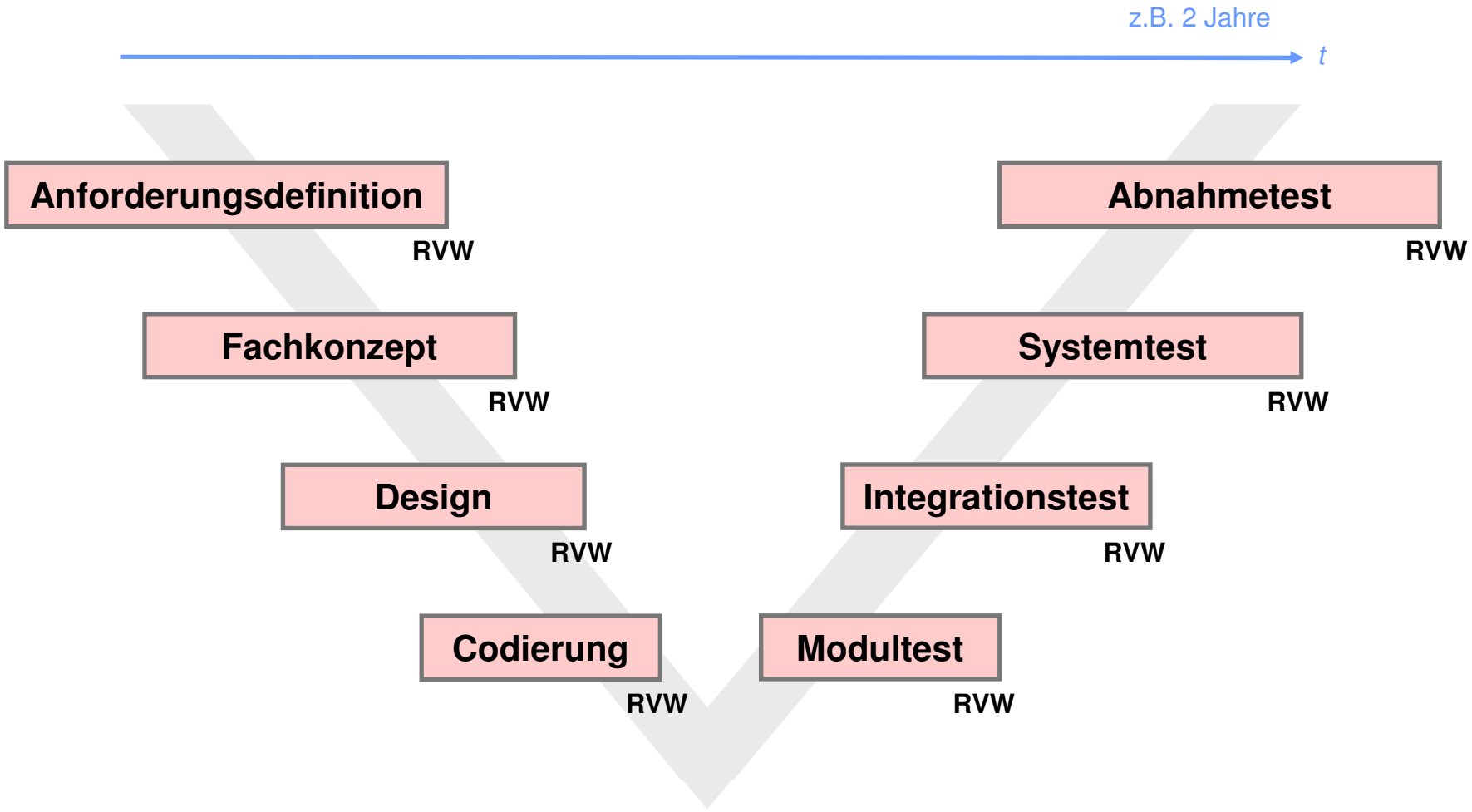
- Definitionen:

  „major defect” (im Gegensatz zu „minor defect”) :

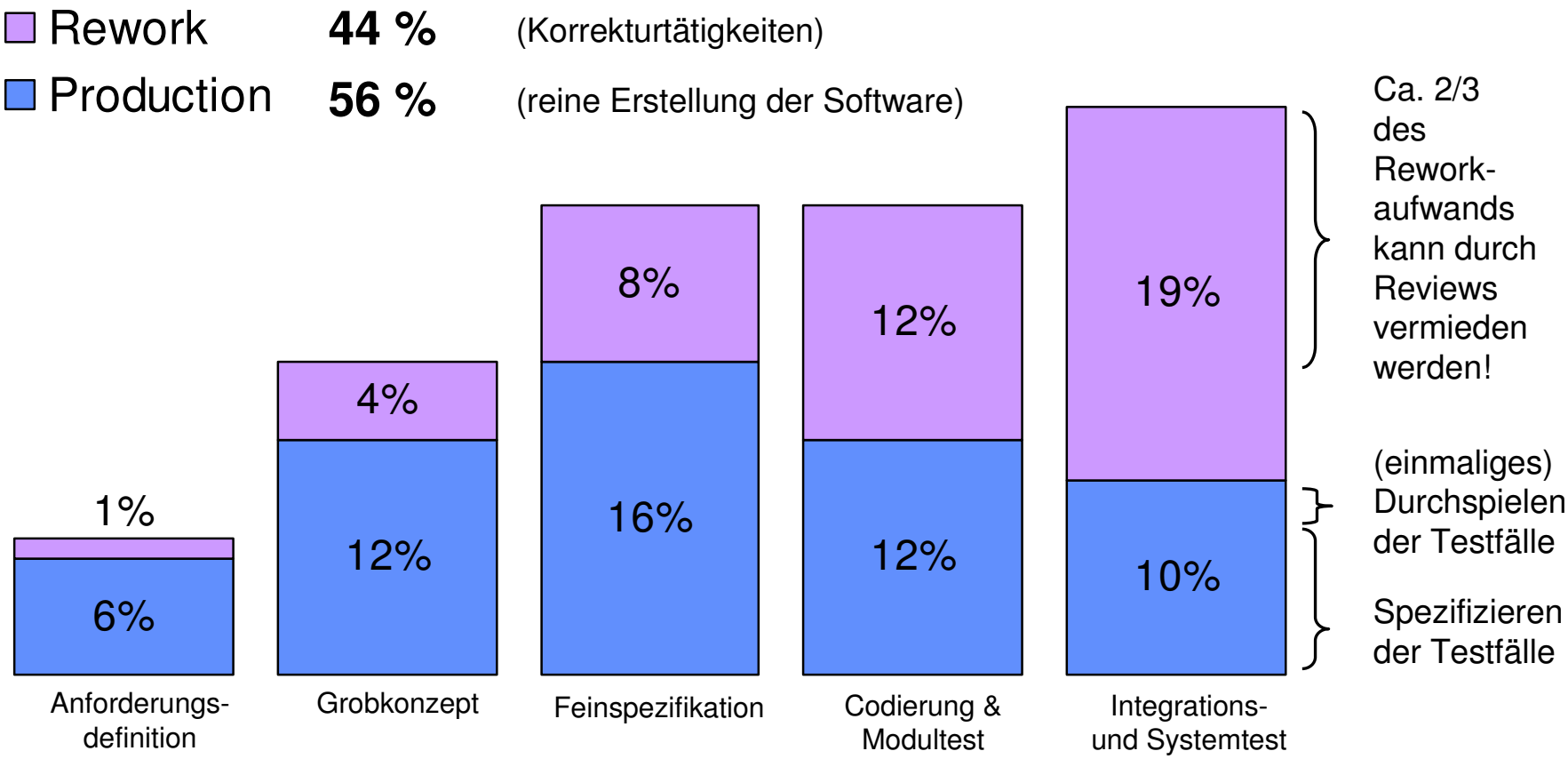
  Fehler, der möglicherweise erheblich höhere Kosten verursacht, wenn er später gefunden wird als jetzt.

