Software Engineering II

6 Inspektion und Code-*Review*

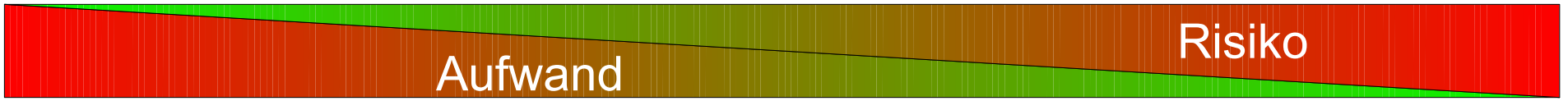
SS 2020

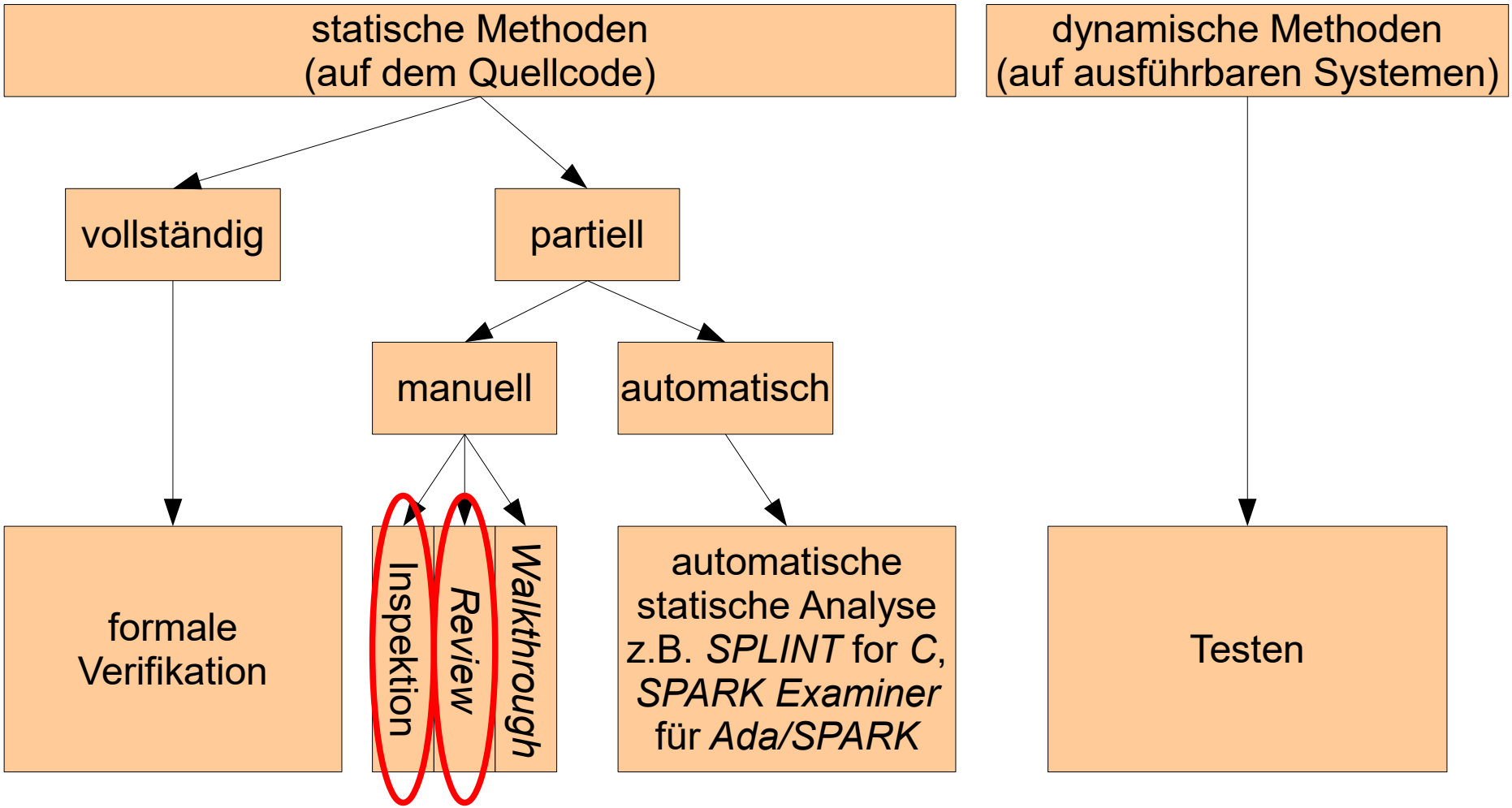
Prof. Dr. Dirk Müller

Übersicht

- Software-Inspektion
- Zielstellung
- Motivation
- Abgrenzung
- *Fagan*-Inspektion
  - Checkliste
  - Prozess
  - Datenerhebung
- Code-*Review* mit *Gerrit*
  - Anpassung mit *Prolog*-Regeln
- Zusammenfassung

