KV Software Engineering
Prof. Dr. Martin Glinz

Kapitel 13

Fehlervermeidung

Universität Zürich
Institut für Informatik

13.1 Systematisches Arbeiten

13.2 Prozesse mit kontinuierlicher Prüfung

13.3 Dokumentier- und Codierrichtlinien

Fehlervermeidung durch systematisches Arbeiten

Systematisches Arbeiten als solches wirkt fehlervermeidend

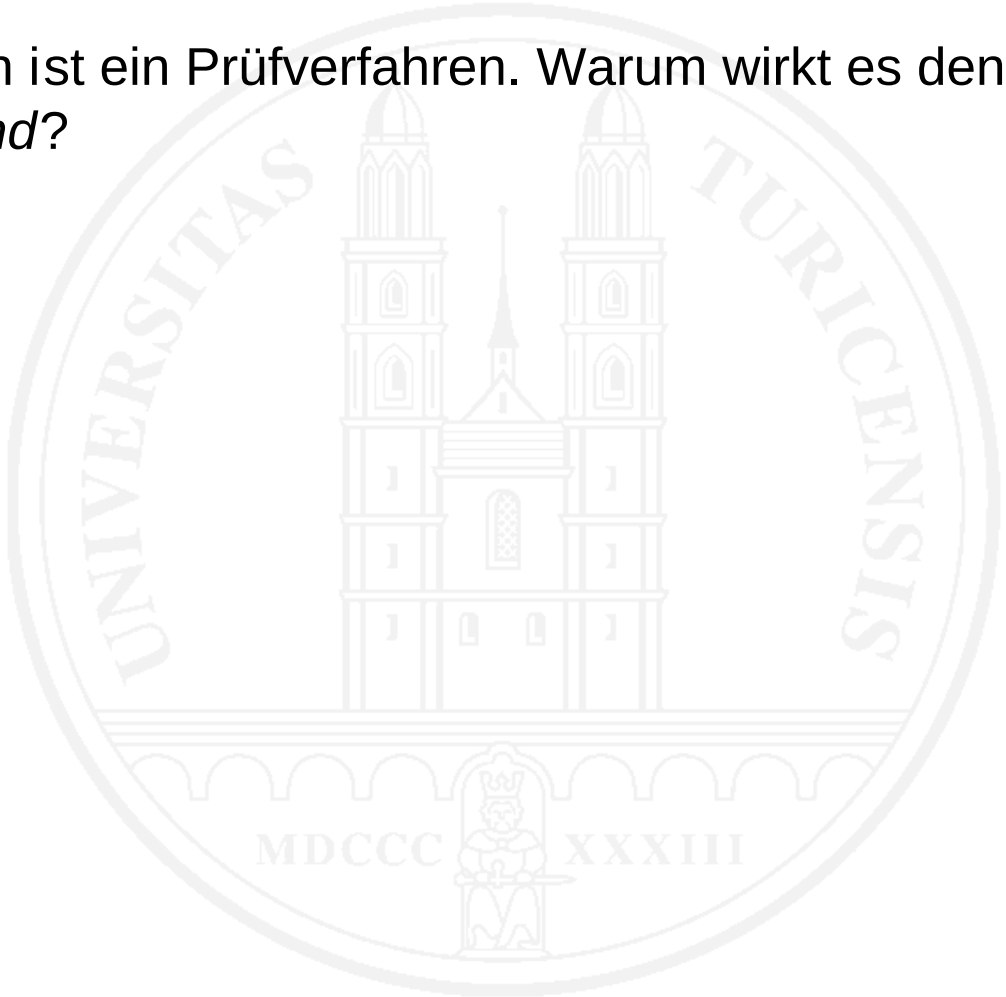
- Sorgfältig erstellte und validierte Anforderungsspezifikation
- Tragfähige Software-Architektur
- Saubere Abgrenzung und Kapselung von Entwurfsentscheidungen und Verantwortlichkeiten beispielsweise durch
  - Information Hiding
  - Vertragsorientierten Entwurf (vgl. Kapitel 3)
- Systematisches Programmieren (vgl. Kapitel 2)

Eliminieren von Fehlern an der Quelle

- Verwendung syntaxsensitiver Editoren
- Fortlaufende Selbstinspektion aller Arbeitsergebnisse
- Autor-Kritiker-Zyklen
- Paarweises Programmieren / Entwickeln
- Testfälle vor dem Code schreiben

Mini-Übung 13.1

Selbstinspektion ist ein Prüfverfahren. Warum wirkt es dennoch *fehlervermeidend*?