  1 Seite = 300 Wörter [4]

Typisches Wasserfall- bzw. V-Modell



Anteil von Korrekturtätigkeiten am Gesamtaufwand

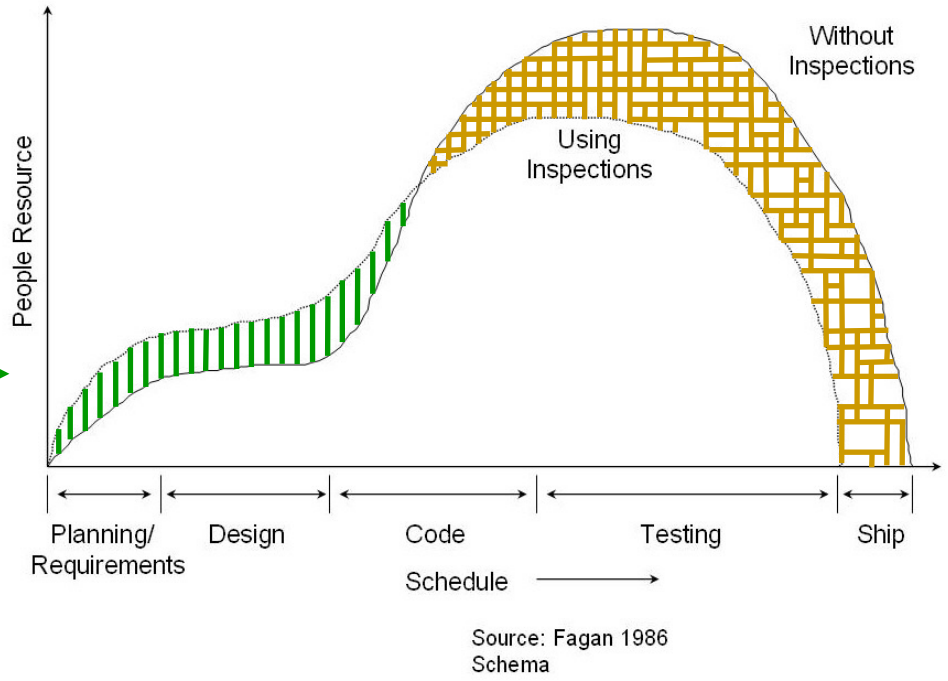


Quelle: Wheeler 1996, Software inspection: an industry best practice, p 9

Aufwand für 100%ige Reviewabdeckung

Gilb: ca. 10-15% des Projektaufwands (Radice: ca. 8-20%)

Aufwand:
z.B. 500 h
verteilt auf 25
Reviews



Einsparung:
z.B. 2000 h
eingespart durch
250 gefundene
Mj defects

ROI = 4:1

Details siehe Notizseite

Review-Varianten (nach Graham)

Walkthrough [Presentation Review] Aktivität: Verstehen

- Autor leitet die Gruppe durch das Dokument und die dahinter stehenden Gedankenprozesse, so dass alle dasselbe verstehen und Einigkeit erzielt werden kann, was zu ändern ist.

Management-Review [Projektstatus-Review] Aktivität: Entscheiden

- Gruppe diskutiert das Dokument und trifft Entscheidungen über den Inhalt, z.B. wie etwas gemacht werden soll, Ja/Nein-Entscheidungen.

Inspektion Hauptaktivität: Fehler finden

- Formale Einzel- und Gruppen-Prüfaktivität nach genau festgelegten Regeln, unter Verwendung von Vorgängerdokumenten und Standards.