Software-Qualitätssicherung



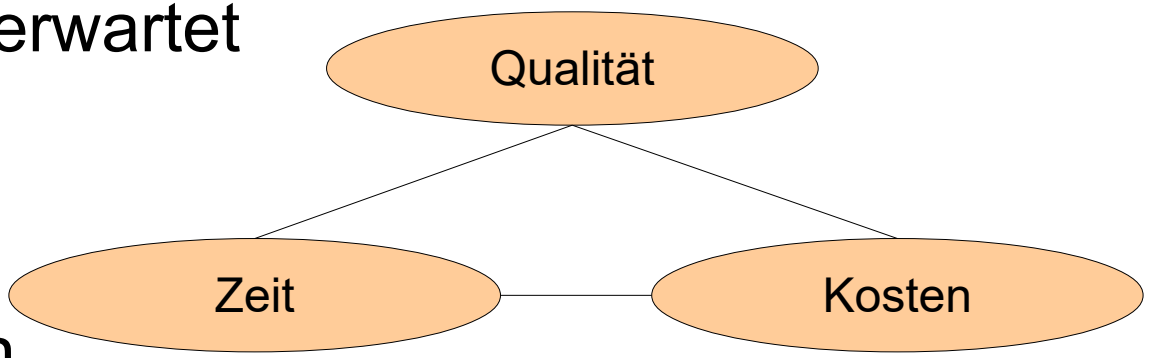


Definition

- Software-Inspektion
  „Manuelle, formalisierte Prüfmethode, um schwere Defekte in schriftlichen Dokumenten anhand von Referenzunterlagen zu identifizieren und durch den Autor beheben zu lassen.“
  Quelle: [Bal08], S. 497
- manuelle statische Prüfmethode
- Identifizierung durch Gruppe
- Behebung durch Autor
- automatische statische Analyse kann Semantik-Prüfungen nicht voll abdecken
  - Software-Inspektion wird weiter relevant sein

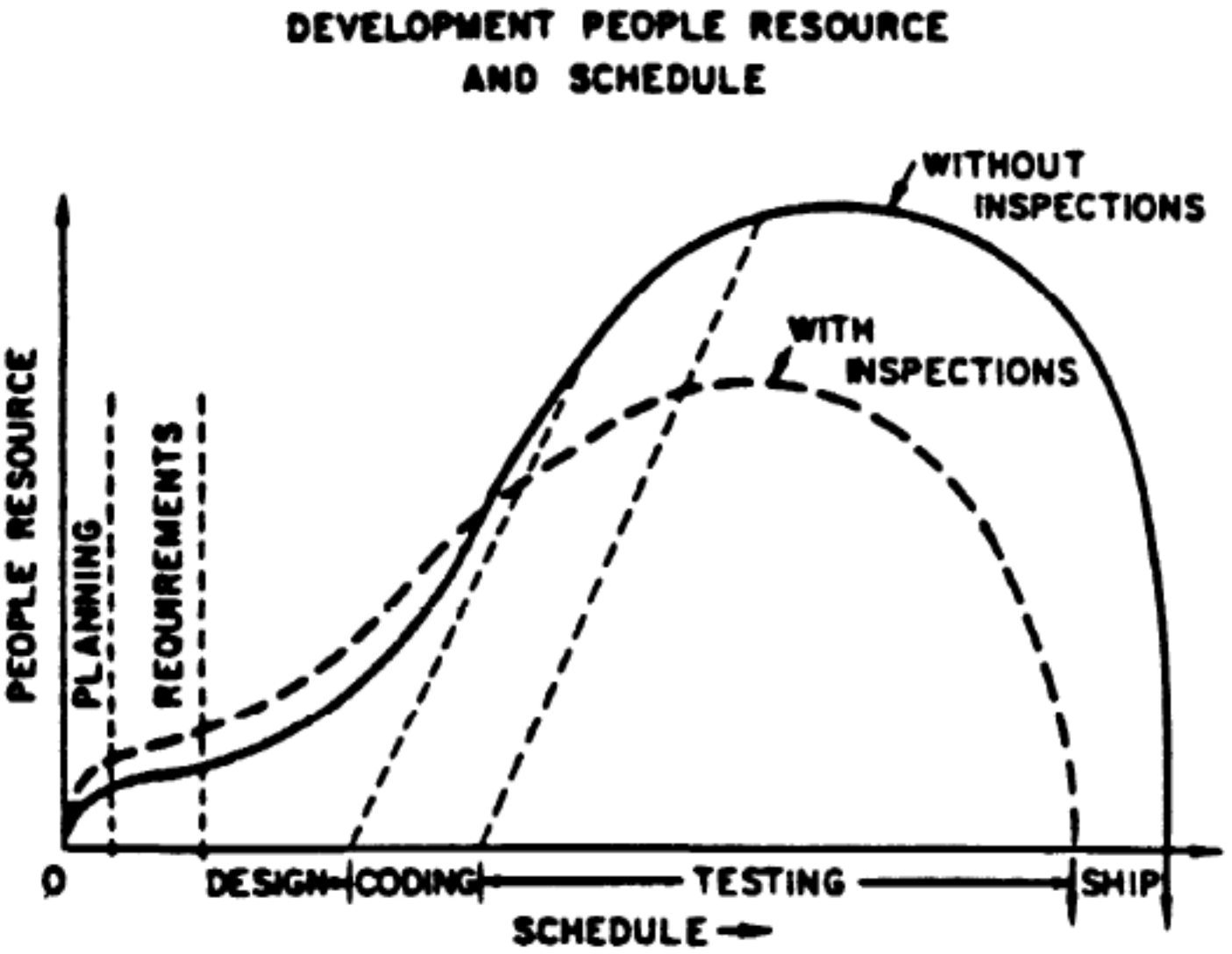
Ziele von Software-Inspektionen

- Fehler so früh wie möglich finden
  - einfacher zu beheben
  - billiger
- Verbesserungen in allen drei Dimensionen des magischen Dreiecks erwartet



- Korrektur von Fehlern
  - *Major Defect*: Fehler, der zu (beobachtbarem) kritischen Fehlverhalten führt
  - *Minor Defect*: Fehler, der nicht zu einem (beobachtbaren) kritischen Fehlverhalten führt

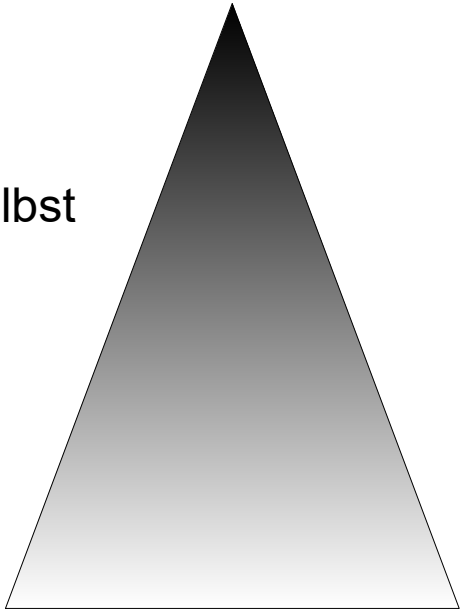
Motivation: Einfluss von SW-Inspektionen auf den Projektverlauf



Quelle: [1]

Abgrenzung

- automatische statische Analyse mit gleichem Ziel, nutzt aber Werkzeuge statt Personen
  - *SPLINT* für *C*
  - *SPARK Examiner* für *Ada/SPARK*
- viel früher einsetzbar als dynamische Methoden (Testen)
  - Änderbarkeit, Zuverlässigkeit, Sicherheit schwer zu testen
- Formalisierungsgrad
  - *Walkthrough*
    - Präsentation der Arbeit durch den Autor selbst
    - keine Vor- und Nachbereitung
  - *Review*
    - keine Checklisten, keine Datenerhebung
  - SW-Inspektion
    - klare Rollenverteilung
    - sehr strukturiert



Inspektion vs. Testen

- Vorteile der Software-Inspektion gegenüber dem Testen
  - Wechselwirkungen nicht relevant für statische Inspektion; Fehler können andere Fehler maskieren, Seiteneffekte, aufwendige Regressionstests als Abhilfe
  - unvollständige Versionen ohne Zusatzkosten überprüfbar; Testharnische müssen extra entwickelt werden
  - Qualitätssicherung in einem weiteren Sinne abgedeckt: z. B. Einhaltung von Standards, Portabilität, Änderbarkeit
- Defektarten, die nur durch Testen entdeckt werden können
  - unerwartete Wechselwirkungen zwischen Programmteilen
  - Fehler im Zeitverhalten
- Inspektion nicht als Ersatz fürs Testen, am besten in Kombination
  - Inspektion von Testfällen

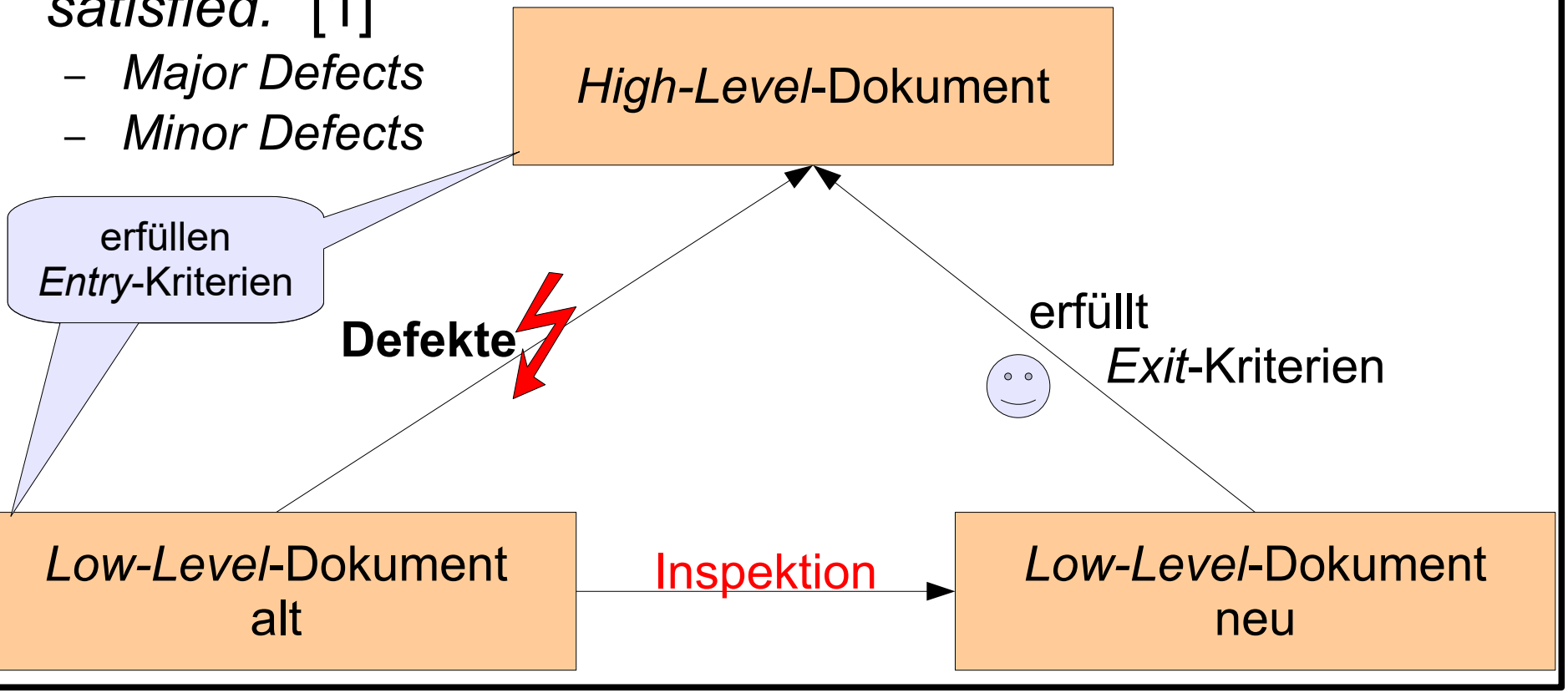
Quelle: [Som10], S. 208 f.

Fagan-Inspektion

- bewährte, sehr strukturierte Methode
- 1976 von *Michael Fagan* bei IBM eingeführt [1][2]
- Inspektions-Treffen als Höhepunkt (max. 2 h), aber Vor- und Nachbereitung
- Checkliste spezifisch für Programmiersprache, Domäne und Unternehmen
- Rollen:
  - Autor: des Codes/des Dokuments, muss Fehler beheben
  - Inspektor(-en): Suche nach Fehlern, Lücken und Inkonsistenzen
  - Leser: präsentiert Code/Dokument auf dem Treffen
  - Moderator: Prozess-Manager
  - (Schriftführer)
- für Code, Testpläne, Spezifikationen, Dokumentationen

Inspektion von etwas gegen etwas

- *Low-Level*-Dokument (z. B. Code) wird gegen *High-Level*-Dokument (z. B. Spezifikation) gecheckt
- „*A defect is an instance in which a requirement is not satisfied.*“ [1]
  - *Major Defects*
  - *Minor Defects*



Beispiel einer Checkliste (1/2)

- Datenfehler
  - nicht-initialisierte Variablen
  - unbenannte Konstanten
  - *Off-by-one-Error*, z. B. obere Feldgrenze ist `n` oder `n-1`
  - Pufferüberläufe
- Kontrollfehler
  - boolesche Bedingungen
  - korrekte Klammerung von Unterausdrücken
  - Abdeckung aller Fälle in Switch-Anweisung, Break in Zweigen?
- E/A-Fehler
  - Nutzung aller Eingabe-Variablen?
  - Zuweisung eines Werts an alle Ausgabe-Variablen?
  - Handhabung unerwarteter Eingaben (z. B. *SQL-Injection*)

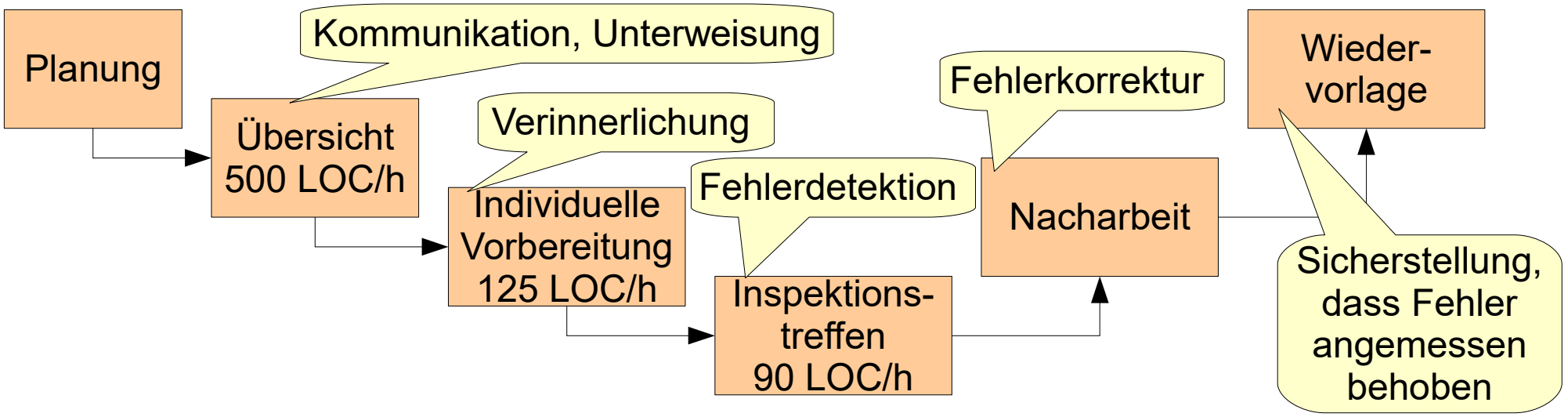
Quelle: [Som10]

Beispiel einer Checkliste (2/2)

- Schnittstellenfehler
  - Anzahl der Parameter jeweils korrekt?
  - Übereinstimmung formaler und aktueller Parametertyp?
  - Korrekte Reihenfolge der Parameter?
  - bei Zugriff auf *Shared Memory*: Alle mit gleichem Modell?
- Speichermanagementfehler
  - Korrektes Neusetzen aller Zeiger bei Bearbeitung einer verlinkten Struktur?
  - Dynamischer Speicher alloziert?
  - Explizite De-Allozierung wenn nicht mehr benötigt?
- Ausnahmebehandlungsfehler
  - Abdeckung aller Ausnahmeszenarien?

Quelle: [Som10]

Inspektionsprozess



- Planung: Team, Zeit, Ort, *Entry*-Kriterien
- Übersicht: Präsentation des *Low-Level*-Dokuments, Verteilung der Checkliste, Rollenverteilung
- Treffen: nur Fehlerdetektion, keine Korrektur, max. 2 h
- Nacharbeit: Korrektur durch Autor
- Wiedervorlage: Moderator/Team prüft, ob *Exit*-Kriterien erfüllt

Datenerhebung

- Erhebung genauer Statistiken sinnvoll
  - gefundene Fehler
  - Klassifizierung von Fehlern
  - Aufwand (Zeit und Personal: Personenstunden)
- Ergebnisse
  - Aufwand-Nutzen-Verhältnis: In Zukunft auch Inspektionen?
  - kontinuierliche Prozessverbesserung
  - dürfen nicht für persönliche Beurteilung von Programmierern genutzt werden, stattdessen Ergebnisse des Testens dafür zu nutzen [2]

Empirische Ergebnisse

- 15% bis 20% Zusatzaufwand
- Nettonutzen (unter Berücksichtigung des Prüfaufwands) ca. 20% in Entwicklung und 30% in Wartung
- 60% bis 70% Abdeckung

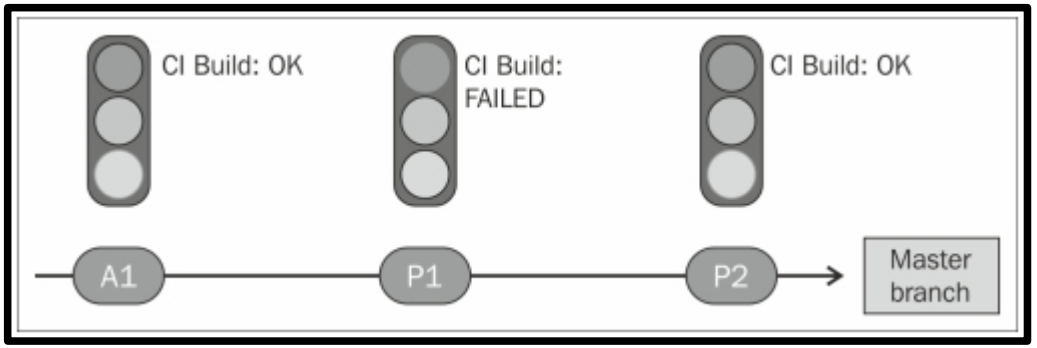
Quelle: [Bal08], S. 513

- Anteil der Fehlerdetektion durch Vorbereitung und Treffen
  - stark abhängig von Projekt und Checkliste
  - Kombination aus individueller und Gruppenphase macht die Inspektion als Methode so stark

Code-*Review*

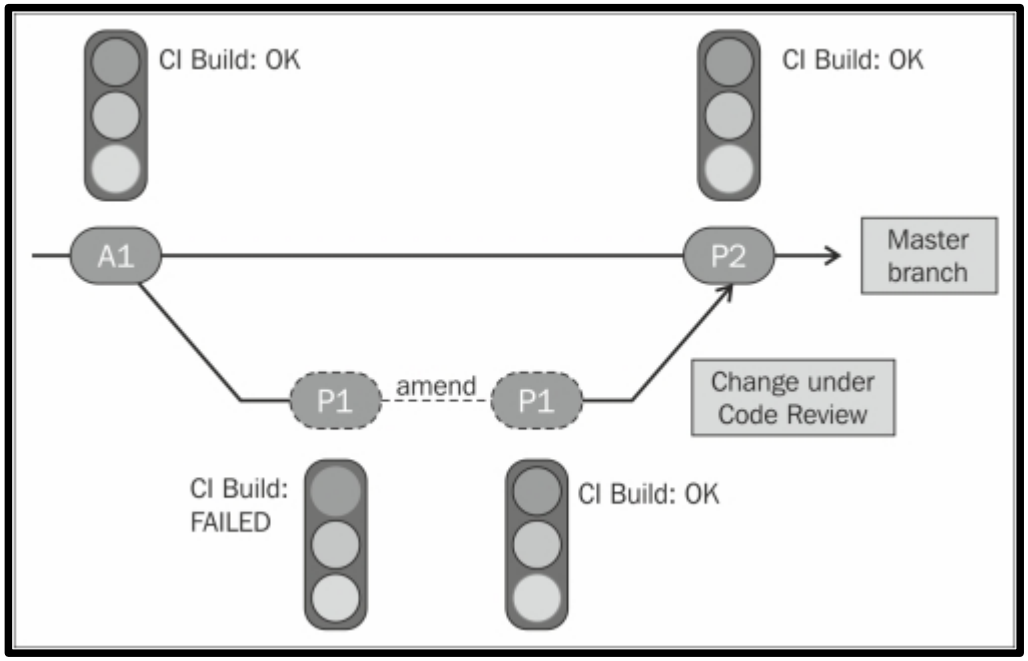
- „**Re|view**, das *oder* der; -s, -s, *auch* die; -, -s (kritische Besprechung eines Buchs, Films u. a.)“ Quelle: Duden, 25. Aufl. 2009
- Code-*Review* als ein agiler Ansatz bei kontinuierlicher Integration unter Nutzung von *Gerrit*

