Testen und Debuggen

by

Dr. Günter Kolousek

Motivation zu Testen

- Schätzungen
  - im Schnitt: 25 Fehler pro 1000 Zeilen Code
  - gute Programme: 2 Fehler pro 1000 Zeilen Code
  - SW für kritische Systeme
    - sollte eine Größenordnung darunter liegen
- Testen… eine ungeliebte Tätigkeit
  - oft: Bananensoftware (reift bei Kunden)
    - unzufriedene Kunden!
    - hohe Supportkosten!
- je später ein Fehler erkannt wird $\leadsto$ höhere Kosten!!

Ariane 5

- 4.6.1996: stürzte nach ca. 36.6s ab!
  - SW Modul von Ariane 4 auf Ariane 5 übernommen
    - nur mit **eingeschränkter** Parameterauswahl getestet
    - **Überlauf** einer Variable!

Ariane 5 – Analysen

- Typumwandlung war nicht abgesichert
- **Überlauf** bei 64 Bit-Gleitkommazahl $\rightsquigarrow$ 16 Bit-Ganzzahl
- **Vermutung**: übergebene Geschwindigkeit wird nicht so groß
- Betriebsbedingungen **nicht** entsprechend dokumentiert!
- Ariane 5 hat **anderes** Flugprofil!
- Modul führte zu Ausfall des Navigationssystem
- daraufhin: Ausfall des (identischen) Reservesystems!
- Steuerungssystem $\rightsquigarrow$ Kurskorrektur (30°/s) $\rightsquigarrow$ droht auseinanderzubrechen $\rightsquigarrow$ Selbstzerstörung
- Modul liefert sinnvolle Daten nur **vor** dem Start!
- aktiv bis **40s** nach dem Start (Anforderung von Ariane 4)
- bei Ariane 5 **sinnlos**, da andere Startsequenz
- Technische Leitung lehnte Simulation der Funktion ab
- aus **finanziellen** Gründen!

4 Gründe zu testen

- sicherstellen, dass Code so funktioniert wie sich der Entwickler vorstellt.
- sicherstellen, dass Code nach Änderungen weiterhin funktioniert.
- sicherstellen, dass der Entwickler die Anforderungen versteht.
- sicherstellen, dass Code ein wartbares Interface aufweist.

Validation und Verifikation

- Softwareentwicklung bzgl.
  1. Anforderungen
  2. Spezifikation
  3. Implementierung
- Validation: Spezifikation entspricht Anforderungen
- Verifikation: Implementierung entspricht Spezifikation

Teststufen

- Komponententest, Modultest, Unittest
  - testet abgrenzbare Teile, wie Funktionen, Klassen, Module, Pakete oder ganze Programme.
- Integrationstest
  - testet Zusammenarbeit voneinander unabhängiger Komponenten
- Systemtest
  - testet System gegen funktionale und nicht-funktionale Anforderungen
  - wird in eigener Testumgebung mit Testdaten durchgeführt
- Abnahmetest
  - Test durch Auftraggeber

Vollständiges Testen

- unmöglich!
- Beispiele
  - Faktorielle

    ```
    unsigned fak(unsigned i);
    ```

    - 32 Bit-System $\rightarrow 2^{32}$ mögliche Eingaben
  - Größter gemeinsamer Teiler

    ```
    unsigned ggT(unsigned x, unsigned y);
    ```

    - 32 Bit-System $\rightarrow 2^{32} \times 2^{32} = 2^{64}$ mögliche Eingaben

Codeabdeckung

1. Funktionsabdeckung (engl. function coverage)
   - Jede Funktion aufgerufen?
2. Anweisungsabdeckung (engl. statement coverage)
   - Jede Anweisung aufgerufen?
3. Entscheidungsabdeckung (engl. branch coverage)
   - Jeder Zweig aufgerufen?
4. Pfadabdeckung (engl. path coverage)
   - Jeder Pfad durchlaufen? (z.B. 2 `if` hintereinander)
5. Bedingungsabdeckung (engl. condition coverage)
   - Einfachbedingungabdeckung: Jede boolsche Variable (Bedingung) einmal zu `true` und einmal zu `false` ausgewertet?
   - Mehrfachbedingungabdeckung: Jede Kombination, also 2^n bei n boolschen Operanden