13.1 Systematisches Arbeiten

13.2 Prozesse mit kontinuierlicher Prüfung

13.3 Dokumentier- und Codierrichtlinien

Grundlagen

- Fortlaufende und rigoros durchgesetzte Prüfungen
  - decken Fehler frühzeitig auf
  - vermeiden die Weiterarbeit mit falschen Vorgaben
- Sorgfältige Reviews (Inspektionen) aller Dokumente einschließlich des Codes sind das Mittel der Wahl für solche Prüfungen
- Es gibt Entwicklungsprozesse, die spezielle auf Fehlervermeidung ausgelegt sind, zum Beispiel
  - der Cleanroom-Prozess
  - der Persönliche Software Prozess (PSP)

Der Cleanroom-Prozess

- Bei IBM Mitte der 80er Jahre speziell zur Fehlervermeidung geschaffen (Cobb und Mills 1990, Linger 1994)
- Grundideen
  - Inkrementelle Entwicklung (Wachstumsmodell)
  - Strenge (aber faktisch nicht formale) Spezifikation
  - Verifikation aller Programme durch Inspektion und strenges Argumentieren
  - Programmierer dürfen ihren Code weder übersetzen noch testen; es gibt weder Modul- noch Integrationstests
  - Eine unabhängige Testgruppe unterwirft die fertige Software einem statistischen Systemtest
  - Das Testziel ist eine Zuverlässigkeitsprognose, nicht das Finden von Fehlern

Kritik an Cleanroom

- Verschiedene Experimente sollen die Überlegenheit von Cleanroom belegen (Basili und Green 1994, Selby, Basili und Baker 1987)
- Der Cleanroom-Prozess wird heute in Frage gestellt (z.B. Beizer 1997)
  - Neuere softwaretechnische Erkenntnisse, insbesondere Information Hiding und vertragsorientierter Entwurf werden ignoriert
  - Alle Fortschritte in der Testtechnologie werden ignoriert
  - Statistisches Testen erfordert Testfallauswahl nach dem erwarteten Benutzungsprofil der Software
    - Die Bestimmung des Benutzungsprofils kann schwierig oder unmöglich sein
    - Gemäß Profil seltene Fälle werden nur wenig getestet, unabhängig davon, ob diese Fälle kritisch sind oder nicht
  - Unklar, ob die Cleanroom Experimente die Überlegenheit von Cleanroom oder nur die von systematischem Entwickeln zeigen

Der Persönliche Software Prozess (PSP)

- Von Humphrey (1995) als systematischer Prozess zur Fehlerver-meidung auf der Stufe individueller Software-Entwickler eingeführt
- Projektplanung und -lenkung auf der Stufe der Arbeit jedes Einzelnen
- Konzentriert sich auf Detailentwurf und Codierung
- Kernideen:
  - Der Aufwand für jede Arbeit wird vorgängig geschätzt, alle tatsächlichen Aufwendungen werden gemessen
  - Über alle Fehler und deren Behebung wird penibel Buch geführt
  - Durch Analyse der Messwerte können eigene Schwachstellen erkannt und eliminiert werden
  - Rigorose Selbstinspektion jedes Arbeitsergebnisses
- Soll einen fehlervermeidenden Prozess auf Projektstufe unterstützen

Stärken und Schwächen des PSP

+ Unterstützt die Gewinnung quantitativer Erfahrungswerte über den Aufwand für die Bearbeitung von Aufgaben
+ Bessere Prognosen möglich bei der Übernahme von Aufgaben
+ Hilft, Schwachstellen und Probleme im persönlichen Arbeitsstil zu erkennen und zu beheben
+ Fördert die Arbeitshaltung, Dinge lieber etwas langsamer, aber auf Anhieb richtig zu machen als schnell und dreimal falsch

– Papierkrieg
– Einseitig auf Defektminimierung optimiert
– Differenziert nicht nach individuellen Problemen und Systemproblemen
– Vernachlässigt Anforderungsspezifikation und Systemarchitektur
– Vernachlässigt Gruppenarbeit

13.1 Systematisches Arbeiten

13.2 Prozesse mit kontinuierlicher Prüfung

13.3 Dokumentier- und Codierrichtlinien

Allgemeines

- Dokumentier- und Codierrichtlinien enthalten Vorgaben zu Inhalt, Struktur und Stil der zu erstellenden Artefakte
- Vor- und Nachteile solcher Reglementierungen sind gegeneinander abzuwägen
- Richtlinien sind nur nützlich, wenn
  - sie nur relevante Dinge regeln (keine Vorschriften über die Form von i-Punkten)
  - ihre Einhaltung überprüft wird
  - Verstöße gegen die Richtlinien Sanktionen nach sich ziehen

Dokumentier-Richtlinien

- Legen fest
  - Welche Dokumente zu erstellen sind
  - Für jedes Dokument: Gliederung und erwartete Inhalte
- Sorgen für ein einheitliches, lesbares Erscheinungsbild der Dokumente
- Unterstützen die Erstellung der notwendigen Dokumentation

