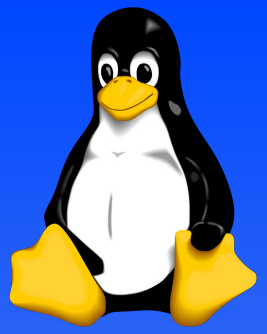
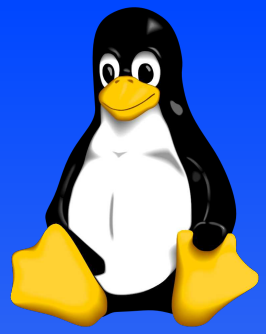
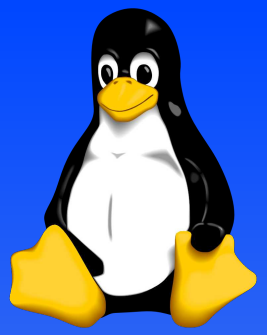


Hard- und Software Zuverlässigkeit von gnu/Linux-basierten eingebetteten Systemen

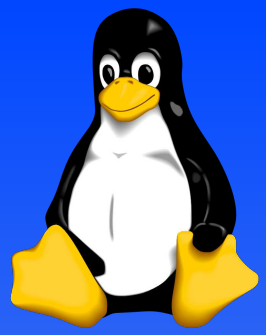


Wolfram Luithardt und Daniel Gachet

**Hochschule für Technik und Architektur,
Fribourg, Schweiz**



Zuverlässigkeit



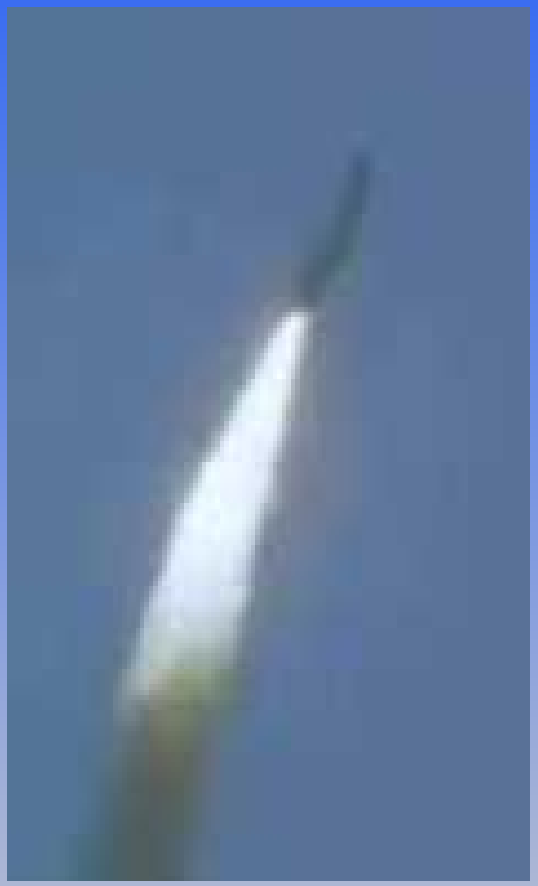
Definitionen (nach Wikipedia):

Die **Zuverlässigkeit** eines technischen Produkts oder Systems ist eine Eigenschaft (Verhaltensmerkmal), die angibt, wie verlässlich eine dem Produkt oder System zugewiesene Funktion in einem Zeitintervall erfüllt wird

Software-Zuverlässigkeit ist definiert als „Wahrscheinlichkeit der fehlerfreien Funktion eines Computer-Programms in einer spezifizierten Umgebung in einer spezifizierten Zeit“. Damit gehört Software-Zuverlässigkeit zu den objektiven, messbaren oder schätzbaren Kriterien der Softwarequalität und gehört somit zu den Software-Metriken.

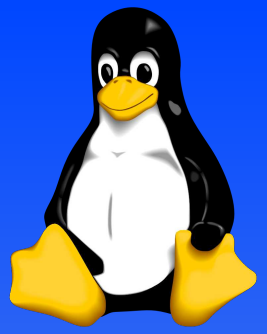
Das prominenteste Beispiel Ariane 501, 4.6.1996

H0+36 s

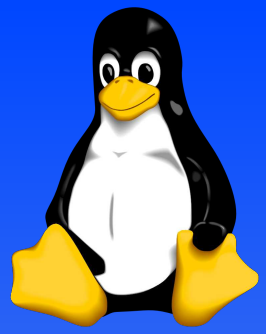


H0+38 s

H0+39 s



Ariane 501

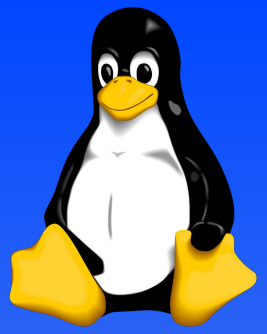


```
declare
vertical_veloc_sensor: Float;
horizontal_veloc_sensor: Float;
vertical_veloc_bias: Integer;
horizontal_veloc_bias: Integer;
…
begin
      declare
            pragma suppress (numeric_error, horizontal_veloc_bias);
      begin
            sensor_get(vertical_veloc_sensor);
            sensor_get(horizontal_veloc_sensor);
            vertical_veloc_bias:= Integer(vertical_veloc_sensor);
            horizontal_veloc_bias:= Integer(horizontal_veloc_sensor);
            …;
      exception
            when numeric_error => calculate_vertical_veloc;
            when others => use_irs1;
      end
;
```

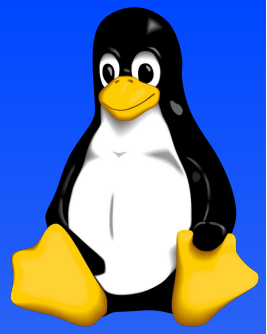
Die kontrollierte Ausnahme- Behand-lung wurde explizit ausgeschaltet

Hier entstand ein Überlauf

Das Programm schaltet auf einen anderen Rechner um, der aber dasselbe Problem hatte.



Ariane 501



Was kann man daraus lernen:

- Software kann gefährlich sein, auch wenn sie gar nicht gebraucht wird.

- Sicherheitsmechanismen helfen nichts, wenn sie ausgeschaltet sind.

- Bei nicht ganz korrektem Pflichtenheft entstehen leicht gefährliche Fehler.

- Redundanzen nützen natürlich nichts, wenn in allen redundanten Systemen der gleiche Fehler auftritt.

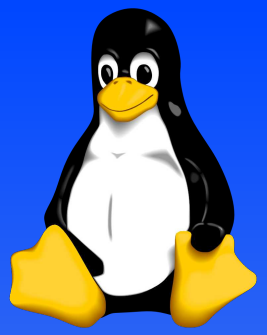
- Fehler die eigentlich gar nicht auftreten können, treten trotzdem auf....

- 3 Mio Linien Sourcecode werden nicht über Nacht neu geschrieben....

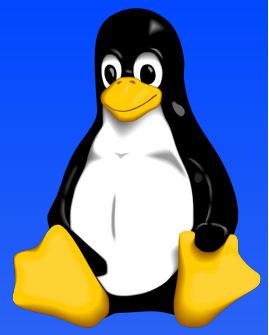
Küzliches Beispiel: Olypische Spiele 2012

Softwarefehler: Computer wertete 2 genau gleiche Ergebnisse als nur ein Ergebnis, wodurch sich die gesamte Rangliste verschob.