Codeabdeckung – Beispiel

```
int foo(int x, int y) {
    int z{};
    if (x > 0 && y > 0) {
        z = x;
    }
    return z;
}
```

1. Funktionsabdeckung: `foo(0, 0)`
2. Anweisungsabdeckung: `foo(1, 1)`
3. Entscheidungsabdeckung: `foo(1, 1)`, `foo(1, 0)`
4. Pfadabdeckung…
5. Bedingungsabdeckung: `foo(1, 0)`, `foo(0, 1)`

Testarten

- funktionale Tests
- nicht-funktionale Tests: Sicherheit, Zuverlässigkeit, Bedienbarkeit,…
- Fehlertests: Verhalten im Fehlerfall
- Wiederinbetriebnahmetest
- Installationstest
- Lasttests: hohe Speicher, CPU, Netzwerk-Anforderungen
- Stresstests: Verhalten in Ausnahmesituationen (zu hohe Anforderungen)
- Performancetests: entspricht System gestellten Anforderungen bzgl. Performance

Testvarianten

- White-Box-Tests
  - Kenntnisse über innere Funktionsweise
  - Korrektheit testen: Testfälle werden aus Code abgeleitet
    - Problem: ”durch Betriebsblindheit um Fehler herum testen”
- Black-Box-Tests
  - **Keine** Kenntnisse über innere Funktionsweise
  - Funktion testen: Übereinstimmung mit der Spezifikation
    - Problem: Aufwändig
- Grey-Box-Tests
  - Wie White-Box: Werden von Entwicklern des Systems erstellt.
  - Wie Black-Box: Anfänglich Unkenntnis über innere Funktionsweise.
    - Wie das? $\rightsquigarrow$ zuerst Tests schreiben…

Testablauf

1. Entwerfen der Testfälle
2. Erstellen der Testdaten
3. Ausführen des Programmes
4. Vergleich der Ergebnisse mit den Testfällen

Testfalldokumentation

1. Beschreibung der Ausgangssituation
   - genaue Beschreibung des Systemzustandes
2. Eingaben bzw. Aktionen
   - genaue Beschreibung welche Funktionen ausgeführt werden soll
   - genaue Dokumentation der Eingaben
3. Soll-Resultat
   - Welche Ausgaben bzw. welches Verhalten wird vom System erwartet?

TDD

- ”if it’s not tested, it’s broken”
- Test Driven Development
  1. Schreibe einen Test, der beweisen soll, dass Code funktioniert.
  2. Der Test wird fehlschlagen: kein Code bis jetzt!
  3. Schreibe Code, sodass Test erfolgreich bestanden wird.
  4. Beginne wieder bei 1.

Testen und Debuggen

- Ausgaben auf `stderr` (nicht so gut: `stdout`)
  - Normale Ausgabe vermischt mit Debuginformation
  - Ein- bzw. Ausschalten der Debuginfo nicht möglich
  - Entfernen der Anweisungen nicht möglich
- Loggen
  - Python: Modul `logging`
  - Java: Paket `java.util.logging`
  - C++: Header-only Bibliothek `spdlog`
- Debugger