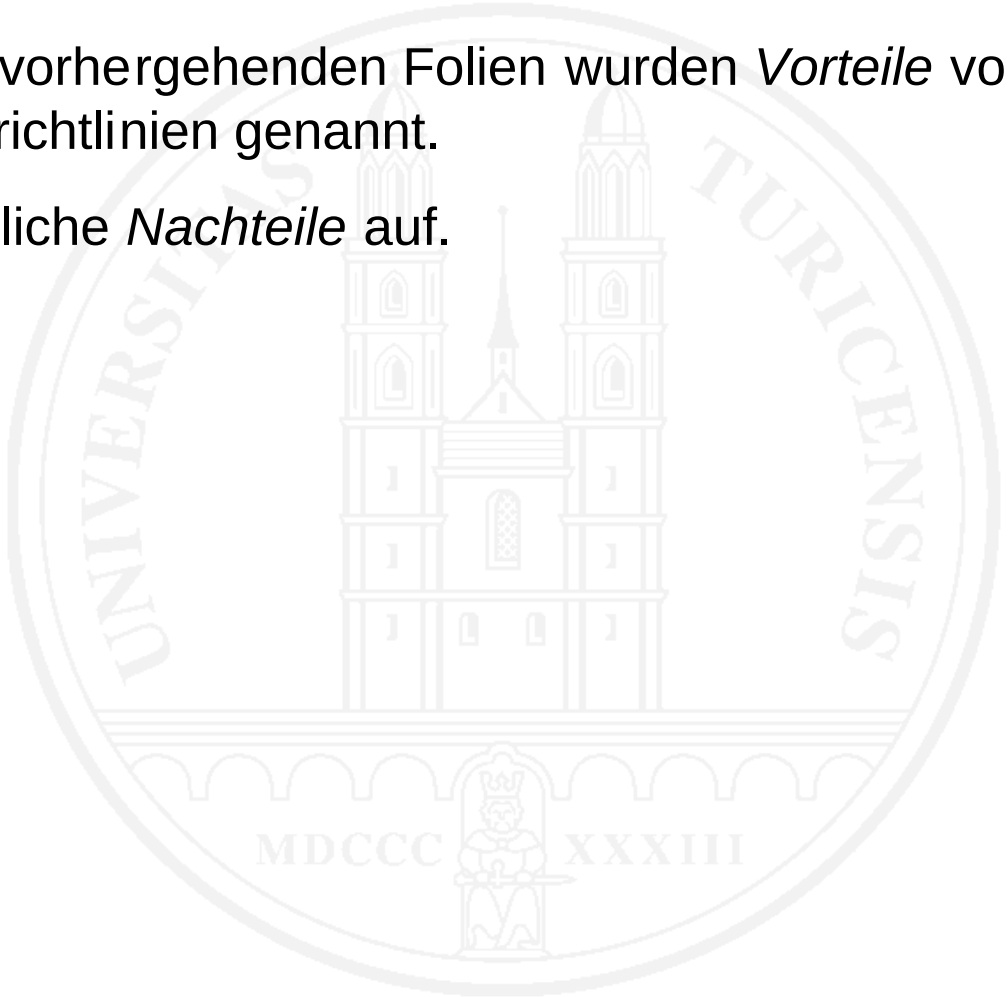
Codier-Richtlinien

- Sorgen für einen gut lesbaren, nach einheitlichen Kriterien strukturierten Code
- Vereinheitlichen den Programmierstil
- Verhindern die Verwendung schlechter oder gefährlicher Konstrukte
- Sind eine Voraussetzung für vergleichbare Messungen der Produktgröße in Anzahl Codezeilen

Mini-Übung 13.2

Auf den beiden vorhergehenden Folien wurden *Vorteile* von Dokumentier- und Codierrichtlinien genannt.

Zählen Sie mögliche *Nachteile* auf.



Literatur

Basili, V. and S. Green (1994). Software Process Evolution at the SEL. *IEEE Software* **11**, 4 (Jul 1994). 58-66.

Beizer, B. (1997). Cleanroom Process Model: A Critical Examination. *IEEE Software* **14**, 2 (Mar/Apr 1997). 14-16.

Berner, S., S. Joos, M. Glinz (1997). Entwicklungsrichtlinien für die Programmiersprache Java. *Informatik/Informatique* **4**, 3 (Jun 1997). 8-11.

Berner, S., M. Glinz, S. Joos, J. Ryser, S. Schett (2001). *Entwicklungsrichtlinien für Java-Software*. Revidierte Fassung. Institut für Informatik, Universität Zürich.
http://www.ifi.unizh.ch/req/publications/java.html

Cobb, R.H., H.D. Mills (1990). Engineering Software under Statistical Quality Control. *IEEE Software* **7**, 6 (Nov 1990). 44-54.

Dyer, M. (1992). *The Cleanroom Approach to Software Quality*. New York: John Wiley & Sons.

Humphrey, W.S. (1995). *A Discipline for Software Engineering*. Reading, Mass., etc.: Addison-Wesley.

Linger, R.C. (1994). Cleanroom Process Model. *IEEE Software* **11**, 2 (Mar 1994). 50-58.

Selby, R.W., V.R. Basili and T. Baker (1987). Cleanroom Software Development: An Empirical Evaluation. *IEEE Transactions on Software Engineering* **SE-13**, 9. 1027-1037.