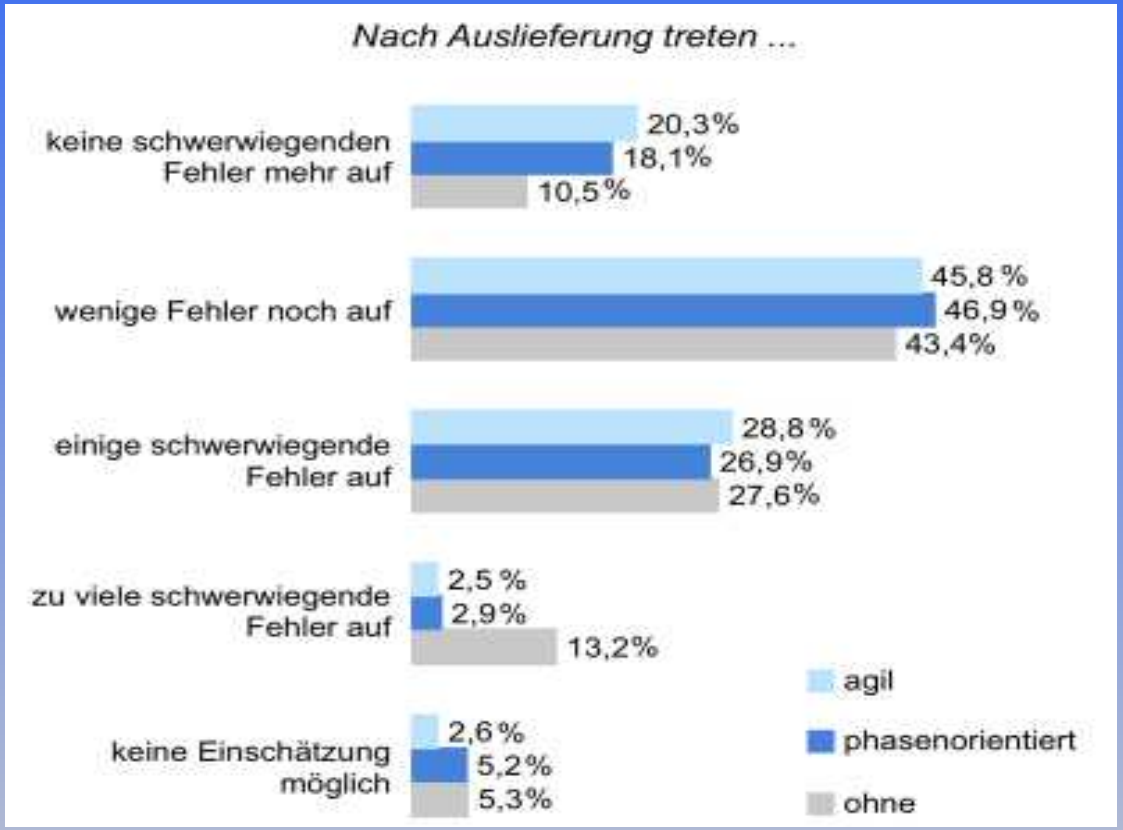
http://www.spiegel.de/sport/sonst/olympia-2012-software-fehler-schuld-an-heidler-drama-im-hammerwurf-a-849533.html



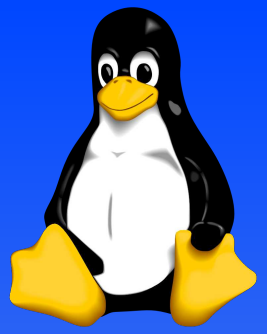
Softwarefehler



Amüsant zu lesen: Die 10 übelsten Softwarefehler 2012:
http://www.inside-it.ch/articles/31318

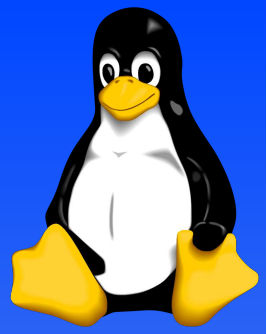


http://greiterweb.de/spw/FehlerReduzierungswege.htm



Klassifikation von SW-Fehlern

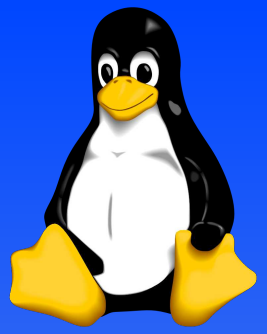
(nicht vollständig)



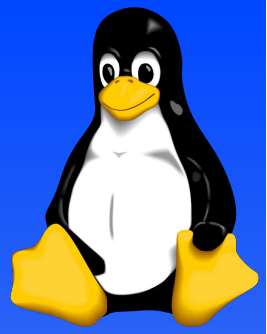
Syntaktische und **lexikalische Fehler:** Vom Compiler erkannt (aber was passiert bei interpretierten Sprachen?).

Semantische und **logische** Fehler: Können häufig bei Tests entdeckt werden.

Zuverlässigkeitsrelevante Fehler: Treten (meist) nicht reproduzierbar auf.



Zuverlässigkeitsrelevante Fehler



Memory Leaks

Nicht geprüfte Rückgabewerte

(kurzzeitige) Hardwarefehler

Nicht geprüfte Eingabewerte

Falsches Pointer Handling

Zuverlässigkeits-relevante Fehler

Race Conditions

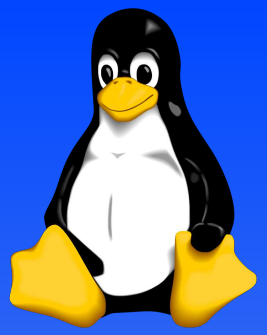
Stack-Überlauf

Nicht konvergierende Algorithmen

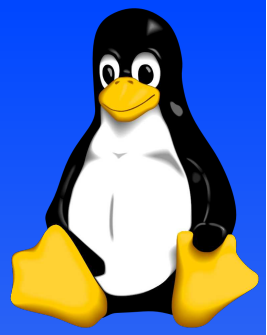
Bufferüberlauf

Variablenüberlauf

Präzision von mathematischen Operatoren



Zuverlässigkeitsrelevante Fehler



Memory Leaks

Nicht geprüfte Rückgabewerte

(kurzzeitige) Hardwarefehler

Nicht geprüfte Eingabewerte

Falsches Pointer Handling

Zuverlässigkeits-relevante Fehler

Race Conditions

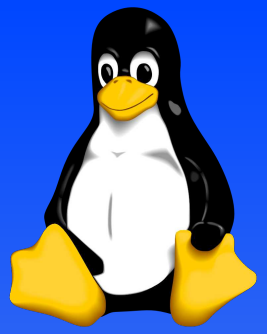
Stack-Überlauf

Nicht konvergierende Algorithmen

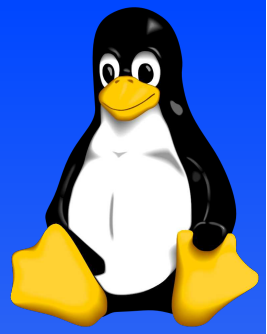
Bufferüberlauf

Variablenüberlauf

Präzision von mathematischen Operatoren



Typische zuverlässigkeitsrelevante Fehler

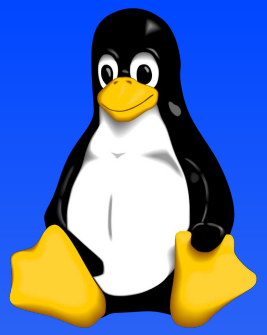


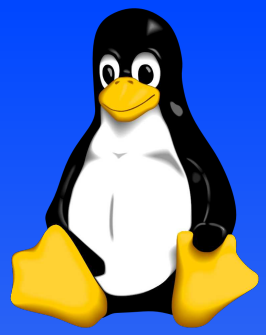
Nicht geprüfte Rückgabewerte

```
int fd, ret, i = 12345;   // some variable
char buffer[100];
fd = open(....);
ret = read (fd, buffer, sizeof(buffer)-1);
buffer[ret] = '\0';
......
```

Read ist blockierend bis die Daten vorhanden sind (kanonischer Mode)

--> Unterbrechung durch Signal
    --> ret = -1
    --> Variable i wird verändert !!!





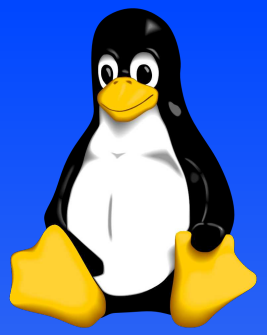
Bessere Lösung

Bessere Implementierung (unterbrechungssicher):

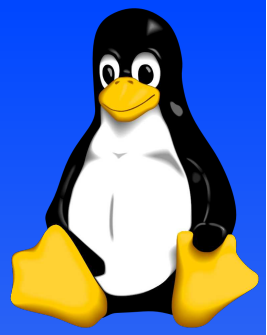
```
char buffer[100]; char* b=buffer;
int fd, ret, number=sizeof(buffer)-1;
fd = open (....);
while (number !=0 && (ret = read (fd, b, number)) != 0){
    if(ret == -1) {
        if (errno == EINTR) continue;
        perror(″read″);
        break;
    }
    number -= ret;
    b += ret;
}
*b = '\0';
```

--> **Defensives Programmieren:** Konsequente Überprüfung von Benutzereingaben

-- > Konsequente Überprüfung von Funktionsrückgabewerten



Race Conditions



Treten i. A. beim Zugriff auf gemeinsame Ressourcen auf !