hier: Quellcode



ohne Code-*Review*

- weniger Fehler im *Build*-Prozess
- stabilerer Hauptzweig



mit Code-*Review*

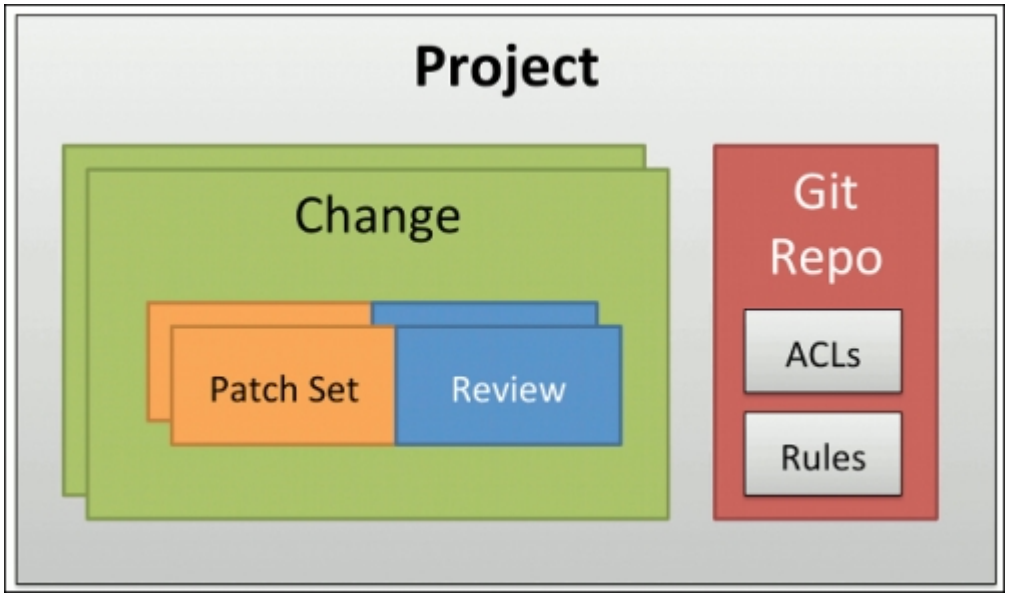
Quelle: [3]

Gerrit

- kollaboratives *Review*-System für *Git*
  - komfortable Diskussion von Änderungen an einer Software
  - Integration solcher Änderungen (über Versionsverwaltung *Git*)
- aktuelle Version 3.2 vom Mai 2020
- von *Google* entwickelt
- läuft seit Version 2 auf *Java EE*
- für die Entwicklung von *Android* genutzt
- Verwendung im Zusammenspiel mit einem *Git*-Repository
- kann als integriertes Werkzeug für Konfigurationsmanagement angesehen werden
  - Versionsmanagement: *Git*
  - Änderungsmanagement: *Gerrit*
  - *Build*-Management: explizit vorgesehen, frei wählbar

besser: integrierendes

Gerrit: Architektur



Quelle: [3]

- *Git*-Repository: Codebasis und Änderungsvorschläge in Begutachtung
- Referenzen auf Änderungen (*Gerrit commit-id*)
- *Access Control Lists*: Rollen mit Rechten für Zweige
- *Prolog*-Regeln: Standardregeln, aber anpass- und erweiterbar

Code-*Review*: Motivation

- Motivation zur Mitarbeit an *Open-Source*-Projekten
  - öffentliches Review als eine generelle Motivation für quelloffene Software
- Risiko des Blockierens des Hauptentwicklungszweigs deutlich reduziert
- gegenseitiges Lernen voneinander
  - keine Selbstrezensionen zulassen
  - Taktgefühl bei Formulierungen in Rezension nötig, keine persönlichen Angriffe
  - sonst Gefahr der Ablehnung des Ansatzes
- Teammitglieder können Privilegien erwerben/verlieren
  - basiert auf bisherigen Beiträgen

Quelle: [3]

Gerrit: Bewertungsskala für Code

- Code-*Review*, manuelle Begutachtung
  - -2 Veto, Code kann so in keinem Fall eingecheckt werden
  - -1 Code sieht gut aus, benötigt aber noch Änderungen vor dem Einchecken
  - 0 ohne Bewertung, nur Kommentare
  - +1 Code sieht gut aus, aber ich brauche noch mehr *Reviewer*
  - +2 Freigabe, Code sieht gut aus und kann so eingecheckt werden
- *Validate*, funktional verifiziert mittels Tests und *Builds*
  - -1 Code läuft/funktioniert nicht
  - 0 ohne Bewertung, nur Kommentare
  - +1 Code läuft/funktioniert

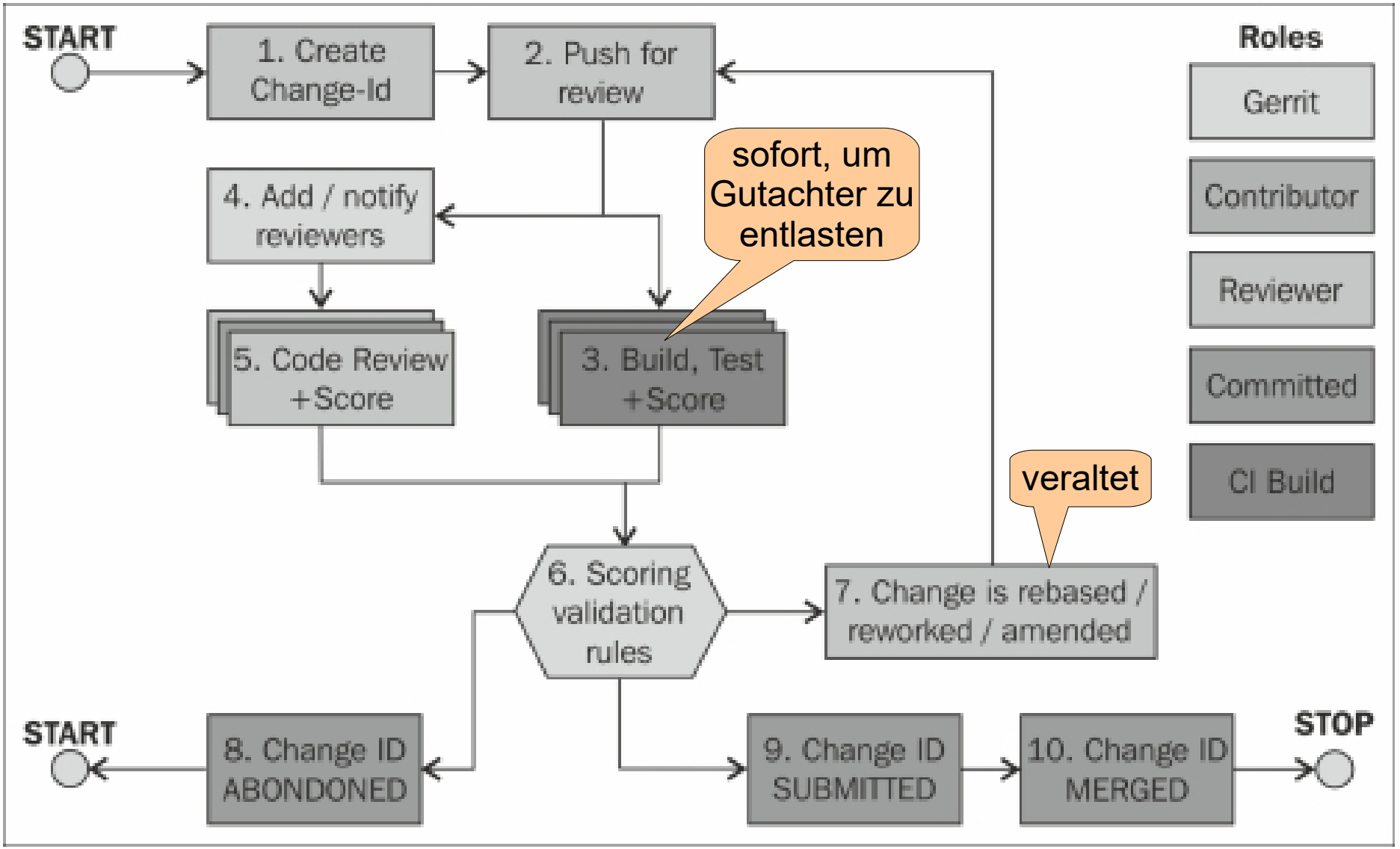
Quelle: [3]

Code-*Review*-Rollen unter *Gerrit*

- *Contributor*: prinzipiell jeder, Lesezugriff auf min. 1 Zweig und dann Hochladen eines Änderungsvorschlags
- *Reviewer*: Begutachtung von Code und Vergabe einer (numerischen) Bewertung im Bereich -1 bis +1
- *Committer*: Entscheider bei der Begutachtung von Code, Bewertung -2 oder +2, also Veto oder Freigabe
- *Maintainer*: Chef bzw. Manager eines Projekts, der Rollen verteilt und Zugriffsrechte vergibt

Quelle: [3]

Gerrit: Workflow



Quelle: [3]

Prolog-Regeln

- `rules.pl` definiert Regeln
- Logik-Regeln lassen sich ideal in *Prolog* ausdrücken
- schnell genug, durch Compilierung in *Java*-Bytecode
- Standard-Einstellung für Regeln

```
submit_rule(submit(V, CR)) :-
    gerrit:max_with_block(-2, 2, 'Code-Review', CR),
    gerrit:max_with_block(-1, 1, 'Verified', V).
```

- Wiederverwendung des Standards mit Anpassung

```
submit_rule(S) :- gerrit:default_submit(R),
                  project_specific_policy(R, S).

project_specific_policy(R, S) :- ...
```

Quelle: [4]

gerrit:max_with_block

```
44  public class MaxWithBlock extends CategoryFunction {
45    public static String NAME = "MaxWithBlock";
46
47    @Override
48    public void run(final ApprovalType at, final FunctionState state) {
49      boolean rejected = false;
50      boolean passed = false;
51      for (final PatchSetApproval a : state.getApprovals(at)) {
52        state.normalize(at, a);
53
54        rejected |= at.isMaxNegative(a);
55        passed |= at.isMaxPositive(a);
56      }
57
58      // The type must not have had its max negative (a forceful reject)
59      // and must have at least one max positive (a full accept).
60      //
61      state.valid(at, !rejected && passed);
62    }
63  }
```

Quelle: https://github.com/sudosurootdev/gerrit/blob/master/gerrit-server/src/main/java/com/google/gerrit/server/workflow/MaxWithBlock.java
Download am 17.4.2016

Prolog-Regeln: Anpassung auf minimale Summe

```
submit_rule(submit(CR, V)) :-
    sum(2, 'Code-Review', CR),
    gerrit:max_with_block(-1, 1, 'Verified', V).

% Sum the votes in a category. Uses a helper function score/2
% to select out only the score values the given category.
sum(VotesNeeded, Category, label(Category, ok(_))) :-
    findall(Score, score(Category, Score), All),
    sum_list(All, Sum),
    Sum >= VotesNeeded,
    !.
sum(VotesNeeded, Category, label(Category, need(VotesNeeded))).

score(Category, Score) :-
    gerrit:commit_label(label(Category, Score), User).

% Simple Prolog routine to sum a list of integers.
sum_list(List, Sum)   :- sum_list(List, 0, Sum).
sum_list([X|T], Y, S) :- Z is X + Y, sum_list(T, Z, S).
sum_list([], S, S).
```

Quelle: [4]

Prolog-Regeln: Selbstrezension verbieten

```
submit_rule(submit(CR, V)) :-
    base(CR, V),
    CR = label(_, ok(Reviewer)),
    gerrit:commit_author(Author),
    Author \= Reviewer,
    !.

submit_rule(submit(CR, V, N)) :-
    base(CR, V),
    N = label('Non-Author-Code-Review', need(_)).

base(CR, V) :-
    gerrit:max_with_block(-2, 2, 'Code-Review', CR),
    gerrit:max_with_block(-1, 1, 'Verified', V).
```

Quelle: [4]

Probleme beim *Linux*-Kernel?

- sehr komplexes Software, > 25 Mio. SLOC
- zu wenig Nachwuchs-*Maintainer*
- kompetente Leute mit dem Schwerpunkt des *Code Reviews* statt des Implementierens

> "*Naively extrapolating the relative trend predicts that around the year 2025 large numbers of kernel maintainers will do nothing else than be the bottle-neck, preventing everyone else from getting their work merged and not contributing anything of their own. The kernel community imploding under its own bureaucratic weight being the likely outcome of that.*" - Daniel Vetter, zitiert in [5]

- „Je weniger die *Maintainer* selbst entwickeln, desto mehr guter Code geht dem Kernel durch die Lappen." [5]
- keine fairen Standards bei der Qualitätssicherung
  - *Patches* der *Maintainer* weniger genau geprüft als die anderer
  - Netzwerk-Subsystem: nur 9 % der *Commits* mit *Code-Review*
  - Grafiktreiber-Subsystem: 83 %, deutlich besser
- Ausweg: Nachwuchsarbeit

Quelle: [5]

Zusammenfassung

- Inspektion als strukturierte, statische Methode zur Qualitätssicherung von Software durch eine Gruppe
- Einsatz schon sehr früh möglich (vs. dynamische Methoden wie Testen), früh erkannte Fehler
  - einfacher zu korrigieren, billiger
- Zusatzaufwand ist meist gerechtfertigt
  - individuelle und Gruppenphase: Entdeckung sehr vieler Fehler
  - Transparenz und Zusammenhalt in Gruppe (Teamgeist) gestärkt
  - Datenerhebung zu Nutzen, Aufwand und Fehlerklassen sinnvoll, um für zukünftige Projekte zu lernen
- Programmierer nicht nach Inspektionsergebnissen beurteilen, sondern nach Testergebnissen (Kosten!)
- Code-*Review* mittels *Gerrit* ermöglicht kooperative *Reviews* mit Abstimmungen, kontinuierliche Integration
- aber: Gefahr von zu viel Bürokratie, Bsp. *Linux*-Kernel

Literatur

[1] Fagan, M.E.: “Advances in Software Inspections”, July 1986, IEEE Transactions on Software Engineering, Vol. SE-12, No. 7, pp. 744-751

[2] M.E. Fagan (1976): “Design and Code inspections to reduce errors in program development”, IBM Systems Journal 15 (3): pp. 182–211

[3] Luca Milanesio: „Learning Gerrit Code Review“, Packt Publishing, Sep 2013

[4] “Gerrit Code Review - Prolog Submit Rules Cookbook”, Version 2.12.2-1887, Download am 15.4.2016, https://gerrit-review.googlesource.com/Documentation/prolog-cookbook.html

[5] Fabian A. Scherschel: „Linux-Entwickler: Kernel-Community wird unter eigener Bürokratie zusammenbrechen“, 24.04.2018, Heise-Online, Download am 24.04.2018, https://heise.de/-4030460