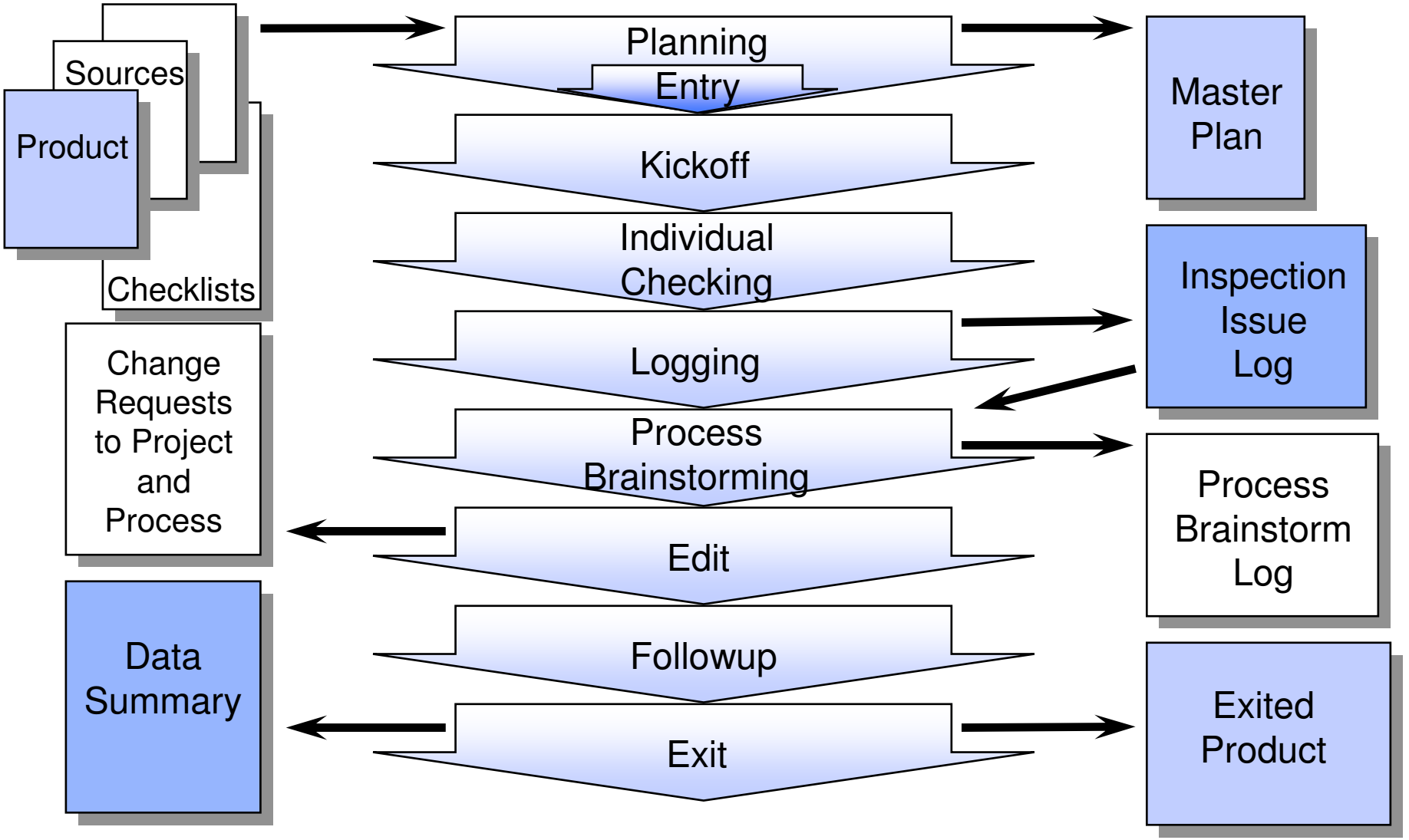
Der IEEE Std 1028-2008 nennt noch „Audits“ und „Technical Reviews“

Rollen in einer Inspektion

- Moderator
- Autor
- Protokollführer
- Reviewer
- Vorleser/Reader (nur wenn „double checking“ gemacht wird)

Ein Teilnehmer kann mehrere Rollen übernehmen.
Einzige Einschränkung: der Autor darf zusätzlich höchstens die Rolle eines Reviewers übernehmen.

Phasen einer Inspektion nach Gilb/Graham



Source: Tom Gilb,
Team Leader Course

Phase „Individual Checking“

- Von den Fehlern, die das Reviewteam insgesamt entdeckt, werden 80% im „Individual Checking“ gefunden und 20% im „Logging meeting“, sofern dort mit „Double Checking“ erneut Fehlersuche betrieben wird. [2]
- Da in den meisten durchgeführten Reviews auf „Double Checking“ verzichtet wird, werden in der Praxis eher 95% der durch das Reviewteam entdeckten Fehler im „Individual Checking“ gefunden. [3]

Nutzen von Reviews

Für Wasserfall-Projekte:

- geringere Entwicklungskosten (25-35%)
- kürzere Entwicklungszeiten (25-35%)
- geringere Wartungskosten (Faktor 10-30)
- höhere Zuverlässigkeit (10-100 mal weniger Fehler)

Source: Gilb1988, Principles of Software Engineering Management, Table 12.2
vgl. auch SI p 24

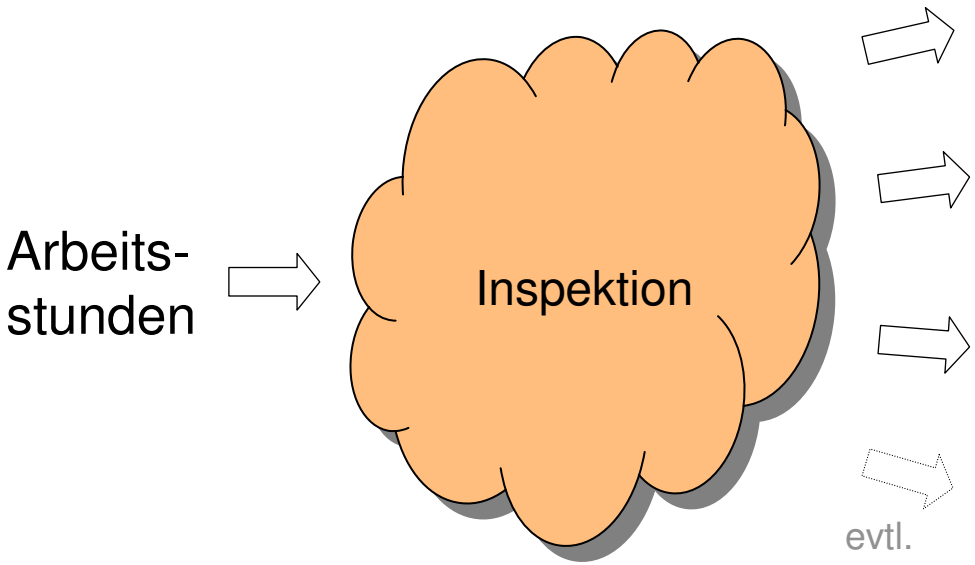
Black-Box-Sicht auf Inspektionen

In

(Kosten)

Out

(Nutzen)



gefundene Major Defects
d.h. eingesparte Reworkstunden

Prozessverbesserungs-Vorschläge

Kennzahlen
Fehlerdichte etc.

Lernkurve des Autors
geringere zukünftige Fehlerrate

Source: Peter Rösler, www.reviewconsult.de, 2010

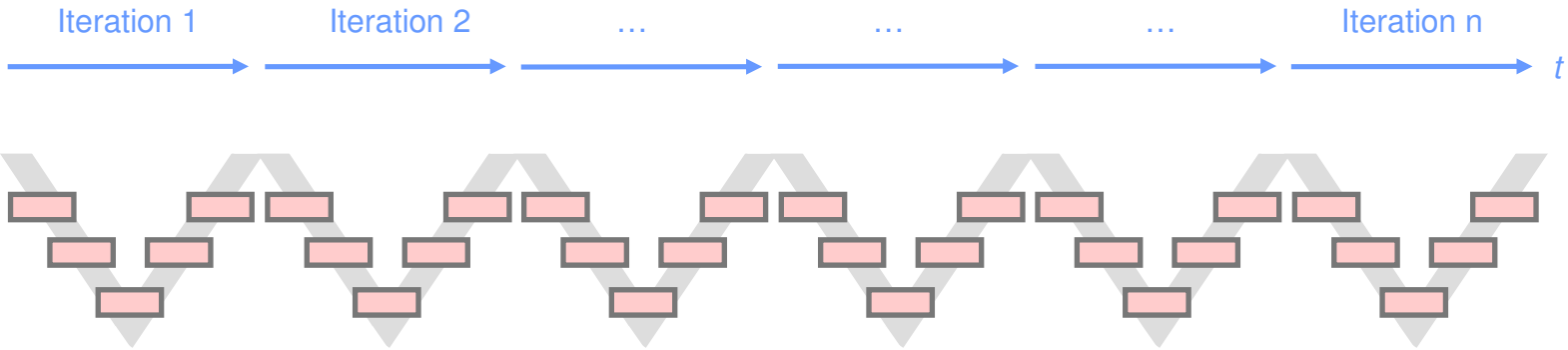
2. Agile Software-Entwicklung
2.1 Prinzipien

Agiles Manifest (2001, Utah) als Wertesystem:

Individuen und Interaktionen	sind wichtiger als	Prozesse und Werkzeuge
Funktionierende Software	ist wichtiger als	umfangreiche Dokumentation
Zusammenarbeit mit dem Kunden	ist wichtiger als	Vertragsverhandlungen
Reaktion auf Änderungen	ist wichtiger als	Verfolgung eines festgelegten Plans

17 Erstunterzeichner, darunter Kent Beck (XP), Alistair Cockburn (Crystal), Ken Schwaber und Jeff Sutherland (Scrum).

Das System entsteht in vielen Iterationen



- Iterationen (oder „*sprints*“) dauern üblicherweise 1 bis 4 Wochen.
- Am Ende jeder Iteration steht ein getestetes und lauffähiges System („*potentially shippable code*“) mit neu dazu gekommener Funktionalität.

Vorteile gegenüber Wasserfall

- Schnelle Reaktionsmöglichkeit auf sich ändernde Anforderungen oder Prioritäten
- Sehr früher Return on Investment
  (Schon nach der ersten Iteration hat der Kunde ein lauffähiges System, das er produktiv einsetzen kann.)
- Ein komplettes Scheitern des Projekts (kein lauffähiger Code), wie bei Wasserfall-Projekten oft der Fall (vgl. CHAOS-Report), ist wenig wahrscheinlich.
- … (und viele weitere Vorteile)

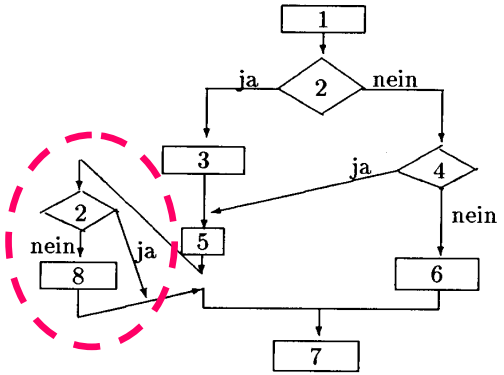
Testautomatisierung

- Viele Iterationen bedeuten auch viel Testaufwand.
- In jeder Iteration muss nicht nur die neue Funktionalität, sondern wegen möglicher Seiteneffekte die gesamte Software durchgetestet werden.
- Manuelle Tests sind aufwändig
  (Schreckensszenario: in jeder Iteration sitzen Tester da, starten manuell Skripts, um die Datenbank zu füllen, klappern die gesamte Benutzeroberfläche manuell ab, denken jedes Mal nach, ob die angezeigten Werte stimmen, …)
- Lösung: möglichst vollautomatische Regressionstests

s.a. Folien 23-24 (Test Driven Development)

Problem Aufwandskurve

- Wenn die neue Funktionalität in jeder Iteration auf die jeweils bequemste Art und Weise eingebaut wird (z.B. durch „Rucksäcke“ oder andere Anti-Pattern wie „Spaghetti-Code“ oder „Copy-and-Paste“), dann wird die Software schnell unwartbar.



aus: Wunderatsch, 1999

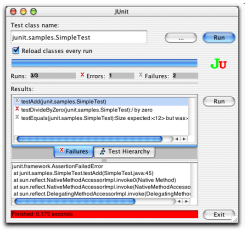
- Die Produktivität des Teams („*velocity/speed*“) wird in den Folge-Iterationen immer geringer. Die „Aufwandskurve“ (Arbeitstage je Feature) steigt also steil an.
- Vorbeugende Lösung: ständiges „Refactoring“

Refactoring

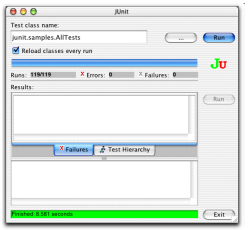
- Refactoring (Refaktorisierung/Restrukturierung): Umbau von Code, um ihn in eine optimale Form zu bringen, also wartbarer zu machen (ohne dabei die Funktionalität zu verändern)
- Beim manuellen Refactoring können ungewollt wieder Fehler eingebaut werden.
  => „Sicherheitsnetz“ aus automatischen Regressions-tests (mit hohem Testüberdeckungsgrad) nötig
- Entwicklungsumgebungen (z.B. Eclipse-SDK) unterstützen durch automatisierte Refactorings (Verschieben von Klassen, Umbenennen von Bezeichnern, etc.)

Test-Driven Development (TDD)

- Neuer Code wird zyklisch wie folgt erstellt:
- „Red-Green-Refactor“-Zyklus:
  1. Ein Test wird geschrieben, der zunächst fehlschlägt.
     (Im Test-Framework **rot** dargestellt.)
  2. Genau so viel neuer Code wird implementiert, dass der Test erfolgreich durchläuft.
     (Wird **grün** dargestellt, sobald der Code richtig ist.)
  3. Code (und Test!) werden refaktorisiert.
     (Wird **grün** dargestellt, wenn beim Umbau keine Fehler eingebaut wurden.)
- Ein Zyklus ist meist wenige Minuten lang.



z.B. JUnit



Test-Driven Development (2)

- Prinzipien von TDD: kontinuierliche Designverbesserung, einfaches Design und Test-First
- TDD ist also keine Test-, sondern eine Designstrategie.
- Das Test-First Prinzip von TDD fördert einen hohen Testüberdeckungsgrad und eine möglichst vollständige Testautomatisierung.

← T-Shirt von codesmack

2.2 Pair Programming und anderes „Reviewartiges“ in der agilen Software-Entwicklung

Drei Beispiele aus unterschiedlichen agilen Methoden werden vorgestellt:

- Pair Programming in XP (Extreme Programming)
- Design- und Code-Reviews in FDD (Feature Driven Development)
- Sprint Review Meetings in Scrum

Pair Programming in XP

- Der „Pilot“ („*Driver*“) schreibt den Code.
- Der „Navigator“ („*Observer*“) prüft den Code und denkt über Verbesserungen am Design nach. Wenn nötig, wird sofort diskutiert.
- Die Rollen werden oft gewechselt, z.B. alle paar Minuten. („Let me drive“)
- Die Paarzusammensetzungen im Team werden beispielsweise zweimal am Tag gewechselt.

aus http://en.wikipedia.org/wiki/Pair_programming

Pair Programming in XP (2)

Vorteile:

- Höhere Qualität (je nach Studie 15% bis 50%)
  - viele Fehler können sofort entdeckt werden
  - ein besseres Design kann gefunden werden
- Die Partner lernen voneinander.
- Mindestens zwei Entwickler kennen sich mit jedem Teil des Codes sehr gut aus. (Gut für den „Lastwagenfaktor“)

Aufwand:

- Nicht etwa 100% mehr als bei Einzelprogrammierung, sondern (je nach Studie) 15% bis 85% mehr.

(Diesem Mehraufwand stehen durch die verbesserte Qualität Einsparungen später im Projekt gegenüber.)

Design- und Code-Reviews in FDD

- Für jedes zu entwickelnde Feature gibt es folgende Meilensteine:
  - Domain Walkthrough
  - Design
  - **Design Inspection** (Peer review your design and acceptance tests with stakeholders)
  - Code
  - **Code Inspection** (Peer review code and acceptance tests)
  - Promote to Build
- Der Chef-Programmierer als Leiter des Teams bestimmt den Grad der Formalität der jeweiligen Inspektion.

Design- und Code-Reviews in FDD (2)

- Inspektionen in FDD: Gefahr von psychologischen Problemen ist minimiert, weil der Owner des zu prüfenden Artefakts das gesamte Feature-Team ist (statt einer Einzelperson)
- FDD bezieht sich auf wissenschaftliche Untersuchungen: Code-Inspektionen beseitigen Fehler mit weniger Aufwand als Tests.

  Weitere positive Wirkungen: Wissenstransfer, Programmierrichtlinien und andere Prozess-Standards setzen sich durch, etc.
- Fazit: FDD vertraut voll auf klassische Inspektionen.

Sprint Review Meetings in Scrum

- An Ende eines Sprints erfolgt ein informelles Review durch Team, Product Owner und Stakeholder.
- Dazu wird das Ergebnis des Sprints (die laufende Software, keine Folien!) durch das Team vorgeführt. („The demo“)
- Product Owner und Stakeholder geben Feedback, das in die weitere Arbeit mit einfließt.

Source: Mountain Goat Software

Sprint Review Meetings in Scrum (2)

- Kein Review im Sinne einer Inspektion, welche Fehler auf Dokument-Ebene finden will.
- Es gibt eher einige Merkmale eines Abnahmetests.

Source: Mountain Goat Software

- Behauptung: Scrum hat keine „reviewartigen“ Mechanismen im Sinne von Inspektionen.
- Empfehlung: Scrum (mindestens) um Pair Programming oder Inspektionen ergänzen.

3. Agile Inspektionen

- Agile Inspektionen* wurden von Tom Gilb im Jahr 2005 vorgestellt [1].

Tom Gilb

- Autor des Buches „Software Inspection“ (1993, Coautor Dorothy Graham)
- Mit seinem Buch „Software Metrics“ (1976) war er der Ideengeber für CMM/CMMI Level 4.
- „Erster Agilist“
  Mike Cohn: „I’ve always considered Tom to have been the original agilist. In 1989, he wrote about short iterations (each should be no more than 2% of the total project schedule). This was long before the rest of us had it figured out.“

Tom Gilb
www.gilb.com

- Agile Inspektionen sind besonders in Wasserfall-Projekten nützlich, weil es dort keine kurzen Feedback-Schleifen (z.B. in Form von Iterationen) gibt.

* Andere Bezeichnungen: „Extreme Inspections“, „Agile Specification Quality Control (SQC)“ [5]

Eine Beobachtung als Ausgangspunkt

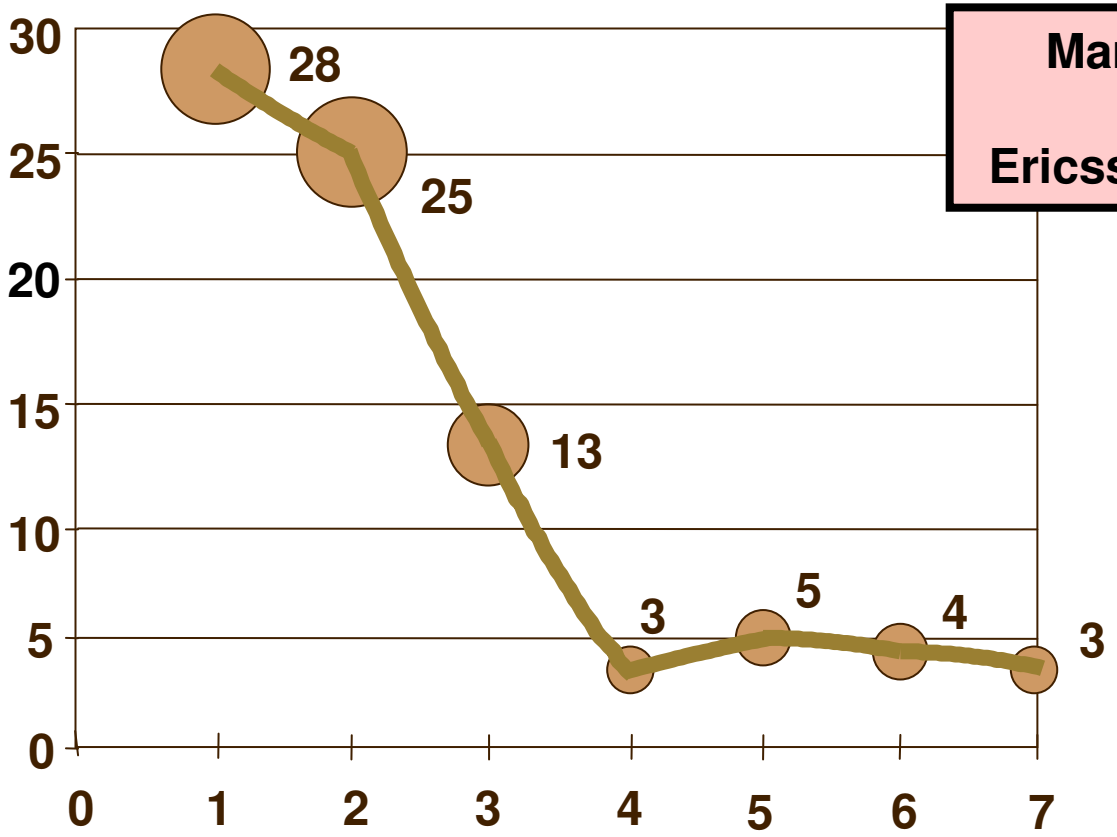
Bei *klassischen* Inspektionen:

- Die Autoren beginnen manchmal aufgrund der Review-Erfahrungen, deutlich fehlerfreier zu arbeiten.
- Eine signifikante Lernkurve kann erreicht werden.

Es folgen vier Folien mit Beispielen.

Marie's persönliche Lernkurve

Geschätzte verbleibende Major defects / Seite

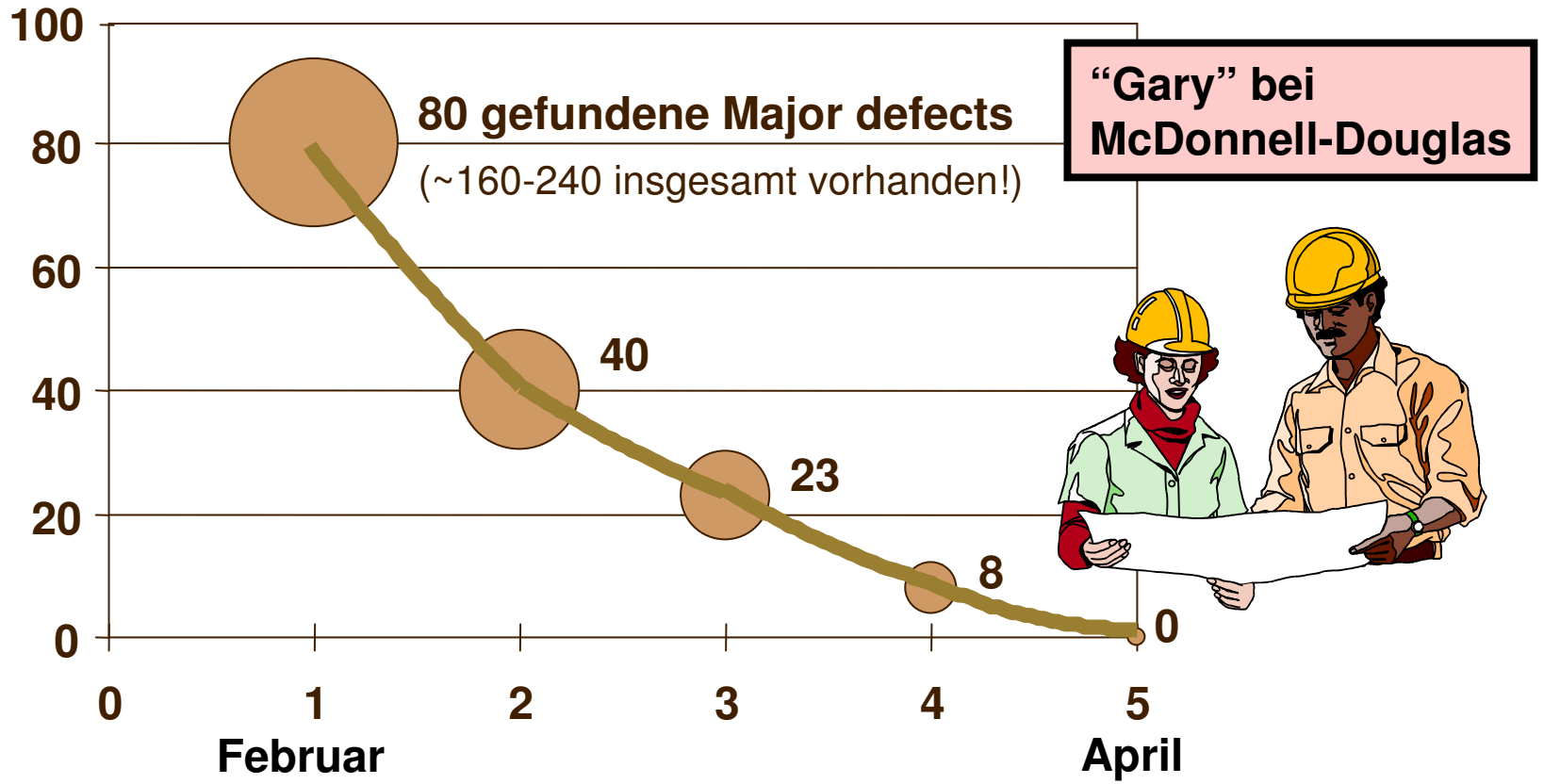


Marie Lambertsson's Lernkurve, Ericsson, Stockholm, 1997

Reihenfolge der Dokumente, die einem Review unterzogen wurden

Gary's persönliche Lernkurve

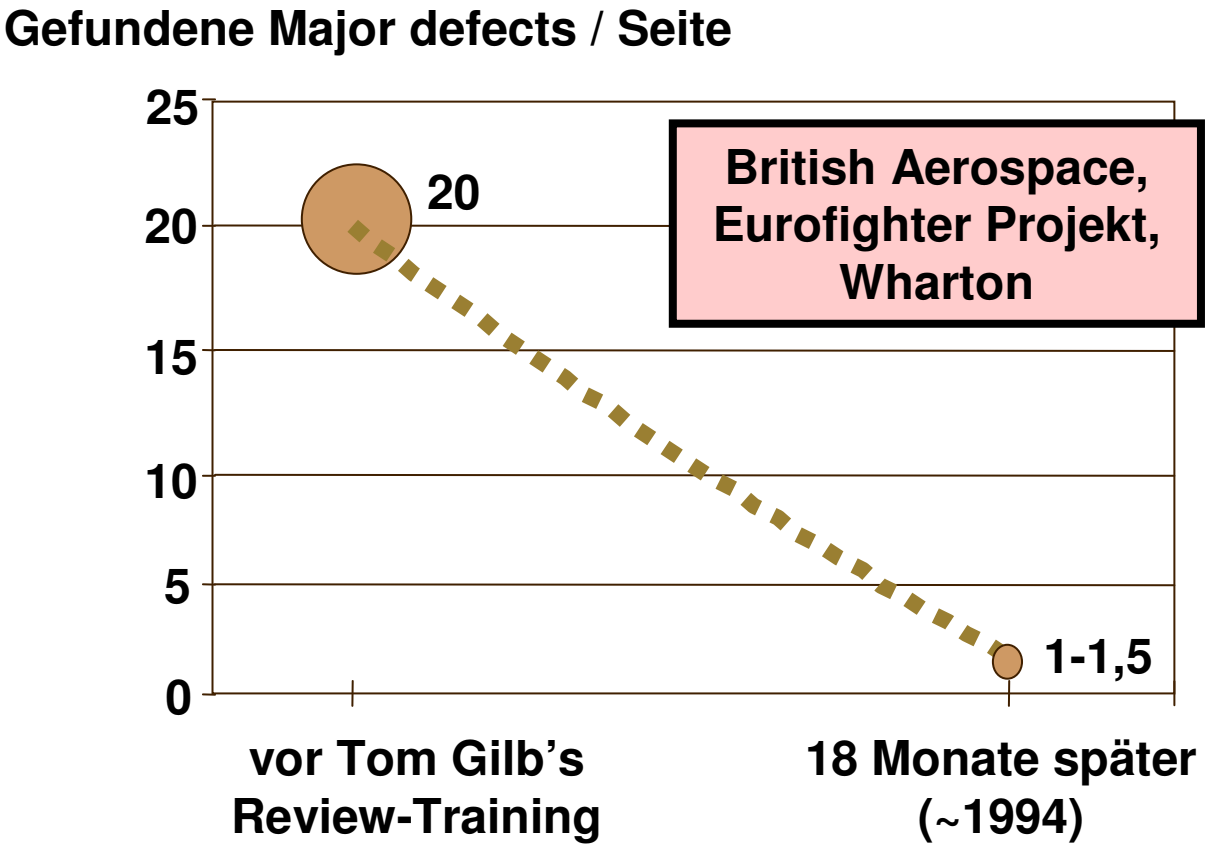
Gefundene Major defects / Seite



Reviews von Gary's Design-Dokumenten

Quelle: Douglas Aircraft, 1988, private report to Tom Gilb

Lernkurve einer gesamten Organisation



Weiteres Beispiel bei Douglas, 1988: innerhalb des ersten Jahres stieg die Qualität um Faktor 60, gemessen am Anteil der Konstruktions-zeichnungen, die abgelehnt und korrigiert werden mussten (0.5% gegenüber ca. 30% vorher).

Weiteres Beispiel bei Raytheon: siehe Notizansicht der Folie.

Quelle: Tom Gilb
www.gilb.com

Lernkurve und positive Motivation

- Der normale Entwickler ist i.A. fähig, seine Fehlerrate um 50% zu senken nach jedem Zyklus von persönlichem Lernen und Feedback. (Tom Gilb's pers. Erfahrung, 1988 und später).
- Diese Reduktion ist nur zu erwarten, wenn die Autoren positive Motivation haben, Dokumente mit guter Qualität zu produzieren. Quantifizierte Exit-Kriterien, die vom Management verlangt werden, können eine solche Motivation sein.
- Es gibt keine Hinweise, dass die Autoren signifikant länger brauchen, um diese Dokumente zu produzieren! (Tom Gilb, Review Symposium 06/2005)

Niels Malotaux:
»Die 'Null-Fehler'-Haltung führt meiner Erfahrung nach sofort zu 50% weniger Fehlern.«