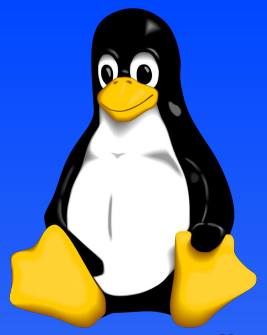
Bei heutigen Prozessoren sind atomare Zugriffe eher selten:

i++; ⟶ LD rx, (address)
inc rx
ST (address), rx

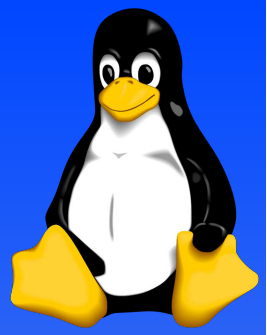
Gegenseitiges Ausschliessen:
- Mutexes
- Semaphores

Es gibt viele interessante Methoden, Race Conditions aufzuspüren: Guter Überblick: http://www.cs.cmu.edu/~nbeckman/papers/race_detection_survey.pdf

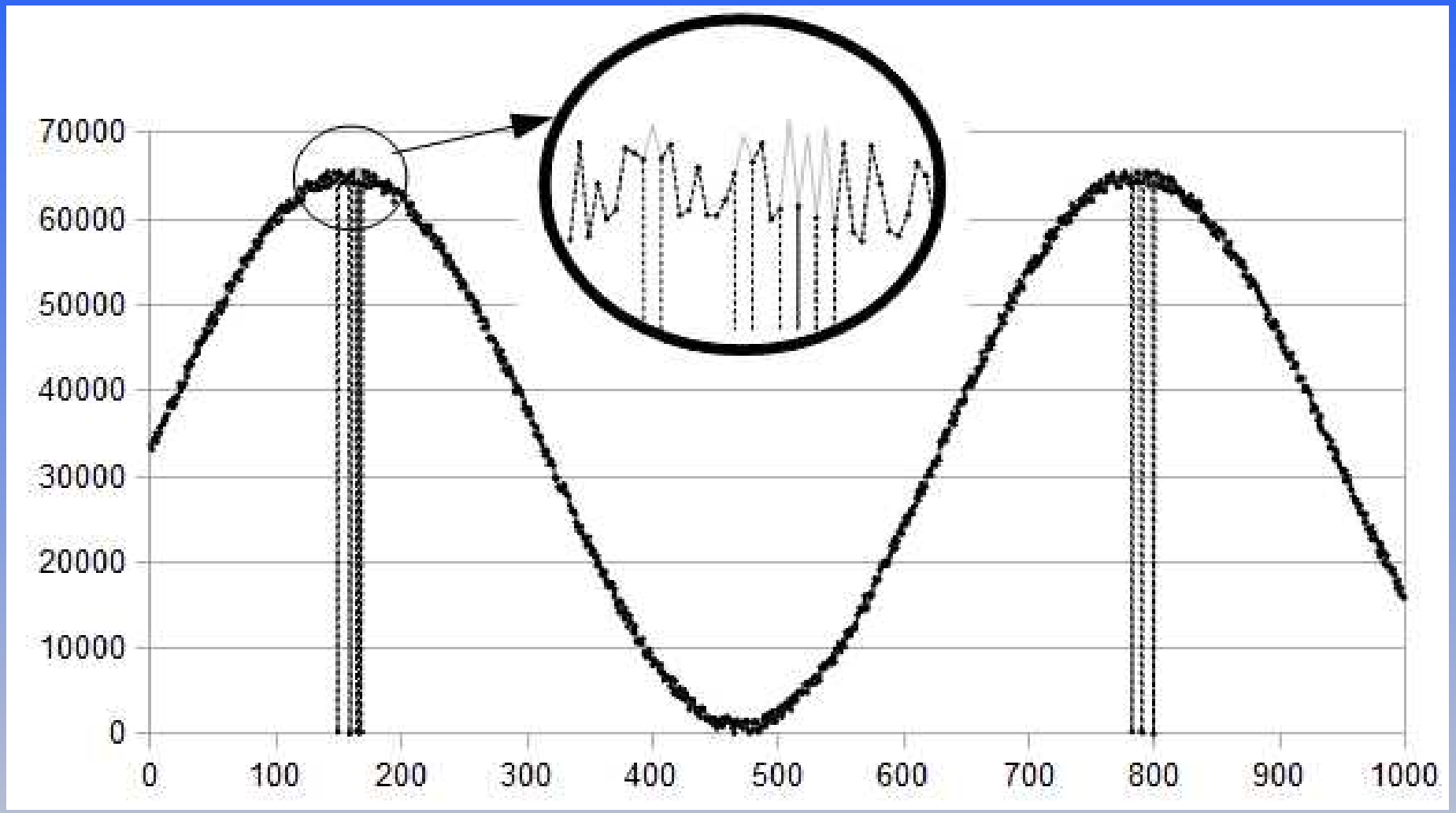
Achtung: Race Conditions können häufig in Debugging- oder Logging-Modi nicht reproduziert werden --> Heisenbug

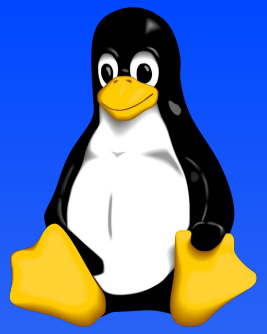


Überlauf von Variablen

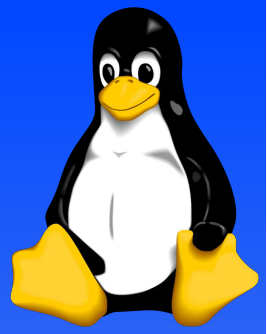


Überlauf von Variablen (hier: unsigned short)





Vermeidung von Variablenüberläufen



z.B. **Framework:** **safe_iop**

safe_mul()
safe_div()
safe_add()
safe_sub()
safe_shl()
safe_shr()

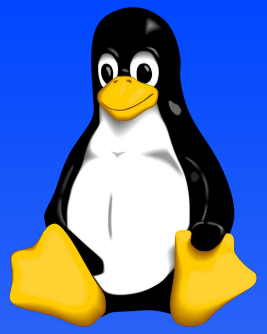
Beispiel:

```
uint32_t a = 100, b = 200, c = 0;
int ret = safe_mul(&c, a, b);          // returns true if OK, false if overflow
```

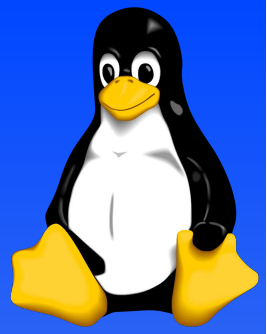
- Umschreiben von bestehender Software ist sehr mühsam bzw. unmöglich

- Müsste vom Compiler gemacht werden!

- Bei Sprachen, die ein Überladen von Funktionen oder Operatoren ermöglichen, ist es einfacher....



Bufferüberläufe



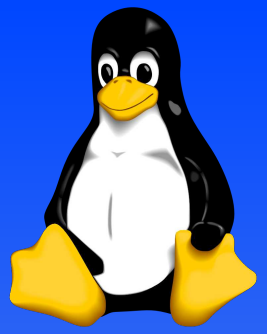
Eine spezielle Art von Variablenüberläufen

Vermutlich der häufigste Fehler in Programmen, mit teils schwerwiegenden Folgen.

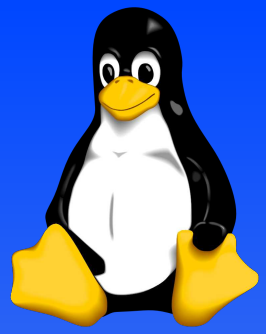
Grosse Disziplin notwendig:
- Standardbibliothek verwenden
- **strncpy(buf, s, BUFSIZE – 1);** ist besser als **strcpy(buf,s);**

Durch eine gute Initialisierung kann zumindest ein deterministisches Verhalten erzwungen werden:

memset(the_array, '\0', sizeof(the_array))



Nicht konvergierende Algorithmen



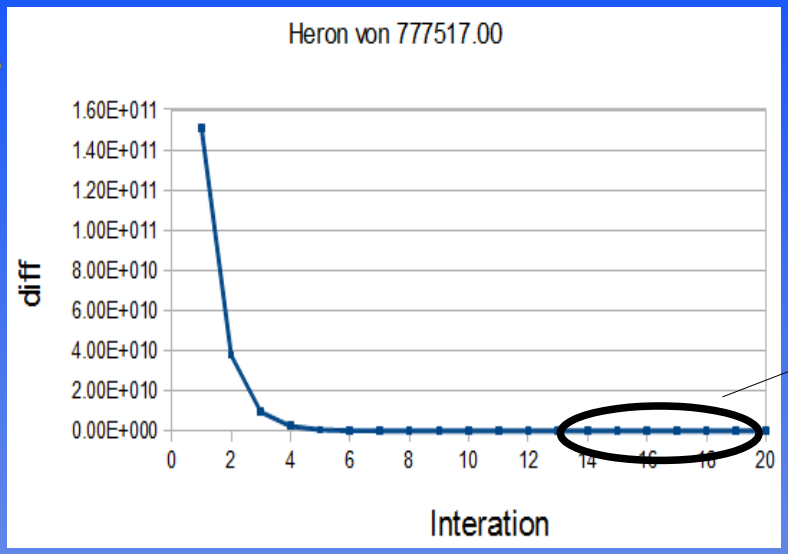
Beispiel: Heron-Verfahren zur Berechnung der Quadratwurzel:

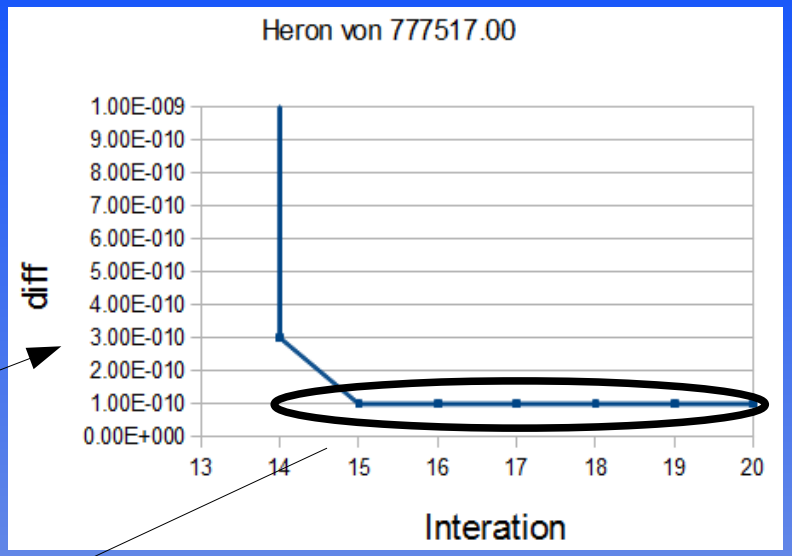
Die Quadratwurzel einer zahl A berechnet sich nach folgender iterativen Regel: $x_{n+1} = \frac{1}{2} \cdot \left(x_n + \frac{a}{x_n} \right)$

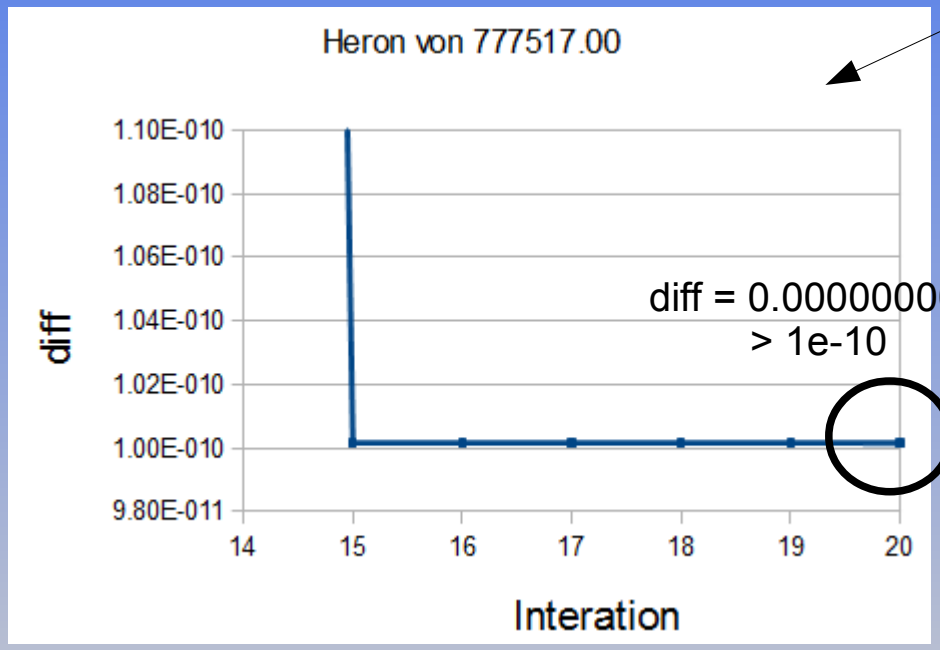
```
double sqrt(double a, double precision){
    double diff=1;
    double y, x=1.0;
    int iterations = 0;
    if(a<0){return(-1);}
    do {
        y = 0.5*(x+a/x);
        diff = y*y  - a;
        if (diff<0) diff = - diff;
        x=y;
    } while(diff > precision);
    return(y);
}
```

Dieser Algorithmus funktioniert sehr gut, ausser z.B. bei **777517.00** mit einer Genauigkeit von 1e-10

Nicht konvergierende Algorithmen





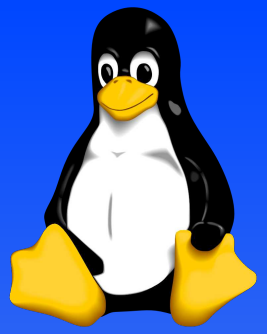


diff = 0.000000000100158104 03
> 1e-10

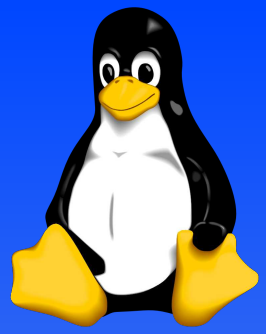
diff = y*y - a;

Subtraktion zweier grosser Zahlen ist sehr ungenau !!!

Anderes Abbruchkriterium wählen!



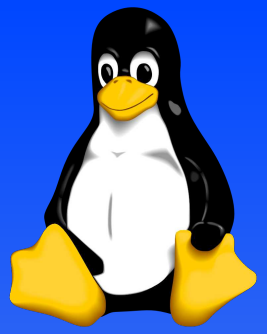
Genauigkeit von Fliesskomma-zahlen



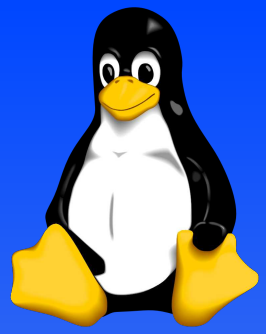
```
void main(){
   double zahl = 0.00;
   int i;
   for(i = 0; i < 20; i++){
     printf("%0.20f\n", zahl);
     Zahl += 0.01;
   }
}
```

```
0.00000000000000000000
0.01000000000000000021
0.02000000000000000042
0.02999999999999999889
0.04000000000000000083
0.05000000000000000278
0.06000000000000000472
0.07000000000000000666
0.08000000000000000167
0.08999999999999999667
0.09999999999999999167
0.10999999999999998668
0.11999999999999998168
0.12999999999999997669
0.13999999999999998557
0.14999999999999999445
0.16000000000000000333
0.17000000000000001221
0.18000000000000002109
0.19000000000000002998
```

Regel: float: ~ 7 Stellen sind genau
double: ~ 16 Stellen sind genau



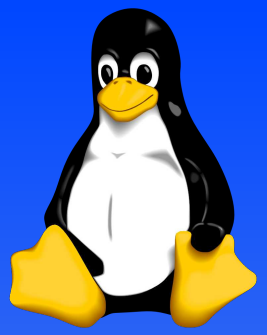
Memory Leaks



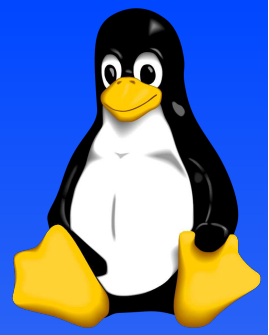
```
int foo (void){
    struct Struc *s= malloc (sizeof(struct Struc));
    if (s == NULL) return(-1);
    ....
    if (some_error) return(-2);
    ...
    free(s);
    return(0);
 }
```

Durch sauberes Funktionsdesign können solche Fehler vermieden werden

Leicht zu testen: free, top, ps memwatch.
oder über Funktionen im Programm mit mtrace();

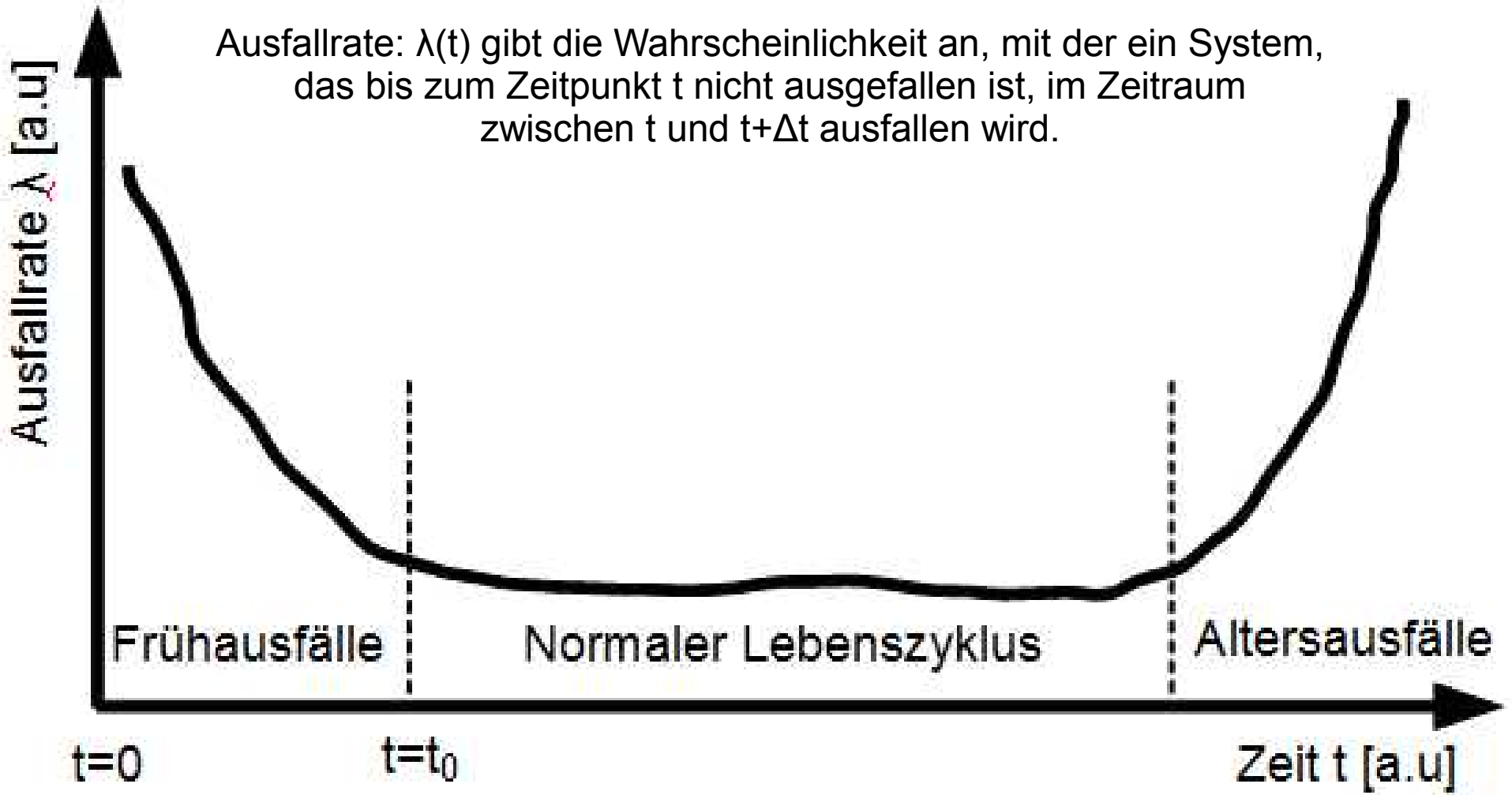


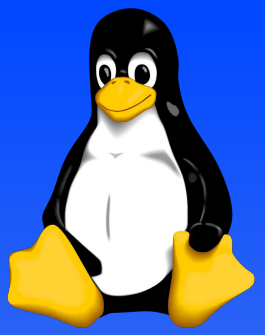
Klassische Zuverlässigkeitstechnik



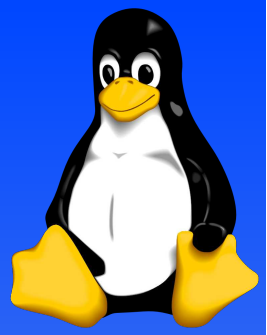
Ausfälle treten zufällig auf

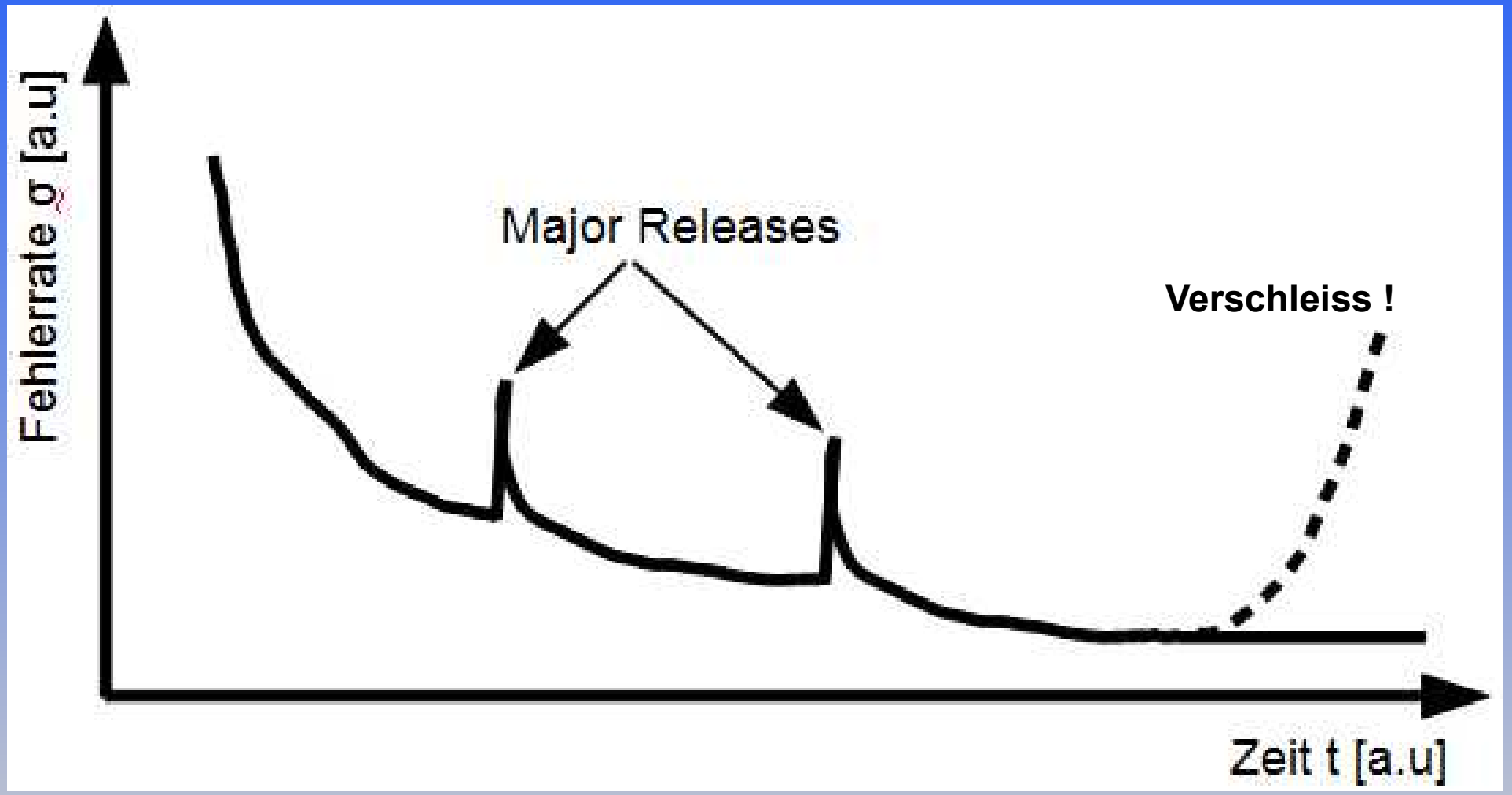
Ausfallrate: λ(t) gibt die Wahrscheinlichkeit an, mit der ein System, das bis zum Zeitpunkt t nicht ausgefallen ist, im Zeitraum zwischen t und t+Δt ausfallen wird.

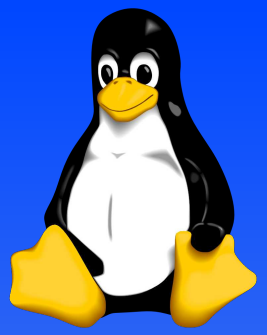




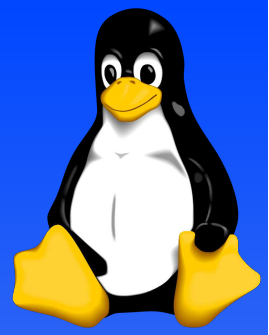
Fehlerrate bei Software







Vergleich HW / SW

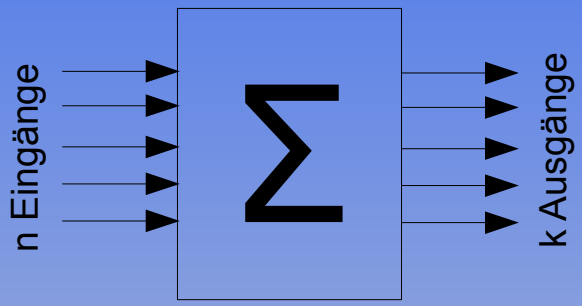


	HW	SW
Ausfallrate der einzelnen Komponenten reduzieren	Derating, ...	Modul-Tests und Fehlerreduktion
Burn-In	wird oft gemacht	Kann nicht angewandt werden
Redundanzen	wird oft angewandt	Muss von unterschiedlichen SW-Teams erstellt werden
Wartung	reinigen/schmieren etc.	System-Wartungsarbeiten (Aufräumen im System)
Reparatur	Erneuerung von einzelnen Komponenten	neue Releases

HW- Zuverlässigkeit und -Test

Systeme
- MTBF-Berechnungen
- Failure Mode and Effectif Analysis (FMEA)
- Fehlerbaumanalyse FTA

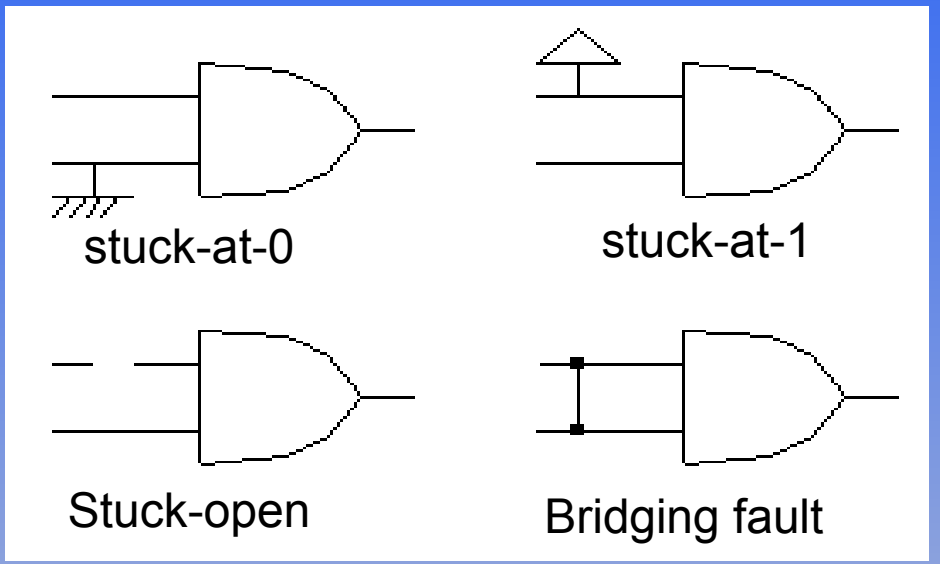
Mikroelektronik:



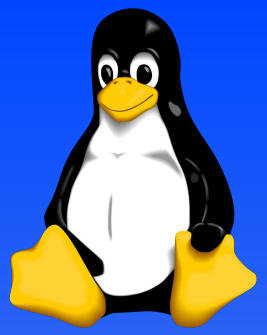
Anzahl Tests = $k \cdot 2^n$

Diese Zahl explodiert bei sequentiellen Systemen!

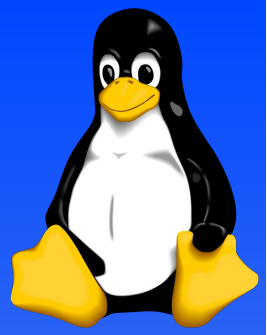
Fehlermodelle:



Diese Modelle führen zu automatisch generierten Testvektoren



SW-Zuverlässigkeitsmodelle

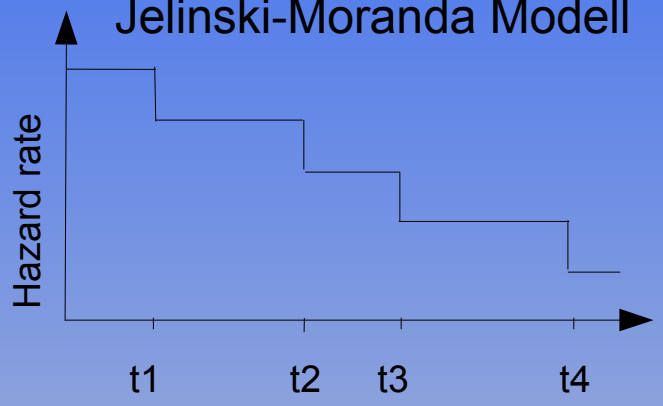


Prof. Matthias Werner, TU Chemnitz: Modelle für SW-Verlässlichkeit

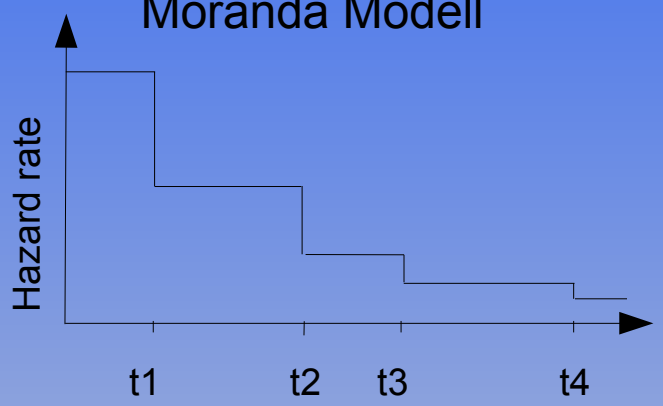
SoftwareZuverlässigkeitsWachstumsModell (SZWM)

- Position der Fehler sind nicht bekannt
- Exakte Sequenz der Verwendung der Programme ist nicht bekannt

Jelinski-Moranda Modell

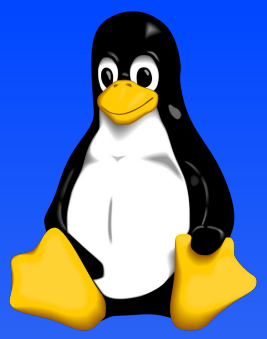


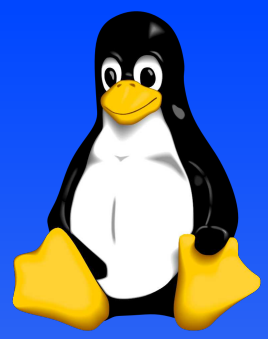
Moranda Modell



Andere Modelle/Annahmen:
- Anzahl der Fehler in einer Software ist proportional zur Anzahl von Codezeilen im System
- Hazardrate (Gefährlichkeitsrate) ist proportional zur Anzahl verbleibender Fehler.
- .....

Automatische Unit-Tests, CUnit, uCUnit; testgetriebene Entwicklung, Mocking....





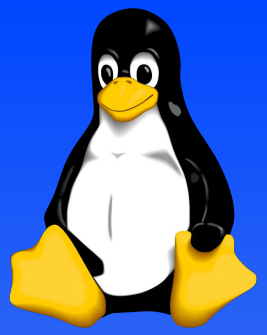
Zuverlässig-keits-zyklus

1.) Zuverlässig-keitsoptimiertes Design

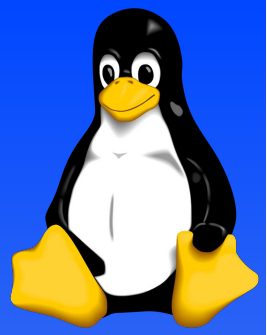
2.) Vermeidung von Softwarefehlern in der Implementierungs-phase

3.) Analysen und Tests

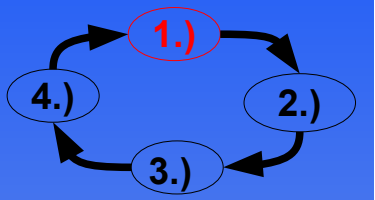
4.) Reduktion der Auswirkung von Software-fehlern



Zuverlässigkeitsoptimiertes Design



Multithreading ⟷ **Multiprocessing**



Race conditions?

Komplizierte IPC

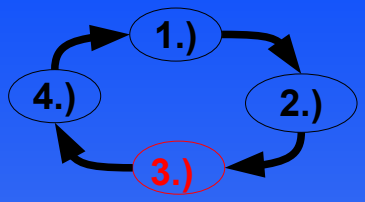
Wahl der richtigen Programmiersprache: Bitte die Entscheidung an objektiven und weniger an subjektiven Kriterien festmachen

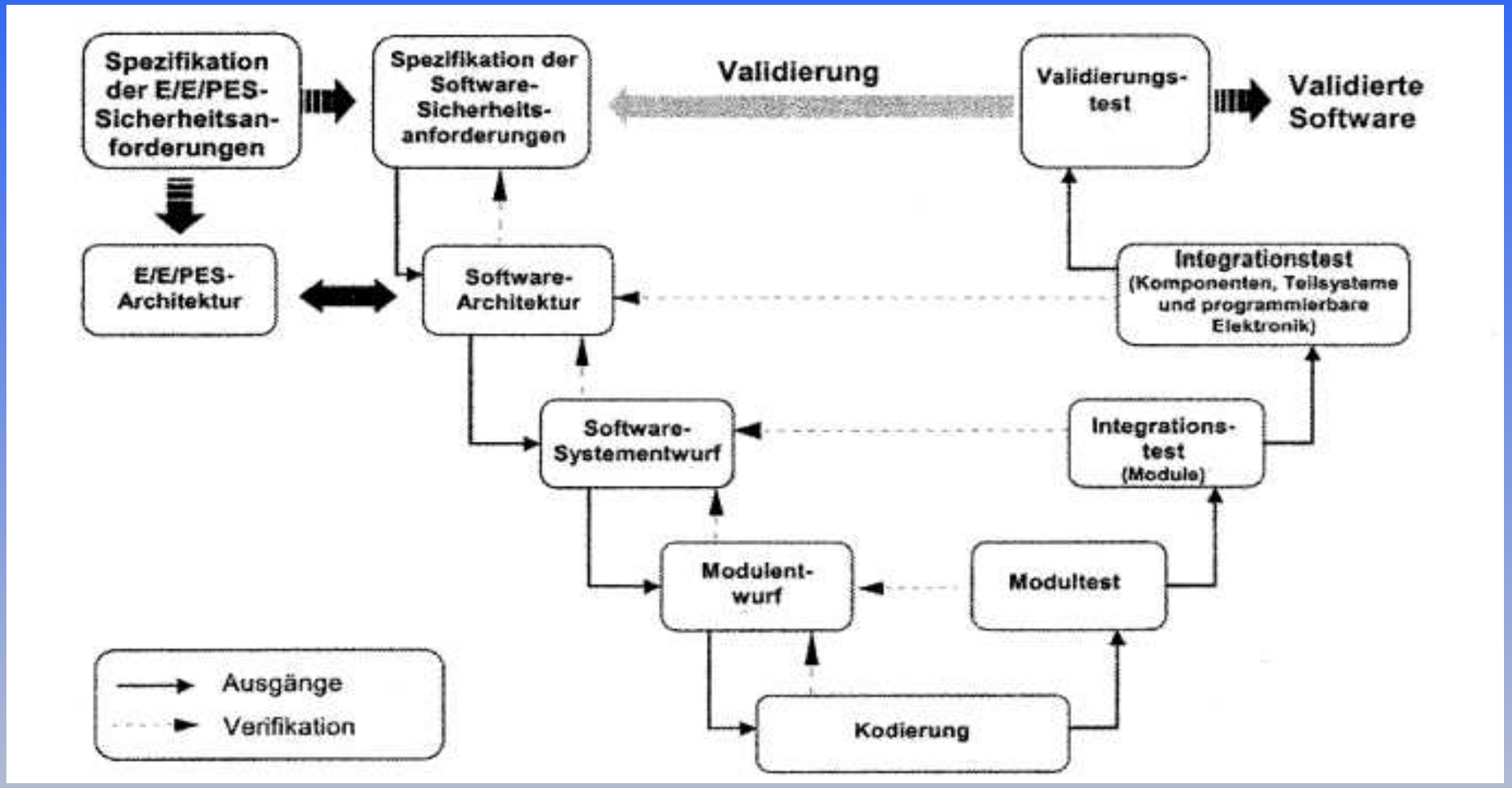
SW-Architektur und SW-Design bereits vor der Implementierungsphase analysieren

SW sollte möglichst einfach gehalten werden: Zu grosse Komplexität ist der häufigste Grund für Fehler und damit für Ausfälle

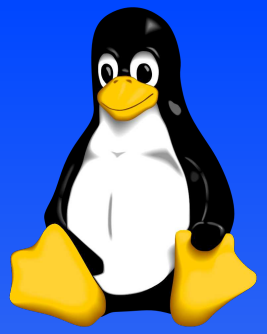
Transaktionen etc. vorsehen

Testen von SW nach dem V-Modell

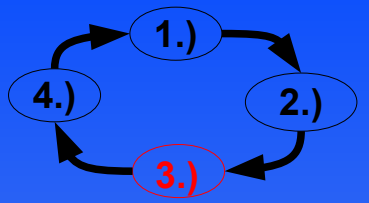


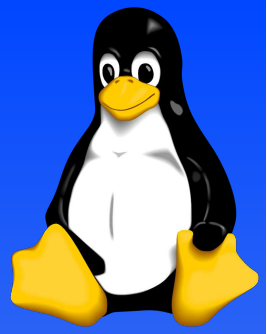


Nach: Anselm Schaefer, Grundlagen der Zuverlässigkeitstechnik, http://www.isar.tum.de/tum/t07



Analysieren von Software





Validierung: **„Building the right system“**

Verifizierung: **„Buildling the system right“**

Die Möglichkeiten des Compilers ausnützen: -Wall -Wextra

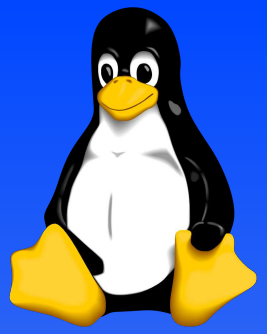
Langfristige Beobachtung des Speicherbedarfs einer Software z.B. mit top

Statische Codeanalyse-tools: Splint, Sparse, CODAN (Eclipse Plugin)

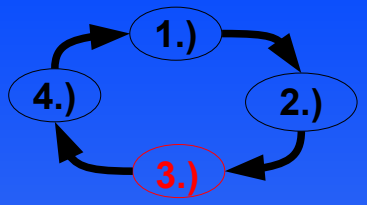
Syslog bietet hervorragende Möglichkeiten Software zu analysieren (zumindest in der Entwicklungs- und Testphase

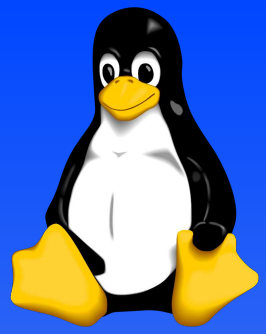
Performancemessungen am laufenden System

Dynamische Codeanalyse (z.B. mit Valgrind)

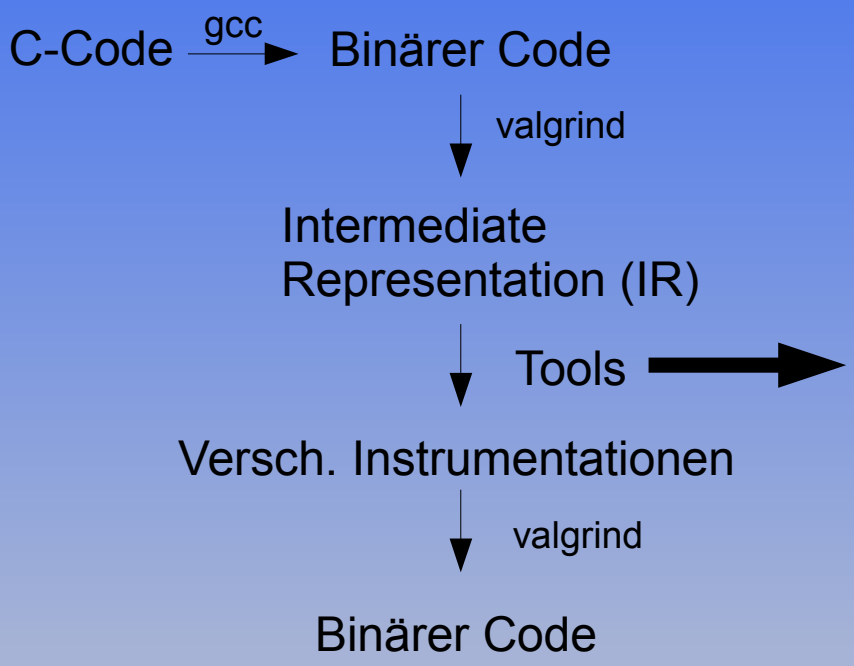


Valgrind





Valgrind ist ein Framework, um das dynamische Verhalten von Programmen zu untersuchen. Es ist eine virtuelle Maschine, die den Code vor dem Ausführen erst in einen Zwischencode compiliert, der dann durch unterschiedliche Tools modifiziert werden kann. Erst danach wird ein Maschinencode erstellt.



Memcheck: findet nicht initialisierten Speicher, Schreiben über Speichergrenzen, Memory-Leaks

SGcheck: ähnlich wie Memcheck, benützt aber andere Methoden zum Auffinden von Speicherzugriffsfehlern

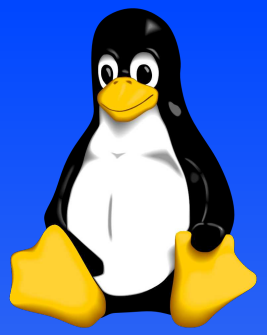
Callgrind: Misst Anzahl CPU-Takte, Cache-Zugriffe

Helgrind: Thread-Fehler detektor Auffinden von Race Conditions

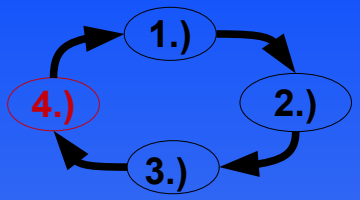
DRD: ähnlich wie helgrind, benützt aber andere Algorithmen

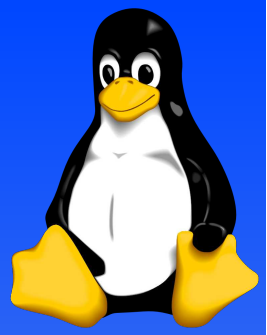
Massif: Heap und Stack-Profiler

DHAT: Heap-Profiler, findet Speicher-Layout-Probleme



Reduktion der Auswirkung von Softwarefehlern





Wenn es das Programmiermodell bzw. die programmierpsrache erlaubt: **Exception Handling** anwenden. Unter gnu/Linux können z.B. Signalhandler implementiert werden.

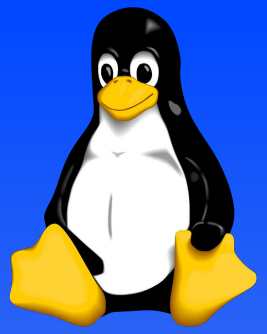
Konsequent **Timeouts** implementieren: Was passiert bei einem Timeout? Abbruch oder Neuversuchen?

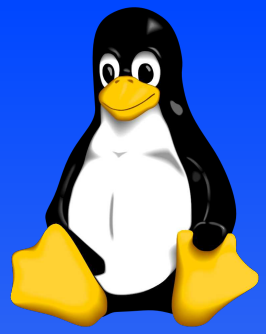
Prozesse können **neu gestartet** werden (z.B. über init). Achtung in diesem Fall auf die richtige Initialisierung achten.

Automatische Erkennung der verbleibenden **Speichermenge** (/proc/meminfo)

Ressourcenisolierung von Prozessen: gnu/Linux bietet über die Control-Groups hervorragende Möglichkeiten dafür.

Watchdogs verwenden





Wichtiger Event

Donnerstag, 6.6.2013: Hochschule für Technik und Architektur, Freiburg, Schweiz
17:00 – 19:30

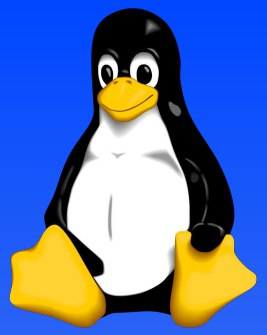
4. Freiburger Seminar über embedded Linux

Thema: Zuverlässigkeit von embedded Linux Systemen

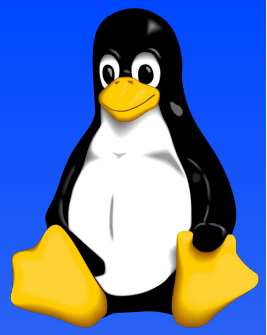
u.a. Vortrag von Fa. Verisign: SW-Sicherheit und Codeanalyse

Weitere Informationen ab Mai unter: http://portal.hefr.ch/eifr/else

oder: www.eia-fr.ch



Und Schluss....



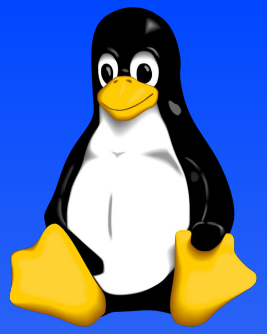
Niklaus Wirth (1999):

Anstelle von hektischen Umtrieben und Effekthascherei wäre mehr Augenmerk auf Zuverlässigkeit, Überschaubarkeit und Solidarität zu wünschen

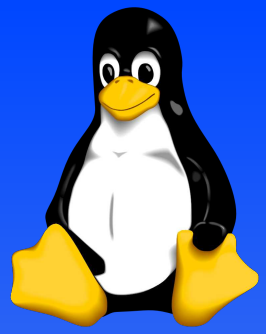
Wir wünschen Euch immer sicheres Programmieren!!!

Vielen Dank für die Aufmerksamkeit !

wolfram.luithardt@hefr.ch



Einige interessante Literaturtipps



- U.Vigenschow: Testen von Software und Embedded Systems, dpunkt Verlag (2010).

- D. Mölle: Testsieger, Stabile Software durch Design for Testability, iX 11/2012, 86-89.

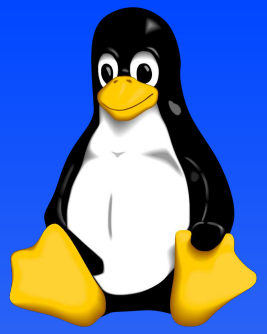
- A. Meyna und B. Pauli, Zuverlässigkeitstechnik, 2. Auflage, Hanser 2010.

- M. Grottke: Prognose von Softwarezuverlässigkeit, Softwareversagensfällen und Softwarefehler, in P. Mertens und S. Rässler(Hrsg), Prognoserechnung, 7. Auflage, Springer/Physica, 2012, S. 585-619.

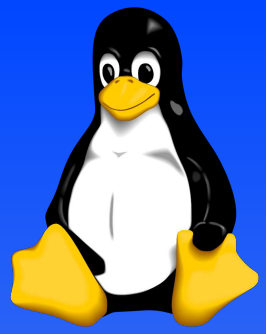
- A. Schäfer, Grundlagen der Zuverlässigkeitstechnik, Manuskript des Moduls T7, SS2011, Institute for Safety and Reliability.

- M. Meitner and F. Sagletti: Software Reliability Testing Convering Subsystem Interactions, in Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance (Jens B. Schmitt ed.), Lecture Notes in Computer Science, volume 7201, Springer-Verlag 2012.

- J. Pan: Software Reliability, carnegie Mellon University, 18-849b Dependable Embedded Systems, 1999.



Murphy'sche Gesetze über Zuverlässigkeit



Wenn die Möglichkeit besteht, daß verschiedene Fehler auftreten, wird der schlimmste eintreten.

If a program has not crashed yet, it is waiting for a critical moment before it crashes.

Software bugs are impossible to detect by anybody except the end user.

A patch is a piece of software which replaces old bugs with new bugs.

The probability of bugs appearing is directly proportional to the number and importance of people watching.