Fachhochschule
Münster University of
Applied Sciences

# Embedded Software

# Produktqualität

---

*Nichtfunktionale Anforderungen:* Zuverlässigkeit, Funktionale Sicherheit, Echtzeitverhalten und Ressourcenverbrauch, Wart- und Erweiterbarkeit.

Prof. Dr. Nikolaus Wulff

# Embedded Produkt Entwicklung

Die Entwicklung eines Produkts, bestehend aus Hard- und Software, erfolgt in der Industrie häufig nach dem V-Modell. Die wesentlichen Schritte sind:

- Anforderungsanalyse
- Architektur & Design Erstellung
- Implementierung & Modultests
- Integration & Integrationstest
- Über- und Abnahme in den Wirkbetrieb

Die einzelnen Schritte der Erstellung werden in Zeitabschnitte eingeteilt und die Planerfüllung mittels erreichten Meilensteine markiert & verifiziert.

# Modul Systemerstellung

- SE-1 Systemanforderungsanalyse
- SE-2 Systementwurf
- SE-3 SW/HW Anforderungsanalyse
- SE-4 SW/HW Grobentwurf
- SE-5 SW/HW Feinentwurf
- SE-6 SW/HW Implementierung
- SE-7 SW/HW Integration
- SE-8 Systemintegration
- SE-9 Überleitung in den Produktivbetrieb

Das V Modell detailliert die Planungsschritte...

# Nichfunktionale Anforderungen

- Eingebettete Systeme müssen häufig ohne Konsole oder Display im 365x24 Betrieb wartungslos und *ausfallsicher* arbeiten.
- Dies erfordert eine hohe *Fehlertolerenz* gegenüber Rauschen, Störungen und Ausreißern von Sensor-messwerten bedingt durch Umwelteinflüssen etc.
- Wichtig ist es nicht nur zu spezifizieren, was die Soft- und Hardware im *Normalfall* leisten, sondern auch wie sie sich im *Fehlerfall* verhalten soll.
- All dies muss sowohl gründlich *dokumentiert* als auch *getestet* werden.

# Nichfunktionale Anforderungen II

- Sollen externe Maschinen & Aktoren gesteuert und geregelt werden, so ist die *Antwortzeit* wichtig.
- Häufig werden Messungen und die Antwort darauf in einer (Endlos)Schleife ausgeführt.
- Dies erfordert den Einsatz eines *echtzeitfähigen* Systems und härtere Restriktionen an die eingesetzten Algorithmen. => *Laufzeitanalyse*
- Werden Ressourcen mittels Semaphore oder Barrieren geschützt muss auf *Deadlocks* geachtet werden, insbesondere falls Prozesse oder Threads mit unterschiedlicher Priorität verwendet werden.

# Sicherheit

Das deutsche Wort Sicherheit zerfällt im Englischem in die zwei zentralen Bereiche:

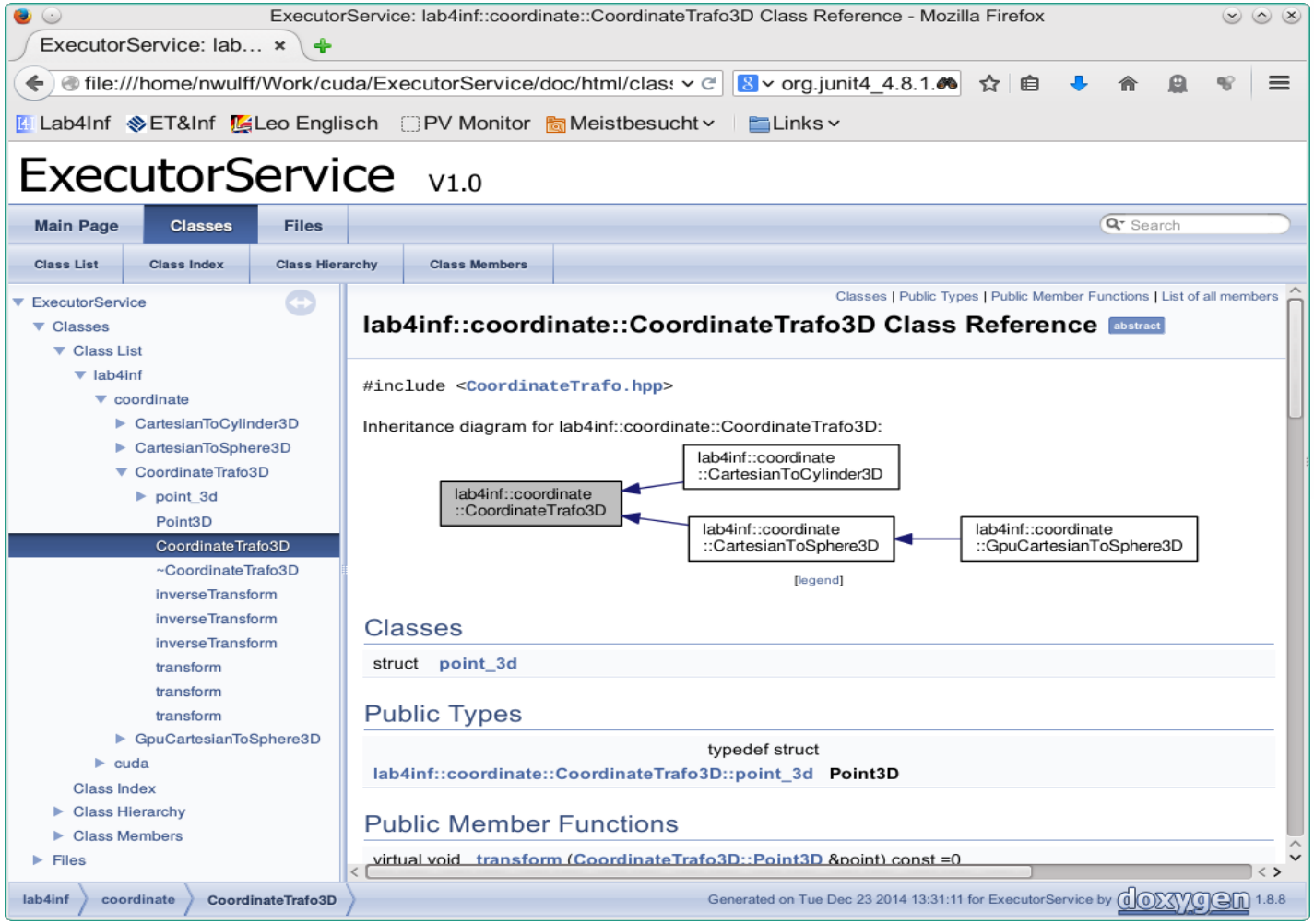
- *Safety* / Funktionale Sicherheit
  - Das System gefährdet seine Umgebung nicht durch unerlaubte Aktivitäten.
  - Wichtig bei Embedded Systeme die physisch Aktoren steuern oder regeln!
- *Security* / Zugriffssicherheit
  - Das System wird durch seine Umgebung nicht gefährdet.
  - Zu berücksichtigen sobald das System Datenschnitt-stellen „nach Außen“ hat (also fast immer).

Beides muss im Lastenheft spezifiziert werden!

# Dokumentation

- Embedded Software wird häufig über Jahrzehnte eingesetzt. Dies erfordert wart- und erweiterbaren Code, der entsprechend professionell dokumentiert sein muss.
- Wird Code nicht nur aus MATLAB/Simulink oder UML Werkzeugen erstellt, so ist eine interne Dokumentation im Quelltext um so wichtiger.
- Mit *Doxygen* läst sich ähnlich zu Javadoc auch in C/C++ Projekten eine HTML, RTF oder PDF Dokumentation erstellen, die eingebettet im Quelltext immer synchron zum Programm ist.

# Doxygen Dokumentation




- *doxygen*+*graphviz* generiert UML Diagramme

# Testüberdeckung

Die Qualität von Tests wird meist klassifiziert nach dem Grad $C_x$ der getesteten Codeabdeckung:

- $C_0$ – Jede Codezeile wird ausgeführt.
- $C_1$ – Jede Alternative wird ausgeführt
- $C_2$ – Alle möglichen Pfade werden ausgeführt.
- $C_3$ – Jede kombinatorische Bedingung einer Verzweigung wird ausgeführt.

$C_3$ erfordert erhebliche Anstrengungen und ist meist nur für sicherheitskritsche Anwendungen bezahlbar.

# Anweisungsüberdeckung

- $C_0$ kennzeichnet den Bruchteil an getesteten Anweisungen (*statement coverage*):

$$C_0 = \frac{\textit{getestete Anweisungen}}{\textit{alle Anweisungen}}$$

- $C_0$ ist die einfachste zu vermessende Metrik und kennzeichnet im Wesentlichen den Bruchteil an durch Tests geprüfte Codezeilen. Eine hohe $C_0$ Überdeckung sollte immer erreicht werden.

# Zweigüberdeckung

- $C_1$ misst den Anteil der getesteten Verzweigungen im Programmfluß (*branch coverage*):

$$C_1 = \frac{getestete\ Verzweigungen}{alle\ Verzweigungen}$$

- Eine hohes $C_1$ bedingt meistens auch ein höheres $C_0$ (solange in allen Verzweigungen die selbe Größenordnung an Anweisungen erfolgt), bzw. falls $C_1=1$ dann gilt auch $C_0=1$.

# Pfadüberdeckung

- $C_2$ misst den Anteil der getesteten Pfade durch das Programm (*path coverage*):

$$C_2 = \frac{getestete\ Pfade}{alle\ Pfade}$$

- $C_2$ und $C_1$ sind nicht identisch, gehören aber eng zusammen, da meist ein Pfad in Abhängigkeit von einer Bedingung gewählt wird.

# Bedingungsüberdeckung

- $C_3$ misst den Anteil der getesteten Terme (*komplexe if-else* Bedingungen) (*condition coverage*):

$$C_3 = \frac{getestete\ Bedingungen}{alle\ möglichen\ Bedingungen}$$

- Alle Terme einer Bedingung müssen sowohl mit *true* als auch mit *false* getestet werden. Hierbei gilt es besonders auch auf Seiteneffekte der „Kurzschluß Operatoren && und || zu achten.
  - Es ist möglich eine Verzweigung/einen Pfad einzuleiten ohne alle Bedingungen getestet zu haben...
- Ein hohes $C_3$ erfordert exponentiellen Aufwand.

# Wer testet den Tester?

- Gute Tests zu entwickeln ist nicht einfach und erfordert viel Gespür und Erfahrung, gerade wenn es um eine $C_2$ – $C_3$ Abdeckung geht.
- Jede funktionale und nichtfunktionale Anforderung sollte systematisch getestet sein.
- Die automatisierten Tests müssen selber getestet sein. Hierzu mag es helfen mit *Mock-Objekten* zu arbeiten. Die Tests müssen dann fehlschlagen und zeigen, dass sie auf Fehler der Mocks anspringen.

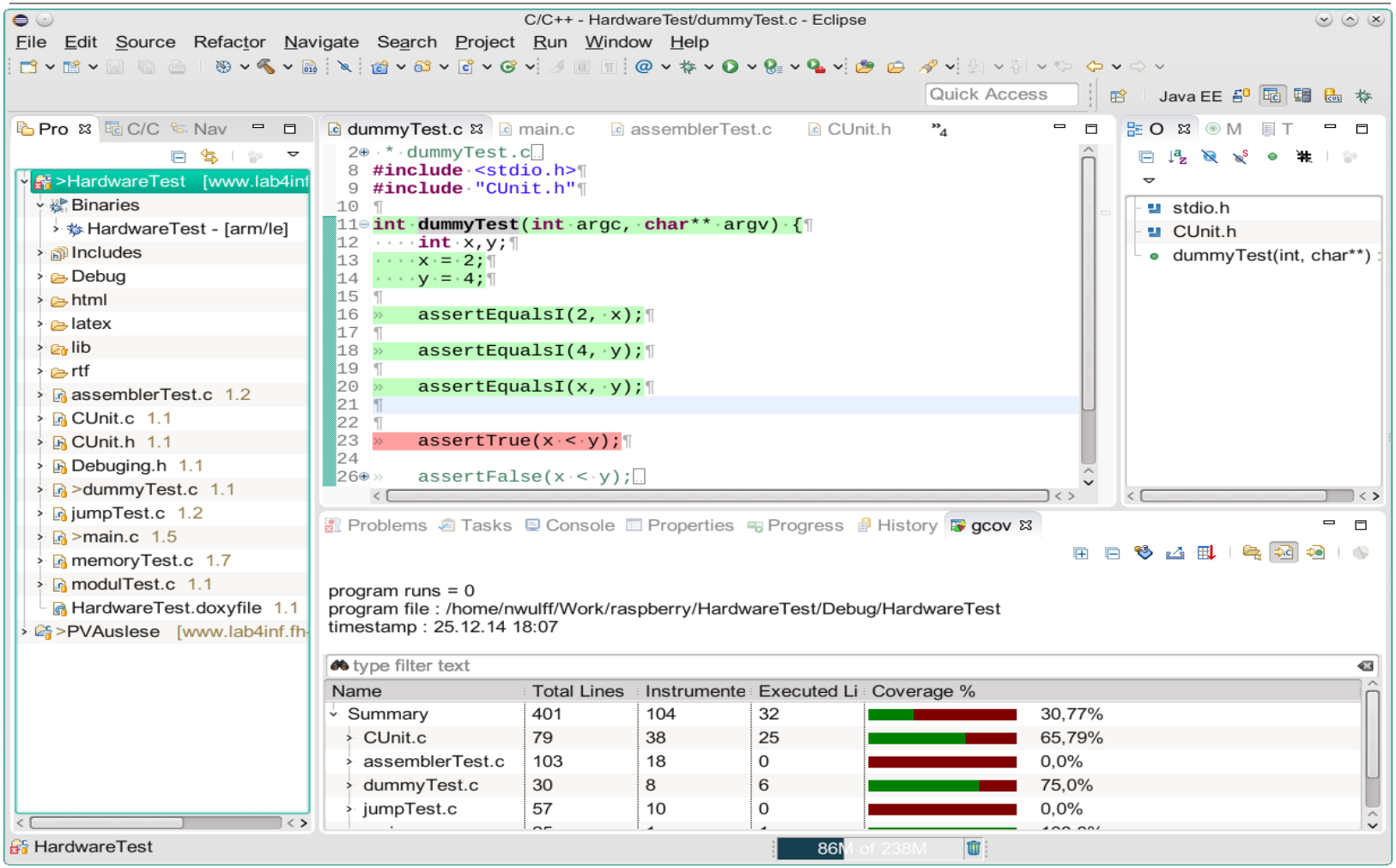
**Eine nicht testbare Software ist ein klares Indiz für ein schlechtes Design!**

Mock: engl. Attrape

# Testüberdeckung messen

- Mit Hilfe von *gcov* und einem entsprechendem Eclipse Plugin lassen sich C/C++ Anwendungen instrumentieren, um zu ermitteln welche Anweisungen während eines Testlaufs ausgeführt wurden.
- Um dies auch bei Cross-Compiler Projekten durchführen zu können, müssen vor dem Ausführen der Anwendung auf der Zielplattform die zwei Umgebungsvariablen GCOV_PREFIX und GCOV_PREFIX_STRIP gesetzt werden, da ansonsten versucht wird auf dem Remote Zielrechner die Pfade des lokalen Build Rechners aufzulösen, was meist fehlschlägt...

# Eclipse Gcov Plugin




- ARM RaspberryPi Testlauf in der Eclipse...

# Speichermanagement

- Im PC Bereich wird Software unter der Kontrolle eines Betriebssystem ausgeführt und Hochsprachen, wie Java oder C++, stellen ein automatisches Speichermanagement mit Garbage Collection bereit.
- Gerade dies steht der Echtzeitfähigkeit im Wege: Sollte während eines zeitkritischen Abschnitts der Garbage Collector zuschlagen, lassen sich harte Anforderungen an das Laufzeitverhalten eines Algorithmus kaum realisieren.

# Speicherlecks

- Mit der Toolsuite *valgrind* lässt sich der Speicherverbrauch und fehlende freigegebene Ressourcen sowie falsches Thread Handling identifizieren.
- Valgrind ist sowohl für Intel PCs als auch für ARM Architekuren als Open-Source erhältlich.
- Das Programm muss lediglich mit dem Flag *-g* und ohne Optimierung übersetzt werden, dann sind Zeileninformationen in den Ausgaben enthalten.

# Valgrind Testlauf

```
==30230==    by 0x488B26B: exit (exit.c:100)
==30230==    by 0x487082F: (below main) (libc-start.c:276)
==30230==  If you believe this happened as a result of a stack
==30230==  overflow in your program's main thread (unlikely but
==30230==  possible), you can try to increase the size of the
==30230==  main thread stack using the --main-stacksize= flag.
==30230==  The main thread stack size used in this run was 8388608.
==30230==
==30230== HEAP SUMMARY:
==30230==     in use at exit: 14,554 bytes in 2 blocks
==30230==   total heap usage: 3 allocs, 1 frees, 14,914 bytes allocated
==30230==
==30230== LEAK SUMMARY:
==30230==    definitely lost: 0 bytes in 0 blocks
==30230==    indirectly lost: 0 bytes in 0 blocks
==30230==      possibly lost: 0 bytes in 0 blocks
==30230==    still reachable: 14,554 bytes in 2 blocks
==30230==         suppressed: 0 bytes in 0 blocks
==30230== Rerun with --leak-check=full to see details of leaked memory
==30230==
==30230== For counts of detected and suppressed errors, rerun with: -v
```

- *valgrind* deutet auf ein Speicherleck hin...

valgrind ./HardwareTest

# Echtzeitsysteme

- Bei der Regelung physikalischer Systeme ist das Einhalten zeitlicher Rahmenbedingungen wichtig.
- Hierzu sind meist Echtzeitsysteme notwendig.
- Echtzeitfähig ≠ schnell.
- *Echtzeitfähig* bedeutet ein korrektes Ergebnis innerhalb eines vorbestimmten Zeitintervalls bereitzustellen.
- Hart echtzeitfähig: Zeitintervall wird garantiert eingehalten.
- Weich echtzeitfähig: Zeitintervall wird in der Regel eingehalten.

# Echtzeitfähigkeit

Probleme:

- Multitasking
- Speicherverwaltung (Swapping)
- Locks auf Ressourcen
- …

Lösungen:

- Echtzeitbetriebsystem
  (VxWorks, QNX, LynxOS...)
- Erweiterungen von „normalen Betriebsystemen“
- Zusätzliche Hardware

# Laufzeitverhalten

- Unabhängig davon ob eine Anwendung echtzeit-fähig sein soll oder nicht, lohnt es sich das Lauf-zeitverhalten zu untersuchen.
- Das Werkzeug *Gprof* bietet hierzu die benötigten Informationen. Beim Übersetzen mit *gcc* lediglich das Flag *-gp* setzen – oder in den Eclipse Projekt Einstellungen – und schon werden Zeitmessungen ohne großen Aufwand möglich.
- Die Methoden mit den meisten Aufrufen und/oder dem größten Zeitaufwand – und somit Kandidaten für eine Optimierung – sind schnell identifiziert...

# Gprof Testlauf

```
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) no time propagated

index % time    self  children    called     name
                0.00    0.00      32/32          walkDataBus [9]
[1]      0.0    0.00    0.00      32         writeTo [1]
-----------------------------------------------
                0.00    0.00       2/2           memoryTest [5]
[2]      0.0    0.00    0.00       2         bitout [2]
-----------------------------------------------
                0.00    0.00       1/1           runAllTests [8]
[3]      0.0    0.00    0.00       1         assemblerTest [3]
                0.00    0.00       1/1           mov2Ptr [7]
                0.00    0.00       1/1           mov [6]
-----------------------------------------------
                0.00    0.00       1/1           runAllTests [8]
[4]      0.0    0.00    0.00       1         dummyTest [4]
```

- Callgraph und Zeitmessung einer C Anwendung auf dem RaspberryPi mittels gprof Aufruf:

  gprof ./HardwareTest gmon.out

# Zusammenfassung

- Embedded Anwendungen in C lassen sich genauso komfortabel dokumentieren und testen wie die Pendants moderner Java Anwendungen.
- Mit etwas Phantasie und Einsatz lassen sich entsprechende Werkzeuge sowohl in eine automatisierte Build, als auch als Plugins in die Eclipse, NetBeans, etc. einbinden.
- Einem professionellem Einsatz steht nichts im Weg und es gibt „keine Entschuldigung“ für fehlende Professionalität im embedded Bereich.