Grundidee der Agilen Inspektionen

Der Schwerpunkt der Inspektionen wird verschoben

- weg vom frühzeitigen Fehler finden und korrigieren („cleanup“-Modus)
- hin zum Schätzen der Fehlerdichte der Dokumente, um die Entwickler zu motivieren, zu lernen wie man von vorne herein fehlerfreier arbeitet.

Die Kosten für Inspektionen sinken enorm:

- Stichproben reichen aus (statt 100% der Dokumente zu prüfen), denn der Zweck ist jetzt „Messen“ statt „cleanup“-Modus.

Ziel von Agilen Inspektionen

Hauptziel:

- Die Entwickler zu motivieren, ihre Fehlerrate zu senken

Zusätzliche (klassische) Inspektions-Ziele:

- Verhindern, dass zu fehlerhafte Dokumente in die folgenden SW-Entwicklungsphasen gelangen
  (Terminverzögerungen und Qualitätsprobleme wären sonst die Folge)
  Haupttaktik: numerische Exit-Kriterien wie „maximal 1,0 Mj / p“.
- Gültige Prozess-Standards lehren und durchsetzen

Allgemeine Prinzipien von Agilen Inspektionen

- Wenige Seiten auf einmal (z.B. 1 bis 3 Seiten)
- Evtl. frühzeitig (erste 5% eines großen Dokuments)
- Kontinuierlich (z.B. jede Woche), bis die Arbeit fertig ist
- Für jeden Entwickler (jeder einzelne Autor muss persönlich motiviert und trainiert werden)

Ablauf einer Agilen Inspektion (1)

- Eine Stichprobe wird ausgewählt (z.B. 1 Seite) und gegen ca. 3 bis 7 Regeln geprüft, z.B.
  - Clarity („clear enough to test“)
  - Unambiguous („to the intended readership“)
  - Consistent („with other statements in the same or related documents“)
  - Completeness („compared to sources“)
  - Für Anforderungsdokumente: „no design“
- Die Reviewer sollen alle Abweichungen von diesen Regeln identifizieren und klassifizieren. Die Mj Defects werden an den Moderator berichtet.

Ablauf (2)

- An der Prüfsitzung nehmen beispielsweise zwei Reviewer teil, Dauer ca. 30-60 Minuten. Geprüft wird mit optimaler Inspektionsrate. Ein trainierter Moderator ist anwesend und leitet den Prozess.
- Danach wird die geschätzte Anzahl der tatsächlich vorhandenen Fehler aus der Gesamtzahl der gefundenen Fehler berechnet.
  (Berechnungsgrundlage: typischerweise findet das Team in diesem Setting ein Drittel der vorhandenen Fehler.)

Details siehe Notizseite

Ablauf (3)

- Als Exit-Kriterium wird anfangs ein Wert wie beispielsweise „maximal 10 Mj / p“ angesetzt.
  - Nach ein paar Monaten „Kulturänderung“ sollte man das Limit eher bei „maximal 1 Mj / p“ ansetzen.
- Maßnahmen werden festgelegt
  - Bei sehr hoher Fehlerdichte (z.B. 10 Mj / p oder mehr):
    Es wäre unökonomisch, die anderen Seiten des Dokuments zu prüfen, um „alle Fehler“ zu finden, oder die bisher gefundenen Fehler zu korrigieren. Es würden trotzdem zu viele Mj Defects unentdeckt bleiben.
  - Beste Alternative: Autor oder jemand anderes schreibt Dokument **neu**. (Hier sieht man, dass eine frühzeitige Agile Inspektion Sinn macht, z.B. schon wenn 5% des Dokuments fertig sind.)

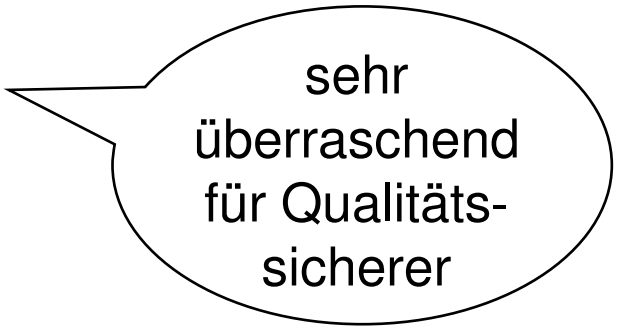
Prüfung ohne Vorgängerdokumente

- Anders als bei klassischen Inspektionen wird nicht gegen die Vorgängerdokumente geprüft, sondern nur gegen das Wissen der Reviewer.
- Dadurch wird die Prüfung beschleunigt, allerdings ist das Ergebnis ungenauer.
  (Wenn aber die Stichprobe ergibt, dass die Fehlerdichte 50 Mj / p oder mehr beträgt, was laut Gilb [1] durchaus normal ist, dann besitzt allein das schon genug Aussagekraft.)

Gilb's Paradigmenwechsel

Die unmittelbare Lösung gegen hohe Fehlerdichten

- ist nicht, die gefundenen Fehler aus dem Dokument zu entfernen,
- und ist nicht, den SW-Entwicklungs-Prozess zu ändern!



Die effektivste praktikable Lösung:

- Sicherstellen, dass jeder Autor das Exit-Kriterium der maximalen Fehlerdichte ernst nimmt.

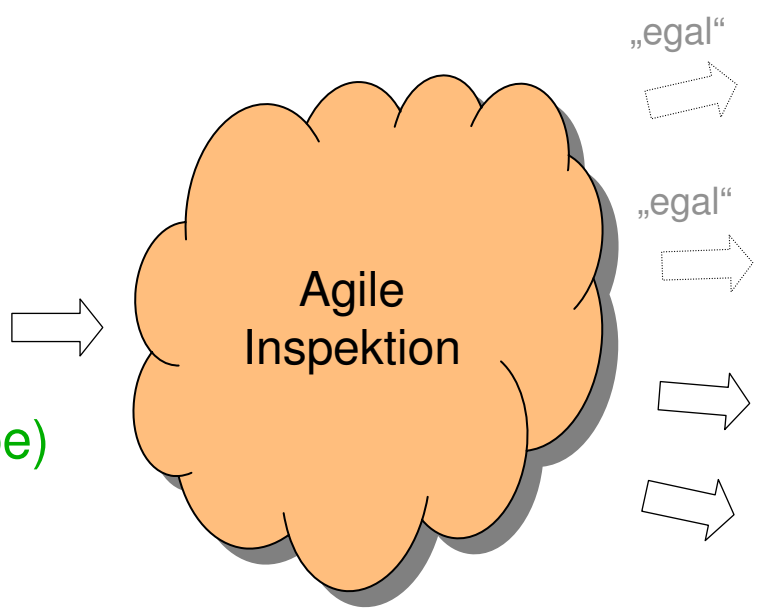
  (Im Durchschnitt sollte nach jeder Agilen Inspektion die Fehlerrate eines Autors um ca. 50% sinken.)

Black-Box-Sicht auf Agile Inspektionen

In

(Kosten)

wenige Arbeits-stunden (Stichprobe)



Out

(Nutzen)

gefundene Major Defects

Dienen nur zum Messen der Fehler-dichte und um dem Autor zu zeigen, welche Art von Fehlern er macht.

Prozessverbesserungs-Vorschläge

Kennzahlen (Fehlerdichte)

Lernkurve des Autors

Geringere Fehlerrate (ca. 50%. Kumuliert über mehrere Inspektionen ist eine um eine Größenordnung niedrigere Fehlerrate gut erreichbar, vgl. Folien 29-31)

Source: Peter Rösler, www.reviewconsult.de, 2010

4. Wie agil sind Agile Inspektionen wirklich?

Contra:

- Da Agile Inspektionen besonders in Wasserfall-Projekten nützlich sind, ist das Wort „agil“ in dieser Hinsicht irreführend.

Pro:

- Agile Inspektionen sind viel leichtgewichtiger als klassische Inspektionen. (Einfacherer Prozess, deutlich weniger Aufwand)
- Mit Agilen Inspektionen sind viele kurze Feedback-schleifen möglich. (Das entspricht voll dem Grundgedanken der agilen Vorgehensmodelle.)

Literatur zu Inspektionen

1. Gilb, Tom: Agile Specification Quality Control. Cutter IT Journal, Vol. 18, No. 1, pp. 35-39, January 2005.
2. Gilb, Tom / Graham, Dorothy: Software Inspection, Addison-Wesley, 1993,
3. Radice, Ronald A.: High Quality Low Cost Software Inspections, Paradoxicon Publishing, 2002
4. www.gilb.com (Download Center, Stand 23.09.2005), “Optimizing Inspection” von Tom Gilb, S. 2
5. Gilb, Tom: Competitive Engineering, Butterworth-Heinemann, 2005