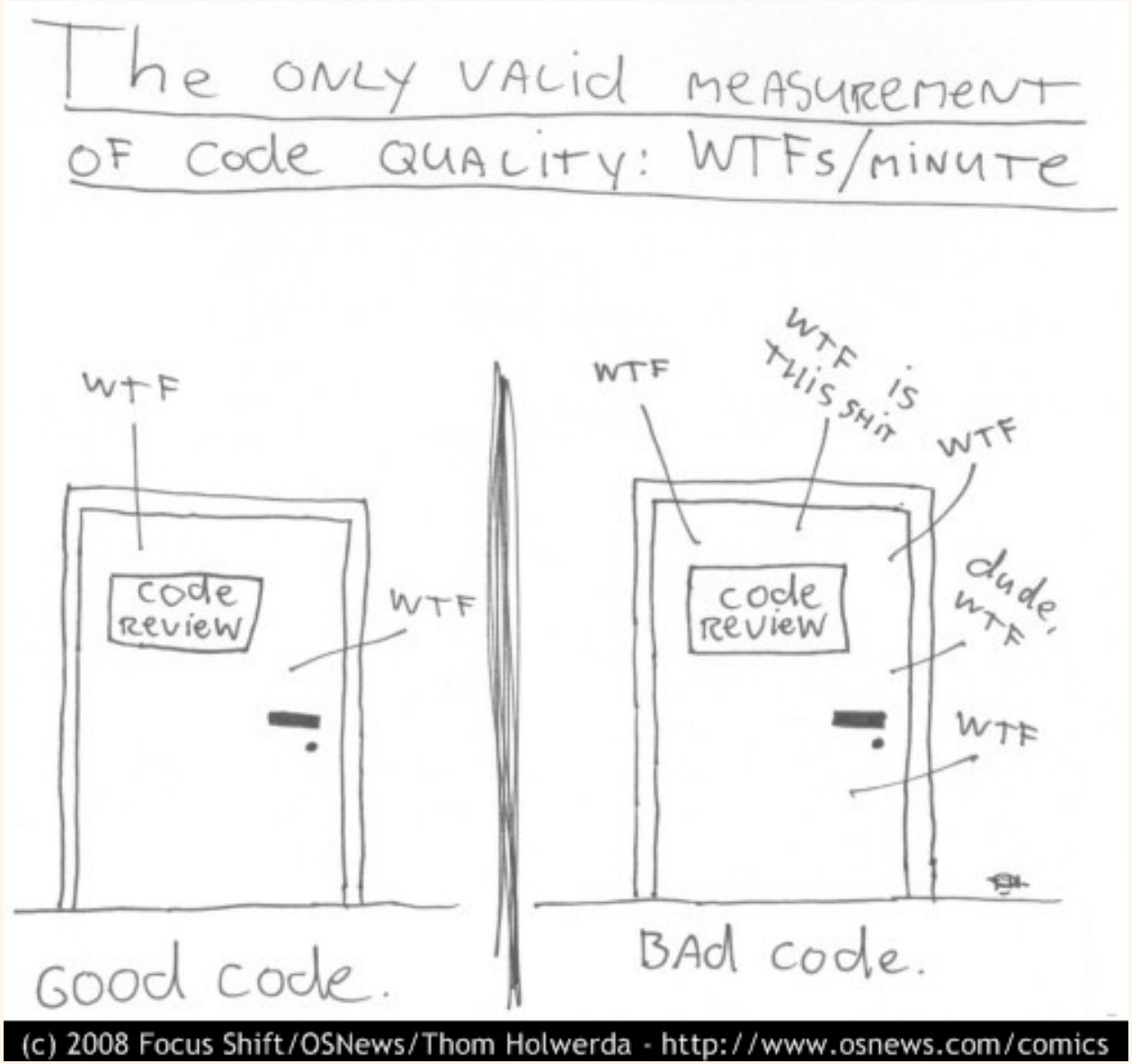
  Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. – Brian Kernighan

- Testtools, z.B. Unit-Tests

Analysen

- Überprüfung der Coding-Conventions (style checker)
- Metriken
- statische Analyse, z.B.
  - nicht initialisierte Variabe
  - nicht verwendete Variablen, Funktionen, Klassen
  - Finden von Speicherleaks
  - nicht erreichter Code, Endlosschleifen, überflüssige `if`…
  - …
- Code coverage
- Code-Reviews
- formale Verifikation

Qualität?


The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE
WTF
code review
WTF
Good code.
WTF
WTF is THIS SHIT
WTF
code review
dude, WTF
WTF
BAd code.
(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics