

# Effizient programmieren

## Teil I: Effiziente Entwicklung

Institut für Aerodynamik und Gasdynamik

Sommersemester 2018



**Vorlesung** 12 x Doppelstunde, Donnerstag, 9:45-11:15, V21.01

**Dozent** Dr. Manuel Keßler

**Kontakt** Institut für Aerodynamik und Gasdynamik,  
Pfaffenwaldring 21, EG 0.37, 0711/685-63419

**E-Mail** kessler@iag.uni-stuttgart.de

**Sprechstunde** nach Vereinbarung

**Skript** Vortragsfolien und erläuternder Text  
→ ILIAS, Passwort: prog2go

**WWW** Terminverschiebungen, Infos, Materialien, ... → ILIAS

**Prüfung** Konzipiert als informative Veranstaltung für Interessierte  
6 LP. möglich für Programmierprojekt und Vortrag

1	UNIX-Werkzeuge	7
2	Shells	25
3	Make	40
4	GNU Autotools	59
5	Versionsmanagement	76
6	Testen	93
7	Debugging	128
8	Speicherfehler: dmalloc und Valgrind	149
9	Coding Standards	162
10	Dokumentation I: Code – doxygen	203
11	Codebereinigung und Konsolidierung	210
12	Entwicklungsprozesse	217
13	Entwicklungsumgebungen	232
14	Releasemanagement	243
	Zusammenfassung	249

Codeentwicklung an der Uni hat sich verändert:

- Nicht mehr individuelle Entwicklung, sondern Teams
- Gewachsene Codes über mehrere Dissertationen, manchmal über Jahrzehnte
- Hochkomplexe Anwendungen (gekoppelte Physik), Algorithmen (unstrukturiert/hybrid, implizit, Mehrgitter) und Umgebungen (massiv parallele Systeme)
- Sehr große, wenig überschaubare und ungenügend strukturierte Codesysteme

Erschwerend kommt hinzu:

- Kaum Ausbildung in Programmierung für Ingenieure
- Altertümliche Low-Level Implementierungen

**Fazit:** Heroische Einzelkämpfer sind heute überfordert!

(In vielen anderen Einrichtungen ist es kaum besser.)

Diese Veranstaltung sollte daher verbessern:

- Verständnis der grundsätzlichen Problematik
- Techniken zum effizienteren Einsatz der knappen Ressource Programmiererzeit (Personenmonate)
  - Programmierumgebung, Tests, Automatisierung, Fehlersuche und -behebung, Dokumentation
- Techniken zur effizienteren Nutzung der Rechnerressourcen (FLOPs, vorrangig in Teil II: Effiziente Programme)
  - Hardware, Parallelisierung, Techniken für Programmierprobleme, zukünftige Konzepte

Verfahren, Techniken und Werkzeuge werden vorgestellt, aber

- eigene Nutzung ist unbedingt notwendig!

Deshalb ergeht die Einladung:

- Nicht nur zuhören, sondern auch nachfragen!
- Evtl. Notebook mitbringen, Gehörtes gleich ausprobieren.
- Erlebte Probleme mitbringen (evtl. mir vorher zuschicken), gegebenenfalls mit gefundener Lösung vorstellen.

Veranstaltung lebt von der Interaktion!

# 1 UNIX-Werkzeuge

UNIX-artige (Linux, BSD, ...) Systeme, teilweise auch OS X und selbst Windows bieten viele nützliche Tools (evtl. nachrüstbar):

- Basics: cd, ls, cp, mv, rm, mkdir, rmdir, cat, more
- Dateisystem: ln, du, touch, dd, df, chmod, chown, find
- Dateien: cmp, od, split
- Textdateien: wc, tac, head, tail, sort, uniq, fold, diff, patch, grep
- Textbearbeitung: tr, sed, awk, expand/unexpand, cut, paste, join, printf
- Prozesse: ps, nice, kill, killall, fg, bg, pstree
- Nützliches: man, info, file, env, sleep, date, time, tee, tar, xargs, gzip, dirname, basename, uname, who, finger, chroot, at, cron, install, su, hostname, seq, nslookup
- Shellbefehle: pwd, which, set, test, alias, echo, export

## Weitere Tools zum Programmieren

- aclocal, autoconf, automake, m4, make, cmake
- ar, as, cpp, c++filt, strings, nm, readelf, ld, objdump, ranlib, strings, strip
- cc/gcc, fc/gfortran, CC/g++
- gdb, ddd, ltrace/strace, valgrind
- bison, flex, yacc
- doxygen
- emacs, vi/vim, joe, pico

## Zusätzliche Anwendungen

- dos2unix/unix2dos, enscript, gnuplot, L<sup>A</sup>T<sub>E</sub>X, xfig, dot, mmv/mcp, xv
- zip, bzip2, troff/groff, locate, iconv, mail, convert (ImageMagick), rsync, wget, ssh/scp, zdiff/zmore/zgrep/zcmp/zcat, ping

## Systemkommandos

- ldconfig, mktexlsr, passwd, mount/umount



Auch häufig genutzte Kommandos haben außer den bekannten:

- a/-A alle/fast alle Dateien (also auch mit „.“ beginnend/ohne „.“ und „..“)
- l langes Anzeigeformat (Rechte, Eigentümer, Gruppe, Größe)
- t sortiert nach Datum/Zeit (neues zuerst)
- S sortiert nach Größe (groß zuerst)
- r umgekehrt sortiert

einige weitere nützliche Optionen:

- d Verzeichnisnamen anzeigen statt Inhalte (bei Wildcards)
- l einspaltig
- R rekursiv in alle Unterverzeichnisse
- h lesbare Größenangaben bei -l (35k, 738M, 1.5G)

Einige Optionen sind relativ universell für sehr viele Tools verfügbar:

`--help` Hilfetext, manchmal auch `-h`

`--version` Versionsangabe, manchmal auch `-v` oder `-V`

Häufig gibt es kurze (`-r`) und lange (`--reverse`) Optionen gleicher Bedeutung.

Manche Optionen erfordern Parameter, die dann unterschiedlich anzugeben sind, beispielsweise

`-I/--ignore` Dateien, die einem Muster entsprechen, sollen nicht angezeigt werden: `-I '*~' / --ignore='*~'`

Aus dem Rahmen fallen etwas

- keine Option, sondern Kennzeichen für die Standardeingabe (oder -ausgabe) anstelle eines Dateinamens
- Ende der Optionen, alles danach ist keine Option mehr, selbst wenn es so aussieht

grep (Varianten egrep/fgrep) sucht nach Zeilen mit regulären Ausdrücken. Wichtige Optionen:

- i Groß-/Kleinschreibung ignorieren
- l nur Anzahl Treffer ausgeben
- v Zeilen ausgeben, die den Ausdruck *nicht* enthalten

GNU grep Erweiterungen:

- r rekursiv in Unterverzeichnissen suchen
- w ganze Wörter finden
- C NUM NUM Zeilen vorher und nachher als Kontext
- H Dateiname voranstellen
- L nur Dateinamen, in denen nichts gefunden wurde

Möglichkeit, Klassen von Zeichenketten zu definieren

**Text** buchstabengetreue Definition

- beliebiges Zeichen: `gro.` passt auf `grob` und `groß`, aber nicht auf `gram`
- ? vorheriges Element optional: `derb?` passt auf `der` und `derb`
- + vorheriges Element mindestens einmal: `derb+` passt auf `derb`, `derbb`, `derbbb`, nicht aber `der`
- \* vorheriges Element beliebig oft: `derb*` passt auf `der`, `derb`, `derbb`, `derbbb`

$\{N\}/\{N,\}/\{N,M\}$  spezifiziert genaue Anzahl

?, + und \* sind am Ausdruckende kaum sinnvoll

**Vorsicht!** Reguläre Ausdrücke werden von zahlreichen Tools benutzt, mit immer leicht unterschiedlicher Syntax und Ausdrucksfähigkeit: `grep`, `sed`, `awk`, Shells, `perl`, `python`, `make`, ...

- | entweder das Element links oder das rechts passt: `der|das` passt sowohl auf `der` als auch auf `das`
- [...] Zeichenklassen: eines der Zeichen innerhalb der eckigen Klammern: `[0-9]` passt auf eine Ziffer, `[a-zA-Z]` auf einen Buchstaben (groß oder klein)
- `[[:alnum:]]` spezielle Zeichenklassen: in diesem Fall ein Buchstabe oder eine Ziffer
- `[^a-zA-Z0-9]` das Caret `^` am Anfang kehrt die Klasse um: alles *außer* Buchstabe oder Ziffer
- `[[:alpha:]]_ [[:alnum:]]*` Name nach C-Regeln
- `(:)?([[:alpha:]]_ [[:alnum:]]*::)*[[:alpha:]]_ [[:alnum:]]*` qualifizierter Name nach C++-Regeln
- `[+-]?([[:digit:]]+\.[[:digit:]]*)|(\.[[:digit:]]+)([eE][+-]?[[:digit:]]+)?` eine Fließkommazahl

## Verankerungen

`\b / \B` Wortkante (Anfang oder Ende) / Wortinneres

`\< / \>` Wortanfang / -ende

`\w / \W` Wortzeichen (`[[[:alnum:]]]`) / Nicht-Wortzeichen (`[[^[:alnum:]]]`)

`^ / $` Zeilenanfang / -ende

Rückbezüge `\N` (also `\1`, `\2`, ...) wiederholen den N-ten zurückliegenden Klammerausdruck, beispielsweise

`(\w)(\w)(\w)\3\2\1` für ein sechsbuchstabiges Palindrom (hannah)

`? + | { ( )` nur in „extended syntax“ mit Sonderbedeutung (egrep oder grep -e), nicht in „basic regular expressions“ (dort aber über `\?`, `\+`, `\|`, `\{`, `\(` und `\)` zugänglich)

**find DIR [spec...]** sucht nach Dateien (auch in Unterverzeichnissen) mit spezifischen Eigenschaften

Die Spezifikation, was zu finden ist, kann sich aus mehreren Komponenten zusammensetzen:

- name '\*.sik'** Dateinamen mit Wildcards in Anführungszeichen (Shell expansion), **-iname** ohne Groß-/Kleinschreibung
- mtime -3** Datei wurde vor höchstens drei Tagen modifiziert
- cmin +60** Datei wurde vor mindestens 60 Minuten erzeugt
- type d/f/l** Ist ein Verzeichnis/normale Datei/symbolischer Link
- user iagmanu** Gehört iagmanu
- not/-a/-o** Kombination von Spezifikationen: Negation, Und-, Oder-Verknüpfung, Klammerung für Rangfolge möglich.
- exec chmod g+w '{} ' ;'** Führt das Kommando aus, wobei '{}' ersetzt wird durch den gefundenen Dateinamen und ';' das Kommando abschließt.

**tr** anslate übersetzt oder löscht zeichenweise:

**tr** [Optionen] SET1 [SET2]

SET1/SET2 bestehen aus einzelnen Zeichen (abcd), Escapes (\n, \b, \t, \\), Bereichen (a-z), Zeichenklassen (siehe grep) und Wiederholungen ([a\*5] ist aaaaa). Optionen sind -s (squeeze, Wiederholungen löschen) und -d (delete). Jedes Zeichen in SET1 wird ersetzt durch das an der gleichen Position stehende Zeichen in SET2:

**tr a-z A-Z** ersetze Kleinbuchstaben durch Großbuchstaben (Vorsicht: nur bei ASCII, portabler wäre **tr '[:lower:]' '[:upper:]'**)

**tr , .** ersetze Komma durch Punkt (Fließkommazahlen von deutscher in englische Notation)

**tr -s '\n'** ersetze aufeinanderfolgende Zeilenenden durch einzelne (Leerzeilen entfernen)

**tr -d '\015'** lösche die Wagenrückläufe (CR) aus Windows-Texten (CR/LF als Zeilenende)



**sed** (stream editor) liest zeilenweise in einen Puffer ein, arbeitet darauf Kommandos ab (typisch Suchen/Ersetzen) und druckt das Ergebnis:

sed 's/PAT1/PAT2/Flags' ersetzt jedes Vorkommen von PAT1 durch PAT2. An Flags ist vor allem 'g' von Interesse, das global alle Vorkommen von PAT1 in der Zeile ersetzt (ansonsten nur das erste Auftreten). Statt / kann jeweils ein beliebiges Trennzeichen verwendet werden. PAT1/PAT2 sind reguläre Ausdrücke mit leicht anderer Syntax als in grep: \+, \?, \| anstelle von +, ?, |. Wichtige Optionen:

**-i/--in-place** Überschreibt die Originaldatei mit der modifizierten

**-r** extended syntax der regulären Ausdrücke

**-f SCRIPTFILE** Kommandos sind in SCRIPTFILE abgelegt (Beispiel: L<sup>A</sup>T<sub>E</sub>X-Umlaute: 's/\\\"a/ä/g s/\\\"A/Ä/g s/\\\"o/ö/g ...', mit jedem s/././g-Kommando auf einer eigenen Zeile)

awk (Aho, Weinberger, Kernighan) arbeitet ebenfalls zeilenweise, sucht nach Mustern und führt spezifische Aktionen aus, gegebenenfalls auf Spalten. awk-Skripte sind ideal zur (Vor-) Verarbeitung von Rohdaten. Sie bestehen aus einer Reihe von Mustern und zugehörigen Kommandos:

`awk '/foo/ { print $0 }'` äquivalent zu `grep foo`: alle Zeilen die „foo“ enthalten, werden gedruckt

Ab hier nur die awk-Programme

`/foo/ { print $2,$5 }` druckt die zweite und fünfte Spalte aller Zeilen mit „foo“

`$1 ~ /real/ { r=$2 ; getline ; u=$2 ; getline ; s=$2 ; print r, u, s }`  
Falls die erste Spalte „real“ enthält, wird die zweite Spalte dieser, der folgenden und der übernächsten Zeile gedruckt

```
#!/usr/bin/awk -f
# usage: average [-N] [file(s) ...]
# If the first argument is a minus sign, followed by a number, the
# corresponding column is averaged, otherwise the first column.
# File names may follow, otherwise stdin is taken
BEGIN {
    if ( (ARGC>0) && (ARGV[1]<0) ) {
        col=-ARGV[1]
        ARGV[1]=" "
    }
    else col=1
    sum=0 ; lines=0 ; squares=0
}
{ sum=sum+$(col) ; squares=squares+$(col)*$(col) ; ++lines }
END {
    if (lines>0) print sum/lines
    if (lines>1) print "+/-" sqrt((squares-(sum*sum/lines))/(lines-1))
}
```

Zur Automatisierung ist es nützlich, Programme (oder Skripte) zu einem vorher festgelegten Zeitpunkt zu starten. Kommandos werden von der Kommandozeile gelesen, die Zeit als Parameter angegeben. Standardausgabe und -fehler werden per E-Mail verschickt, falls nicht leer.

`echo "ls -lR / > /ls-lR" | at 01:00` Kommende Nacht ein Kompletttlisting anlegen

`echo "mail friend@email.com < birthday.txt" | at noon 15012012`  
Vorher geschriebene Geburtstagsmail während des Urlaubs verschicken

`atq` Noch auszuführende Jobs anzeigen

`atrm 3` Noch nicht angelaufenen Job entfernen

`batch -m me@email.com -f script` Job mit Kommandos in Datei `script` ausführen, sobald Maschine hinreichend leer ( $\text{load} < 0,8$ ), Benachrichtigungsmail nach Ende schicken.

at ist für einmalige Jobs sehr nützlich, bei wiederkehrenden Aufgaben ist cron besser geeignet. Damit können Jobs zu definierten Zeiten immer wieder angestoßen werden: Stündlich, täglich, jeden Ersten im Monat, ... Spezifiziert wird das in einer Tabelle, die mit „crontab -e“ editiert wird (normalerweise vi). Format:

**Minute Stunde Tag Monat Wochentag Kommando** führt das Kommando immer zur angegebenen Zeit aus. Nützlich ist die Angabe „\*“ für „beliebig“, aber auch „/4“ für „jede(r) 4.“ (z.B. Stunde, Monat) oder „6-7“ (bei den Wochentagen für das Wochenende).

`15 */2 * * * /usr/local/bin/checkFilesystemFull` jede zweite Stunde um viertel nach wird das angegebene Skript ausgeführt

`30 04 * * Sun /opt/ws/sbin/ws_cleaner >> /opt/ws/log/ws_cleaner.log 2>&1`  
löscht die abgelaufenen Workspaces auf Prandtl immer Sonntag Nacht um halb fünf.

Sowohl at als auch cron brauchen jeweils einen Daemon (einen Prozess, der kontinuierlich im Hintergrund läuft), der auf manchen Distributionen nicht standardmäßig aktiviert ist. Start mit `/etc/init.d/atd start` beziehungsweise `/etc/init.d/cron start` oder das jeweilige systemspezifische Äquivalent.

Falls nötig, entsprechende Links in `/etc/init.d/rc.5` (oder welcher Runlevel auch immer) eintragen, damit der Start automatisch erfolgt. Sowohl at- als auch cron-Jobs sind benutzerspezifisch. Welcher Nutzer Jobs absetzen darf oder nicht, wird über `at.allow/at.deny` beziehungsweise `cron.allow/cron.deny` im Verzeichnis `/etc` festgelegt.

... | **xargs** **CMD** [**PARAMS...**] Nimmt die Standardeingabe und gibt sie als Argumente an den Befehl **CMD** (gegebenenfalls mit führenden Argumenten) weiter. **xargs -n X** gibt immer nur **X** Argumente weiter und ruft entsprechend häufig auf.

**ls -d -1 old/\* | sed "s%^old/./.%"** | **xargs rm** Zeigt alle Dateien und Verzeichnisse im Verzeichnis **old** an, ersetzt das den Verzeichnisnamen **old/** durch das aktuelle Verzeichnis **./** und löscht dort. Alle Dateien, die es in **old** gibt, werden also im aktuellen Verzeichnis gelöscht, alle die es nur hier gibt, bleiben erhalten.

**touch DATEI** Setzt Datum und Zeit der letzten Änderung von **DATEI** auf jetzt. Falls **DATEI** nicht existiert, wird sie leer erzeugt. Nützlich, um einen Zeitstempel zu dokumentieren:

**touch TIMESTAMP** Hält beispielsweise in einem Skript oder für ein Makefile die aktuelle Zeit fest.

```
find . -iname '*.jpg' -exec chmod g+w '{}' ';'
```

Alle Bilder für Gruppe beschreibbar machen

```
find . -iname '*.jpg' | xargs chmod g+w
```

äquivalente Alternative



```
find . -iname '*.jpg' -exec chmod g+w '{}' ';'
```

Alle Bilder für Gruppe beschreibbar machen

```
find . -iname '*.jpg' | xargs chmod g+w
```

äquivalente Alternative

*ABER*

```
find . -iname '*.jpg' -exec chmod g+w '{}' ';' 
```

Alle Bilder für Gruppe beschreibbar machen

```
find . -iname '*.jpg' | xargs chmod g+w
```

äquivalente Alternative

*ABER*

nur fast äquivalent: Namen mit Leerzeichen, Zeilenumbrüchen, ...

```
find . -iname '*.jpg' -exec chmod g+w '{}' ';'
```

Alle Bilder für Gruppe beschreibbar machen

```
find . -iname '*.jpg' | xargs chmod g+w
```

äquivalente Alternative

*ABER*

nur fast äquivalent: Namen mit Leerzeichen, Zeilenumbrüchen, ...

*KORREKT*

```
find . -iname '*.jpg' -print0 | xargs -0 chmod g+w
```

```
find . -iname '*.jpg' -exec chmod g+w '{}' ';'
```

Alle Bilder für Gruppe beschreibbar machen

```
find . -iname '*.jpg' | xargs chmod g+w
```

äquivalente Alternative

*ABER*

nur fast äquivalent: Namen mit Leerzeichen, Zeilenumbrüchen, ...

*KORREKT*

```
find . -iname '*.jpg' -print0 | xargs -0 chmod g+w
```

oder gegebenenfalls

```
find . -iname '*.jpg' -print0 | xargs -0 -n 1 chmod g+w
```

## 2 Shells

Der große Mehrwert der zahlreichen UNIX-Tools liegt in der praktisch unbegrenzten Zahl von Möglichkeiten, sie miteinander zu verknüpfen. Dazu bedarf es eines Kommandoprozessors, der Shell, die Funktionen zur Abarbeitung der diversen Kommandos zur Verfügung stellt. Am weitesten verbreitet sind

**bash** Bourne-Again-Shell, eine Nachbildung der originalen (und in POSIX spezifizierten) UNIX-Shell von Stephen Bourne mit zahlreichen Erweiterungen

**tcsh** Nachfolger der csh (Berkely UNIX C-Shell, also BSD, mit C-ähnlicher Syntax) mit zahlreichen Erweiterungen

**ash, ksh, sash, zsh** weitere Varianten mit mehr oder minder sinnvollen Zusatzfeatures

Mit allen lassen sich Kommandos verketteten (Listen, Pipelines), Ein-/Ausgaben umlenken, Skripte schreiben, Kontrollstrukturen definieren, die Laufzeitumgebung verändern, nur die Syntax ist unterschiedlich.

Bei der Ausführung einer Kommandozeile arbeitet eine Shell typischerweise folgende Schritte ab:

0. **Kommentar** Beginnt die Zeile mit `#` wird sie als Kommentar ignoriert
1. **Tokenzerlegung** Zeile wird in „Worte“ und Operatoren zerlegt unter Beachtung der Quoting-Regeln (Backslash, einfache und doppelte Anführungszeichen), Aliase werden ersetzt.
2. **Parsing** Worte und Operatoren werden als Kommandos, Pipelines und Listen beziehungsweise Kontrollkommandos (for, while, if, case, ...) interpretiert.
3. **Expansions** Ersetzen verschiedener Konstrukte: geschweifte Klammern, Tilde, Parameter und Variablen, Arithmetik, Kommandos (Anführungszeichen rückwärts), Wortzerlegung, Dateinamen (\*,?,[ ]), zuletzt Entfernen überflüssiger Backslashes und Anführungszeichen
4. **Umleitungen** Ein- und Ausgabe können in Dateien umgeleitet oder direkt aus Skripten umgelenkt werden (Here-Dokument)

**5. Ausführen** Falls ein Kommando übrig blieb, wird es ausgeführt: ohne Slashes als definierte oder eingebaute Shell-Funktion beziehungsweise nach Suche in \$PATH, mit Slashes direkt. Falls nicht asynchron (also ohne &) wird auf Beendigung gewartet und der Rückgabewert bereitgestellt.

Das Kommando wird dann in einer spezifischen Umgebung ausgeführt, die die Shell bereitstellt, und die enthält

**offene Dateien** entsprechend der vorgenommenen Umleitungen

**aktuelles Verzeichnis** und Maske für die Dateierzeugung (umask)

**Shell-Variablen und -Funktionen** soweit als „export“ markiert (oder äquivalent, z.B. setenv in (t)csh)

**Argumente** die nach dem eigentlichen Kommando folgen und beispielsweise an main(argc, argv[ ]) übergeben werden

Etwas weniger bekannte Aspekte sind geschweifte Klammern, Kommandoersetzung und Here-Dokumente:

**PRE{STR1,STR2[,...]}POST** erzeugt Wörter PRESTR1POST PRESTR2POST ... ähnlich Dateinamen, nur dass sie nicht als Dateien existieren müssen. Folgen sind mit Zahlen {N1..N2} und Zeichen {C1..C2} möglich:

**mkdir /usr/local/man/man{1..9}** Erzeugt alle man-Unterverzeichnisse.

**`COMMAND`** Setzt an dieser Stelle der Befehlszeile die Ausgabe von COMMAND ein:

**mv libiag.a `uname`/libiag.a** Verschiebt die Datei ein das der Maschine entsprechende Verzeichnis.

**<<DELIM** Lenkt die Eingabe auf Text aus dem laufenden Skript um bis zum begrenzenden Wort DELIM. Beispiel:

**cat <<DELIM**

Laufender Text, unter Umständen über mehrere Zeilen, kann sogar \$PATH Variablen expandieren, läuft bis zum DELIM



Das eigentliche Kommando kann entweder einfach sein, oder mehrere sind in einer Pipeline (|) oder Liste (&, ;, ||, &&) miteinander verbunden. In einer Pipeline ist die Standardausgabe des vorderen Kommandos verbunden mit der Standardeingabe des nächsten, so dass die Ausgabe gleich weiter verwendet wird:

`ls -l | grep "^d" | grep -v 2009` Zeigt (fast) alle Verzeichnisse an, die nicht 2009 geändert wurden.

Wichtig: In einer Pipeline läuft jedes Kommando in einer eigenen Subshell und kann deswegen die Umgebung nicht verändern:

`cd prog | ls` Zeigt das aktuelle Verzeichnis, nicht das Unterverzeichnis prog!

`;` und `&` Führen die Kommandos nacheinander aus, wobei `&` das erste im Hintergrund ausführt.

`||` und `&&` Bilden die ODER-beziehungsweise UND-Operatoren, wobei ersterer das zweite Kommando nur ausführt, falls das erste einen Nicht-Null Rückgabewert liefert, bei `&&` ist es gerade andersherum.

Übliche Schleifenkonstrukte werden unterstützt:

**until TEST-COMMANDS ; do COMMANDS ; done**

COMMANDS wird ausgeführt, solange TEST-COMMANDS Nicht-Null zurückgibt.

**while TEST-COMMANDS ; do COMMANDS ; done**

COMMANDS wird ausgeführt, solange TEST-COMMANDS Null zurückgibt.

**for NAME [ in WORDS ] ; do COMMANDS ; done**

COMMANDS wird ausgeführt, wobei NAME nacheinander jeder Eintrag in WORDS zugewiesen wird. Falls „in WORDS“ fehlt, werden die Argumente dafür benutzt.

**for (( EXPR1 ; EXPR2 ; EXPR3 )) ; do COMMANDS ; done**

Alternative Form entsprechend der for-Schleife in C: EXPR1 wird vorneweg ausgeführt, danach COMMANDS, solange EXPR2 nicht Null ergibt, nach jedem Durchlauf EXPR3 (was normalerweise den Zähler erhöht).

**break** und **continue** ermöglichen eine weitere Durchlaufkontrolle

Des weiteren gibt es die bedingten Befehle:

```
if TEST-COMMANDS ; then COMMANDS ;  
[ elif TEST-COMMANDS ; then COMMANDS ; ]  
[ else COMMANDS ; ]  
fi
```

außerdem noch die Auswahl nach Mustern

```
case WORD in  
[ ([ PATTERN [ PATTERN ] ... ) COMMAND-LIST ;;]...  
esac
```

sowie die Möglichkeit, ganz einfach interaktive Menüs anzuzeigen  
und abzuarbeiten

```
select NAME [ in WORDS ... ] ; do COMMANDS ; done
```

Die Auswertung von Ausdrücken ist vom Kontext abhängig.

**(( EXPR ))** Auswertung mit üblichen arithmetischen Regeln.

Rückgabewert Null, falls das Ergebnis von 0 verschieden ist, ansonsten 1. Arithmetische Auswertung auch in den drei Ausdrücken der C-Variante der for-Schleife.

**[[ EXPR ]]** Bedingter Ausdruck, ähnlich test oder [ ] für until/while/if/elif.

Ausgewählte Bedingungen (Rückgabewert Null, falls wahr, sonst 1):

**-e FILE** Datei existiert.

**-d DIR** Verzeichnis existiert.

**-f FILE** Reguläre Datei existiert (also kein Verzeichnis oder Special).

**-w FILE** Datei existiert und ist beschreibbar (analog -r, -x).

**STRING == PATTERN** Zeichenkette passt auf Muster  
(entsprechend !=, und =~ für erweiterten regulären Ausdruck)

**! && || ( )** Übliche Verknüpfungen und Vorrangregelung

Kommandos können, wo nötig, gruppiert werden, beispielsweise für gemeinsame Ausgabeumlenkung oder bedingte Ausführung.

( **CMD-LIST** ) Liste von Kommandos, die in einer *Subshell* ausgeführt werden (und deswegen nicht die Umgebung verändern können).

{ **CMD-LIST** ; } Liste von Kommandos, die in der *aktuellen* Shell ausgeführt werden.

```
for DIR in * ; do ( cd $DIR && { echo `pwd`: ; \
    echo ; ls ; echo -e "\n—\n" ; } | tee ../$DIR.list ) ; done
```

Gibt eine Liste der Inhalte aller Unterverzeichnisse aus, mit Verzeichnisname als Überschrift und Trennungsstrichen unten, und erzeugt zusätzlich entsprechende Listen-Dateien.

Funktionen können ebenfalls definiert und auch aufgerufen werden:

`[„function”] NAME () COMPOUND-COMMAND`

COMPOUND-COMMAND ist üblicherweise eine Kommandoliste in geschweiften Klammern, kann aber auch eine Schleife oder sogar Bedingung sein. Übergebene Parameter können via  $\${N}$  oder  $\$N$  benutzt werden. Das gilt für die Funktion wie auch an ein Skript insgesamt.  $\$0$  steht dann für den Namen des Skriptes.

Sind viele Argumente gleichermaßen zu behandeln, bietet sich eine Schleife an, mit shift zum Weiterschieben ( $\$2 \rightarrow \$1$ ,  $\$3 \rightarrow \$2$ , ...) und mit return wird zurückgekehrt.

Spezielle Parameter:  $\$*$  und  $\$@$  repräsentieren alle Parameter ( $\$@$  zum Erhalt aller Wortgrenzen),  $\$\#$  ihre Anzahl.  $\$?$  steht für den Rückgabewert des letzten Kommandos,  $\$\$$  für die Prozessnummer der Shell,  $\$!$  für die des letzten Hintergrundprozesses.

Beispielskript mvto (schiebt alle Dateien ab dem zweiten Parameter in das Verzeichnis des ersten, also quasi ein mv mit dem letzten Argument vorneweg):

```
#!/bin/sh
[ $# -lt 2 ] && echo "Syntax: mvto DEST SRC [SRC...]"
               && exit 2

DEST=$1
shift
mv "$@" $DEST
```

Aufruf

```
mvto 2011-F "Aufgabe 1.pdf" Aufgabe2.pdf A3.pdf
```

ist äquivalent zu

```
mv "Aufgabe 1.pdf" Aufgabe2.pdf A3.pdf 2011-F
```

Variablen werden mit  $\$$  referenziert ( $\{\text{VAR}\}$  oder  $\$VAR$ ) und mit  $=$  oder  $+$  zugewiesen (mit  $+=$  angehängt). Ein  $!$  direkt nach der öffnenden Klammer  $\{$  führt dazu, dass der Inhalt der Variable als Variablenname interpretiert und anschließend dieser expandiert wird (Indirektion). Falls  $VAR$  also  $DIR$  enthält ( $VAR=DIR$ ), ist  $\{\!|\text{VAR}\}$  gleichbedeutend mit  $\{DIR\}$ . Manchmal nützlich sind noch die Ersetzungen:

$\{\text{VAR}:-\text{WORD}\}$  Falls  $VAR$  nicht gesetzt oder leer ist, ist das Ergebnis  $WORD$ , ansonsten der Inhalt von  $VAR$ .

$\{\text{VAR}:+\text{WORD}\}$  Falls  $VAR$  nicht gesetzt oder leer ist, ist das Ergebnis leer, ansonsten der Inhalt von  $WORD$ .

$\{\text{VAR}:=\text{WORD}\}$  Falls  $VAR$  nicht gesetzt oder leer ist, wird  $WORD$  an  $VAR$  zugewiesen. Ergebnis ist der Wert von  $VAR$ .



(Eindimensionale) Arrays von Variablen gibt es auch, der Zugriff findet über 0-basierte Integer ( $ARR[0]$ ,  $ARR[1]$ , ...) statt.  $ARR$  alleine ist äquivalent zu  $ARR[0]$ .

$ARR=(e0\ e1\ e2)$  Zuweisung mehrere Elemente

$ARR+=(eN)$  Anhängen des nächsten Elementes

Ähnlich der speziellen Parameter  $\$*$  und  $\$@$  kann auch auf alle Einträge gemeinsam zugegriffen werden:

$\{ARR[*]}\text{ oder }\{ARR[@]}$  Alle Elemente hintereinander ( $e0\ e1\ e2$ )

$\{ARR[*]}$  Gibt ein langes Wort mit allen Einträgen (" $e0\ e1\ e2$ ").

$\{ARR[@]}$  Der Anzahl entsprechend viele einzelne Worte (" $e0$ " " $e1$ " " $e2$ ").

$\{\#ARR[*]}\text{ Anzahl der Einträge (genauso mit } @)$

Oftmals muss die Sonderbedeutung bestimmter Zeichen unterdrückt werden, beispielsweise, wenn ein Dateiname Leerzeichen enthält, die Shell aber an dieser Stelle dann den Namen in zwei aufspaltet. Es gibt:

- `\` Erhält die wörtliche Bedeutung des nächsten Zeichens, sei es ein Leerzeichen (Wortseparator), ein `$` (Variablenreferenz), ein ``` (Kommandoersetzung) oder einfach `\`.
- `' '` Alles zwischen einfachen Hochkommata wird 1:1 übernommen, ohne jede Interpretation. Das gilt auch für `\`, so dass kein `'` vorkommen kann.
- `" "` Innerhalb von doppelten Hochkommata werden noch `$`, `\` und ``` interpretiert, so dass über `"` auch ein solches Zeichen vorkommen kann. Vorsicht: `$*` und `$@` werden in doppelten Hochkommata speziell behandelt (s.o.).
- `\newline` Ein Backslash gefolgt von einem Zeilenende sorgt dafür, dass beides gelöscht und die logische Zeile damit in die nächste physikalische fortgesetzt wird.

- . SCRIPT** Führt das angegebene Skript in der aktuellen Shell aus. Typisches Beispiel ist `. ~/.profile`. Synonym ist `source`.
- alias NAME=VAL** Definiert eine Kurzform für einen Aufruf. `unalias` dient zum Löschen. Typisch: `alias ll='ls -l'`, `alias cd.='cd ..'`
- eval [ARGS]** Baut die angegebenen Argumente zu einem Kommando zusammen und führt es in der aktuellen Shell aus.
- export VAR** Markiert Shell-Variable zur Übergabe an Kindprozesse (Umgebung).
- read VAR** Liest eine Zeile von der Standardeingabe in die angegebene Variable ein.
- umask MODE** Legt den Standardzugriff neuer Dateien fest, indem die *nicht* zu gewährenden Rechte aufgezählt werden. Typisch ist `umask 022` (keine Schreibrechte für Gruppe und Andere).
- unset VAR** Löscht die angegebene Variable.

### 3 Make

Make automatisiert den mehrschrittigen Zusammenbau von Systemen, typischerweise beim Kompilieren von Software. Die Schritte umfassen

- ① Erzeugen der Quellen (gelegentlich bei Programmgeneratoren, aber auch Präprozessorläufe, sofern nicht im Compiler)
- ② Kompilieren der Quellen
- ③ Erzeugen von Bibliotheken (falls in solche zerlegt)
- ④ Linken der Anwendung
- ⑤ Test
- ⑥ Installation

sind teilweise voneinander abhängig und beim Programmieren möchte man, dass möglichst schnell weiter gearbeitet werden kann, also keine unnötigen Compiler-/Linkeraufrufe.

Abhängigkeiten und Erzeugungsvorschriften werden im Makefile (auch makefile ist ok) beschrieben. Es besteht aus Regeln:

**ZIEL:** Abhängigkeiten ...

<TAB>COMMAND1

<TAB>COMMAND2

Wichtig: Vor COMMAND1,... muss ein echter Tabulator stehen, also *nicht* nur ein paar Leerzeichen!

Ziel und Abhängigkeiten sind Dateinamen, und sobald Ziel älter ist als mindestens eine der Dateien, von denen es abhängt, wird COMMAND1,... ausgeführt, was das Ziel hoffentlich neu erzeugt (z.B. Compileraufruf). Die COMMAND1,...-Zeilen (alles, was mit <TAB> beginnt) sind make selbst völlig egal und werden zur Ausführung 1:1 der Shell weitergereicht, deren Syntax und Semantik dann dafür gelten.

Abhängigkeiten können auch fehlen, beispielsweise wenn das Ziel gar keine zu erzeugende Datei ist:

clean:

```
-rm *.o  
-rm *~
```

make clean löscht dann alle herumliegenden .o-Dateien, denn ein an make übergebenes Argument legt das zu erstellende Ziel fest. Das Minuszeichen vor dem Löschbefehl sorgt dafür, dass die Bearbeitung des Makefiles nicht abbricht, falls rm einen Fehler meldet (beispielsweise keine .o-Dateien mehr zum Löschen vorhanden). Ein automatisierter Testlauf nutzt das gezielt aus und bricht beim ersten Fehler (Ausgabe weicht von Vorlage ab) sofort ab:

check: run

```
run < check1.input | diff - check1.output  
run < check2.input | diff - check2.output
```

Eine Anwendung run besteht aus den drei Quelldateien run.c, input.c, exec.c sowie den Headerdateien defs.h, welche von allen drei Quelldateien eingebunden wird, sowie cmd.h, die nicht von run.c eingebunden wird. Kompiliert wird mit dem Standard-C-Compiler, zusätzlich wird die libz gelinkt. Makefile dafür:

```
run: run.o input.o exec.o
    cc -o run run.o input.o exec.o -lz
```

```
run.o: run.c defs.h
    cc -c run.c
```

```
input.o: input.c defs.h cmd.h
    cc -c input.c
```

```
exec.o: exec.c defs.h cmd.h
    cc -c exec.c
```

```
clean:
    -rm run.o input.o exec.o run
```

make liest zuerst das gesamte Makefile ein und baut einen Abhängigkeitsbaum aus den Regeln auf. Falls kein Ziel angegeben ist, wird das erste (run) genommen und geprüft, ob es neuer ist als alle Abhängigkeiten, und transitiv weiter durch den Baum. Ist irgend eine Datei neuer, werden alle davon abhängigen durch Ausführen der entsprechenden Kommandos neu erzeugt, was zu weiteren Neuerzeugungen führt.

Ist also nur run.c verändert, wird run.o neu erzeugt und dann run. Wurde cmd.h modifiziert, werden input.c und exec.c compiliert und run wird gelinkt.



Wiederholung ist nervig und fehleranfällig. Sinnvollerweise können deshalb Variablen eingesetzt werden:

```
objects=run.o input.o exec.o
```

```
run: $(objects)  
      cc -o run $(objects) -lz
```

```
...
```

```
clean:  
      -rm $(objects) run
```

Variablen werden am Anfang des Makefiles deklariert, eine Nutzung erfolgt mit \$(VAR). Diese Syntax wird von make erfordert, falls Variablen in Kommandos genutzt werden sollen (z.B. HOME), so muss an die Shell ein einzelnes \$-Zeichen übergeben werden, was mit \$\$ passiert, so dass es dann \$\$HOME (oder \${HOME}) heißt.

Variablenreferenzen werden erst bei ihrem Einsatz in einer Regel ausgewertet. Das heißt

```
CFLAGS=$(includes) -O2
```

```
includes=-I /opt/gnu/include
```

wird bei der Verwendung von `$(CFLAGS)` zu

`"-I /opt/gnu/include -O2"` expandiert. Das ist anders als in Programmiersprachen gewohnt, aber manchmal sehr praktisch. Falls man das nicht möchte, benutze man

```
CFLAGS:= $(includes) -O2
```

Dann wird `$(includes)` sofort beim Einlesen des Makefiles ausgewertet, spätere Zuweisungen bleiben unerheblich. Für beide Fälle kann mit

```
CFLAGS+=-O2
```

noch etwas angehängt werden, der Charakter der Variablen (sofortige oder aufgeschobene Auswertung) bleibt dabei erhalten.

Umgebungsvariablen werden 1:1 als Make-Variablen übernommen, können auf der Kommandozeile und/oder im Makefile aber ergänzt und überschrieben werden.

Variablennamen können alle Zeichen enthalten außer : # = und Leerraum (Leerzeichen, Tab). Alles außer Buchstaben, Ziffern und dem Unterstrich ist aber nicht zuverlässig portabel und daher zu vermeiden.

Bei Zuweisungen an Variablen werden Leerzeichen nach dem = entfernt. Das macht es schwierig, ein Leerzeichen gezielt unterzubringen. So geht es doch:

`empty:= #` Gar nix

`space:=$(empty) #` Ein Leerzeichen vor dem Kommentar

`empty` ist dann wirklich völlig leer (aber definiert)! Damit habe ich ein Leerzeichen in einer Variablen und kann das unterbringen, wo es gebraucht wird.

Für viele häufig gebrauchte Regeln gibt es vorgefertigte Standards, die impliziten Regeln (die auch selbst definiert werden können). Beispielsweise können C-Dateien automatisch compiliert werden:

`input.o: defs.h cmd.h`

Falls eine Objektdaten input.o gebraucht wird (beispielsweise zum Linken), wird nach der zugehörigen Quelldatei gesucht und diese mit dem richtigen Compiler übersetzt. Die Abhängigkeit `input.c` → `input.o` ist dabei impliziert, nur die Header müssen deshalb extra angegeben werden.

Da gegen die libz gelinkt werden muss, kann dies in einer (in der impliziten Linkregel genutzten) Standardvariable angegeben werden. Das Makefile wird dann sehr einfach:

```
objects=run.o input.o exec.o
```

```
LDLIBS=-lz
```

```
run: $(objects)
```

```
run.o: defs.h
```

```
input.o: defs.h cmd.h
```

```
exec.o: defs.h cmd.h
```

```
clean:
```

```
    -rm $(objects) run
```

Alternativ können die Abhängigkeiten von den Headern auch umgekehrt angegeben werden:

```
run.o input.o exec.o: defs.h
```

```
input.o exec.o: cmd.h
```

Was man geschickter findet, ist Geschmackssache. Wie oben gezeigt, nutzen implizite Regeln diverse Variablen, die standardmäßig leer oder sinnvoll vorbelegt sind. Beispielsweise zum Kompilieren verschiedener Quelldateien:

```
$(CC) -c $(CFLAGS) $(CPPFLAGS) für .c
```

```
$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) für .cpp, .cc und .C
```

```
$(FC) -c $(FFLAGS) für .f
```

```
$(FC) -c $(FFLAGS) $(CPPFLAGS) für .F
```

CC=cc, CXX=g++, FC=f77, alle anderen Variablen sind leer.

Eingebaute implizite Regeln setzen auf Endungstransformationen auf:  $.c \rightarrow .o$  und ähnliche Regeln können selbst definiert werden.

Beispielsweise zur Transformation von Bildern:

`%.png: %.eps`

`convert $< $@`

% dient dabei als Platzhalter (wie \* in der Shell) und darf im Ziel links nur einmal auftauchen. Bei den Abhängigkeiten wird es entsprechend ersetzt. Die Variable `$<` wird ersetzt durch die erste Abhängigkeit, (`$?` für alle neueren Abhängigkeiten, `$^` für alle ohne Dubletten, `$+` für alle), und `$@` durch das Ziel. `$*` steht für den Stamm, also das, was im Muster für % eingesetzt wurde. Durch Anhängen von D oder F, also `$(<D)` oder `$(@F)`, kann nur der Verzeichnis- beziehungsweise Dateianteil des Namens ausgewählt werden.

make erlaubt relativ umfangreiche Textmanipulationen mit verschiedenen Funktionen. Diese haben die Gestalt

`$(FUNCTION ARGS)` und formen ARGS entsprechend der gewählten FUNCTION um.

Falls mehrere Argumente erforderlich sind, werden diese durch Kommata getrennt. Wesentliche Beispiele:

`$(subst FROM,TO,TEXT)` Ersetzt in TEXT (häufig eine Variablenreferenz) das Auftreten von FROM durch TO.

`$(patsubst PATTERN,TO,TEXT)` Analog für das Muster PATTERN, das % als Platzhalter (im Sinne des Shell-\*) enthalten kann, und das dann auch in TO ersetzt wird. Typisch: `$(patsubst %.o,%.c,$(objects))`. Äquivalent dazu sind

`$(objects:%.o=%.c)` und für diesen häufig auftretenden Fall sogar noch kürzer:

`$(objects:.o=.c)`



`$(strip TEXT)` Entfernt Leerzeichen und Tabs am Anfang und Ende von TEXT.

`$(filter PATTERNS...,TEXT)` Liefert die Worte in TEXT zurück, die dem Muster oder den Mustern entsprechen, wieder mit % als Platzhalter. Beispiel:

`c_sources=$(filter %.c,$(sources))` Wählt aus einer Menge von .c und .f-Quelldateien nur die .c-Dateien aus.

`$(sort LIST)` Sortiert und entfernt Duplikate.

`$(basename NAMES...)` Entfernt das Suffix der angegebenen Dateinamen.

`$(addsuffix SUFFIX,NAMES...)` Hängt ein neues Suffix an.

`$(addprefix PREFIX,NAMES...)` Hängt ein neues Prefix davor, beispielsweise bestehend aus einem Verzeichnis.

`$(wildcard PATTERN)` Expandiert das Muster mit Shell-Platzhaltern auf die verfügbaren Dateinamen.

`$(shell COMMAND...)` Ruft die Shell auf, Ergebnis ist die Standardausgabe von COMMAND.

Ziele, die eigentlich keine Dateien sind, können als solche markiert werden:

**.PHONY clean check** Falls (versehentlich) eine Datei clean oder check existiert, wird die Aktion trotzdem auf jeden Fall ausgeführt, auch wenn die Datei ganz neu ist.

**include Datei** Setzt an dieser Stelle die genannte Datei ein. Das können auch mehrere sein (mit Shell-Platzhaltern), auch durch Variablen definiert. Besonders geschickt für automatisch generierte Abhängigkeitslisten:

**gcc -M input.c > input.deps** Erzeugt eine Datei input.deps des Inhalts input.o : input.c deps.h cmd.h (und vermutlich viele andere wie /usr/include/stdio.h etc.), die mit include eingebunden werden kann.

**\newline** Ein Backslash unmittelbar vor dem Zeilenende sorgt dafür, dass beides entfernt und die logische Zeile mit der nächsten physikalischen fortgesetzt wird, genau wie in der Shell.

**-COMMAND** Wie im clean-Beispiel benutzt, wird ein eventuell auftretender Fehler ignoriert und die Bearbeitung des Makefiles fortgesetzt

**@COMMAND** Normalerweise werden alle ausgeführten Befehle ausgedruckt, mit vorangestelltem @ unterbleibt das.

Ganz wichtig: Alle Kommandos zu einer Regel werden in ihrer eigenen Shell abgearbeitet, es kann also keine Information von einem zum nächsten Kommando weitergegeben werden. Das schließt beispielsweise die Verwendung von `cd` und Variablen aus. Lösung: Nur ein einzeliges Kommando, mit entsprechend vielen `\newline` zur besseren Eingabe und Darstellung.

Komplexere Programme sind oft in mehreren Unterverzeichnissen untergebracht, die separat kompiliert (unter Umständen zu Bibliotheken gepackt) und zuletzt zusammengelinkt werden. Oft ist es sinnvoll, in den Unterverzeichnissen eigene separate Makefiles zu haben, die unabhängig voneinander laufen können. Das Makefile im Hauptverzeichnis ruft dann die Makefiles der Unterverzeichnisse auf:

```
subdirs=edit file gui output
```

```
LDLIBS=$(addprefix -l,$(subdirs))
```

```
LDFLAGS=$(addprefix -L ,$(subdirs))
```

```
run: run.o $(subdirs)
```

```
$(CC) $(LDFLAGS) -o $@ $< $(LDLIBS)
```

```
$(subdirs):
```

```
$(MAKE) -C $@
```

Für jedes Unterverzeichnis in `$(subdirs)` wird eine eigene Regel erzeugt, in der einfach `make` für das jeweilige Verzeichnis aufgerufen wird. Die Option `-C DIR` sorgt dafür, dass zuvor in das Verzeichnis `DIR` gewechselt wird.

Bei größeren Softwaresystemen dauert eine einzelne Übersetzung oft relativ lange, so dass auch dabei eine parallele Ausführung wünschenswert wäre. Da im Makefile die Abhängigkeiten der einzelnen Übersetzungsstufen spezifiziert sind, kann make anhand des Abhängigkeitsbaumes feststellen, welche Aufrufe von Präprozessoren, Compilern, Bibliotheksgeneratoren und Linkern unabhängig voneinander sind und parallel ausgeführt werden können. Mit

`make -j [N] [-l LOAD]` wird make aufgefordert, maximal N (ohne: unbegrenzt) viele Aktionen parallel zu bearbeiten, solange die aktuelle Last des Systems unter LOAD bleibt.

Vorsicht: Nur einer der parallelen Prozesse bekommt eine gültige Standardeingabe (falls das überhaupt sinnvoll ist), und die Ausgaben (Warnungen, Fehlermeldungen) der verschiedenen Prozesse erscheinen durcheinander. Sobald ein Prozess mit (nicht ignoriertem) Fehler abbricht, werden die noch laufenden bis zum Ende ausgeführt und danach gestoppt.

Wichtige Optionen:

**TARGETS...** Spezifisch das angegebene Ziel (eventuell mehrere) erstellen anstelle des ersten auftretenden. Vor allem für virtuelle Ziele verwendet, wie clean, check, install und dist.

**VAR=VALUE** Variablenzuweisung als Argument überschreibt die im Makefile und auch in der Umgebung, beispielsweise make CC=icc.

**-f MAKEFILE** Alternative Datei anstelle von Makefile und makefile.

**-n** Nicht ausführen, sondern Kommandos nur ausgeben (Probelauf).

**-q** Abfrage, ob das angegebene Ziel aktuell ist.

**-B** Ziel wird vollständig neu erstellt, inklusive aller Abhängigkeiten  
Rückgabewert ist 0, falls make erfolgreich war, 1, falls das Ziel nicht aktuell war (Option -q) und 2, falls ein Fehler aufgetreten ist.

## 4 GNU Autotools

Makefiles zu schreiben ist manchmal mühsam, insbesondere auch Standardziele wie clean und check. Glücklicherweise gibt es einen Satz an Werkzeugen, der einen hier gut unterstützt. Die GNU autotools bestehen dabei aus

- ① autoscan, aclocal, autoheader
- ② autoconf
- ③ automake

Ausgehend von einer Beschreibungsdatei `configure.ac` für das Projekt und einer Schablone `Makefile.am` für jedes Verzeichnis wird ein Shell-Skript `configure` gebaut, das dann bei Aufruf Systemspezifisches abprüft und mit dem Resultat ein Makefile für jedes Verzeichnis generiert.

Es beginnt mit der Projektbeschreibung. Diese sieht im einfachsten Fall so aus (und wird bei Bedarf auch von autoscan generiert):

```
# Primitive configure.ac
dnl Process this file with autoconf to produce a configure script
AC_INIT([niftyapp], [0.1])
AC_CONFIG_AUX_DIR([admin])
# Checking host/target/build systems, for make, install etc.
AC_CANONICAL_SYSTEM
AM_INIT_AUTOMAKE
AC_PREFIX_DEFAULT([/usr/local])
# AC_CONFIG_HEADERS([config.h])

AC_CONFIG_FILES([Makefile edit/Makefile file/Makefile gui/Makefile \
    output/Makefile docs/Makefile docs/pdf/Makefile])
# check for compilers and library handling
AC_PROG_CC
AC_PROG_CXX
AC_PROG_F77
AC_PROG_RANLIB
AC_OUTPUT
```



Im Projektverzeichnis:

```
SUBDIRS=edit file gui output docs .  
EXTRA_DIST=niftyapp.kdevelop  
AUTOMAKE_OPTIONS=foreign  
bin_PROGRAMS=niftyapp  
niftyapp_SOURCES=main.cpp  
niftyapp_LDADD=-lz
```

Im Verzeichnis edit:

```
noinst_LIBRARIES=libedit.a  
libedit_a_SOURCES = mouse.cpp buffer.cpp wrap.c  
AM_CPPFLAGS=-DFILE64  
INCLUDES = $(all_includes) -I $(top_srcdir)
```

Die GNU Autotools nutzen den m4-Makroprozessor mit einer etwas eigenwilligen Syntax. Quoting geschieht mit eckigen Klammern [...], die in vielen (aber nicht allen) Fällen auch weggelassen werden können. Zu Beginn generiert autoscan ein Gerüst für configure.ac, indem alle Unterverzeichnisse abgesucht und entsprechende Informationen eingefügt werden. Zahllose Tests können eingebaut werden (auf Systemheader, Bibliotheken, Pfade zu externen Werkzeugen und Komponenten, Compiler und ihre Features/Fehler). Anhand der Tests in configure.ac generiert autoheader bei Bedarf eine Schablone config.h.in, in die die Testergebnisse eingebaut werden, um einen Header config.h zu erzeugen, der eine Menge von C-#define's enthält, die zur Übersetzungszeit abgefragt werden können (beispielsweise mit #ifdef HAVE\_STDLIB\_H). Außerdem wird mit aclocal eine Sammlung von dafür notwendigen m4-Makros abgelegt.

Falls unterschiedliche Varianten gebaut werden können (beispielsweise eine serielle und eine parallele Version) oder externe Funktionalität optional ist, werden entsprechende Optionen in `configure.ac` aufgenommen, die dann als (dokumentierte!) Parameter für das `configure`-Skript generiert werden. Die Parameterwahl schlägt sich zum einen in `config.h` nieder, aber auch in spezifischen Ersetzungen für die Makefiles.

Im Projektverzeichnis und jedem Unterverzeichnis wird dann nämlich ein `Makefile.am` angelegt, das die wesentlichen Informationen enthält: Welche Quelldateien gibt es, und zu welcher Bibliothek oder Applikation gehören sie. Nach Bedarf können auch Compilerflags wie Include-Verzeichnisse oder Makrodefinitionen zugefügt werden, so dass die Schablone stückweise spezifischer angepasst wird.

Mit `autoconf` wird nun aus `configure.ac` ein Shellskript `configure` generiert. Des weiteren wird mit `automake` aus den verzeichnisspezifischen `Makefile.am`-Dateien jeweils ein `Makefile.in` erzeugt, das viele Standardziele und -funktionen enthält.

Durch Ausführen des configure-Skriptes (das ebenso wie die Makefile.in-Dateien mit der Anwendung verteilt wird) entstehen die eigentlichen Makefiles. Vorteil: Der Anwender braucht nur die ohnehin vorhandenen Werkzeuge Shell und make zum Übersetzen (also den Dreiklang `./configure ; make ; sudo make install`), nicht die ganzen autotools (solange er nichts an den `configure.ac` und `Makefile.am` ändert), trotzdem sind systemspezifische Belange berücksichtigt und Anwendungsoptionen (beispielsweise seriell oder parallel) bleiben wählbar.

Sehr praktisch ist, dass im Makefile Regeln erzeugt werden, die dafür sorgen, dass bei einer Änderung von `configure.ac` oder `Makefile.am` die entsprechenden Werkzeuge (`autoconf`, `automake`, `configure`) wieder automatisch ausgeführt werden, so dass während der Entwicklung einfach immer nur `make` aufgerufen wird.

Für typische Portierungsprobleme gibt es jede Menge vorgefertigte Tests:

**AC\_PROG\_MKDIR\_P** Setzt MKDIR\_P auf ein Programm, das sicherstellt, dass ein Verzeichnis einschließlich eventuell notwendiger Elternverzeichnisse existiert oder erstellt wird.

**AC\_FUNC\_ALLOCA** Versucht, die Funktion alloca verfügbar zu machen.

**AC\_FUNC\_STRTOLD** Prüft, ob strtold vorhanden und voll funktionstüchtig ist.

**AC\_HEADER\_STDBOOL** Falls stdbool.h existiert und volle C99-Funktionalität bereitstellt, wird HAVE\_STDBOOL\_H definiert, falls \_Bool vorhanden ist, auch HAVE\_\_BOOL

**AC\_STRUCT\_TIMEZONE** Stellt fest, wie die Zeitzone zu ermitteln ist (entweder über struct tm.tm\_zone oder tzname[ ]).

Zahlreiche andere Tests sind mittlerweile überflüssig, da 23 Jahre nach der ersten POSIX-Standardisierung die meisten Systeme entweder verschieden (IRIX, Tru64, HP-UX) oder weitestgehend konform sind.

Ein einfaches Beispiel für einen Debug-Modus, der im Code auf NaNs prüft:

```
# check for --with-nan
AC_MSG_CHECKING([for nan])
AC_ARG_WITH(nan,
    [AC_HELP_STRING([--with-nan],
        [enable debug support with NaNs])],
    [ac_with_nan="yes"],
    [ac_with_nan="no"]
)
AC_MSG_RESULT($ac_with_nan)
if test "x$ac_with_nan" = "xyes"; then
    AC_DEFINE(DEBUG_NAN, 1, [Define to 1 if you want debug
        support with NaNs, prefer --with-nan in configure])
fi
```

Anwendung im Code:

```
#ifdef DEBUG_NAN
if (std::isnan(rho))
    IAG_ERROR("NaN in density, computed from p=" << p);
#endif
```

Es existieren Makros, mit denen spezifische Code-Sequenzen probenhalber übersetzt oder Bibliotheken auf Existenz und Verfügbarkeit bestimmter Funktionen getestet werden können:

```
# check for FP exception availability
AC_MSG_CHECKING([for FP exception handling])
AC_COMPILE_IFELSE([[#include <fenv.h>]], [
    AC_MSG_RESULT([yes])
    AC_DEFINE([HAVE_FENV_H], 1, [Define to 1 if fenv.h
        is available])
    AC_CHECK_LIB([m], [feenableexcept],
        AC_DEFINE([HAVE_FEENABLEEXCEPT], 1, [Define to 1 if
            feenableexcept is available in fenv.h])
    ]
],
AC_MSG_RESULT([no])
)
```

So kann nach Werkzeugen gesucht werden, die nicht zwingend vorhanden sein müssen und notfalls durch Alternativen ersetzt werden können:

```
AC_MSG_NOTICE([looking for numerical comparison tool])
AC_PATH_PROG([NCMP], [ncmp], [diff], \
  [$PATH:$LIB_DIR_iaglib/src/util])
if test "$NCMP" = "diff" ; then
  AC_MSG_NOTICE([Sorry, not found, defaulting to diff])
else
  NCMP="$NCMP -b -a 2e-7 -r 1e-5"
fi
```

Verwendung dann im Makefile beispielsweise mit

```
%.result: %.now %.ref
  if $(NCMP) $(NCMPFLAGS) $*.ref $*.now ; then \
    $(ECHO) success > $@ ;\
  else \
    $(ECHO) failure > $@ ;\
  fi
```



Eigene Funktionen können definiert werden, beispielsweise zur Spezifikation externer Komponenten:

```
dn1 check for external library, usage:
dn1 IAG_CHECK_EXTERNAL_LIB(library name, header, library, default
AC_DEFUN([IAG_CHECK_EXTERNAL_LIB], [
  AC_MSG_CHECKING(for $1 directory)
  AC_ARG_WITH($1-dir,
    [AC_HELP_STRING([--with-$1-dir=DIR],
      [directory of $1 library (default is $4) ])],
    [ac_with_$1=$withval],
    [ac_with_$1=$4]
  )
  ac_with_$1='(cd ${ac_with_$1} ; pwd)'
  AC_CHECK_FILE([${ac_with_$1}/$2],
    AC_CHECK_FILE([${ac_with_$1}/$3],
      [ac_cv_have_$1="yes"],
      [ac_cv_have_$1="no"]
    ),
    [ac_cv_have_$1="no"]
  )
])
```

## Fortsetzung der Funktionsdefinition:

```
AC_MSG_RESULT($ac_cv_have_$1)
if test "x${ac_cv_have_$1}" = "xno"; then
  AC_MSG_ERROR([$1 library not found])
fi
AC_SUBST(LIB_DIR_$1, ${ac_with_$1})
])
```

Die Anwendung in `configure.ac` ist dann recht einfach:

```
IAG_CHECK_EXTERNAL_LIB([cgnspp], [cgnspp/cgns++.h],
  [cgnspp/libcgnspp.a], [$srcdir/./cgnspp])
```

und gegebenenfalls kann ein Alternativpfad konfiguriert werden:

```
./configure --with-cgnspp-dir=/opt/cgnspp-0.9.6
```

Benutzt wird das Ergebnis dann folgendermaßen im `Makefile.am`:

```
AM_CPPFLAGS=-I$(top_srcdir) -I @LIB_DIR_cgnspp@/cgnspp
niftyapp_LDFLAGS = -L @LIB_DIR_cgnspp@/cgnspp
niftyapp_LDADD = -lcgnspp
```

Testprogramme können ebenfalls erzeugt werden:

```
check_PROGRAMS=edittest  
edittests_SOURCES=edittests.cpp  
edittests_LDADD=-ledit -lunit  
TESTS=edittests testscript.sh
```

Damit wird ein Ziel check erzeugt, bei dessen Aufruf zunächst das Programm edittests gebaut wird (falls nicht aktuell) und dann ausgeführt, gefolgt von testscript.sh. Beide müssen einen Rückgabewert von 0 liefern, um Erfolg zu signalisieren.

Außer den GNU autotools gibt es noch andere Werkzeuge, um Software zu bauen. Relativ bekannt und verbreitet ist beispielsweise CMake, das eine einzelne Beschreibungsdatei namens CMakeLists.txt nutzt – und gegebenenfalls weitere in Unterverzeichnissen. Mit dem gleichen Beispiel wie zuvor, nur diesmal direkt aus dem Hauptverzeichnis erstellt (in einzelnen Bibliotheken pro Verzeichnis ist natürlich auch möglich) sieht das dann so aus:

```
cmake_minimum_required(VERSION 2.6)
project(niftyapp)
# Allow for building a package
set(CPACK_PACKAGE_VERSION "2.71828")
set(CPACK_PACKAGE_INSTALL_DIRECTORY "niftyapp")
set(CPACK_GENERATOR "STGZ;TGZ;ZIP")
set(CPACK_SOURCE_GENERATOR "STGZ;TGZ;ZIP")
include(CPack)
```

Nach dem relativ unspezifischen Prolog, der weitestgehend Standard ist, geht es weiter

```
enable_language("Fortran")
include_directories(.)
set(niftyapp_FILES
    output/ascii.cpp output/binary.cpp
    file/write.f file/read.f
    gui/graph.c gui/window.cpp
    edit/wrap.c edit/mouse.cpp edit/buffer.cpp
    main.cpp)
add_executable(niftyapp ${niftyapp_FILES} )
install(TARGETS niftyapp RUNTIME DESTINATION bin)
```

`cmake .` erzeugt dann ein Makefile, das normal verwendet werden kann und auch Regel zu seiner eigenen Regeneration enthält (falls sich CMakeLists.txt geändert hat).

Außer Makefiles kann `cmake` aber auch Projektdateien für KDevelop, Eclipse CDT, Xcode und sogar Visual Studio hervorbringen. Ansonsten ist die Funktionalität vergleichbar mit den autotools, die Syntax der Beschreibung – gerade für Anfänger – vielleicht etwas klarer als die m4-bedingt etwas kryptischen Kommandos von `configure.ac`.

Außer den GNU Autotools und CMake gibt es noch – teilweise sprachspezifische – alternative Buildsysteme:

**SCons** Python-basiert, Konfigurationsdateien sind Python-Skripte, make-Ersatz.

**qmake** (Nachfolger von tmake) Bestandteil der Qt-Bibliothek von Trolltech, erzeugt Makefiles

**MPC** (Make, Project and Workspace Creator) Perl-Skripte, unterstützt neben make auch Visual Studio.

**Ant** Das Standard-Buildsystem für (und in) Java.

**Maven** Konkurrenz zu Ant.

Praktisch alle integrierte Entwicklungsumgebungen (IDEs, z.B. KDevelop, Eclipse, Netbeans, Visual Studio, XCode) haben eigene integrierte Buildsysteme. Teilweise können Tools deren Dateien generieren (cmake, MPC).

## 5 Versionsmanagement

Kontinuierlich erfolgreiche Entwicklung erfordert

- Verfolgbarkeit von Änderungen
- Reproduzierbarkeit
- Dokumentation zu Änderungen
- Konfliktmanagement bei mehreren Beteiligten

Das leisten Versionsmanagementsysteme. Mehr oder minder weit verbreitet sind

- Zentrale Systeme: CVS, Subversion (GCC)
- Verteilte Systeme: Git (Linux Kernel, X), Mercurial (Mozilla, OpenOffice, Python)
- Flexible Systeme: Bazaar (Ubuntu, MySQL)



Subversion als Nachfolger von CVS behebt einige von dessen Schwächen:

- Umbenennungen erhalten die Historie.
- Binärdateien bleiben grundsätzlich identisch.
- Versionsnummern sind global.
- Branches und Tags sind sehr schnell.

Grundlegende Konzepte finden sich ähnlich in allen Systemen zur Versionskontrolle.

Ein Subversion-Repository erscheint für den Benutzer als (über Netzwerk zugängliches) Dateisystem mit

- Gedächtnis/Historie (alte Fassungen bleiben erhalten)
- Metadaten (Informationen zusätzlich zum Inhalt)

jeweils einzeln für Dateien, aber auch für den gesamten Baum.

- Das Repository enthält den versionierten Entwicklungsbaum, direkt im Dateisystem (CVS) oder als Datenbank (Subversion).
- Erst beim Einstellen (commit) wird der Zustand festgehalten, dazu ist eine kurze Beschreibung anzugeben.
- Veränderte Dateien überschreiben ältere, die aber rekonstruiert werden können.
- Zeitpunkt und Autor jeder Änderung ist ermittelbar.
- Unabhängige Änderungen verschiedener Entwickler werden automatisch zusammengeführt, Konflikte zur manuellen Bereinigung gemeldet.
- Spezifische Fassungen können gesondert festgehalten werden (als Tags), zum Beispiel zur Auslieferung.
- Zum Ausprobieren können Seitenpfade der Entwicklung eröffnet (Branches) und auch wieder zusammengeführt werden.

- Beschreibung und Autor werden für jeden Commit gespeichert.
- Jede Zeile ist einem Autor zuordenbar.
- Änderungen sind an jeder Zeile nachvollziehbar.
- Jeder Zustand zu einer bestimmten Zeit ist reproduzierbar.
- Für Fälle, die mal funktionierten und jetzt versagen, kann die genaue Fehlerquelle eingegrenzt werden: Quellcodemodifikation, Autor, Nachricht.

## Vorbereitung:

- Falls nötig, zuerst Repository auf Server anlegen (svnadmin create /pfad/zum/repository),
- danach eine erste Fassung einstellen (svn import projectdir file:///pfad/zum/repository/trunk/projectname)
- und an passender Stelle lokal wieder holen (svn checkout file:///pfad/zum/repository/trunk/projectname).

## Wiederkehrender Entwicklungszyklus:

- Aktualisieren (svn update),
- editieren und ergänzen (svn add/copy/remove/move) nach Bedarf,
- Änderungen lokal testen!
- Aktualisieren (svn update), gegebenenfalls Konflikte manuell beheben,
- dabei Unterschiede zur lokalen Version bewerten (svn diff), und
- einstellen (svn commit).

Nicht-lokales Repository ist zugänglich (je nach Installation) über  
[ssh-Zugang](#)

svn+ssh://svn.iag.uni-stuttgart.de/srv/svn/heli/sunwint/trunk

[Web-Server](#) svn://svn.iag.uni-stuttgart.de/srv/svn/heli/sunwint/trunk

[Apache-Modul](#) http://svn.iag.uni-stuttgart.de/srv/svn/heli/sunwint/trunk

Je nach Authentifizierungsmethode können Benutzergruppen unterschiedliche Rechte auf unterschiedlichen Zweigen eingeräumt werden.

- svn status** Gibt den momentanen Zustand der Arbeitskopie an. Zugefügt, gelöscht, kopiert, modifiziert, gelockt und unversioniert sind mögliche Attribute.
- svn info** Umfangreichere Informationen über einzelne Datei oder Verzeichnis, einschließlich Repository, Pfad, Version, letzte Änderung, gegebenenfalls Infos zu Locks.
- svn diff** Zeigt Unterschiede der Arbeitskopie zur Ausgangsversion. Mit `-r HEAD|BASE|42` wird verglichen zur aktuellsten, Ausgangs- (Default), Version 42, mit `-rBASE:HEAD` können Unterschiede zwischen verschiedenen Repository-Versionen festgestellt werden.
- svn revert DATEI** Setzt die Datei auf den letzten Zustand ohne lokale Modifikationen zurück. Mit `--recursive` auch für Unterverzeichnisse (hier ausnahmsweise nicht standardmäßig).

Beim gemeinsamen Entwickeln entstehen gelegentlich Konflikte: beide modifizieren die gleiche Datei, möglicherweise sogar die gleiche Stelle (und sei es nur, eine alte Funktion zu entfernen). Subversion kann in den meisten Fällen automatisch auflösen.

- Unterschiedliche Dateien: problemlos
- Unterschiedliche Stellen in einer Datei: automatisch, solange mime-type etwas Text-artiges enthält (Binärdateien besser mit Locks)
- Gleiche Stelle in einer Datei: automatisch, falls identische Änderung (beide gelöscht, Namen geändert, ...)
- Ansonsten: Konflikt wird gemeldet, Benutzer muss manuell auflösen:
  - Lokale oder Repository-Version verwenden
  - Direkt editieren
  - Später editieren: Zwei Dateien mit .rOLD und .rREP werden angelegt, lokale Kopie enthält Marker: <<<<<<, ===== und >>>>>>

- Komplette Bäume werden versioniert.
- Versionsnummern sind Repository-weit.
- Revisionshistorie einer einzelnen Datei ist in der Regel sehr lückenhaft.
- Oft sind nur Teilbäume von Interesse (trunk, branch, tag, vendor)
- Separate Projekte – separate Repositories
- Commits sind atomar.
- Kopien sind billig.
- SVN-spezifische Informationen werden in .svn-Verzeichnissen gespeichert, unter anderem der Ort des Repositorys und die Position darin, aber auch die letzte Repository-Version jeder Datei. Damit ist `svn diff` lokal und `svn update/commit` benötigen keine URL-Angabe!
- Eigenschaften (properties) können zu jeder Datei archiviert werden und enthalten zusätzliche Information: standardmäßig zum Beispiel `svn:executable`, `svn:mime-type`, `svn:eol-style`, aber auch beliebige andere (notfalls sogar einen `core-dump`).



**Tag** Benannte Kennzeichnung eines spezifischen Zustandes zu einer bestimmten Zeit, beispielsweise ECD\_20110526 für eine ausgelieferte Version, die reproduzierbar sein muss. Wird nicht weiter verändert.

**Branch** Verzweigung an einer bestimmten zeitlichen Position, um separierte Änderungen vorzunehmen. Weiterentwicklung auf dem Branch, Änderungen auf Branch und Trunk können in beiden Richtungen übernommen werden (merge), auch partiell.

In beiden Fällen wird einfach eine Kopie angelegt:

```
svn copy file:///srv/svn/effprog/trunk \  
file:///srv/svn/effprog/tags/ECD_20110525 \  
-m "Tagging ECD delivered version 25.5.2011"  
svn copy file:///srv/svn/effprog/trunk \  
file:///srv/svn/effprog/branches/iagmanu/restruct \  
-m "Branching for major restructuring work"
```

SVN unterscheidet nicht zwischen Tags und Branches – beides sind einfach Kopien. Nur die Konvention der weiteren Nutzung ist unterschiedlich: An Tags wird nichts mehr verändert (sie sind nur benannte Versionen), Branches erfahren Modifikationen und unter Umständen Zusammenführungen (Merges) mit anderen Branches und dem Hauptzweig (trunk). Arbeiten auf dem Branch:

```
svn checkout file:///srv/svn/effprog/branches/iagmanu/restruct \
    my-restructuring
```

Modifikationen...

```
svn commit -m "Branch changes"
```

Einpflegen der Fixes auf trunk:

```
svn merge ^/trunk
```

Ausführlich testen...

```
svn commit -m "Trunk fixes merged"
```

Rückintegration der Branch-Modifikationen in eine (saubere) Arbeitskopie von trunk:

```
svn merge --reintegrate ^/branches/iagmanu/restruct  
svn commit -m "Branch reintegrated"
```

Eventuell Branch löschen, falls nicht mehr benötigt (bleibt im Repository, wird aber unsichtbar!):

```
svn delete ^/branches/iagmanu/restruct -m "Branch deleted"
```

Branch der Arbeitskopie wechseln (hier von unserem Branch zum trunk)

```
svn switch ^/trunk
```

Cherrypicking (nur einzelne Fixes vom trunk oder einem anderen Branch mergen, hier die Modifikationen von 41 nach 42):

```
svn merge -c 42 ^/trunk
```

Bei späteren Komplettermerges (svn merge ^/trunk) wird dieser Fix dann ausgelassen und nicht noch ein zweites Mal eingefügt.

# Externe Entwicklung importieren

Häufig wird eine Bibliothek (oder auch die ganze Anwendung, wie FLOWer) extern gepflegt, lokal importiert und weiter entwickelt. Damit müssen externe Updates sinnvoll integriert werden können. Man startet von der internen Ausgangsversion

```
svn import app-ext-1 file:///srv/svn/effprog/ext/current \
-m "EXT V1.0"
svn copy file:///srv/svn/effprog/ext/current \
file:///srv/svn/effprog/ext/1.0 \
-m "Tagging EXT V1.0"
svn copy file:///srv/svn/effprog/ext/1.0 \
file:///srv/svn/effprog/trunk \
-m "Bring EXT 1.0 to trunk"
svn checkout file:///srv/svn/effprog/trunk niftyapp
```

Lokale Änderungen vornehmen und schön brav immer wieder einpflegen:

```
svn commit -m "IAG changes"
```

Sobald eine neue Version der externen Bibliothek/Anwendung verfügbar wird, zunächst die letzte Fassung auschecken:

```
svn checkout file:///srv/svn/effprog/ext/current ext
```

Neue Version darüber auspacken (tar xjvf nifty-ext-2.0.tar.bz2) und

```
svn add/delete/move
```

bis Vergleich erfolgreich (diff -r ext nifty-ext-2.0)

```
svn commit -m "EXT V2.0"
```

```
svn copy file:///srv/svn/effprog/ext/current \  
file:///srv/svn/effprog/ext/2.0 -m "Tagging EXT V2.0"
```

Im Verzeichnis der aktuellen IAG-Version:

```
svn merge ^/ext/1.0 ^/ext/current .
```

Konflikte gegebenenfalls lösen (svn resolve --accept working) und wieder einstellen

```
svn commit -m "EXT V2.0 merged"
```

Binärdateien können nicht automatisch zusammengeführt werden.  
Lösung: Locks, also die Sperrung spezifischer Dateien.

```
svn lock blade.cgns -m "Manuel schraubt am Gitter"
```

Lock ist nur erfolgreich, wenn Version aktuell ist und niemand sonst gelockt hat! Nur der Eigentümer kann dann einstellen (und gibt dabei standardmäßig den Lock ab). mit `svn unlock blade.cgns` kann er auch explizit aufgegeben werden. Andere können mit `svn status` den Lock erkennen und mit `svn info` den Eigentümer feststellen. Der Admin kann den Lock zerstören:

```
svnadmin rmlocks /srv/svn/effprog /grids/blade.cgns
```

aber auch ein Kollege, beispielsweise bei einem vergessenen Lock vor dem vierwöchigen Urlaub:

```
svn unlock --force file:///srv/svn/effprog/grids/blade.cgns
```

Mit Properties (`svn propset svn:needs-lock '' blade.cgns`) kann Locking erzwungen werden (ansonsten read only)

Bei Bedarf können Skripte vor und nach manchen Operationen ausgeführt werden, um ihre Zulässigkeit zu prüfen (vorher) und Aktionen in der Folge auszulösen (nachher), wobei auf der Kommandozeile weitere Informationen übergeben werden (beispielsweise Pfad und Benutzername). Ein Rückgabewert ungleich 0 führt eventuell zum Abbruch der Operation (bei start-commit oder pre-XXX) und gibt die Standardausgabe des Skriptes zurück an den aufrufenden Rechner.

**start-commit** Wird aufgerufen, bevor der Commit erzeugt wird – benutzt beispielsweise zur Autorisierung

**pre-commit** Aufruf, bevor der Commit eingebaut wird – beispielsweise Überprüfung von Standards (Changelog, Hinweise, Kommentare, ...)

**post-commit** Nachricht, dass (und wo) ein Commit durchgeführt wurde

**pre-lock/post-lock** Überprüfung ob eine Datei gesperrt werden darf/  
Benachrichtigung, dass die Datei gesperrt wurde

**pre-unlock/post-unlock** Dito beim Aufheben der Sperre

- Erstmal einfach anfangen (ein Repository, nur trunk, keine Properties), dann langsam steigern nach Bedarf.
- Häufig einpflegen, aber nur brauchbare Versionen (mindestens kompilierbar, in der Regel getestet), vor allem auf den Trunk!  
⇒ Tests müssen sehr einfach und schnell sein!
- Verteilte Systeme haben ausgefuchste Merge-Algorithmen, die noch besser funktionieren, aber nur, wenn die einzelnen Commits relativ klein sind.

**Fazit:** Versionskontrolle erleichtert Manches, aber Disziplin und Kommunikation sind beim Programmieren unverzichtbar.



## 6 Testen

Fehler beim Programmieren sind unvermeidlich, deshalb ist es unverzichtbar, diese festzustellen. Das Finden und Beheben von Fehlern fällt unter „Debugging“ und ist damit Bestandteil von Kapitel 7.

Fehler gibt es auf allen Ebenen, und sie lassen sich mit unterschiedlichen Techniken aufspüren. Allen gemeinsam ist, dass sie *vollständig* automatisiert ablaufen können müssen, sonst werden sie nicht in ausreichendem Maße genutzt! Also, ein Make-Ziel (oder mehrere für verschiedene Möglichkeiten) oder Skript, das nur ein einziges Kommando erfordert und ein binäres Ergebnis liefert (geht oder geht nicht):

`make check`

`make quicktests`

`./fulltests`

Testen (und vor allem Fehlerbehebung) ist mühsam, zeitaufwändig und wenig spannend. Deshalb alle Möglichkeiten nutzen, dies zu reduzieren.

- Der am einfachsten zu findende Fehler ist der, der gar nicht erst in den Code kommt. Also: Konzentriert, sorgfältig und defensiv programmieren!
- Syntaxfehler findet der Compiler, also dessen Möglichkeiten nutzen: unverwechselbare Namen, IMPLICIT NONE, auch einfachere Ausdrücke und Anweisungen klammern, ....
- Bei vielen Flüchtigkeitsfehlern kann auch der Compiler warnen, also alles an Warnungen einschalten (-W -Wall), falls möglich auch mit anderen Compilern (Intel, PGI, MS). So weit zumutbar alle Warnungen beheben und darauf achten, dass keine weiteren dazukommen!

Was der Compiler durchlässt, muss geprüft werden. Zuerst manuell oder automatisiert über Konsistenzchecks und Alternativverfahren, danach auf Reproduzierbarkeit (Regressionstests), um zu vermeiden, dass behobene Fehler sich erneut einschleichen, und sicherzustellen, dass erreichte Funktionalität zuverlässig erhalten bleibt.

- Unit-Tests prüfen einzelne Funktionen auf niedriger Ebene: Skalarprodukte, Matrixmultiplikationen, Stringzerlegungen, ... Einmal programmiert, verlässt man sich blind darauf, deshalb sorgfältig testen, vor allem Grenzfälle. Nichts ist ärgerlicher, als auf wackeligem Fundament arbeiten zu müssen.
- Modultests betrachten ein ganzes Subsystem, wofür unter Umständen größere Datenstrukturen erst aufgebaut werden müssen: Gitter einlesen, Randbedingungen behandeln, Parallelisierungsgrenzen beachten, ... Hier wird die Zusammenarbeit verschiedener Einzelfunktionen sichergestellt, wechselseitige Annahmen abgedeckt, Konventionen beachtet.

Systemtests untersuchen das Komplettpaket, indem einfache Testfälle abgearbeitet werden, die dem Produktiveinsatz ähnlich sind. Ergebnisse werden mit Referenzen verglichen.

- Alarm bei inakzeptablen Abweichungen (binäres Ergebnis).
- Möglichst unterschiedliche Testfälle, um viele Codeteile abzudecken (Test Coverage kommt später).
- Schnelle Tests (beispielsweise nur 1-2 Zeitschritte, kleine Gitter).
- Typische Einsatzbedingungen, aber auch Fälle mit komplexen Konstellationen (Chimera, Parallel, rotierend, hängende Knoten, gekoppelte Randbedingungen, ...).
- Umfangreiche Ausgabe, um auch Fehler zu finden, die ein Gesamtergebnis (Auftriebsbeiwert) nur geringfügig beeinflussen – oder aber erst auf lange Sicht.

Funktionstests sollten relativ schnell ablaufen (unter 5 Minuten), um jederzeit abgefahren werden zu können, beispielsweise vor einem Commit. Unter Umständen ist es sinnvoll, eine reduzierte Matrix bereitzustellen, die „sofort“ (unter 1 Minute) ein Resultat liefert, um beim Entwickeln ständig (nach jedem Übersetzen) am Ball zu bleiben.

Systemtests dauern oft länger, da komplexe Fälle schon länger brauchen, um überhaupt anzulaufen. Auch hier bietet es sich an, Fälle nach Laufzeit zu gruppieren, um unterschiedliche Reaktionszeiten anbieten zu können:

- 15-30 Minuten (läuft automatisch nach jedem Commit)
- einige Stunden (läuft jede Nacht)
- zwei Tage (läuft am Wochenende)

Realisiert werden kann das über unterschiedliche Make-Ziele, beispielsweise `instantCheck`, `check`, `commitCheck`, `nightlyCheck`, `weeklyCheck`.

Unit-Tests prüfen einzelne Funktionen, dass sie ihre Spezifikation erfüllen, vor allem auch in Grenzbereichen. Dazu gehören auch Fehlermeldungen oder Abbrüche bei Überschreiten. Dazu gibt es ganze Frameworks, z.B. JUnit für Java und deren Portierungen in andere Sprachen(CppUnit, CUnit, PHPUnit, ...). Vorteile davon:

- Reduktion des Aufwandes, Tests zu erstellen: wenige, einfache Makros, die Standard-main() erzeugen, Tests gruppieren, ...
- Hilfsmittel zum Auf- und Abbau noch überschaubarer Datenstrukturen
- einfache Spezifikation erwarteter Ergebnisse
- standardisierte Form der Ergebnisausgabe und -bewertung
- dadurch sehr gut automatisierbar
- Zusammenfassung aller Einzeltests mit binärem Resultat
- vielfach in IDE eingebunden oder einbindbar

Unit-Tests sollten die Funktionalität von Basisbausteinen möglichst vollständig abprüfen. Dazu gehören:

- Korrekte Funktion bei typischen Parameterwerten
- Grenzwertige Parameter (z.B. 0.0 bei Zahlen, 0/1 bei Anzahlen, Fläche entarteter Dreiecke)
- Toleranz gegenüber Fließkommaraunauigkeiten
- Abfangen ungültiger Parameter (Nullzeiger, negative Anzahlen)
- unerwartete Programmumgebung (ungültige globale Variablen, Fehler anderer Funktionen)
- unerwartete Systemumgebung (fehlende Dateien, kein Speicherplatz)

Damit ist ein belastbares Fundament geschaffen, auf dem weiter aufgebaut werden kann.

Unit-Tests bieten in gewissem Maß auch Schutz vor Compilerfehlern (äußerst selten, kommt aber vor), indem Selbstverständliches abgeprüft wird.

Einfach-Framework für unsere C++-Entwicklungen. Per Konvention enthält jedes Anwendungsverzeichnis (beispielsweise mesh) ein Unterverzeichnis tests, das Unit- und gegebenenfalls Modultests enthält. Darin ist eine Datei mit dem Namen des Verzeichnisses und Suffix tests.cpp (im Beispiel also meshtests.cpp) folgenden sehr übersichtlichen Inhalts (der kaum je geändert werden muss):

```
#include <iag/tests.h>
UNIT_MAIN("Unit tests for SUNWinT mesh directory\n"
          "(c) 2005-2011 IAG, University of Stuttgart\n")
```

Damit wird main() generiert, in dem alle Tests dieses Verzeichnisses abgearbeitet und ausgewertet werden (keine explizite Auflistung erforderlich).



Für jede Datei in mesh (cell.cpp) gibt es dann eine entsprechende Quelldatei (celltests.cpp) im tests-Verzeichnis. Im Beispiel:

```
#include "../cell.h"
#include <iag/tests.h>
TEST(LineCell) {
    LineCell l;
    CHECK(LineCell::FACES==2);
    l.mV[0]=2; l.mV[1]=5;
    CHECK(l.mV[0]==2);
    CHECK(l.mV[1]==5);
    std::vector<Position1D> v(6);
    v[2].X()=2.7; v[5].X()=4.1;
    CHECK_CALL(l.setup(v));
    CHECK_CLOSE(l.mSpace, 1.4);
    l.mV[0]=5; l.mV[1]=2;
    CHECK_THROWS(char const *, l.setup(v));
}
```

## Hierarchische Struktur:

- Test-Quelldateien im Verzeichnis
- Tests in Datei
- Prüfungen im Test

Alle Einzelprüfungen geben die getestete Bedingung aus (mit Dateiname und Zeilennummer), führen sie aus und registrieren das Ergebnis (Erfolg, Fehlschlag, Ausnahme, (System-)Fehler). Beispiel:

```

celltests.cpp:33:SUCCESS: LineCell::FACES==2
celltests.cpp:35:SUCCESS: 1.mV[0]==2
celltests.cpp:36:SUCCESS: 1.mV[1]==5
    
```

Am Ende aller Tests wird eine Statistik ausgegeben und bei aufgetretenen Abweichungen (nicht alle Prüfungen erfolgreich) ein Wert  $\neq 0$  zurückgegeben, was im Makefile oder Skript ausgewertet werden kann:

```
25 checks in 13 testcases have been run, of which
25 succeeded,
0 failed,
0 threw an exception and
0 resulted in an error
```

Alle Ausgaben (Einzelprüfungen und Gesamtstatistik) sind leicht parsebar und damit in Auswerteskripte oder Entwicklungsumgebungen zu integrieren.

Vorhandene Einzelprüfungen:

**CHECK(cond)** Prüft, ob die Bedingung ausgeführt werden kann (keine Ausnahme) und true (!=0) zurückliefert.

**CHECK\_EQUAL(x1,x2)** CHECK(x1==x2)

**CHECK\_CLOSE(x1,x2)** Prüft, ob die beiden Fließkommaargumente bis auf Rundungsfehler gleich sind.

**CHECK\_NOT\_CLOSE(x1,x2)** Prüft, ob die beiden Fließkommaargumente sich um mehr als Rundungsfehler unterscheiden.

**CHECK\_CALL(func(...))** Prüft nur auf Ausführung (keine Ausnahme).

**CHECK\_THROWS(ex,func(...))** Prüft, ob der Funktionsaufruf den angegebenen erwarteten Ausnahmetyp auswirft.

Äquivalente Prüfungen finden sich praktisch immer in Frameworks zum Unit-Test. Bei interpretierten Sprachen kann es grundsätzlich noch weitere Ergebnisse (und damit zu prüfende Bedingungen geben: Funktion/Prozedur nicht vorhanden, falsche Parameterzahl, falscher Parametertyp.

Einbau ins Makefile.am:

```
check_PROGRAMS = meshtests
meshtests_SOURCES = $(wildcard *.cpp)
meshtests_LDFLAGS = -L @LIB_DIR_iaglib@/src
meshtests_LDADD = $(top_builddir)/mesh/libmesh.a -liag
INCLUDES = $(all_includes) -I @LIB_DIR_iaglib@/src
TESTS = meshtests
```

**make check** Erzeugt das Testprogramm meshtests (kompiliert alle \*.cpp-Quellen, linkt mit dem Unit-Test Framework) und führt es aus. Abbruch bei fehlgeschlagenem Test.

Typischer Ablauf:

- ① Quellen ändern oder erweitern
- ② make
- ③ make check
- ④ Bei Fehlern: korrigieren und zurück zu 2.  
Ansonsten: weiter bei 1.

Modultests benötigen größere Datenstrukturen, um sinnvoll die Zusammenarbeit verschiedener Funktionen zu testen. Mein bisher bestes Konzept dazu:

- Teil der kompletten Anwendung anfahren (beispielsweise Gitter einlesen)
- `main()` außen herum stricken (die hier eine Gitterdatei entgegennimmt und das Einlesen anstößt)
- Darin spezifische (integrierte) Testfunktionen aufrufen, welche interne Konsistenzprüfungen durchführen

## Integration ins Makefile.am:

```
check_PROGRAMS = meshtests readtests
readtests_SOURCES = readtests.cpp
readtests_LDFLAGS = -L @LIB_DIR_iaglib@/src \
    -L @LIB_DIR_cgnspp@/cgnspp -L @LIB_DIR_cgnspp@/adfpp
readtests_LDADD = $(top_builddir)/mesh/libmesh.a \
    $(top_builddir)/physics/libphysics.a -liag -lcgnspp -ladfpp

MESHFILES = $(wildcard *.cgns)

check-local: $(MESHFILES) readtests
    @set -o pipefail ;\
    for mesh in $(MESHFILES) ; do \
        ./readtests $$mesh | diff - `basename $$mesh .cgns`.ref \
        || exit $$? ;\
    done
```

Diese bestehen im wesentlichen aus ganz normalen Anwendungsfällen, die so weit gekürzt werden, dass die Laufzeit auf ein akzeptables Maß sinkt (Gittergrößen, Zeitschritte). Ausgabewerte werden mit gespeicherten Referenzdaten verglichen, um Rückschritte (Regressions) aufzudecken. Optional gibt ein spezieller Test-Modus auch noch zusätzliche Zwischenergebnisse aus, um die Vergleichsbasis zu verbreitern.

**Wichtig:** Testfälle sollten parallel abgearbeitet werden können, um schnelle Reaktionszeiten zu gewährleisten (make -j X).

Vorgehensweise: Aus jeder Eingabedatei wird (unabhängig voneinander, also potenziell parallel) eine Ausgabe generiert, diese dann (wieder parallel) mit Referenzausgaben verglichen, gleichzeitig die Ergebnisse mit Referenzergebnissen abgeglichen und daraus eine Resultatdatei generiert, die im wesentlichen nur einen Status enthält (Erfolg, Abbruch, Vergleichsfehler, ...). Zuletzt werden alle Resultatdateien zusammengezählt und ein Gesamtstatus generiert ( $N$  Testfälle, davon  $M$  erfolgreich,  $K$  Abbrüche,  $L$  Vergleichsfehler bei Ausgabe,  $J$  Vergleichsfehler bei Ergebnissen). Nur dieser letzte Schritt ist notwendigerweise sequenziell (aber glücklicherweise sehr schnell).



```
SUNWINT_BIN=$(abspath ../../sunwint/sunwint)
ADFCMP=$(abspath $LIB_DIR_cgnspp/adfdiff/adfdiff)
CASES=$(basename $(wildcard *.ini))
PWD=$(shell pwd)
DIR=$(shell basename $(PWD) )
%.now: %.ini $(SUNWINT_BIN)
    @$(SUNWINT_BIN) $(SUNWINT_FLAGS) $< > $@ ;\
    exitcode=$$? ;\
    if [ $$exitcode = 255 ] ; then \
        $(ECHO) skip > $*.result ;\
        $(ECHO) skip >> $@ ;\
    elif [ $$exitcode != 0 ] ; then \
        $(ECHO) exitcode $$exitcode > $*.result ;\
        $(ECHO) exitcode $$exitcode >> $@ ;\
    fi
```

```
%.result: %.now %.ref
@if $(TAIL) -n 1 $< | $(GREP) -q "skip\\|exit" ; then \
    $(TAIL) -n 1 $< > $@ ;\
else \
    if $(NCMP) $(NCMPFLAGS) $*.ref $*.now ; then \
        if [ -a $*-*ref ] ; then \
            TECFILE='basename $*-*ref .ref'.tec ;\
            if $(NCMP) $(NCMPFLAGS) $*-*ref $$TECFILE ; then \
                $(ECHO) success > $@ ;\
            else \
                $(ECHO) tecplot > $@ ;\
            fi ;\
        else \
            $(ECHO) success > $@ ;\
        fi ;\
    else \
        $(ECHO) output > $@ ;\
        $(DIFF) $*.ref $*.now >> $@ || $(TRUE) ;\
    fi ;\
fi
```

```
test: $(addsuffix .result,$(CASES))
    @total=$(words $(CASES)) ; sum=0 ; skip=0 ; run=0 ; \
    exitcode=0 ; success=0 ; output=0 ; tecplot=0 ; \
    for case in $(CASES) ; do \
        result='$(HEAD) -n 1 $$case.result | $(AWK) '{print $$1}'' ; \
        (( ++$$result )) ; \
        declare cases_$$result+=" $$case" ; \
        (( ++sum )) ; \
    done ; \
    (( run=sum-skip )) ; \
    $(ECHO) Altogether $$sum test cases. ; \
    $(ECHO) $$skip \($$cases_skip\) have been skipped, \
        $$run have been run, ; \
    $(ECHO) of which $$success \($$cases_success\) succeeded. ; \
    [ $$exitcode -gt 0 ] && $(ECHO) $$exitcode \
        \($$cases_exitcode\) failed with exit codes, ; \
    [ $$output -gt 0 ] && $(ECHO) $$output \($$cases_output\) \
        failed due to output differences, ; \
    [ $$tecplot -gt 0 ] && $(ECHO) $$tecplot \($$cases_tecplot\) \
        failed for tecplot. ; \
    [ $$run = $$success ] || exit 1
```

Liegen erst einmal automatisierte Test vor, liegt es nahe, diese auch automatisch durchführen zu lassen – gegebenenfalls mit entsprechenden Konsequenzen. Dazu können beispielsweise cron-Jobs verwendet werden, die nächtlich oder am Wochenende ablaufen, außerdem gibt es entsprechende Ansatzpunkte in Versionsmanagementsystemen, die bei jedem einpflegen (commit) ein vorgegebenes (Test-)Skript aufrufen können.

Eine integrierte Lösung stellt buildbot (`trac.buildbot.net`) dar, eine Sammlung von Python-Skripten, die das Erstellen und Testen von Anwendungen erleichtern. Außerdem wird der kontinuierliche Fortschritt des Ablaufs online gestellt (via HTML) und auch der händische Anstoß ist möglich – auch von mehreren Anwendungen, gegebenenfalls auf mehreren Architekturen.

Dahinter steckt ein Master, der Impulse entgegennimmt – beispielsweise von einem Versionskontroll-Hook (bei einem commit), bei Zeitablauf (in der Nacht, am Wochenende) oder einem anderen Ereignis (der Test einer elementaren Bibliothek war erfolgreich). Anhand von anzugebenden Regeln ruft er dann einen Slave auf, der auf der gleichen oder einer anderen Maschine laufen kann, und fordert diesen auf, nach den Regeln tätig zu werden: Quellcode aus dem Repository holen, gegebenenfalls maschinenspezifisch konfigurieren, Bibliotheken übersetzen, Anwendung bauen, Unittests, Modultests und Systemtests fahren. Erfolg und Fortschritt der einzelnen Schritte wird überwacht und online gestellt, bei Fehlern wird abgebrochen und gegebenenfalls der Verursacher benachrichtigt (bei einem commit-Hook).

Mehrere Slaves können angesteuert werden, was oft auch durchaus sinnvoll ist (beispielsweise 32/64-Bit, Linux/Windows, GNU- oder Intel-Compiler).

Nach Installation wird der Master gesteuert über ein Skript (Python-Syntax), die Slaves bekommen ihre Aufgaben vom Master. Hier sind das vier unterschiedliche (nach jedem Commit, jede Nacht, jedes Wochenende und – ebenfalls am Wochenende – mit verschiedenen Compilern):

```
c['projectName'] = "SUNWinT"  
c['projectURL'] = "http://www.iag.uni-stuttgart.de/helicopter/sunwint.h  
c['buildbotURL'] = "http://svn.iag.uni-stuttgart.de:8010/waterfall"  
c['slaves'] = [BuildSlave("bot-heli", "bot-heli", max_builds=2)]  
c['slavePortnum'] = 9989  
c['change_source'] = PBChangeSource()  
...  
  
continuous=Scheduler(name="continuous", branch=None, treeStableTimer=2*  
    builderNames=["continuous"])  
nightly=Nightly(name="night", builderNames=["night"], hour=0, minute=30)  
weekend=Nightly(name="weekend", builderNames=["weekend"], dayOfWeek=4,  
    hour=21, minute=0)  
  
compilers=Dependent(name="compilers", upstream=weekend,  
    builderNames=["compilers"])  
c['schedulers'] = [ continuous, nightly, weekend, compilers.]
```

Das Repository der Quellen ist anzugeben, und zur späteren einfacheren Verwendung können Funktionen und auch Klassen definiert werden: Die Konfiguration besteht aus Python-Code!

```
cvsroot = "/srv/cvs/heli"
cvsmodule = "sunwint"
svnurl = "file:///srv/svn/heli/sunwint/trunk"
...
sourceUpdate = SVN(svnurl=svnurl, mode="update")
sourceExport = SVN(svnurl=svnurl, mode="export")
...
class Prepare(ShellCommand):
    name = "prepare"
    description = ["preparing"]
    descriptionDone = ["prepare"]
    command = ["make", "-fMakefile.cvs"]

class MyCompile(Compile):
    command = ["make", "-fMakefile.cvs", "compile"]
```

## Die Einzelschritte sind zu definieren

```

facContinuous = factory.BuildFactory()
facContinuous.addStep(sourceUpdate)
facContinuous.addStep(Prepare())

facContinuous.addStep(ShellCommand(command=["./configure",
    "CPPFLAGS=-DDEBUG_NAN -Wall", "--with-nan"]))
facContinuous.addStep(MyCompile())

facContinuous.addStep(ShellCommand(command=["make",
    "-fMakefile.cvs", "fastTest"]))
    
```

## Der nächtliche Lauf sieht nur an wenigen Stellen anders aus:

```

facNight.addStep(sourceExport)
facNight.addStep(ShellCommand(command=["./configure"]))

facNight.addStep(ShellCommand(command=["make", "-fMakefile.cvs",
    "compile"], timeout=10000))

facNight.addStep(ShellCommand(command=["make", "-fMakefile.cvs",
    "test"], timeout=1800))
    
```



Etwas komplizierter wird es bei den Compilern, allerdings glücklicherweise dank autoconf/automake jeweils nur im configure-Schritt:

```
facCompilers.addStep(ShellCommand(command=["./configure",  
    "CC=/opt/intel/Compiler/current/bin/intel64/icc",  
    "CXX=/opt/intel/Compiler/current/bin/intel64/icpc",  
    "CPPFLAGS=-DDEBUG_NAN", "--with-nan",  
    "--with-iaglib-dir=../icc/iaglib",  
    "--with-cgnspp-dir=../icc/cgnspp"])))  
  
...  
facCompilers.addStep(ShellCommand(command=["./configure",  
    "CC=/opt/gnu/bin/gcc -V4.1", "CXX=/opt/gnu/bin/g++ -V4.1",  
    "CPPFLAGS=-DDEBUG_NAN", "--with-nan", "LIBS=-lstdc++"])))
```

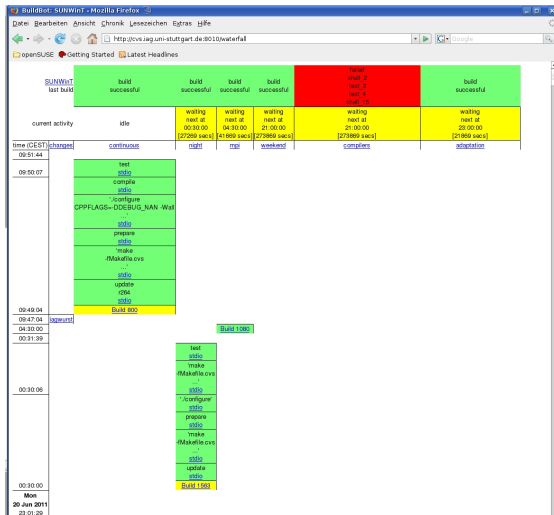
Alles zusammen bauen:

```
builderContinuous = { 'name': "continuous", 'slavename': "bot-heli",  
    'builddir': "continuous", 'factory': facContinuous, }  
builderNight = { 'name': "night", 'slavename': "bot-heli",  
    'builddir': "night", 'factory': facNight, }  
...  
c['builders'] = [ builderContinuous, builderNight, builderWeekend,  
    builderCompilers ]
```

und festlegen, wie die Ergebnisse verwertet werden, nämlich als  
Online-Darstellung und E-Mail-Benachrichtigung im Fehlerfall:

```
c['status'] = [ html.WebStatus(http_port=8010, allowForce=True) ]  
c['status'].append(mail.MailNotifier(mode="failing", builders="continuo  
    fromaddr="buildheli@cvs.iag.uni-stuttgart.de", relayhost="localhost  
    lookup=mail.Domain("iag.uni-stuttgart.de")))
```

Der Fortschritt ist am besten in der Wasserfalldarstellung sichtbar, die kontinuierlich aktualisiert wird:



Im Fehlerfall wird der Verursacher (beim Commit) benachrichtigt, dass er sich gefälligst um die Behebung kümmern möge:

Date: Wed, 23 Mar 2011 17:13:39 +0100  
From: <buildheli@cvs.iag.uni-stuttgart.de>  
To: <iagmanu@iag.uni-stuttgart.de>  
Subject: buildbot failure in SUNWinT on continuous

The Buildbot has detected a failed build of continuous on SUNWinT.  
Full details are available at:

<http://svn.iag.uni-stuttgart.de:8010/waterfallcontinuous/builds/769>

Buildbot URL: <http://svn.iag.uni-stuttgart.de:8010/waterfall>

Buildslave for this Build: bot-heli

Build Reason:  
Build Source Stamp: HEAD  
Blamelist: iagmanu

BUILD FAILED: failed test

sincerely,  
-The Buildbot

Tests helfen vor allem auch dann, wenn sie relevante Teile des Codes abdecken und nicht nur den Standardfall. Viele Spezial- und Grenzfälle werden einmal einprogrammiert, aber niemals aufgerufen (und deshalb auch nicht mit aktualisiert). Deshalb sollten zumindest Systemtests auch diese möglichst vollständig erfassen (aber gerade Fehlercode ist oft schwierig zu triggern). 50% Abdeckung sind schnell geschafft, 90% Abdeckung sollten ohne große Schwierigkeiten erreichbar sein, 95% wären wünschenswert. Deshalb Quantifizierung (und Aufbau zusätzlicher Testfälle) mit einem Werkzeug, das die Testabdeckung misst und Hinweise gibt, an welchen Stellen noch Lücken bestehen. Beispiele:

[gcov](#) GNU coverage testing tool

[codecov](#) Intel Compiler code-coverage tool

Ähnliche Werkzeuge gibt es auch bei anderen Compilern. Zuerst sind die Quellen mit entsprechenden Flags zu übersetzen (`-fprofile-arcs -ftest-coverage` oder `-coverage` bei GNU, `-prof-genx` bei Intel), was statische Aufrufgraphen produziert und den Code instrumentiert, um die Aufrufhäufigkeit jedes Blocks festzustellen. Zur Laufzeit wird diese Information aufgezeichnet und vor Programmende geschrieben. Mehrere Programmaufrufe können kumuliert werden (implizit bei GNU, mit `profmerge` bei Intel), um beispielsweise die Abdeckung einer kompletten Testsuite festzustellen. Anschließend werden die Binärdaten mit `gcov/codecov` in eine lesbare Form gebracht (Text oder HTML/XML). Also:

```
gcc --coverage -o niftyapp output/ascii.cpp output/binary.cpp  
file/write.f file/read.f main.cpp
```

```
./niftyapp data1.input
```

```
./niftyapp data2.input
```

```
gcov --branch-counts output/ascii.cpp output/binary.cpp file/write.f  
file/read.f main.cpp > niftyapp.coverage
```

## Besonderheiten von gcov:

- Alle Quellen sollten aus dem gleichen Verzeichnis heraus übersetzt werden.
- Anwendung sollte aus dem Übersetzungsverzeichnis heraus aufgerufen werden.
- Quellen müssen beim Aufruf von gcov nochmals angegeben werden.
- Mehrfache Aufrufe der Anwendung (beispielsweise für mehrere Testfälle) kumulieren automatisch.
- Auch parallel laufende Anwendungen kumulieren korrekt.
- Für jede Quelldatei (und ihre eingebetteten Header, also auch z.B. iostream oder stdlib.h) wird eine Ausgabedatei geschrieben (Suffix .gcov) mit Aufrufzahlen für jede Zeile: - steht für kein Code generiert, ##### für nicht aufgerufen.

Beispiel aus unserer mesh.cpp für die aktuelle Testmatrix:

```

27: 1651: ZoneBase * ZoneBase::readCGNS(CGNS::Zone iZone, CellType iT
      Mesh const & iMesh) {
54: 1652:     std::auto_ptr<ZoneBase> zone(0);
27: 1653:     switch (iType) {
5: 1654:         case LINE_CELL: zone.reset(new Zone<
-: 1655:             GridTopology<LINE_CELL> >(iMesh)); break;
17: 1656:         case TRI_CELL: zone.reset(new Zone<
-: 1657:             GridTopology<TRI_CELL> >(iMesh)); break;
1: 1658:         case QUAD_CELL: zone.reset(new Zone<
-: 1659:             GridTopology<QUAD_CELL> >(iMesh)); break;
3: 1660:         case TETRA_CELL: zone.reset(new Zone<
-: 1661:             GridTopology<TETRA_CELL> >(iMesh)); break;
1: 1662:         case HEXA_CELL: zone.reset(new Zone<
-: 1663:             GridTopology<HEXA_CELL> >(iMesh)); break;
#####: 1664:         default: IAG_THROW(Mesh::CGNS_Error, \
            "Illegal cell type in zone " << iZone.getName())
-: 1665:     }
27: 1666:     assert(zone.get());
27: 1667:     try { zone->readStructure(iZone); }
#####: 1668:     catch (...) {
#####: 1669:         IAG_THROW(Mesh::CGNS_Error, "Error reading zone "
            "structure of " << iZone.getName());
-: 1670:     }

```



Nicht jede unaufgerufene Zeile stellt ein potenzielles Problem dar!

- Defensive Programmierung bedeutet, auch offenkundig unnötige Sicherheitsabfragen und doppelte Überprüfungen einzubauen.
- Programmteile in Entwicklung enthalten oft Code, der vorerst noch unbenutzte Schnittstellen schafft. Warnungen einbauen!
- Fehlerbehandlung für Folgefehler ist schwer zu triggern.
- Diagnostische Funktionen bleiben manchmal im Code, um sie optional (zum Debugging) wieder einfügen zu können.

Gefundene Lücken sollten trotzdem so weit möglich gefüllt werden.

Zeilenüberdeckung (oder Anweisungsüberdeckung  $C_0$ ) ist noch keine vollständige Abdeckung! Besser (aber auch zunehmend aufwändiger):

- Zweigüberdeckung ( $C_1$ , auch leere else- oder case-Zweige werden abgearbeitet)
- Vollständige Pfadüberdeckung ( $C_2a$ , bei Schleifen unmöglich)
- Partielle Pfadüberdeckung ( $C_2b$ , jede Schleife 0-mal, 1-mal (Boundary) und 2-mal (Interior) ( $N$ -mal,  $C_2c$ ) durchlaufen)
- Bedingungsüberdeckung: einfache ( $C_3a$  entspricht  $C_1$ ), mehrfache ( $C_3b$ ) oder minimale ( $C_3c$ ) bei zusammengesetzten (&&, ||) Bedingungen

## Fehlerelimination auf allen Ebenen:

- Defensive Programmierung
- Übersetzen mit Warnungen auf verschiedenen Compilern/Systemen
- Unit-Tests auf unterer Ebene (Basisbausteine)
- Modultests auf Zwischenebenen, soweit möglich und praktikabel
- Systemtests mit möglichst vollständiger Zeilenabdeckung (mindestens)
- Falls äußerst merkwürdige Dinge passieren und alles andere nicht mehr hilft, Compilerfehler nicht gänzlich ausschließen.

Zum Glück entwickeln wir keine sicherheitskritische Primary Flight System Software (nach RTCA DO-178B/ EUROCAE ED-12B).  
Sorgfältige Tests ersparen aber trotzdem langwieriges Debuggen und sind daher gut investierte Zeit!

## 7 Debugging

Fehlersuche ist lästig und langwierig, deshalb ist die oberste Direktive beim Programmieren:

*Fehler vermeiden!*

oder etwas egozentrischer:

*Fehlersuche vermeiden oder zumindest verkürzen!*

Wie beim in Kapitel 6 beschriebenen Testen angesprochen bedeutet das auf mehreren Ebenen:

- Defensiv programmieren!
- Compilerwarnungen beachten
- systematische Tests (Unit, Modul, System)
- zahlreiche und breit gestreute Konsistenz- und Plausibilitätschecks (Datenstrukturen, positive Drücke, ...)
- Hilfsmittel zum schnellen Finden einbauen und nutzen

Manche Fehler treten bereits zur Übersetzungszeit oder spätestens beim Linken auf und sind deshalb relativ leicht zu finden und beheben. Außer den banalen (Vertipper, Variable vergessen zu deklarieren, Header für Funktionen nicht eingebunden) gehören dazu auch:

**Pfadprobleme** Können immer dann auftreten, sobald unterschiedliche Versionen von Compilern, Bibliotheken, Modulen auf dem Rechner herumvagabundieren. In diesem Fall wird der Compiler mit an Sicherheit grenzender Wahrscheinlichkeit die falsche Variante finden, und der Programmierer kann ewig suchen. Compiler geben Suchpfade üblicherweise mit `-v` aus, ansonsten mal `$PATH` oder `sys.path` ansehen.

**Makro-Expansion** Vor allem in C/C++, wo der Präprozessor doch öfter in nichttrivialer Weise eingesetzt wird, können diese auch fehlerbehaftet sein. Vertauschte Argumente oder fehlende Klammerung von Argumenten, die zu unzulässigen Ausdrücken führen (Glück gehabt!), vergessene Semikola,

falsche Zuordnung von else-Zweigen, ... Das Ergebnis der Expansion ist üblicherweise mit -E zu sehen.

**Linkerfehler** Vor allem im Sprachmix mit C++, welches Argumenttypen mit in den Funktionsnamen einbaut, was andere Sprachen nicht tun: `_Z6squared` bedeutet `square(double)`, `_ZNK4Cell7getAreaEPKd` steht für `Cell::getArea(double const*) const` (der Rückgabewert wird nicht mit verschlüsselt). FORTRAN-Compiler neigen dazu, Unterstriche vorne oder hinten anzuhängen und die Groß/Kleinschreibung zu wechseln: `func` wird zu `func_` oder `_FUNC`. Hilfe durch Inspektion der Objektdateien mit `nm` und gegebenenfalls Übersetzung in eine lesbare Form durch `c++filt`.

Bei manchen Fehlern liegt die Ursache vor allem zwischen den Ohren des Programmiers.

**Zaunlattenproblem** Sehr beliebt beispielsweise bei Punkten und Zellen dazwischen: Hat die Zelle den gleichen Index wie der Punkt links oder der rechts, liegen die Punkte der Zelle  $i$  also bei  $i - 1$  und  $i$  oder bei  $i$  und  $i + 1$ . Besonders heikel bei Phantompunkten/-zellen.

**Eins daneben** Immer wieder gerne gesehen bei Schleifen als erste (0/1) und letzte ( $N-1/N/N+1$ ) Iteration und Vergleichen ( $<$  vs.  $\leq$ ,  $>$  vs.  $\geq$ ).

**Speicherverwaltung** Speicherenden nicht überschreiben, freigegebenen Speicher nicht nochmal verwenden, nicht mehr gebrauchten Speicher freigeben.

Speicherbehandlung ist ein spezielles Problem mit eigenen Fehlern und hat daher ein eigenes Kapitel 8 verdient.

Viele (aber nicht alle) Sprachen teilen sich weitere typische Probleme:

**Uninitialisiert** Viele Sprachen lassen Variablen undefiniert, falls nicht explizit initialisiert. Vorhandener Speichermüll kann alles Mögliche bedeuten.

**Typgrößen** Wie groß ist ein int/long/REAL/DOUBLE/Zeiger? Je nach Maschine und Compiler kann das variieren.

**Verdeckung** Lokale Variable verdeckt globale/äußere Variable gleichen Namens. Bindung von Namen an Speicherobjekte ist manchmal nichttrivial. Gute Namen helfen!

**Überschreiben** Ähnliches kann bei Funktionsnamen passieren, wobei hier in der Regel der Linker meckert. Ausnahme: Funktionen in Bibliotheken (beispielsweise der FORTRAN- oder C-Standardbibliothek) können kommentarlos überschrieben werden.



**Undefinierte Variablen** FORTRAN kennt noch implizite Datentypen, die im Zweifelsfall nie mit dem übereinstimmen, was der Programmierer bräuchte.

**Modifizierte Parameter** FORTRAN übergibt Parameter by-reference, die Funktion kann also Variablen im aufrufenden Code ändern.

**Falsche Aufrufparameter** FORTRAN und C ohne Prototypen: Der Aufrufer legt beliebige Variablen auf den Stack, die Funktion interpretiert die Bits in ihrem eigenen Sinne (oft einfach falsche Reihenfolge oder Anzahl).

**else zu falschem if** Bei zwei geschachtelten if-Anweisungen mit nur einem else-Zweig, gehört dieser zum äußeren oder inneren if?

**Nullzeiger/irregeleitete Zeiger** Das Totschlagargument gegen C/C++ und Verwandte. Tritt aber auch bei FORTRAN auf, wenn Speicherverwaltung über große Arrays und Indizes darin betrieben wird. Indizes können genauso wild in der Gegend herumvagabundieren. Auch hier: Kapitel 8.

**Durchfall** case-Zweige in C/C++ müssen explizit mit einem break; abgeschlossen werden, ansonsten läuft der Ausführungspfad in den nächsten Zweig weiter.

**Leere Schleife** Ein ; hinter der for- oder while-Anweisung ist eine leere Anweisung, die brav wiederholt wird – anstelle des eigentlich gemeinten Schleifenkörpers.

**Operatorenvorrang** Punkt vor Strich ist den meisten noch aus der Schule geläufig (und auch in praktisch allen Programmiersprachen so), aber wie steht es mit << vs. ^ beziehungsweise && vs. || (oder den entsprechenden Äquivalenten)?

**Zuweisung statt Vergleich** if (ch=EOF) handleEof(); ist eine völlig legale Anweisung, aber wohl kaum so beabsichtigt und enorm schwer zu finden. Abhilfe: Warnungen an und/oder Konstante nach links (EOF==ch).

Unterschiedliche Fehler äußern sich auf unterschiedliche Weise, so dass häufig schon aus der Problembeschreibung der Fehlertyp eingegrenzt werden kann.

**Programmabbruch** Die Anwendung verabschiedet sich augenblicklich: Segmentation violation, Signal caught, Unexpected exception, ...  
Bei compilierten Sprachen kann ein Speicherabzug (core Dump) helfen, den Fehler zu lokalisieren, indem der Zustand zum Zeitpunkt des Abbruchs samt Aufrufsequenz und aller Variablen festgehalten wird.

**Kontrollierter Abbruch/Explizite Fehlermeldung** Der Programmierer hat sich Gedanken gemacht über ein potenzielles Problem, konnte (oder wollte) es aber nicht innerhalb des Programmes lösen: fehlende Ressourcen (Speicher, Plattenplatz, Verzeichnisse), unsinnige Eingaben (negative Dichte der Zuströmung) oder unerwartete Zwischenergebnisse (negative Drücke). Bis auf die letzte Problematik meist leicht zu lösen, wenigstens so- lange der Programmierer auch eine hilfreiche (oder zumindest eindeutig zuordenbare) Fehlermeldung spendiert hat.

**Keine Ausgabe** Sind erst einmal NaNs (Not-a-Number) aufgetreten, kann dies die weitere Verarbeitung *signifikant* (1-2 Größenordnungen) verlangsamen, da NaNs oft über Mikrocode in der CPU bearbeitet werden müssen. Passiert noch länger nichts, handelt es sich möglicherweise um eine

**Endlosschleife**, bei der die Abbruchbedingung fehlerhaft ist und/oder das Update des Schleifenzählers nicht funktioniert:  
`for (int i=0; i!=13; i+=2) {}` wird sehr lange laufen!

**Unerklärlicher Kontrollfluss** Manchmal bei zusammengesetzten Bedingungen, Zuweisungen statt Vergleichen (= statt ==) oder vergessenem `break`.

**Unerklärliche Werte** Vor allem in globalen Variablen (Arrays) oder Common-Blöcken können diese auf Speicherfehler hindeuten, die im Prinzip beliebig weit vorher passiert sein können und daher sehr schwierig zu finden sind. Kapitel 8 hilft dabei.

**Falsche Werte** Vorzeichen, Indizes, Variablenbezüge, Vorfaktoren, Operatoren, ... Viel Spaß!

Wegen der bei uns oft auftretenden großen Datenmengen ist es oft problematisch, im Debugger einzelne Werte zu verfolgen. Daher ist es sinnvoll, entsprechend umfangreiche Ausgaben vorzusehen, die im Bedarfsfall systematisch durchsucht oder mit Referenzen verglichen werden können. Dabei können Makros helfen (vor allem in C/C++), mehrfache Tipparbeit zu verringern, beispielsweise in der Form:

```
#define DEB_VAR1(var1) \  
    if (0<DEBUG_LEVEL) std::cerr << #var1 << "=" << (var1) << std::endl  
#define DEB_VAR2(var1,var2) \  
    if (0<DEBUG_LEVEL) std::cerr << #var1 << "=" << (var1) << ", " \  
        << #var2 << "=" << (var2) << std::endl  
#define DEB_VAR3(var1,var2,var3) \  
    if (0<DEBUG_LEVEL) std::cerr << #var1 << "=" << (var1) << ", " \  
        << #var2 << "=" << (var2) << ", " \  
        << #var3 << "=" << (var3) << std::endl
```

Verwendung dann mit

```
DEBUG_LEVEL=1;  
DEB_VAR3(i, mNumVariables, iZone->getName());
```

Also eine intelligente Variante des klassischen write/printf Vorgehens

Gerade bei Vergleichen oder beim manuellen Suchen ist es oft ärgerlich, wenn Rundungsfehler an der Genauigkeitsgrenze zu Abweichungen führen.

**ncmp** vergleicht numerisch und toleriert dabei kleine Abweichungen in einem anzugebendem Maße (relative und absolute Fehler).

**chop** schneidet überflüssige Nachkommastellen ab und glättet beispielsweise  $-3e-15$  zu 0 und  $3.99999998$  zu 4, was deutlich einfacher zu lesen ist.

Eine gewisse Vorsicht ist dabei jeweils geboten, um kleine, aber signifikante Abweichungen nicht versehentlich zu übersehen. Beide Tools werden im Quellcode (POSIX-C) auf der Vorlesungsseite zur Verfügung gestellt.

Änderungen in der Zahl von Leerzeichen (-b) oder Groß-/Kleinschreibung (-i) kann auch diff ignorieren. Darüber hinaus gibt es aber auch Werkzeuge, die nicht nur komplette Zeilen vergleichen, sondern zeichenweise die Unterschiede sichtbar machen. Unter KDE erledigt dies beispielsweise **kompare**.

Debugger lassen ein Programm kontrolliert ablaufen und geben Eingriffsmöglichkeiten unterwegs. In der Regel kann mit dem Quellcode gearbeitet werden (übersetzen und linken mit -g), im Notfall ist auch der Maschinencode zugänglich. Verfügbar sind typischerweise

**Breakpoints** An spezifizierten Stellen (Zeilen, Funktionsaufrufen) im Programm kann abgebrochen werden, gegebenenfalls geknüpft an eine zusätzliche Bedingung (Funktionsparameter oder lokale Variablen haben bestimmte Werte, ...).

**Watchpoints** halten an, sobald ein Ausdruck (eine Variable, eine Speicherstelle) seinen Wert ändert. Extrem langsam, falls keine Hardware-Watchpoints (mehr) zur Verfügung stehen.

**Catchpoints** Stop bei Ereignissen außerhalb des regulären Programmablaufs: Werfen/Fangen einer C++-oder Ada-Ausnahme oder Änderung des Prozesses (exec, fork, vfork).

**Checkpoints** erlauben, den momentanen Zustand des Programmes festzuhalten und später wieder anlaufen zu lassen.



**Anzeige** Beliebige Ausdrücke von Variablen und (zumeist) Funktionsaufrufen können dargestellt werden, um den aktuellen Zustand zu sehen. Variablen können auch modifiziert werden.

**Einzelschritte** Der Fortgang kann schrittweise erfolgen, entweder zeilenweise (next) oder mit der nächsten Anweisung (step). In letzterem Fall werden aufgerufene Funktionen schrittweise abgearbeitet, in ersterem als Ganzes übersprungen. Gegebenenfalls kann der Programmablauf verändert werden (return oder goto) und – sofern nötig oder hilfreich – der Code auf Maschinenebene abgearbeitet.

**Stacktraces** Bei verschachtelten Funktionsaufrufen können höhere Ebenen (einschließlich Parameter und lokaler Variablen) wie auch die Aufrufhierarchie untersucht werden.



gdb ist der Klassiker unter den Debuggern. Die Kommandozeile ist nicht jedermanns Sache, aber er funktioniert gut und zuverlässig und ist immer greifbar. Wichtige Kommandos:

**run [args]** startet das geladene Programm, gegebenenfalls mit Kommandozeilenargumenten.

**break** setzt einen Breakpoint, auf eine Funktion, Quellcodezeile (hier oder in einer anderen Datei) oder Adresse.

**break [POS] if EXPR** setzt einen bedingten Breakpoint, der nur anhält, falls EXPR zu wahr ausgewertet wird.

**tbreak, hbreak** Temporärer (einmaliger) beziehungsweise Hardware-unterstützter Breakpoint.

**rbreak REGEX** richtet einen Breakpoint an allen Stellen ein, die dem regulären Muster entsprechen.

**continue**, **fg** [**COUNT**] lässt nach einem Stop weiter laufen, ignoriert dabei die nächsten COUNT Breakpoints.

**return** [**EXPR**] kehrt (und liefert gegebenenfalls) zum Aufrufer zurück.

**step** [**COUNT**] geht zur (COUNT-) nächsten Quellcodezeile. Falls ein Funktionsaufruf dabei ist, wird in die Funktion gesprungen.

**next** [**COUNT**] geht zur (COUNT-) nächsten Quellcodezeile in der *aktuellen* Funktion. Eventuelle Funktionsaufrufe werden als Ganzes abgearbeitet.

**finish** erledigt den Rest der aktuellen Funktion.

**until** hält erst nach der laufenden Schleife wieder an.

**stepi**, **nexti** funktionieren wie **step** und **next**, allerdings auf Maschinenebene.

**backtrace** zeigt die Aufrufhierarchie bis zur aktuellen Position. Erste Hilfe bei core-Dumps.

**backtrace full** zeigt auch gleich noch alle (jeweils) lokalen Variablen mit an.

**up**, **down** geht eine Ebene hoch (Richtung main) oder runter in der Hierarchie. Der Zustand dort kann dann untersucht werden mit

**info locals**, **info args** zeigt die lokalen Variablen beziehungsweise Parameter der aktuellen Ebene.

**list [POS]** zeigt den angegebenen (Zeile, Funktion) oder aktuellen Quellcode an.

**disassemble** stellt den Maschinencode dar, bei /m gemischt mit dem Quellcode.

`print[/FMT] [EXPR]` ist (erwartungsgemäß) zuständig für die Ausgabe von Daten aller Art. Dazu gehören natürlich vor allem Variablen, aber auch Arrays: `print cell[0]@5` gibt `cell[0]` aus und die vier folgenden. Das Ausgabeformat entspricht dem Typ des Ausdrucks, kann aber modifiziert werden: `/x`, `/o`, `/t`, `/d`, `/u` hexadezimale/oktale/binäre/dezimale/vorzeichenlose Integers; `/c` Zeichen; `/s` String (nullterminierte Zeichenkette; `/a` Adresse (hexadezimal und als Offset); `/f` Fließkomma

`x [/NFU] [ADDR]` (für `examine`) zeigt Speicherinhalte ab der angegebenen Adresse an. `N` steht für die Anzahl, `F` für das Anzeigeformat (siehe `print`, zusätzlich noch `i` für instructions/Maschinenbefehle) und `U` für die Größe: `b`, `h`, `w`, `g` für 1, 2, 4 und 8 Bytes jeweils.

`display[/FMT] EXPR` arbeitet ähnlich wie `print`, zeigt ab sofort den Ausdruck aber jedesmal an, wenn das Programm stoppt, also beispielsweise nach dem nächsten Einzelschritt (`next`, `step`) oder Erreichen eines Breakpoints.

gdb versucht dem Programmierer durch sinnvolle Abkürzungen zu vereinfachen. So können alle Befehle abgekürzt werden: n/s/b/d/p/c für next/step/break/display/print/continue usw. Einfacher Druck auf <ENTER> wiederholt den letzten Befehl, und wo sinnvoll mit der Bedeutung „nächste“: nächste Zeilen im Quellcode, nächste Speicherinhalte bei x, nächster Einzelschritt.

Alle mit print ausgegebenen Werte werden in einer Liste gespeichert, darauf zugegriffen wird mit \$ und fortlaufenden Nummern, also \$1, \$2, \$3, ..., oder von rückwärts \$ für den letzten, \$\$ den vorletzten oder \$\$N für den N-letzten Wert (\$==\$0, \$\$==\$1). Automatisch erzeugt werden weiterhin Adresse (\$\_) und Wert (\$\_\_) des letzten x-Befehls (examine). Eigene Variablen können darüber hinaus erzeugt und genutzt werden: set \$i=5, print \$i++. Diese interferieren nicht mit den Variablen des zu debuggenden Programms. Damit ist gdb grundsätzlich voll skriptfähig. Schließlich gibt es in Ergänzung der Maschinencode-Ansicht noch die maschinenspezifischen Register der CPU: \$pc für den aktuellen Befehlszähler, \$sp für den Stackpointer, aber auch \$eax/\$ebx/... \$xmm0/... \$st(0)/... für die allgemeinen Register.

Neben dem etwas rustikalen gdb gibt es eine Reihe von Alternativen:

**ddd** Ein grafisches Front-End zu gdb für Debugging mit der Maus.  
Nette grafische Möglichkeiten zur Darstellung und  
Live-Aktualisierung komplexer Datenstrukturen (verkettete  
Listen, variable Arrays, ...)

**KDevelop, Eclipse, Netbeans** IDEs mit grafischem Front-End zu gdb  
(und teilweise Intel idb).

**Solaris Studio** IDE mit Compiler und Debugger, auch unter Linux.  
Kann gut mit Threads umgehen, erlaubt Just-in-time-Fixes,  
enthält Speicherdebugger.

**MS Visual Studio** Windows-IDE, auch in kostenfreier Version  
(Express) erhältlich.

**Intel Parallel Studio XE** enthält einen eigenen Debugger, integriert  
in die Intel-Eclipse-IDE, mit vielen parallelen Features.

**ddt** Kommerzieller Debugger von Allinea mit GUI und Stärken vor  
allem bei parallelen Programmen (Threads, Prozesse, verteiltes  
MPI), enthält ebenfalls Speicherdebugger.

- Fehler jeweils so weit wie möglich vermeiden (Programmierstil, Konventionen, Verantwortlichkeiten), kenntlich machen (Compilerwarnungen) oder wenigstens zügig finden (Assertions, Tests)
- Systematisch an die Fehlersuche herangehen, Reproduzierbarkeit sicherstellen
- Komplexität des Fehlerfalles so weit wie möglich reduzieren
- Je nach Fehlervariante unterschiedliche Techniken nutzen
  - core-Dumps mit Backtraces
  - Programmfluss und Datenstrukturen mit interaktivem Debugger
  - Datenverarbeitung mit ausführlichen Zwischenergebnissen und Vergleichen (gegebenenfalls händische Nachrechnung)
  - strace zum Verfolgen von Systemaufrufen

Bei gefundenen Fehlern kurz innehalten

- Habe ich die Ursache gefunden oder nur ein Symptom gepflastert?
- Gibt es Klone dieses Fehlers (copy-paste), die auch korrigiert werden müssen?
- Ist dieser Fehlertyp charakteristisch, taucht er vielleicht noch mehrmals auf?
- Wie hätte ich diesen Fehler vermeiden, erkennen, schneller finden können?



## 8 Speicherfehler: dmalloc und Valgrind

Speicherfehler sind besonders schwierig zu finden (und zu beheben), da sie sich an einer Stelle und zu einer Zeit manifestieren, die unter Umständen nichts mit dem eigentlichen Fehler zu tun hat. Besonders häufig sind sie in C/C++ mit seinen Zeigern und der dynamischen Speicherverwaltung, aber auch in anderen Sprachen, wo eine solche Verwaltung notfalls simuliert wird (FORTRAN-Array mit Indizes). Bibliotheken wie dmalloc und Werkzeuge wie valgrind unterstützen bei der oft unangenehmen Suche.

Die Standardfunktionen der C-Bibliothek sind:

`void * malloc(unsigned size)` Speicherblock anfordern

`void * calloc(unsigned number, unsigned size)` Speicherblock der Größe `number * size` anfordern und mit Nullen füllen

`void * realloc(void * old_ptr, unsigned new_size)` Speicherblock vergrößern (auch von 0) oder verkleinern (auch auf 0), dabei notfalls verschieben (Inhalte werden kopiert)

`void free(void * ptr)` Speicherblock freigeben

`char * strdup(const char * s)` Speicher für Stringkopie anfordern und diese erzeugen

Linux (genauer die glibc) unterstützt mit primitiven Möglichkeiten dank Umgebungsvariable `MALLOC_CHECK` (doppeltes free, off-by-one Bugs): Wenn 0, passiert nichts, bei 1 wird eine Fehlermeldung auf `stderr` ausgegeben, bei 2 abgebrochen (via `abort`).

Dazu zählen Probleme mit der Speicherverwaltung:

- Nicht zugewiesener Speicher: Fällt sofort auf, da kein gültiger Zeiger vorhanden ist, allenfalls uninitialisierter Müll.
- Kein Speicher mehr da: Nicht abgefangener Speichermangel führt beim Zugriff auf den Nullzeiger zum sofortigen Abbruch.
- Nicht oder mehrfach freigegebener Speicher: Ersteres führt zur sukzessiven Speicherfüllung, letzteres oft (aber nicht immer) zum sofortigen Abbruch mit Fehlermeldung.
- Verschobener Speicher: realloc kann beim Vergrößern den Block verschieben, Referenzen/Zeiger (auch innerhalb) darauf muss man dann aber selbst anpassen.

aber auch der Nutzung:

- Lesen/Schreiben bis zum Nachbarn: Der Speicherblock war nicht groß genug (Geisterzellen), so dass noch auf Bereiche (kurz) davor oder dahinter zugegriffen wird. Ganz schwer zu finden ohne Hilfsmittel.

Es gibt einige spezifische Speicherdebugger, die helfen sollen, die genannten Fehler zu finden. Typischerweise hängen sich diese in die Speicherverwaltung ein, bieten also ihre eigenen malloc/calloc/free/realloc-Funktionen, welche die in der Standardbibliothek überschreiben. Diese prüfen dann den Heap auf unzulässige Manipulationen und die aufgeführten typischen Fehler. Ein brauchbares (und im Netz frei verfügbares) Beispiel ist dmalloc ([www.dmalloc.com](http://www.dmalloc.com)). Je nach eingeschalteten Tests (Umgebungsvariable DMALLOC\_OPTIONS) läuft die Anwendung etwas länger (in der Regel marginal), danach findet man die angeforderte Log-Datei. Darin sind die gefundenen Probleme aufgeführt, also nicht oder mehrfach freigegebene Blöcke (einschließlich Quelldatei und Zeile der Anforderung, so weit verfügbar), überschriebene Grenzbytes, zerstörter Heap. Auf Wunsch werden auch alle Speicherverwaltungsaufrufe protokolliert.

## Anwendung von dmalloc:

- Definieren einer Umgebungsvariablen `DMALLOC_OPTIONS` (direkt oder über eine Shell-Funktion `dmalloc` und das gleichnamige Kommando, siehe Tutorials)
- Übersetzen mit Debugging-Informationen (`-g`), `DMALLOC` definiert (`-DDMALLOC`), eventuell auch `DMALLOC_FUNC_CHECK` (prüft auch noch fehlerträchtige Funktionen wie `strcpy`, `bcopy`, ...) und `#include <dmalloc.h>`
- Linken mit der Bibliothek (`-ldmalloc`) *vor* der Standardbibliothek (die normalerweise ganz hinten angefügt wird)
- Laufen lassen

Unter Umständen können auch Anwendungen, die nichts von `dmalloc` wissen, damit getestet werden (über dynamische Bibliotheken), allerdings fehlen dann die Quelldateien und -zeilen als Hilfe (Funktionsnamen sind aber manchmal verfügbar).

Folgende Tests können durchgeführt werden:

- Integrität der Verwaltung: malloc/free werden passend aufgerufen (keine fehlenden oder doppelten Freigaben), auch strdup wird freigegeben
- markierte Grenzbytes vor und nach dem Block bleiben unangetastet (keine off-by-one Fehler, Größe ausreichend)
- angeforderter Speicher wird selbstständig initialisiert (Markierung mit „schlechten“ Werten)
- freigegebener Speicher bleibt unangetastet (Markierung und Überprüfung)
- potenzielle Verschiebung bei realloc wird berücksichtigt (absichtliche Verschiebung)
- freigegebene Zeiger werden nicht neu belegt (immer neue Blöcke)

Da manche Tests die Ausführung spürbar verlangsamen (beispielsweise Integritätscheck der gesamten Heap-Verwaltung bei jedem Aufruf), können diese individuell angefordert werden.

Noch weiter geht valgrind ([www.valgrind.org](http://www.valgrind.org)). Das ist nicht nur ein Speichertester, sondern ein gesamtes System. Der Binärcode wird nicht auf der CPU ausgeführt, sondern in einer virtuellen Maschine emuliert – (fast) jeder fehlerhafte Zugriff kann deshalb unmittelbar am Ort des Geschehens abgefangen werden. Dafür geht die Laufzeit enorm hoch, mindestens eine, manchmal zwei Größenordnungen. Es empfiehlt sich also die Beschränkung auf kleinere Testfälle.

Valgrind selbst ist eigentlich nur die virtuelle Maschine, die Code ausführen kann. Für die eigentlichen Tests gibt es verschiedene Tools, welche die kontrollierte Abarbeitung nutzen. Wichtigstes (und auch standardmäßig aufgerufenes) ist memcheck, das als Speicherdebugger fungiert. Daneben gibt es noch cachegrind, das die Cache-Nutzung sichtbar macht (und callgrind für Aufruf-Graphen), massif, das die räumliche und zeitliche Verteilung von Speicheranforderungen aufdeckt, und schließlich helgrind, das Parallelisierungsprobleme aufdecken hilft (mangelnde Synchronisation zwischen POSIX-Threads).

Memcheck ist das zentrale Werkzeug der valgrind-Familie. Zum Ausgleich für den Performanceverlust gibt es sehr wertvolle Informationen:

Zugriffe außerhalb angeforderter Speicherbereiche (auch Lesen!) werden sofort gemeldet, mit Quelldatei und -zeile samt Aufrufhierarchie und Parametern. Lesen uninitialisierten Speichers wird ebenfalls gemeldet. Außerdem werden auch problematisch Aufrufe von memcpy und ähnlichen Funktionen mit überlappenden Speicherbereichen abgefangen. Dazu alle Probleme, die beispielsweise dmalloc erkennt (fehlerhafte Aufrufe der Speicherverwaltung). Bei Speicherlecks wird auch (meistens zuverlässig) erkannt, ob noch ein Zeiger auf den verlorenen Block existiert, so dass er eventuell freigegeben werden könnte, oder ob er gänzlich verloren ist.



Vor das ausführbare Programm wird einfach ein valgrind gestellt – falls nicht der Speichertest (memcheck), dann ist mit `--tool=XXX` anzugeben, welches Tool verwendet wird (cachegrind, helgrind). Nach einer Startmeldung von memcheck und valgrind läuft dann das Programm wie gewohnt ab – nur natürlich erheblich langsamer. Ausgaben von valgrind sind markiert durch ein `==#####` am Zeilenanfang, wobei `#####` für die Prozessnummer steht. Am Ende steht dann das Gesamtergebnis:

```
==26043== ERROR SUMMARY: 490 errors from 42 contexts (suppressed: 3 from
==26043== malloc/free: in use at exit: 3,234,990 bytes in 292 blocks.
==26043== malloc/free: 1,980 allocs, 1,688 frees, 3,340,922 bytes alloc.
==26043== For counts of detected errors, rerun with: -v
==26043== searching for pointers to 292 not-freed blocks.
==26043== checked 34,451,520 bytes.
```

dann ein Berg von loss records (verlorene Speicherblöcke) und zuletzt

```
==26043== LEAK SUMMARY:
==26043==    definitely lost: 0 bytes in 0 blocks.
==26043==    possibly lost: 1,667 bytes in 58 blocks.
==26043==    still reachable: 3,233,323 bytes in 234 blocks
==26043==    suppressed: 0 bytes in 0 blocks.
```



Ein solcher loss record liest sich so:

```
==26043== 992 bytes in 1 blocks are still reachable in \  
    loss record 63 of 73  
==26043==      at 0x4023825: operator new[](unsigned) (in \  
    /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)  
==26043==      by 0x815E049: Region<GridTopology<(CellType)2> >::\  
    Region(Grid*, Zone<GridTopology<(CellType)2> > const*, bool)  
    (region.cpp:43)  
==26043==      by 0x814BFAD: RegionBase::CreateRegions(Grid*, \  
    ZoneBase const*, std::vector<RegionBase*, \  
    std::allocator<RegionBase*> >&) (region.cpp:159)  
==26043==      by 0x814A205: Grid::Setup() (grid.cpp:39)  
==26043==      by 0x4286F9B: (below main) (in /lib/libc-2.5.so)
```

Eine Meldung (schon während des Laufs) für einen unzulässigen Schreibzugriff *hinter* einem Block sieht so aus:

```
==26043== Invalid write of size 4
==26043== at 0x815DF41: Region<GridTopology<(CellType)2> >::Region( \
    Grid*, Zone<GridTopology<(CellType)2> > const*, bool) (sunwint.h:27)
==26043== by 0x814BFAD: RegionBase::CreateRegions(Grid*, ZoneBase const*
    std::vector<RegionBase*,std::allocator<RegionBase*> >&) (region.cpp:4)
==26043== by 0x814A205: Grid::Setup() (grid.cpp:39)
==26043== by 0x4286F9B: (below main) (in /lib/libc-2.5.so)
==26043== Address 0x43CDEF8 is 0 bytes after a block of size 0 alloc'd
==26043== at 0x4023825: operator new[](unsigned) (in \
    /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==26043== by 0x815DF12: Region<GridTopology<(CellType)2> >::Region(\
    Grid*, Zone<GridTopology<(CellType)2> > const*, bool) (region.cpp:4)
==26043== by 0x814BFAD: RegionBase::CreateRegions(Grid*, ZoneBase const*
    std::vector<RegionBase*,std::allocator<RegionBase*> >&) (region.cpp:4)
==26043== by 0x814A205: Grid::Setup() (grid.cpp:39)
==26043== by 0x4286F9B: (below main) (in /lib/libc-2.5.so)
```

Ein Lesezugriff mitten im Nirgendwo äußert sich so:

```
==26043== Invalid read of size 4
==26043== at 0x8164CE8: Region<GridTopology<(CellType)2> >::evaluatePatch
    Patch const&, EvaluationPoint&, FullMassUpdater&) (region.cpp:922)
==26043== by 0x81651A2: Region<GridTopology<(CellType)2> >::EvaluateLocal
    EvaluationPoint&, FullMassUpdater&) (region.cpp:815)
==26043== by 0x814AE12: Grid::Evaluate(EvaluationPoint&, \
    FullMassUpdater&) (grid.cpp:129)
==26043== by 0x819A6C8: Problem::realEval(EvaluationPoint&, \
    FullMassUpdater&) (problem.cpp:51)
==26043== by 0x4286F9B: (below main) (in /lib/libc-2.5.so)
==26043== Address 0x43CDF0C is not stack'd, malloc'd or (recently) free'd
```

Speicherfehler sind schwierig zu debuggen und zu beheben. Auch hier ist wieder Sorgfalt beim Programmieren die beste Möglichkeit, Fehler schon im Vorfeld zu verhindern:

- Klare Verantwortlichkeiten für dynamische Speicherblöcke, damit auch
- begrenzte (möglichst lokale) Zugriffsmöglichkeiten darauf und
- definierte Aufgabe (kein Recycling von Speicher, der „noch übrig“ ist).

Für die dann noch übrig bleibenden Fehler sind Werkzeuge wie `dmalloc` und `valgrind` sehr wertvolle Hilfen. *Vorsicht:* Ein erstmaliger Einsatz in gewachsenem Code kann Entsetzen hervorrufen!

Mit etwas Mühe lassen sich die allermeisten Probleme aber relativ schnell beheben, beim kontinuierlich wiederkehrenden Einsatz bleibt es dann auch so.

## 9 Coding Standards

Programmierung ist praktisch immer Teamarbeit – wenn nicht im Moment, dann doch zeitlich versetzt (ich habe existierenden Code übernommen und gebe meinen an Nachfolger weiter). Im Extremfall bin ich selbst mein eigener Teampartner, wenn ich Code anfassen muss, den ich vor drei Jahren geschrieben habe. Deshalb sind Coding Standards wichtig, denn sie gewährleisten eine gemeinsame Basis und Entscheidungshilfe durch Konsistenz in

- Namen (Dateien, Klassen, Variablen, Funktionen)
- Modularisierung (Kohäsion, Kopplung)
- Parametern (Anordnung, Semantik)
- Anordnung (Leerzeichen, Tabulatoren, Klammern)
- Fehlerbehandlung (Fehlercodes und -semantik, Ausnahmen)

Komplexe Software ist schwer und aufwändig zu entwickeln, so dass wir alle Hilfe brauchen, die wir kriegen können! Funktional ist natürlich die Korrektheit am wichtigsten, aber die meiste Zeit wird die Software nicht nur angewandt, sondern weiter entwickelt. Deshalb ist auch der Quellcode von entscheidender Bedeutung für die tägliche Arbeit. Wesentliche Ziele, an denen sich untergeordnete Kriterien orientieren sollten, sind deshalb

**Lesbarkeit** Code wird nach verschiedenen Untersuchungen etwa 5-30mal so oft gelesen wie geschrieben! Deshalb so schreiben, dass er schnell und flüssig gelesen und sofort verstanden werden kann. Alle Maßnahmen, die helfen, dass nicht so viel Code gelesen werden muss, sollten ergriffen werden.

**Erweiterbarkeit** Erweiterungen sind unser täglich Brot. Von daher ist es im eigenen Interesse, dies so einfach wie möglich zu gestalten.

**Portabilität** Neue Rechnersysteme werden in kurzen Zeitabständen in Betrieb genommen, die zu Beginn in der Regel relativ gut zugänglich sind. Standardkonforme Programme sind sehr schnell ans Laufen zu bekommen.

**Wiederverwendbarkeit und Wiederverwendung** Viele Aufgaben fallen immer wieder an. Bibliotheken, die in mehreren Programmen verwendet werden, werden auch mehrfach unter verschiedenen Randbedingungen getestet und sind daher zuverlässiger – ganz abgesehen von der gesparten Arbeit.

**Testbarkeit** Klar separierte Module mit spezifischen Aufgaben sind leichter zu testen als große monolithische Blöcke und damit nicht nur leichter zu erweitern und wiederzuverwenden, sondern auch schneller fehlerfrei zu bekommen.

Coding Standards oder Konventionen helfen, indem sie Kriterien an die Hand geben, um Entscheidungen zwischen (nahezu) äquivalenten Alternativen zu treffen. Oft gibt es dafür allenfalls schwache



Die Erarbeitung eines Standards sollte im Konsens geschehen, damit niemand sich gegängelt fühlt. Manches hat sich im Laufe der Jahre in der Gruppe bereit herauskristallisiert und bewährt, für anderes hat sich noch kein Resultat ergeben, obwohl dies wünschenswert wäre. Das kann dann gemeinsam festgelegt werden. Im Standard behandelt werden sollten:

- ① Namen
- ② Formatierung
- ③ Design
- ④ Dokumentation
- ⑤ Tests

Es gibt zahlreiche Konventionen, die teilweise auch im Netz verfügbar sind. Als Ausgangspunkt und Diskussionsgrundlage sind diese sehr hilfreich, ersetzen aber nicht die eigene Auseinandersetzung damit. Ein großer Teil guter Konventionen ist sprachunabhängig und wird für jede verwendete Sprache durch einen individuellen Teil ergänzt, der sich konsistent in den sprachunabhängigen Teil einfügt.

Ganz wichtig ist der Erhalt eines gesunden Maßes an Freiheit innerhalb der festgelegten Regeln.

## Regel 0

Insbesondere gilt als einzige absolute Regel, dass sich für jede absolute Regel immer eine legitime Ausnahme finden lässt.

Der Sinn der Konvention besteht dann darin, sich *fast immer* daran zu halten und im Zweifelsfall nach reiflicher Überlegung bewusst zu entscheiden, warum es *gerade hier* angebracht ist, es anders zu handhaben – und dies am besten auch zu dokumentieren. Ansonsten gilt grundsätzlich

## Regel 1

Quellcode muss gut lesbar und unmittelbar verständlich sein, mit möglichst minimalem Kontextwissen.

Für das Verständnis wichtig ist vor allem auch die Ausdruckskraft des Quellcodes, durch präzise Benennung von Konzepten und klare Strukturen:

## Regel 2

Fasse Dich kurz und prägnant!

*Begründung:* Je weniger Quelltext vorhanden ist, desto weniger muss fehlerbereinigt, gepflegt und erweitert werden. Umständliche Lösungen für einfache Problem verwirren nur. Optimierungen sind fast immer fehl am Platz – der Compiler kann das in der Regel besser als der Programmierer, wenn die Architektur insgesamt stimmt. Auch hier gilt: Sage, was Du denkst, und tue, was Du sagst. Falls etwas übermäßig kompliziert erscheint, noch einmal darüber nachdenken, wie das Vorgehen besser strukturiert werden kann. Eine Definition (Klasse oder Funktion) sollte noch am Stück auf den Bildschirm passen, um überschaubar zu sein!

In die gleiche Richtung geht das DRY (Dont Repeat Yourself)-Prinzip:

## Regel 3

Tue es nur einmal, aber richtig!

*Begründung:* Statt immer weitere Spezialfälle via Copy-Paste zu klonen, erst einmal überlegen, warum ich hier eine neue Variante brauche, was die Unterschiede und Gemeinsamkeiten sind. Gemeinsames lässt sich dann auslagern in eigene Funktionen oder Klassen, notfalls Makros. Hat man schon mehrere Varianten, lohnt es sich, diese auf ihre gegenseitige Beziehung abzuklopfen und so weit wie möglich zu harmonisieren. Die Gemeinsamkeiten werden dann unter mehreren Randbedingungen benutzt und getestet, oft fallen auch weitere Spezialfälle bei der Verallgemeinerung quasi kostenlos heraus.

Die Wahl guter Namen ist das Herzstück beim Programmieren. Sie sollten

- aussagekräftig
- systematisch
- leicht zuzuordnen
- spezifisch

sein, um intuitiv verwendet werden zu können, ohne unbedingt alles nachsehen zu müssen (Regel 1). Dies betrifft die unterschiedlichen Namensräume von Dateien/Verzeichnissen, Modulen/Bibliotheken, Klassen und/oder Datentypen, Funktionen/Prozeduren, Objekten/Variablen in diversen Varianten, Konstanten und Makros.

## Regel N1

Quelldateien weisen ein sprachspezifisches Suffix auf (.f, .c, .cpp, .h, .py) und sind ansonsten nach der Einheit benannt, die sie definieren: Subsystem, Modul oder Klasse. Im Dateisystem wird einheitlich klein geschrieben, zur Wortseparation werden Unterstriche verwendet, Umlaute sind zu vermeiden.

*Begründung:* Suffixe werden von vielen Automatisierungswerkzeugen (make) und Editoren zur Klassifikation verwendet. Für jede Einheit sollte die Definition schnell und einfach zu finden sein. Manche Dateisysteme können Groß-/Kleinschreibung nicht zuverlässig unterscheiden (FAT, NTFS). Als Bezeichner sind ohnehin nur Buchstaben, Ziffern und der Unterstrich zulässig, so dass die Beschränkung auf diese Zeichen im Dateisystem sich fast automatisch ergibt. Die Unterstützung von Leerzeichen und Sonderzeichen durch Tools ist oft noch sehr lückenhaft (Automatisierungswerkzeuge, Backup, ...).

## Regel N2

Module und Bibliotheken werden ebenfalls klein geschrieben und bestehen in aller Regel aus einem einzigen Wort.  
Bibliotheksdeklarationen liegen in ihrem eigenen Namensraum.

*Begründung:* Namen von Bibliotheken und Modulen werden oft auch als Dateinamen(-teil) verwendet, so dass sich hier die Anlehnung an Dateien und Verzeichnisse aufdrängt. Die Nutzung von Namespaces beugt Kollisionen vor und ermöglicht eine natürlichere Wahl von gebräuchlichen Namen.

## Regel N3

Klassennamen bestehen aus einem Substantiv und beginnen mit einem Großbuchstaben. Bei zusammengesetzten Substantiven werden neue Worte mit einem Großbuchstaben begonnen. Dies gilt auch für Schablonen (Templates).

*Begründung:* Klassen repräsentieren eigenständige Einheiten und stehen für (mehr oder weniger) reale Objekte in der Welt. Damit wird ein abgeschlossenes Konzept definiert. Sind mehr als drei Worte notwendig, um den Zweck zu beschreiben, verzettelt sich die Klasse unter Umständen in zu vielen Verantwortlichkeiten.



## Regel N4

Datentypen, die keine Klassennamen sind (oft typedef's), werden klein geschrieben und mit dem Suffix `_type` versehen.

*Begründung:* Diese typedef's tauchen häufig in Templates auf oder kapseln compiler- oder systemspezifische Datentypen weg. Sie repräsentieren damit keine eigenständigen Konzepte. Ausnahmen: Definitionen äquivalent zur Standardbibliothek (iterator, pointer, reference...), die in dieser Hinsicht leider selbst inkonsistent ist (size\_type), und template-Spezialisierungen, die dann klassenartig gebraucht werden (typedef Array<3,double> Position).

## Regel N5

Namen von Funktionen innerhalb von Klassen (Methoden) beginnen mit einem kleingeschriebenen Verb, gegebenenfalls gefolgt von einem oder mehreren Substantiven, die jeweils groß beginnen. Akronyme werden mit einem einzelnen Großbuchstaben begonnen. Freie Funktionen sind möglichst auf Dateiebene lokal zu beschränken (static oder unbenannter Namespace).

*Begründung:* Funktionen führen in der Regel Aktionen aus, durch die Kleinschreibung unterscheiden sie sich von Konstruktoren. Binnenakronyme sind optisch schwer von nachfolgenden Worten zu trennen. Freie Funktionen im globalen Namensraum bergen die Gefahr von Namenskonflikten.

## Regel N6

Iteratorartige Containerzugriffe erfolgen über `begin/end`, gegebenenfalls mit angehängtem Substantiv (`beginCell`, `beginFace`), Anzahlen davon mit Präfix `num-` (`numCell`), binäre Auswertungen mit `isXxx` oder `hasXxx`. Zur Dateneingabe in Klassen wird `setXxx` verwendet, -ausgabe entsprechend mit `getXxx`. Werden Objekte vollständig übergeben (in der Regel als Zeiger und mit Speicherverantwortung), lauten die entsprechenden Funktionen `resetXxx` beziehungsweise `releaseXxx` (analog zu `auto_ptr` und `unique_ptr`).

*Begründung:* Diese Regel sorgt für eine einheitliche und systematische Benennung sehr häufig gebrauchter Funktionstypen. Da die Übergabe von Speicherverantwortung semantische Konsequenzen hat, ist es wichtig, diese anders zu benennen.

## Regel N7

Attribute in Klassen werden durch ein Präfix `mXxx` gekennzeichnet, Klassenvariablen (`static`) durch ein `sXxx`, danach geht es jeweils mit einem Großbuchstaben weiter. Lokale Variablen beginnen ohne Präfix mit einem Kleinbuchstaben, ähnlich wie Funktionen. Globale Variablen sind – sofern unbedingt erforderlich – groß begonnene Substantive (wie Klassen). Statische Dateivariablen bekommen ein `SXxx` vorangestellt.

Lokale Variablen mit *sehr eng* begrenztem Gültigkeitsbereich dürfen auch einfach heißen: `x`, `y`, `z` für Koordinaten, `i`, `j`, `k` für Schleifenzähler (Einzeiler). Ansonsten sind aussagekräftige Namen zu wählen und Variablen nicht für andere Zwecke zu recyceln.

*Begründung:* Mit diesen Regeln sind praktisch alle Variablen in ihrem Bezug und Kontext sofort unterscheidbar. Kurze Schleifen sind zumeist ausreichend überschaubar, um kurze Namen zu legitimieren, da sie Programmierern sehr geläufig sind.

## Regel N8

Parameter von Funktionsaufrufen bekommen ein iXxx, oXxx oder cXxx vorangestellt, je nachdem ob es sich um Eingabe-, Ausgabe- oder modifizierbare Parameter handelt. Ausnahme: Bei arithmetischen Operatoren sind lhs und rhs für die beiden Seiten gebräuchlich, bei Stream-Ein- und -Ausgabe in/out für den Stream.

*Begründung:* Operatoren sind ebenfalls überschaubar und sehr gängig mit einer oft weitestgehend offensichtlichen Implementierung.

## Regel N9

Benannte Konstanten werden je nach Kontext innerhalb von Klassen oder auf Dateiebene definiert. Sie bestehen durchgängig aus Großbuchstaben mit dem Unterstrich zur Worttrennung. Auf Dateiebene (außerhalb von Namensräumen) sind spezifische Namen zu verwenden, niemals NULL, EMPTY, ...

*Begründung:* Konstanten sind eine intelligentere Form von Literalen und somit von Variablen zu unterscheiden. Der Namensraum auf Dateiebene ist global, die Gefahr von Kollisionen somit sehr groß.

## Regel N10

Präprozessormakros werden durchgängig groß geschrieben mit Unterstrich. In Bibliotheken sind sie äußerst vorsichtig einzusetzen, mit einem Präfix aus dem jeweiligen Bibliotheksnamen (IAG\_XXX). Makroargumente sind ebenfalls groß, oft einbuchstabig oder mit angehängtem Index (X1, X2, X3).

*Begründung:* Makros bilden ihren eigenen Namensraum und haben ganz spezielle Probleme, auf die später noch eingegangen wird.

Nach Regel 1 ist die Lesbarkeit des Quelltextes von oberster Priorität. Eine konsistente und logische Formatierung ist daher eine ausgezeichnete Unterstützung beim strukturellen Verständnis. Dazu zählen

- Positionierung von geschweifte Klammern (oder anderen Blockgrenzen)
- Einrückungen von untergeordneten Strukturen
- Anordnung von Kontrollstrukturen
- Zeilenumbrüche zur besseren Unterteilung
- Leerzeichen zur optischen Trennung



## Regel F1

Anweisungsblöcke sind in der Regel einer Kontrollstruktur untergeordnet (if, for, while, ...). Die öffnende Klammer ist – nach einem Leerzeichen – am Ende der übergeordneten Zeile angeordnet, die schließende Klammer auf einer eigenen Zeile, letztere spaltenweise ausgerichtet an der übergeordneten Struktur. Bei sehr langen Blöcken ist ein Kommentar zur logischen Zugehörigkeit angebracht, insbesondere bei mehrfacher Schachtelung.

Nicht zugeordnete Blöcke beginnen auf einer eigenen Zeile, verbunden mit einem Kommentar.

*Begründung:* Der Blockbeginn ist unmittelbar an die Kontrollstruktur gebunden und daher gut auf der gleichen Zeile aufgehoben. Das spart eine ansonsten leere Zeile, das Leerzeichen trennt optisch. Das Blockende gehört weder zum Blockinhalt noch zur nächsten Anweisung, es gibt daher keinen logischen Platz außer einer eigenen Zeile. Die Ausrichtung kennzeichnet deutlich die Zugehörigkeit zum Beginn.

## Regel F2

Untergeordnete Blöcke werden jeweils eingerückt, und zwar mit einem Tabulator. Die Standardeinstellung im Editor beträgt vier Leerzeichen für einen Tabulator.

*Begründung:* Die Einrückung macht die strukturelle Zugehörigkeit deutlich. Mit einem einzelnen Tabulator wird eine 1:1-Relation zwischen Repräsentation (Einrückung) und Bedeutung (Unterordnung) hergestellt. Vier Leerzeichen haben sich als bester Kompromiss zwischen Sichtbarkeit und Praktikabilität bei tiefer Schachtelung bewährt. Einzelne Teammitglieder können notfalls einen anderen Wert nach Geschmack einstellen.

Der Verzicht auf triviale Blöcke erhöht die Lesbarkeit einfachster Strukturen und spart eine ansonsten weitgehend leere Zeile.

## Regel F3

Der Körper von Funktionen wird ebenfalls eingerückt wie ein Block. Ist die Parameterliste zu lang für eine Zeile, wird nach einem Komma umgebrochen und der Rest *zwei* Ebenen eingerückt.

*Begründung:* Der Körper ist auch ein Block, damit gelten die gleichen Regeln. Die zusätzliche Einrückung der weiteren Parameterzeile(n) trennt diese optisch vom eigentlichen Körper der Funktion.

```
void Restart::fillCellData(std::istream & iFile,
    Flux const * iFlux, Basis const * iBasis,
    IAG::Stride iBasisStride, double * oCell) {
    for (int d=0; d<iFlux->dofPerState(); ++d) {
        for (int b=0; b<iBasis->numBases(); ++b, ++oCell)
            iFile >> *oCell;
        oCell+=iBasisStride;
    }
}
```

## Regel F4

Die Anordnung der untergeordneten Blöcke von Bedingungs- und Schleifenanweisungen wurde bereits angesprochen. Kurze, einzeilige Anweisungen nach if/for/while/... dürfen ohne Block geschrieben werden, sind aber (falls in einer eigenen Zeile) ebenfalls einzurücken. Niemals ohne Block schachteln!

else oder else if steht in einer eigenen Zeile unter der vorherigen schließenden Klammer und gleich ausgerichtet, mit der neuen öffnenden hinten dran.

Leere Schleifen haben die leere Anweisung immer auf einer eigenen Zeile und sind kommentiert.

Bei switch sind die case-Labels im untergeordneten Block regulär, also einfach einzurücken, die zugeordnete Anweisung(en), soweit sie nicht mehr in die gleiche Zeile passen, ein weiteres Mal.

*Begründung:* Die Einrückungen machen die Struktur klarer. Einfache Dinge sollten aber auch einfach auszudrücken sein. Schachteln (vor allem mit if) ohne Blöcke birgt eine große Gefahr der falschen Zuordnung, genau wie leere Schleifen, so dass hier extra Aufmerksamkeit gefordert ist.

## Regel F5

Lange Zeilen werden nach 120, spätestens nach 160 Zeichen umgebrochen. Dies erfolgt nach einem Komma (Funktionsaufruf) oder vor einem Operator. Komplexe Ausdrücke können durch Zwischenvariablen aufgeteilt werden. Vor einer neuen Klassen- oder Funktionsdefinition befindet sich eine Leerzeile, innerhalb nur zwischen größeren, voneinander zu trennenden Abschnitten.

Nur eine Anweisung pro Zeile. Ausnahme: Triviale, zusammengehörige Anweisungen dürfen notfalls auf die gleiche Zeile (`startCell=0; finalCell=numCells;`), `break` kann im `switch` ebenfalls hinten angehängt werden.

*Begründung:* Auch breite Monitore sind endlich, und mancher möchte auch gerne sinnvoll am Notebook arbeiten können, ohne zu scrollen. Der Operator am Anfang zeigt die umgebrochene Zeile deutlich an, macht sie für sich genommen syntaktisch unzulässig und beugt somit Fehlern beim Verschieben vor. Sparsamer Gebrauch von Leerzeilen verringert überflüssiges Scrolling in der Vertikalen – genauso wie der Verzicht auf die öffnende Klammer in einer separaten Zeile.

## Regel F6

Ein Leerzeichen gehört jeweils vor den Blockbeginn { und nach Schlüsselwörtern wie for/if/while, dort vor der öffnenden Klammer (. Funktionsaufrufe werden hingegen unmittelbar von der Klammer gefolgt, nach jedem Komma zur Parametertrennung ist ein Leerzeichen zu setzen. Deklaratoren wie \* und & für Zeiger und Referenzen werden beidseitig durch Leerzeichen abgesetzt. Ausdrücke werden in der Regel ohne Leerzeichen geschrieben (außerhalb von Funktionsaufrufen), abgesehen vom ?:-Operator, wo sowohl ? als auch : jeweils beidseitig ein Leerzeichen haben.

*Begründung:* Leerzeichen werden strategisch gesetzt, um Verschiedenes voneinander zu trennen und Gemeinsames zu verbinden. Die Kommaregel orientiert sich an der gewohnten Rechtschreibung, mit Vorsicht zu gebrauchende Zeichen sollen klar heraustreten.

Auch auf der Ebene der Verwendung der jeweiligen Programmiersprache gibt es natürlich viele Möglichkeiten, ein spezifisches Problem zu lösen. Gerade für Anfänger ist es oft schwer abzusehen, welche Vorgehensweise überhaupt denkbar wäre und welche davon vielleicht an dieser Stelle geschickt einzusetzen. Hier hilft eine kleine Sammlung von Best-Practice Hinweisen, die sich ganz allgemein empfehlen. Dieser Teil kann natürlich mit zunehmender Erfahrung immer weiter ergänzt werden, um neu hinzukommenden Teammitgliedern Unterstützung zu bieten, gegebenenfalls mit Hinweisen auf konkrete Anwendungen in Projekten.

Definitionsgemäß sind viele dieser Richtlinien sprachspezifisch. Einige generelle Tipps können allerdings sprachunabhängig gegeben werden.

## Regel P1

„Magische Zahlen“ tauchen nicht einfach im Quelltext auf sondern werden als benannte Konstanten definiert und systematisch benutzt. Je nach Kontext werden sie innerhalb von Klassen oder auf Dateiebene definiert. Sie bestehen durchgängig aus Großbuchstaben mit dem Unterstrich zur Worttrennung.

Ausnahme: Für Ausdrücke wie  $0.5^*$  (Mittelwertbildung) oder  $+1$  (nächster) sind keine Konstanten sinnvoll und notwendig, für `+GHOST_CELLS` natürlich schon (selbst wenn `GHOST_CELLS` gerade 1 ist).

*Begründung:* Nackte Zahlen sind bis auf die genannten offensichtlichen Ausnahmen schwer verständlich. Bei Erweiterungen ist die Chance 100%, dass mindestens eine der Zahlen vergessen wird. Bei benannten Konstanten genügt es, diese zu ändern (und die restlichen Fehler zu suchen).



## Regel P2

Ist eine exakte Größe für eine Variable oder Konstante erforderlich (beispielsweise für I/O), arbeitet man mit compilerspezifischen Typen (int64, float32) definierter Größe, die portabel sind oder zumindest an einer einzelnen Stelle so portabel wie möglich definiert werden. Bei integralen Konstanten mit logischem Zusammenhang (beispielsweise als Diskriminator von Fällen) eignen sich Aufzählungen am besten. Ist die Größe wichtig, dann gibt es – wie für Variablen auch – `static int32 const XXX` (oder `float64, ...`).

*Begründung:* Ist ein Austausch mit anderen Einheiten notwendig (beispielsweise auch bei einer Anwendung aus verschiedenen Sprachen), so muss die genaue Größe bekannt und festgelegt sein. Dies gilt nicht nur für Variablen, sondern natürlich auch für Konstanten, falls sie in dieser Art benutzt werden. Ansonsten sind für integrale Konstanten Aufzählungen besser, da sie auch einen eigenen Typ definieren und damit die Verwechslungsgefahr minimieren.

## Regel P3

Präprozessormakros werden *ausschließlich* dann eingesetzt, wenn es keine bessere Alternative innerhalb der Sprache gibt (Konstanten, inline-Funktionen, Templates). Wie Konstanten werden sie durchgängig groß geschrieben mit Unterstrich. In Bibliotheken sind sie noch vorsichtiger einzusetzen, mit einem Präfix aus dem jeweiligen Bibliotheksnamen (IAG\_XXX). Wo möglich, folgt nach Verwendung gleich ein `#undef`. Makroargumente sind ebenfalls groß, oft einbuchstabig oder mit angehängtem Index (X1, X2, X3). In Ausdrücken werden Argumente generell geklammert, genau wie gegebenenfalls das Resultat. Makros, die zu einzelnen Statements expandieren, sind ohne abschließendes Semikolon zu definieren, komplexere Ausdrücke (gerade auch mit `if`) werden in `do { XXX } while (false)` eingepackt.

*Begründung:* Der Präprozessor ist gleichermaßen mächtig wie dumm, da er außerhalb der Sprache operiert. Makros sind sehr schwer zu debuggen, da sich die Fehlermeldung des Compilers auf die nicht im Quelltext sichtbare expandierte Fassung bezieht. Auf der anderen Seite können mit Makros sich wiederholende Quelltextteile generiert und so Copy-Paste-Fehler vermieden werden.

Die Dokumentation eines Projektes richtet sich an verschiedene Benutzer im weitesten Sinne. Das sind zunächst die Anwender, die wissen möchten, an welchen Stellen sie welche Eingaben zu machen haben, vielleicht noch wie die zugrundeliegende Theorie aussieht.

### Regel D1

Für Dokumentation gibt es im Projekt ein Verzeichnis doc. Ein Anwendungshandbuch beschreibt dort die Bedienung des Programmes, notwendige Eingabegrößen, Ausgabeformate und die zugrundeliegende Theorie.

Bei Bedarf gibt es ein weiteres Dokument, in dem die Architektur des Gesamtsystems beschrieben wird, einschließlich Diagrammen und Begründungen für Designentscheidungen. Beide werden in einem einfachen Textformat geschrieben (L<sup>A</sup>T<sub>E</sub>X, L<sup>A</sup>X, docbook, XML).

*Begründung:* Anwendungs- und Systemhandbuch sind nur relativ lose an die Implementierung gebunden und bestehen im wesentlichen aus Prosa. Ein mergebares Textformat ermöglicht gemeinsames Arbeiten mit (meist) automatischer und nachvollziehbarer Konfliktbehandlung.

Dann gibt es die Teammitglieder, die am Quelltext arbeiten. Auch die teilen sich in zwei Gruppen, nämlich jene, die ein Modul, eine Teilbibliothek nur nutzen möchten und deshalb nur die Schnittstelle zu kennen brauchen, und auf der anderen Seite jene, die solche Teile implementieren und erweitern müssen.

### Regel D2

Der Quelltext wird durch strukturierte Kommentare mit doxygen dokumentiert. Für Benutzer von Teilen (Module, Bibliotheken) wird die Schnittstelle in den entsprechenden Headerdateien spezifiziert. Die eigentliche Implementierung wird dann in den Quelldateien genauer beschrieben.

*Begründung:* Erfahrungsgemäß wird Dokumentation (besonders für den Quelltext) um so weniger gepflegt, je weiter sie vom Code entfernt ist. Die einzige praktikable Möglichkeit ist deshalb, diese im Quelltext selbst direkt an der entsprechenden Stelle zu haben – und das ist schon schwer genug.

## Regel D3

In strukturierten doxygen-Kommentaren werden alle Klassen und Funktionen dokumentiert, deren Aufgabe nicht anhand der Namen offensichtlich ist. „Offensichtlich“ ist dabei relativ eng auszulegen. Weitere Kommentare sind sparsam einzubringen, im Zweifelsfall ist eher die Schnittstelle oder der Code zu vereinfachen als durch einen Kommentar zu erklären.

Implementierungsdetails oder Designentscheidungen lokaler Natur (beispielsweise deque statt vector) sind ebenfalls so zu dokumentieren, allerdings auf einer konzeptionellen Ebene, die mutmaßlich mehrere Revisionen überdauern wird.

*Begründung:* Kommentare sind wertvoll und problematisch gleichermaßen. Wertvoll, weil sie Dinge erklären können, die nicht im Quellcode ausdrückbar sind. Problematisch, weil sie bei der kontinuierlichen Entwicklung zwangsläufig schnell veralten. Nur der Code selbst ist definitionsgemäß aktuell, was die Bedeutung von guten Namen, klarer Struktur und definierten Verantwortlichkeiten unterstreicht.

Auch Kommentare müssen gelesen werden und sollten deshalb einen entsprechenden Informationsgehalt haben. Niemand will sowas lesen

## Regel D4

Strukturierte doxygen-Kommentare beginnen mit `///`. Für die meisten Funktions- und Parametererklärungen sollte eine einzelne Zeile genügen. Längere Dokumentationen sind oft für Module und Klassen angebracht.

Funktionsparameter, Aufzählungskonstanten und Attribute werden in der Regel am Zeilenende mittels `///  
[in]`, `///  
[out]` oder `///  
[in,out]` dokumentiert, bei Funktionsparametern kann via `///  
[in]`, `///  
[out]` oder `///  
[in,out]` die Richtung der Übergabe angegeben werden.

*Begründung:* doxygen bietet viele Möglichkeiten der Markierung von Kommentaren an, eine einheitliche Verwendung ist aber wünschenswert. Der dokumentierende Kommentare sollte räumlich so nah wie möglich am übersetzten Element positioniert sein. Die explizite Angabe der Übergaberichtung ist in der Regel nicht erforderlich, da bereits durch das Präfix (iXxx, oXxx, cXxx) erkennbar.

Grundsätzlich sind aussagekräftige Namen einer notwendigen Erklärung vorzuziehen. Von daher ist es optimal, wenn keine oder kaum mehr doxygen-Kommentare mehr notwendig sind.

## Regel D5

Strukturelle doxygen-Tags sind überflüssig, wenn der Kommentar unmittelbar vor (`/// Erklärung) oder nach (///< Erklärung) dem zu dokumentierenden Element steht, was auch im Sinne der Konsistenz durchaus sinnvoll ist. Zusammengehörige Klassenfunktionen oder Attribute können mit @{ und @} gruppiert werden, beispielsweise ein beginXxx/endXxx/numXxx Trio, oder entsprechend hasXxx/resetXxx/releaseXxx.`

Veraltete Schnittstellen erhalten ein `/// \deprecated, problematische oder nur partiell funktionierende Abschnitte ein \warning beziehungsweise \todo. Implementierungsdetails werden hinter \internal beschrieben, \bug hält Fehler fest, bis sie hoffentlich behoben werden.`

*Begründung:* Obwohl doxygen zahlreiche Auszeichnungen bereit stellt, ist es sinnvoll, sich auf einen beschränkten Satz zu konzentrieren, der die wesentlichen benötigten Funktionen enthält.

Eine Sammlung guter Tests ist sehr wertvoll für die kontinuierliche Entwicklung, da sie gewährleistet, dass erreichte Funktionalität auch bei notwendig gewordenen größeren Umbaumaßnahmen nicht wieder zerstört wird. Sinnvollerweise finden diese auf mehreren Ebenen statt:

- Unittest für einzelne Funktionen
- Modultests für größere Komponenten
- Systemtests für die gesamte Anwendung mit realistischen Testdaten.



## Regel T1

Jedes Projekt enthält eine Sammlung von Tests mit unterschiedlichem Kontext. Diese können automatisiert durch ein einzelnes Kommando abgearbeitet werden. `make check` übersetzt und arbeitet alle Unit- und Modultests ab, für die Systemtests stehen die `make`-Ziele `fastTest`, `mediumTest`, `longTest` und `fullTest` zur Verfügung.

*Begründung:* Automatisierte Testmatrizen mit einem einzelnen Kommando sind notwendig, um die Durchführung der Tests zu erleichtern. Die Unit- und Modultests sollten nicht wesentlich länger als das Übersetzen dauern. `make fastTest` sollte bereits einen signifikanten Teil der Funktionalität abdecken, wenn auch nicht notwendig in allen möglichen Kombinationen, und weniger als eine Viertelstunde dauern. `mediumTest` könnte über die Mittagspause laufen, `longTest` einige Stunden in der Nacht. `fullTest` umfasst dann alle automatisierten Tests und wird beispielsweise einmalig vor Auslieferung einer neuen Version abgefahren – beispielsweise am

## Regel T2

Als Rahmengerüst für Unittests wird die entsprechende Funktionalität aus der iaglib verwendet. Jede Funktion, die ohne Aufbau größerer Datenstrukturen aufrufbar ist, erhält einen Testfall. Enthalten sind darin mehrere Test mit verschiedenen Parametern, insbesondere grenzwertigen und gegebenenfalls auch ungünstigen. Alle Unittests eines Moduls liegen in einem Unterverzeichnis tests.

*Begründung:* Die Basisfunktionalität sollte sorgfältig getestet sein, damit die darauf aufbauenden Funktionalitäten auf einem soliden Fundament stehen. Ein Framework für die Unittests vereinfacht wiederkehrende Aufgaben und sorgt für eine konsistente Ausführung aller Tests.

Oft ist es nicht ungeschickt, zuerst die Unittests zu schreiben (die logischerweise zunächst fehlschlagen) und dann die Implementierung anzugehen (Test driven design).

## Regel T3

Außer den Unittests liegen im Unterverzeichnis tests auch noch Modultests. Diese erproben größere zusammenhängende Funktionalitäten eines Subsystems, beispielsweise das Einlesen eines Gitters. Modultests werden nach den entsprechenden Unittests ausgeführt.

*Begründung:* Modultests sind in unseren Anwendungen meist relativ schwierig zu schreiben, da sie auf komplexe Datenstrukturen angewiesen sind, die zuerst aufgebaut werden müssten, beispielsweise Gitter und Diskretisierungen. Für einzelne Subsysteme kann die Aufgabe jedoch einigermaßen separiert werden, und dies sollte dann auch genutzt werden.

## Regel T4

Systemtest liegen im Unterverzeichnis tests des Projektes beziehungsweise dessen Unterverzeichnissen. fast, medium, long und full bieten sich dafür an. Für jede spezifische Funktionalität wird ein expliziter Testfall generiert, der auf einem kleinen Problem diese ausführt. Die meisten davon sollten in fast einsortiert werden können. Kombinationen von Funktionalitäten und größere Probleme werden nach Bedarf in den längeren Ordnern untergebracht.

*Begründung:* Die Notwendigkeit unterschiedlicher Testintensitäten wurde bereits angesprochen. Wichtig ist, dass selbst der schnellste Test bereits relativ belastbare Resultate liefert und eine entsprechend hohe Testabdeckung aufweist. Bei größeren Änderungen kann es durchaus auch notwendig werden, Testfälle nach oben oder unten zu verschieben.

## Regel T5

Die Testabdeckung der Unittests sollte über 50% liegen, die von fastTest über 75%. fullTest sollte 95% übersteigen. In allen Fällen ist expliziter Fehlerbehandlungscode (fehlende Dateien, Speichermangel, falsche Eingabedateien) von der Zählung ausgenommen. Die Testabdeckung wird nach größeren Änderungen oder Ergänzungen neu überprüft.

*Begründung:* Auch häufig durchgeführte Tests helfen nur dann wirklich gut, wenn der ganze Code getestet wird. Dies sollte kontinuierlich gewährleistet bleiben.

## Regel T6

Ausführungshäufigkeit und -geschwindigkeit werden ebenfalls nach größeren Änderungen oder Ergänzungen anhand von repräsentativen Datensätzen (beispielsweise einer der Testmatrizen) neu überprüft (Profiling).

*Begründung:* Ein regelmäßiges Profiling verschafft mit der Zeit ein gutes Gefühl, wo Geschwindigkeit liegen bleibt und Handlungsbedarf besteht – gegebenenfalls auch durch strukturelle Änderungen an der Architektur. Besonders hier zeigt sich dann der Wert einer zuverlässigen Testmatrix.

## 10 Dokumentation I: Code – doxygen

Das Dokumentationswerkzeug doxygen wurde bereits mehrfach angesprochen. Es extrahiert speziell formatierte Kommentare im Quelltext und formatiert diese zusammen mit dem Quelltext in ansprechender Form für unterschiedlicher Dateiformate, unter anderem  $\text{\LaTeX}$  (und damit PDF) und HTML. Dabei werden beispielsweise – sofern technisch möglich – Querverweise zwischen einzelnen Elementen verlinkt, diverse Verzeichnisse erstellt, Aufrufgraphen erzeugt und Beziehungsdiagramme generiert. Es existieren auch noch einige Alternativen, diese beschränken sich jedoch oft auf eine einzelne Sprache (javadoc, Qt) oder haben einen wesentlich geringeren Funktionsumfang. doxygen genießt deshalb eine sehr weite Verbreitung und wird für zahllose – auch kommerzielle – Projekte eingesetzt.

Strukturierte Kommentare werden von doxygen in unterschiedlicher Form unterstützt. Sie alle zeichnen sich durch ein zusätzliches Zeichen nach dem Kommentarbeginn aus:

**///*Dokumentation*** Drei Schrägstriche leiten einen von doxygen interpretierten Kommentar ein.

**///*Dokumentation*** Alternativ tut es auch ein Ausrufezeichen.

**/\*\**Dokumentation*** Ein C-Kommentar mit einem zusätzlichen Stern geht auch.

**/\*!*Dokumentation*** In diesem Fall ist das Ausrufezeichen gleichermaßen möglich.

Normalerweise bezieht sich ein doxygen-Kommentar auf das nachfolgende Element. Meint man das Element vor dem Kommentar (beispielsweise wenn der Kommentar am Zeilenende steht), ist zusätzlich zu den genannten drei Zeichen noch ein Kleinerzeichen < anzuhängen, zusammen also dann ///**<**, ///**<**, /\*\*< oder /\*!**<**.



Einige der doxygen-Tags wurden bereits in Regel D4 vorgestellt. Darüber hinaus gibt es noch zahlreiche weitere, beispielsweise falls die Dokumentation vom Quelltext getrennt werden soll (was wahrscheinlich keine gute Idee ist). In diesem Fall müsste immer spezifiziert werden, auf welches Element sich die Dokumentation bezieht und damit der Inhalt des Quellcodes dupliziert werden (und dann mit gepflegt). Andere Möglichkeiten bestehen in der Integration von zusätzlichen Diagrammen zur besseren Verdeutlichung (automatisch mit `\callgraph` beziehungsweise `\callergraph`, oder manuell mit `\dot` oder `\msc`), aber auch  $\text{\LaTeX}$ -Formeln können eingebaut werden: eingeschlossen in `\f$` für Formeln in der Zeile oder in `\f[` und `\f]` für abgesetzte Formeln. Sinnvoll können auch noch kleine Beispiele sein, die zwischen `\code` und `\endcode` stehen können.

Für die optische Gestaltung stehen auch noch jede Menge Möglichkeiten zur Verfügung, angefangen von Kursiv- (`\em`) und Fettdruck (`\b`) über Schreibmaschinenschrift (`\c`) bis hin zu schachtelbaren Listen, entweder nur mit Markierungspunkten (`\item`) oder mit Markierungspunkten und Indizes (`\item[i]`).

Zur Darstellung des Quelltextes analysiert doxygen diesen entsprechend den Regeln der jeweiligen Sprache. Da doxygen kein „richtiger“ Compiler ist, gelingt dies mehr oder minder gut, insbesondere bedingt dadurch, dass doxygen keine Zuordnung von Bezeichnern zu unterschiedlichen Namensräumen treffen kann. Des weiteren bilden beispielsweise C++-Templates ja eine eigene Sprache für sich, die von doxygen ebenfalls nicht ausgeführt wird. Im allgemeinen funktioniert der Parser jedoch relativ gut, nur bei der Zuordnung gleicher Namen zu unterschiedlichen Modulen braucht er gelegentlich etwas Unterstützung.

Gerade wenn man an der einen oder anderen Stelle dann doch mit Makros arbeiten muss, ist es hilfreich, dass doxygen auch einen eigenen Präprozessor mitbringt, dem genau mitgeteilt werden kann, welche Makros expandiert werden sollen und welche besser im Quelltext stehen bleiben. Die genaue Einstellung ist da zwar manchmal etwas aufwändig, aber das Ergebnis rechtfertigt den Einsatz.

Gesteuert wird doxygen über eine Eingabedatei, die das Ausgabeformat, die zu lesenden Dateien und zahllose andere Möglichkeiten festlegt. Wer das nicht per Hand eingeben möchte (doxygen kann eine dann individuell einstellbare Schablone ausgeben), bedient sich der grafischen Benutzeroberfläche doxywizard, die (fast) alle Einstellmöglichkeiten einigermaßen übersichtlich darstellt und die jeweils möglichen Optionen anbietet. Dazu zählen natürlich die zu erstellenden Ausgabeformate, aber auch globale Informationen wie der Projektname, gegebenenfalls Logos oder Über- und Unterschriften. Des weiteren finden sich Einstellungen zur (Quell- und Dokumentations-) Sprache, zum angesprochenen Präprozessor sowie die Definition der abzusuchenden Quellen und der daraus zu generierenden Information. In der Regel wird man wohl vor allem verlinktes HTML erzeugen wollen, bei großen Projekten ist ein PDF für den Ausdruck nur noch bedingt praktikabel.

Ein Beispiel aus CGNS++:

```
/// Possible data types of the ADF node
enum DataType {
    MT, ///< No data
    I4, ///< Signed integer, 4 bytes
    I8, ///< Signed integer, 8 bytes
    U4, ///< Unsigned integer, 4 bytes
```

...

```
/// Iterator to the first child.
Child_iterator_t beginChild() const;
/// Iterator one past the last child.
Child_iterator_t endChild() const;
/// Ask if there is a child of this name.
bool hasChild(std::string const & iChildName) const;
/// Ask if the given node is a child of this node.
bool isChild(Node const & iChildNode) const;
```

## enum ADF::DataType

Possible data types of the node.

### Enumerator:

<i>MT</i>	No data.
<i>I4</i>	Signed integer, 4 bytes.
<i>I8</i>	Signed integer, 8 bytes.
<i>U4</i>	Unsigned integer, 4 bytes.

### Public Member Functions

**Child\_iterator\_t** **beginChild** () const

Gets an iterator to the first child.

**Child\_iterator\_t** **endChild** () const

Gets an iterator one past the last child.

**bool** **hasChild** (std::string const &iChildName) const

Ask if there is a named child.

**bool** **isChild** (Node const &iChildNode) const

Ask if this is a child node.

## 11 Codebereinigung und Konsolidierung

- Nach Entwicklungsetappen wieder Ordnung schaffen!
- Zwischendurch darf gemogelt werden, hinterher wird aufgeräumt.
- Ansonsten geht Erreichtes schnell verloren:
  - Übersichtlichkeit
  - Wartbarkeit
  - Erweiterbarkeit
  - Allgemeingültigkeit
- Schlimmer: Fehler werden nicht erkannt!

1. Hat man einmal den Status eines weitgehend fehlerfreien, gut wartbaren und erweiterbaren Quellcodes erreicht, so ist es natürlich äußerst wünschenswert, diesen Zustand möglichst beizubehalten oder gar noch zu optimieren. Andererseits gibt es manchmal Situationen, wo man einfach zusehen muss, zu einem festen Termin noch eine gewisse Funktionalität oder eine Fehlerkorrektur einfach fertig zu bekommen – zumindest für den benötigten Anwendungsfall. Da ist es dann durchaus legitim, *vorübergehend* gewisse Kompromisse in Bezug auf Übersichtlichkeit, Allgemeingültigkeit und Flexibilität einzugehen. Die Betonung liegt hier auf „vorübergehend“. Es ist ganz wichtig, nach solchen Stressphasen wieder Ordnung zu machen, einen Überblick über das Erreichte zu gewinnen und aufzuräumen.

Unter Refactoring wird eine Umarbeitung des Quellcodes ohne Veränderung der Funktionalität verstanden. Dazu gehören

- Vereinheitlichung** der Darstellung (Leerzeichen, Zeilenenden, ...),
- Umbenennung** von Variablen, Klassen, Funktionen, Dateien,
- Verschiebung** von Funktionen oder Teilen davon in der Klassenhierarchie oder zwischen Modulen,
- Verallgemeinerung** und Zusammenführen von (geklonten) Varianten.

Besonders wichtig ist hierbei eine gute Testmatrix, um sicherzustellen, dass tatsächlich die Funktion erhalten bleibt. Technisch gesehen kein Refactoring, aber trotzdem in dieser Phase wichtig:

- Dokumentation** der erzielten Funktionalität.



- Konsolidierung des Neuen mit Abstand
  - zeitlich
  - persönlich (Kollege)
- Verbesserungsmöglichkeiten
  - Einschränkungen
  - Ineffizienz
  - Vereinfachung
  - Fehlerbereinigung
- Dokumentation von Funktionalität und Entscheidungen
- Netto (mit Nachfolgeaktion) ergibt sich Zeitersparnis!

1. Auch wenn die nächste Erweiterung noch so dringend erscheint, ist es wichtig, das Erreichte erst einmal zu festigen. Dazu ist es günstig, mit einem gewissen zeitlichen und geistigen Abstand noch einmal über den Code zu schauen – im Idealfall auch noch von einem Kollegen. Erfahrungsgemäß finden sich dabei noch einige unnötige Einschränkungen, ineffiziente Implementierungen, nicht (mehr) erforderliche Komplexitäten und schlichte Fehler. Nicht zuletzt wächst dabei auch das Verständnis, weshalb an dieser Stelle gerade jene Möglichkeit gewählt wurde, und dass die eigentlich offensichtliche Alternative hier unter Umständen unbrauchbar gewesen wäre.

Abgesehen davon tut es dem Code einfach auf Dauer gut, gelegentlich überarbeitet zu werden, ohne den Druck, etwas Neues einbauen zu müssen. Die dafür investierte Zeit wird zumeist vielfach bei der nächsten Erweiterung eingespart, die dann auf einer soliden Basis viel einfacher zu erzielen ist. Natürlich darf dies nicht so weit gehen, dass man sich längere Zeit nur mit sich selbst beschäftigt, letztlich muss man da ein gesundes Maß finden: Beim Durchsehen sollten keine offenkundigen und einfach zu behebenden Mängel mehr sichtbar sein.

- Regelgerechte Setzung von Leerzeichen und Zeilenschaltungen
- Vorsicht bei Teamentwicklungen:
  - Zahlreiche marginale Änderungen überall
  - Viele unnötige Konflikte (zeilenweises Merging)
  - Branches helfen nur bedingt
  - Idealerweise in ansonsten inaktiven Phasen
- Wiederherstellung der Übersichtlichkeit (Quelltest wird oft gelesen) ist es wert!

1. Das Vereinheitlichen des Quelltextes in Bezug auf Leerzeichen und Zeilenenden ist zwar technisch nicht notwendig und nur mäßig spannend, kann aber dafür auch einmal ohne große Probleme in der Mittagsschlafzeit nach dem Essen ;- ) durchgeführt werden. Trotzdem ist es sinnvoll, auch hier einige Sorgfalt walten zu lassen, um die Übersichtlichkeit zu erhalten oder wiederherzustellen. In der Regel wird man dafür aber keinen separaten Durchgang vorsehen, sondern die einzelnen Schritte mehr oder minder gleichzeitig durchführen. Dabei kann man dann bei Bedarf hier und da noch ein Leerzeichen einfügen oder wegnehmen und die Zeilenumbrüche anders anordnen. Da hier oft größere Mengen an Quelltext modifiziert werden, bietet es sich an, diese Veränderungen zu einer Zeit durchzuführen, in der andere Teammitglieder wenig aktiv sind. Ansonsten würden deren Erweiterungen unter Umständen erheblich gestört, wenn immer wieder viele Zeilen geändert werden, in denen nur kleine Verschiebungen stattgefunden haben. Versionskontrollsysteme arbeiten zeilenweise, so dass dann unnötig viele Konflikte erzeugt würden, die manuell aufgelöst werden müssen.

- Regelgerechte Namen von Variablen, Klassen, Funktionen und Dateien
- Potenziell ähnlich störend im Team wie Leerzeichen und Zeilenenden, aber lokaler
- Integriere Entwicklungsumgebungen unterstützen
  - Vorsicht bei gängigen Bezeichnern
- Namen an Funktion anpassen

1. Es bietet sich hier auch an, bessere Namen einzuführen. Zum einen natürlich solche, die den einmal vereinbarten Konventionen entsprechen (Regeln N1-N9), zum anderen aber auch den jeweiligen Zweck treffender wiedergeben. Unter Umständen hat sich im Laufe der Entwicklung die Verwendung geändert, manche Variablen werden gar nicht mehr benötigt, oder Ausdrücke sind komplex genug geworden, um ein Zwischenergebnis in einer Variablen unterzubringen. Dies betrifft natürlich nicht nur lokale Variablen, sondern besonders auch Funktionen und Methoden. Einige Entwicklungsumgebungen bieten hierfür spezielle Unterstützung an, sind allerdings durch die fehlende semantische Analyse des Quelltextes in der korrekten Zuordnung von Bezeichnern zu Namensräumen etwas eingeschränkt. Die Werkzeuge funktionieren dann gut, solange Namen wirklich überall eindeutig sind. Sobald es übereinstimmende Namen in unterschiedlichen Kontexten gibt (`getName()` in mehreren Klassen), kann eine Umbenennung schwieriger werden und muss größtenteils manuell durchgeführt werden. Der grundsätzliche Ansatz, Namen nach einheitlichen Regeln und im jeweiligen Kontext passend zu wählen, sollte trotzdem beibehalten werden.

- Verschiebung von Attributen und Methoden nach strikten Kriterien:
  - Verantwortlichkeit
  - Zugriffsmöglichkeiten
  - Vermeidung von geklontem Code
- Klassenhierarchien erweitern oder kürzen, falls notwendig, also
  - Zwischenklassen einfügen oder entfernen.
- Umbenennen!

1. Gelegentlich zeigt es sich, dass ein bestimmtes Attribut nicht richtig aufgehoben ist oder eine Aufgabe besser nicht in der Klasse durchgeführt wird, in der sie implementiert wurde<sup>2</sup>, sondern eher nach oben (falls es äquivalente Aktivitäten in Geschwisterklassen gibt) oder in eine private Methode verschoben wird. In der Regel ist das relativ einfach, und Entwicklungsumgebungen bieten eine gewisse Unterstützung. Trotzdem ist es im wesentlichen ein manueller Prozess. Aspekte wie Zugriffskontrolle und Geschwindigkeit müssen berücksichtigt werden, nicht zuletzt auch die Zukunft, also inwieweit Erweiterbarkeit oder Allgemeingültigkeit an dieser Stelle notwendig ist.

Nach den diversen Umbauaktionen immer die Testmatrix abfahren, um zu sehen, ob noch jeder Stein auf dem anderen liegt, auf den er gehört. Da sich dabei immer auch die genaue Aufgabe von Klassen oder Methoden ändert, wird es häufig notwendig sein, die Namensgebung an dieser Stelle gleich mit anzupassen. Das Finden sinnvoller Namen führt fast immer zu einem besseren Verständnis der Codearchitektur.



- Entstandene Klone wieder zusammenführen
- Nebeneinander stellen (diff, diff3, kompare), Unterschiede minimieren
- Notwendige Unterschiede genau analysieren, Selektionskriterium finden
- Gemeinsamen Code und Variationen separieren und geschickt (!) verknüpfen

1. Dass beim Entwickeln Codeteile kopiert, modifiziert und wiederverwendet werden, ist völlig normal. Allerdings sollte man, wenn alles wie gewünscht funktioniert, sich Mühe geben, sehr ähnliche Abschnitte wieder anzugleichen und die identischen Teile zu extrahieren. Zum einen, um Fehler, die in einer Variante behoben werden, nicht auch in den anderen suchen zu müssen, zum anderen auch, um weitere Entwicklungen nicht parallel pflegen zu müssen. Schließlich wird Code, der unter unterschiedlichen Randbedingungen gerufen wird, besser und umfassender getestet, und zu guter letzt muss weniger Code verstanden werden. Oft ergeben sich dabei auch weitere, bisher nicht abgedeckte Sonderfälle, die bei einer hinreichend allgemeinen Formulierung automatisch mit erledigt werden. Dabei ergibt sich auch ein tieferes Verständnis, worin jetzt die genaue Ursache der Variantenbildung liegt, und wie diese zukünftig vielleicht bereits im Vorfeld vermieden werden kann.

## 12 Entwicklungsprozesse

Softwareentwicklung ist nicht unproblematisch. Definierte Prozesse sollen die Programmierung besser planbar machen – mit mehr oder weniger Erfolg. Verbreitete Prozesse:

- Wasserfallmodell (1970)
- Spiralmodell (1986)
- V-Modell (1997)
- RUP (Rational Unified Process) (1999)
- XP (Extreme Programming, 1995)
- Scrum (1986 für Produktion, 1991 für Software)

Ältestes, relativ starres, vollständig lineares Modell der Entwicklung mit klar definierten, voneinander abgegrenzten Phasen:

- Anforderungsdefinition
- Analyse
- Design
- Implementierung
- Test, Verifikation
- Abnahme, Betrieb, Pflege

Erfahrung zeigt: Wasserfallmodell ist zu starr

- Anforderungen ändern sich kontinuierlich
- fehlendes Feedback von späteren Phasen (Implementierung, Test) auf frühere (Design, Implementierung).

Konsequenz:

- Wiederholte (iterative) Abfolge von Wasserfallphasen
- spiralförmige Erweiterung der Funktionalität
- linear innerhalb einer Iteration

Sehr stark formalisiert und dokumentenlastig, besteht aus zwei getrennten Erstellungs- und Testphasen, die auf verschiedenen Ebenen einander zugeordnet sind:

- Anforderungen – Abnahmetests
- Architektur – Systemtests
- Entwurf – Integrationstests
- Implementation – Modultests

Erstellungsphase wird von oben nach unten abgearbeitet, die Testphase dann umgekehrt. Enge Kopplung zwischen benachbarten Ebenen, iterative Anwendung möglich. Zahlreiche Vorschriften für organisatorische Abwicklung und zu erstellende Dokumente. Teilweise vorgeschrieben für Bundesprojekte.

Rational Unified Process, kommerziell (Prozess+Tools) von Rational Software (IBM). Sehr schwergewichtig, stark strukturiert, architekturzentriert, dokumentenintensiv. Kernaufgaben (disciplines):

- Geschäftsprozessmodellierung (Business Modeling)
- Anforderungsanalyse (Requirements)
- Analyse & Design (Analysis & Design)
- Implementierung (Implementation)
- Test
- Auslieferung (Deployment)

Unterstützende Arbeitsschritte (supporting disciplines):

- Konfigurations- und Änderungsmanagement (Configuration & Change Management)
- Projektmanagement (Project Management)
- Infrastruktur (Environment)

Jeder Arbeitsschritt wird in vier Phasen abgearbeitet (mehr oder minder intensiv):

- Konzeption
- Planung
- Konstruktion
- Übergabe

Dazu kommen Best Practices:

- Iterative Softwareentwicklung
- Projektbegleitendes Qualitätsmanagement
- Komponentenbasierte Architektur
- Visuelle Modellierung
- Kontrolliertes Änderungsmanagement
- Anforderungsmanagement

Sehr komplex (>30 Rollen, >130 Aktivitäten), sehr viel Meta-Arbeit (>100 Artefakttypen), lohnt nur bei sehr großen Projekten (>10 Leute)



Statische und schwergewichtige Prozesse verkennen, dass Software keine Immobilie ist: kontinuierliche Änderungen der Anforderungen sind unausweichlich, schnelle Reaktionen darauf unerlässlich. Agile Methoden (XP, Scrum) versprechen mehr Flexibilität, bessere Qualität und höhere Produktivität. Gemeinsam ist allen das Bekenntnis zum „Agilen Manifest“:

- Individuen und Interaktionen haben Vorrang vor Prozessen und Werkzeugen.
- Lauffähige Software hat Vorrang vor ausgedehnter Dokumentation.
- Zusammenarbeit mit dem Kunden hat Vorrang vor Vertragsverhandlungen.
- Das Eingehen auf Änderungen hat Vorrang vor strikter Planverfolgung.

XP wurde von Kent Beck, Ward Cunningham und Ron Jeffries bei Chrysler für ein Lohnabrechnungssystem entwickelt, das nach der Fusion mit Daimler eingestellt wurde.

- Agile, leichtgewichtige Methode
- Kunde vor Ort
- keine formale Spezifikation
- wenig Dokumentation
- kontinuierliches Testen (oft Testfälle vor Implementierung)
- tägliche Kurzbesprechungen
- paarweises Programmieren
- gemeinsame Verantwortlichkeit für den Code
- einfaches Design
- häufige Releases

XP besteht aus Werten, Prinzipien und Praktiken:

**Werte** Kommunikation, Einfachheit, Rückmeldung, Mut und Respekt

**Prinzipien** Menschlichkeit, Wirtschaftlichkeit, beidseitiger Vorteil, Selbstgleichheit, Verbesserungen, Vielfältigkeit, Reflexion, Lauf, Gelegenheiten wahrnehmen, Redundanzen vermeiden, Fehlschläge hinnehmen, Qualität, kleine Schritte sowie akzeptierte Verantwortung

**Praktiken** Pair-Programming, kollektives Eigentum, permanente Integration, testgetriebene Entwicklung bzw. permanentes Testen, Kundeneinbeziehung, Refactoring, keine Überstunden, Iterationen, Metapher, Coding-Standards, einfaches Design und Planning-Game

Nach fünf Jahren Erfahrung wurden die Praktiken neu definiert, umbenannt und aufgeteilt:

**Hauptpraktiken** räumlich zusammen sitzen, informativer Arbeitsplatz, Team, Pair-Programming, energiegelante Arbeit, entspannte Arbeit, Storys, wöchentlicher Zyklus, quartalsweiser Zyklus, 10-Minuten-Build, kontinuierliche Integration, Test-First-Programmierung und inkrementelles Design

**Begleitpraktiken** richtige Kundeneinbeziehung, inkrementelles Deployment, Team-Konstanz, schrumpfende Teams, ursachliche Analysen, geteilter Code, Codierung und Testen, eine zentrale Codebasis, tägliches Deployment, verhandelbarer und vertraglicher Funktionsumfang, Zahlen-pro-Nutzung

Das Einhalten von Coding Standards wird interessanterweise mittlerweile als so selbstverständlich betrachtet, dass es gar nicht mehr aufgeführt wird!

Scrum (Gedränge) ist eine weitere agile Methode, kommt eigentlich aus der Produktion und wurde von Ken Schwaber, Jeff Sutherland und Mike Beedle auf Softwareentwicklung übertragen.

Grundprinzipien:

- Softwareentwicklung ist ein sehr komplexer, schwer planbarer Prozess
- deshalb Selbstorganisation des Teams innerhalb eines relativ losen Rahmens
- enge Einbeziehung des Kunden
- stark zeitlich strukturiert
- kein hierarchisches Management von oben

Es gibt feste Rollen:

**Product Owner** Festlegung und Priorisierung der notwendigen Features

**Scrum Master** überwacht Rollenverteilung, Hindernisbeseitigung und gute Arbeitsbedingungen für das Team, steht selbst aber außerhalb

**Team** idealerweise 5-9 Leute, selbst organisiert, schätzt ab und

Entwicklung findet statt in zeitlich fixierten Sprints, die jeweils *fertige* Funktionalität ausliefern und ca. 4 Wochen dauern. Ablauf:

**Planungstreffen 1** Abstimmung von Product Owner und Team, Priorisierung der Funktionseinheiten, Festlegung der Sprintziels für Abnahme

**Planungstreffen 2** Abstimmung innerhalb des Teams, Zerlegung der Funktionseinheiten, Verteilung der Aufgaben

**Sprint** Umsetzung der vereinbarten Aufgaben, Implementierung und Test der vereinbarten Funktionalität.

**Daily Scrum** Tägliches kurzes (Viertelstunde) Treffen zur Diskussion des Standes und notwendigen Abstimmung, empfohlen nach dem Mittagessen

**Review** Abnahmesitzung zwischen Team und Product Owner

**Retrospective** Betrachtung des zurückliegenden Sprints, aufgetretene Probleme, Verbesserungspotenziale, positive Erfahrungen

Während des Projektes werden diverse Dokumente geführt und kontinuierlich fortgeschrieben, hauptsächlich

**Product Backlog** Features des zu entwickelnden Produktes einschließlich Priorisierung und Aufwandsschätzung, Detaillierungsgrad je nach Priorität

**Selected Backlog** Ausgewählte Elemente aus dem Product Backlog für den jeweiligen Sprint, maßgeblich für die Abnahme

**Sprint Backlog** Zerlegte Teilaufgaben für den aktuellen Sprint, mit Verantwortlichem

**Burndown Chart** Grafische Darstellung der aufgestellten und abgearbeiteten Teilaufgaben aus dem Sprint Backlog, ermöglicht Abschätzung des Fortschrittes

**Impediment Backlog** Hindernisse, die es auszuräumen gilt (fehlende Abstimmungen, Ressourcen, Werkzeuge, Schulungsbedarf)

[www.iag.uni-stuttgart.de](http://www.iag.uni-stuttgart.de)



[www.iag.uni-stuttgart.de](http://www.iag.uni-stuttgart.de)

[www.iag.uni-stuttgart.de](http://www.iag.uni-stuttgart.de)

[www.iag.uni-stuttgart.de](http://www.iag.uni-stuttgart.de)

[www.iag.uni-stuttgart.de](http://www.iag.uni-stuttgart.de)

[www.iag.uni-stuttgart.de](http://www.iag.uni-stuttgart.de)



Prozesse sind für andere Softwaretypen konzipiert (eher Datenbank-artig, hochgradig interaktiv, viele relativ separate Nutzungsfälle) als bei uns (Batch-Anwendungen, viele hoch integrierte Funktionen, hoher Abdeckungsgrad für viele Anwendungsfälle). Außerdem arbeiten Einzelpersonen an wenig vernetzten Baustellen über längere Zeit. Manches ist aber durchaus sinnvoll zu übernehmen

**Lauffähiger Code** Die Entwicklung findet statt, ohne dass der Code längere Zeit unbrauchbar ist. Die Restfunktionalität bleibt immer erhalten. Sichergestellt durch

**Kontinuierliche Tests**, die schnell und automatisiert ablaufen. Des weiteren ist angesagt

**Häufiges Refactoring** um das System flexibel zu halten und an Erweiterungen anzupassen. Ermöglicht wird die Flexibilität unter anderem durch

**Einfaches Design** KISS (Keep it simple, stupid) und YAGNI (You ain't gonna need it), also nur die Features implementieren, die benötigt werden, Erweiterungen erst bei Bedarf

## 13 Entwicklungsumgebungen

Integrierte Entwicklungsumgebungen (IDEs) unterstützen bei der Programmierung. Verbreitet sind:

- KDevelop (KDE)
- Anjuta (GNOME)
- QtCreator (Qt)
- Eclipse (Intel Parallel Studio)
- Netbeans (Oracle Solaris Studio)
- Microsoft Visual Studio (Intel Parallel Studio)

Daneben gibt es auch noch einige kleinere oder auf bestimmte Sprachen eingeschränkte IDEs, sowie Plug-Ins oder Makrosammlungen für Editoren (Emacs, Vim).

Praktisch überall verfügbar sind

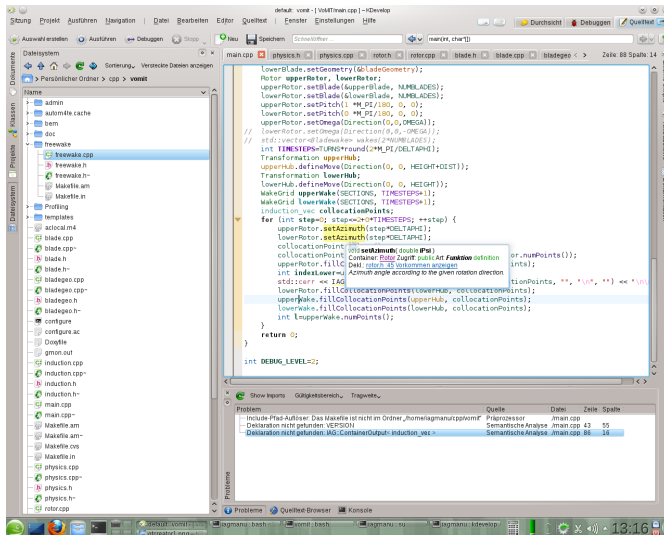
- Syntax Highlighting für verschiedene Programmiersprachen
- Mehrere gleichzeitig offene Dateien
- Intelligentes Editieren
  - Einrücken
  - Kommentieren
  - Kennzeichnen (Klammern, Blöcke)
  - Faltung (Ausblenden momentan uninteressanter Abschnitte)
- Übersetzen und Start auf Tastendruck

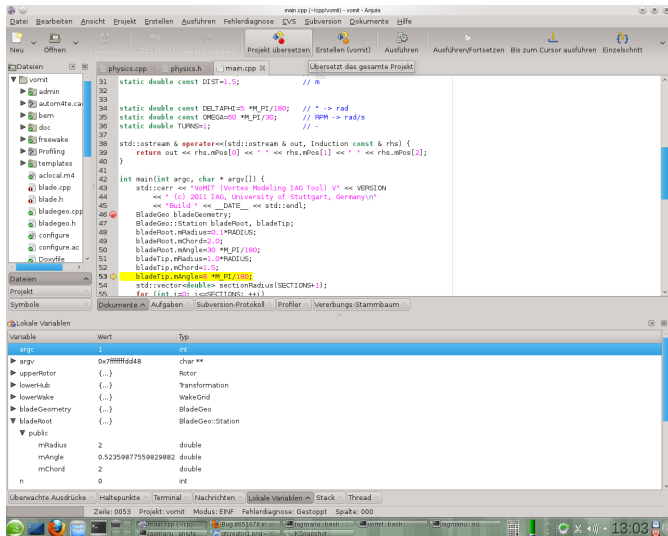
Darüber hinaus gibt es – vor allem natürlich bei den vollwertigen IDEs –

- Editierhilfen
  - Automatisches Vervollständigen
  - Information (Funktionsparameter, Dokumentationskommentare, Standardbibliothek)
  - Makros, Schablonen, Wizards
  - Formatierung
- Vielfältige Navigation: Verwendung ↔ Deklaration ↔ Definition, Marker, Quelldateien
- Projektverwaltung (Quelldateien, Skripte, Abhängigkeiten, eventuell mehrere gleichzeitig)
- Integrierte Werkzeuge
  - Versionskontrolle
  - Übersetzung, Fehlernavigation
  - Debugger
  - Speicherprüfung
  - Testumgebung
  - Dokumentation: doxygen, UML-Modellierung, ...

Schließlich gibt es noch die Sonderfeatures

- Analyse des Quelltextes
  - Meldung von Syntaxfehlern
  - Extraktion und Zuordnung von Typen und Objekten
  - kontextabhängige Editierhilfen
  - Klassen- und Aufrufdiagramme
- Refactoring-Unterstützung (jeweils klassen- und dateiübergreifend)
  - Umbenennen
  - Verschieben
  - Extraktion
  - Erweiterung





```

31 static double const DIST=1.5; // m
32
33
34 static double const DELTA PHI=0.1 * M_PI/180; // * -> rad
35 static double const OMEGA=90 * M_PI/30; // RPM -> rad/s
36 static double TURNS=1; //
37
38 std::ostream & operator<<(std::ostream & out, Induction const & rho) {
39     return out << rho.nPos[0] << " " << rho.nPos[1] << " " << rho.nPos[2];
40 }
41
42
43 int main(int argc, char * argv[]) {
44     std::cerr << "VoMIT [Vortex Modeling IAG Tool] V" << VERSION
45     << " (c) 2011 IAG, University of Stuttgart, Germany\n";
46     << "Build " << __DATE__ << std::endl;
47     BladeGeo bladeGeometry;
48     BladeGeo::Station bladeRoot, bladeTip;
49     bladeRoot.mRadius=0.1 * RADIUS;
50     bladeRoot.mChord=2.0;
51     bladeTip.mAngle=0.1 * M_PI/180;
52     bladeTip.mRadius=1.0 * RADIUS;
53     bladeTip.mChord=1.5;
54     bladeTip.mAngle=0.1 * M_PI/180;
55     std::vector<double> sectionRadius(SECTIONS+1);
56     for (int i=1; i<=SECTIONS; ++i)

```

Variable	Wert	Typ
argc	1	int
argv	0x7ffffffd48	char **
upperRotor	{...}	Rotor
lowerRotor	{...}	Transformation
lowerWake	{...}	WakeGrid
bladeGeometry	{...}	BladeGeo
bladeRoot	{...}	BladeGeo::Station
public		
mRadius	2	double
mAngle	0.52359877559829882	double
mChord	2	double
n	0	int

Überwachte Ausdrücke | Haltepunkte | Terminal | Nachrichten | Lokale Variablen | Stack | Thread

Zeile: 0053 Projekt: vornit Modus: ENF Fehlerdiagnose: Gestoppt Spalte: 000

man cp /tmp/ Bug #51678 in ... ingmanu: bash ingmanu: su vornit: bash ingmanu: anjuta gtfcrat01.png KSnapshot

13:03

The screenshot shows the Qt Creator IDE interface. The main window displays the source code of `main.cpp` in the `vomit` project. The code defines a `BladeGeo` class and a `main` function that creates a `BladeGeo` object and sets its geometry.

The variable explorer on the right shows the following variables:

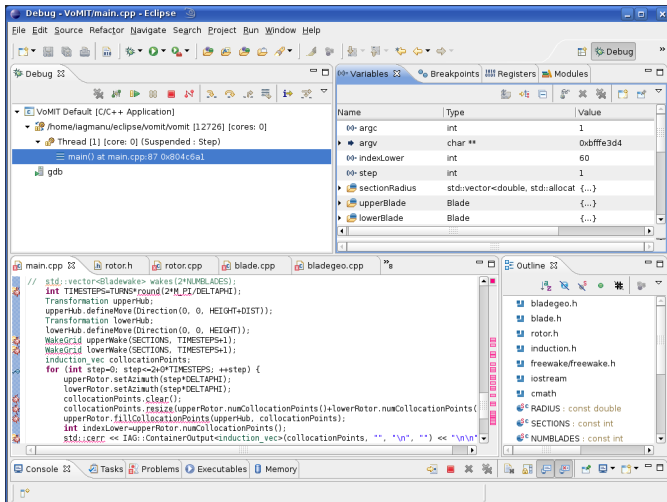
Name	Wert	Typ
<code>main</code>		<code>int</code>
<code>argc</code>	1	<code>int</code>
<code>argv</code>	-ein Element-	<code>char**</code>
<code>bladeGeometry</code>		<code>BladeGeo</code>
<code>bladeRoot</code>		<code>BladeGeo::Station</code>
<code>mAngle</code>	0.52359877559829882	<code>double</code>
<code>mChord</code>	2	<code>double</code>
<code>mRadius</code>	2	<code>double</code>
<code>bladeTip</code>		<code>BladeGeo::Station</code>
<code>colocationPoints</code>	<nicht zugänglich>	<code>induction_vec</code>
<code>lowerBlade</code>		<code>Blade</code>
<code>lowerHub</code>		<code>Transformation</code>
<code>lowerRotor</code>		<code>Rotor</code>
<code>lowerWake</code>		<code>WakeGrid</code>
<code>n</code>	0	<code>int</code>
<code>sectionRadius</code>	<nicht zugänglich>	<code>std::vector&lt;double&gt;</code>
<code>sectionRadius</code>		<code>double</code>

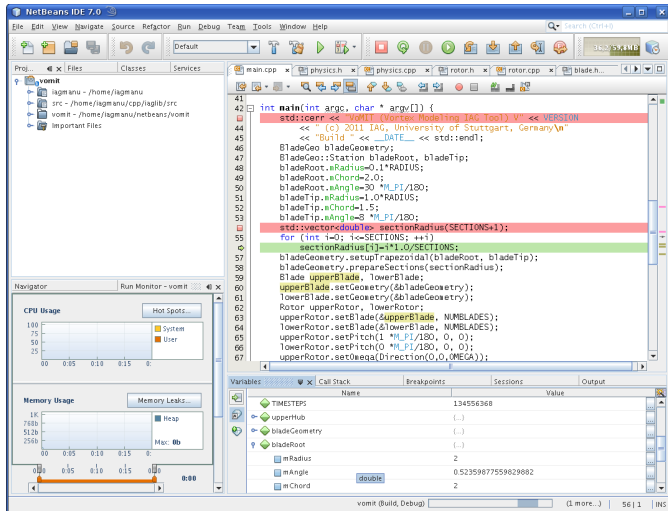
The console window at the bottom shows the output of the `make` command, indicating that the build process was successful.

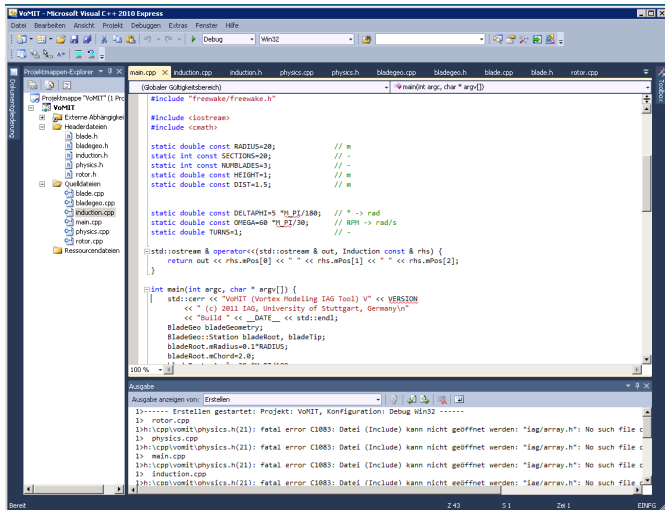
```

make[1]: Für das Ziel 'vomit' ist nichts zu tun.
make[1]: Leaving directory '/home/agmanu/cp/vomit/bem'
Making all in freewake
make[1]: Entering directory '/home/agmanu/cp/vomit/freewake'
make[1]: Für das Ziel 'vomit' ist nichts zu tun.
make[1]: Leaving directory '/home/agmanu/cp/vomit/freewake'
Making all in
make[1]: Entering directory '/home/agmanu/cp/vomit'
make[1]: Für das Ziel 'vomit' ist nichts zu tun.
make[1]: Leaving directory '/home/agmanu/cp/vomit'
Making all in doc
make[1]: Entering directory '/home/agmanu/cp/vomit/doc'
make[1]: Für das Ziel 'vomit' ist nichts zu tun.
make[1]: Leaving directory '/home/agmanu/cp/vomit/doc'
Der Prozess 'ustbin/make' wurde normal beendet.
  
```









VoMIT - Microsoft Visual C++ 2010 Express

Projekt: VoMIT (1 Proje...

Quelldateien

- blade.h
- bladegeo.h
- induction.h
- physics.h
- rotor.h
- blade.cpp
- bladegeo.cpp
- induction.cpp
- main.cpp
- physics.cpp
- rotor.cpp

```

(Globaler Gültigkeitsbereich)
#include "freewake/freewake.h"

#include <iostream>
#include <cmath>

static double const RADIUS=20;           // m
static int const SECTIONS=20;           // -
static int const NUMBLADES=3;           // -
static double const HEIGHT=1;           // m
static double const DIST=1.5;           // m

static double const DELTAPI=5 * M_PI/180; // * -> rad
static double const OMEGA=60 * M_PI/30; // RPM -> rad/s
static double TURNS=1; // -

std::ostream & operator<<(std::ostream & out, Induction const & rhs) {
    return out << rhs.mPos[0] << " " << rhs.mPos[1] << " " << rhs.mPos[2];
}

int main(int argc, char * argv[]) {
    std::cerr << "VoMIT (Vortex Modeling IAG Tool) V" << VERSION
    << " (c) 2011 IAG, University of Stuttgart, Germany\n"
    << "Build " << __DATE__ << std::endl;
    BladeGeo bladeGeometry;
    BladeGeo::State bladeRoot, bladeTip;
    bladeRoot.mRadius=0.1*RADIUS;
    bladeRoot.mChord=2.0;
    }
    
```

Ausgabe anzeigen von: Erstellen

```

1>----- Erstellen gestartet: Projekt: VoMIT, Konfiguration: Debug Win32 -----
1> rotor.cpp
1>h:\c\p\vo\mit\physics.h(21): fatal error C1083: Datei (Include) kann nicht geöffnet werden: "iag/array.h": No such file c
1> physics.cpp
1>h:\c\p\vo\mit\physics.h(21): fatal error C1083: Datei (Include) kann nicht geöffnet werden: "iag/array.h": No such file c
1> main.cpp
1>h:\c\p\vo\mit\physics.h(21): fatal error C1083: Datei (Include) kann nicht geöffnet werden: "iag/array.h": No such file c
1> induction.cpp
1>h:\c\p\vo\mit\physics.h(21): fatal error C1083: Datei (Include) kann nicht geöffnet werden: "iag/array.h": No such file c
    
```

Bereit Z 43 S 1 Zeil EINF

- Integrierte Entwicklungsumgebungen können große Unterstützung leisten.
- Wesentliche Merkmale sind in allen IDEs verfügbar.
- Individuelle Gegebenheiten können angepasst werden (Kammersetzung, Einrückung, ...).
- Großer Produktivitätsgewinn nach Eingewöhnungsphase möglich.

Also ausprobieren, es könnte sich lohnen!

## 14 Releasemanagement

Ist die Entwicklung so weit gediehen, dass sie ausgeliefert werden kann, sollte auch dies systematisch und reproduzierbar geschehen:

- Automatisierte Erstellung
- Test auf Lauffähigkeit und Funktion auf verschiedenen Systemem (soweit möglich ebenfalls automatisiert)
- Dokumentation des ausgelieferten Produktes

Zur eindeutigen Identifizierung eines Releases ist es enorm hilfreich, eine Möglichkeit zur Ausgabe vorzusehen, so dass entweder generell oder zumindest auf Anforderung die Versionsnummer erkennbar wird.

Vor der Auslieferung steht die Erstellung. Dazu sind nötig:

- ❶ Letzte Aktualisierungen (README, INSTALL, ChangeLog, Dokumentation)
- ❷ Extraktion der eigentlichen Quellen ohne überflüssige Artefakte (Objektdateien, Sicherheitskopien)
- ❸ Fertigstellung der Dateien (Einfügen von Copyright-Vermerken, Erzeugung von hilfreichen Zwischendateien wie configure oder Makefile.in)
- ❹ Zusammenpacken der erforderlichen Dateien, einschließlich eventuell notwendiger weiterer Bibliotheken (für die jeweils die gleichen Schritte gelten)

Danach geht es mit Tests weiter:

- ① Transfer des entstandenen Päckchen auf die in Frage kommenden Zielsysteme
- ② Erzeugen der Anwendung
- ③ Globales Skript oder Makefile, vor allem bei mehreren Teilen
- ④ Falls Fehler auftreten, zurück zum Start
- ⑤ Keine schnellen Korrekturen auf dem einen oder anderen System!

Da unter Umständen mehrere Iterationen durchlaufen werden müssen ist auch eine weitestgehende Automatisierung des Ablaufes so hilfreich.

Erst wenn die Anwendung auf allen Zielsystemen reibungslos gebaut werden kann und die hoffentlich vorhandenen Tests jeweils durchlaufen, wird das Release freigegeben.

- ① Markierung im Versionskontrollsystem (Tagging)
- ② Erzeugung eines Astes zur Fehlerbehebung (kann, ausgehend vom Tag, zur Not auch später bei Bedarf erfolgen)
- ③ Falls binär ausgeliefert wird, Anwendung auf der Zielplattform (gegebenenfalls mehrere) erzeugen, mit erforderlichen Hilfsdateien verpacken
- ④ Quell- oder Binärpäckchen archivieren und dokumentieren
- ⑤ Päckchen verschicken



Zur Kennzeichnung unterschiedlicher Releases gibt es zahllose Möglichkeiten. In der Praxis hat sich eine dreistufige Nummerierung bewährt, bestehend aus Haupt- und Nebenversionsnummer und dem Release.

**Hauptversionsnummer (Major)** wird erhöht, sobald wesentliche Änderungen vorgenommen wurden, die unter Umständen auch signifikante Inkompatibilitäten hervorrufen. Bei Bibliotheken sind Quellcodeänderungen erforderlich, da sich Funktionen geändert haben, Parameter weggefallen oder dazugekommen sind. Minor und Release werden zurückgesetzt.

**Nebenversionsnummer (Minor)** wird erhöht bei kleineren Erweiterungen, die aber im wesentlichen die Kompatibilität wahren und nur Funktionalität hinzufügen. Bei Bibliotheken muss nur neu übersetzt werden, Quellcodeänderungen sind im allgemeinen nicht notwendig, falls die neuen Funktionen nicht genutzt werden sollen. Release wird zurückgesetzt.

**Release** betrifft lediglich Fehlerbehebungen oder allenfalls marginale Erweiterungen und offensichtliche kleine Lücken. Bei Bibliotheken sollte es genügen, neu zu linken, unter Umständen auch nur dynamisch.

Die eigentliche Kennzeichnung der einzelnen Nummern kann dann unterschiedlich erfolgen, beispielsweise als fortlaufende Nummern (1.5.2), mit der Jahreszahl als Hauptversionsnummer (2011.2.1) oder auch mit Buchstaben für das Release (4.6b).

Falls einfache Bugfixes normalerweise nicht auftreten (fehlerfrei per Definitionem), sondern nur immer neue Funktionalität ausgeliefert wird, kann auf das Release auch verzichtet und ein zweistufiges System genutzt werden. Eine Trennung und Kennzeichnung von kompatiblen/nicht kompatiblen Versionen ist aber in jedem Falle empfehlenswert.

Codenamen (Dapper Drake, Snow Leopard, Indigo) sind zwar hübsch, verfehlen aber den Zweck der eindeutigen Einordnung in eine Entwicklungshistorie und sind daher nur bedingt tauglich.

# Zusammenfassung

Im ersten Teil der zweisemestrigen Veranstaltung ging es im wesentlichen um das Drumherum:

**Werkzeuge** Kleine Helferlein, Shells und automatisierte Erstellung, Versionsmanagement

**Techniken** Systematisches Testen, Fehlerbehebung, Quellcode bereinigen und sauber halten

**Entwicklung** Prozesse, IDEs, Release

Der zweite Teil gliedert sich wiederum in zwei große Abschnitte:

**Technische Aspekte** Hardwareaspekte und ihre Nutzung, Portabilität und Dokumentation

**Programmierung** Sinnvolle Anwendung der vielfältigen Möglichkeiten von Programmiersprachen

- Prozessorhardware** CPU-Architekturen, Speichertechnologien, Caches, SMP und NUMA, Speichertechnologien, Latenz und Bandbreite, Performance Counter, Cache-Optimierung, Prefetching
- Parallelisierung** Gemeinsamer (SMP) oder verteilter (MPI) Speicher, Ressourcenzuweisung, Synchronisation, Deadlocks, Nachrichtenversand, Punkt-zu-Punkt und gemeinsamer Austausch, Latenz und Bandbreite, OpenMP und MPI
- Portabilität** Betriebssysteme und Standards, Ressourcenverwaltung, grafische Benutzeroberflächen, Standardbibliothek, Sprachstandards für C, C++ und Fortran, Bibliotheken verwenden und schreiben, gängige Bibliotheken
- Dokumentation II und Programmarchitektur** Anforderungen und Analyse, HPC-Erwägungen, Bedienungskonzept, Kommunikation und Datentransfer, Verteilung und Parallelisierung, Spezifikation, Entwurf und Design, Diagramme

- Objektorientierte Programmierung** Kapselung und Objekte, Verantwortlichkeiten, Vererbung und LSP, Patterns, Erzeugung und Zerstörung, Assoziationen
- Python** Syntax, Versionen, Datentypen, Klassen und Funktionen, Operatoren und Lambdas, Standardbibliothek, NumPy und SciPy, Wrapping und Interaktion mit anderen Sprachen
- C++** Syntax, Klassen, Objekte, virtuelle Funktionen, Zeigerkonzept, Interaktion mit anderen Sprachen, Templates und Operatoren, Containerklassen, Optimierungen, Ausnahmen, Standardbibliothek, Speicherverwaltung, Verantwortlichkeit und RAII, Idioms
- GPU-Programmierung** Architektur, Problemanalyse, Algorithmenanpassung, Datentransfer, Genauigkeit, Speicherzugriffe, parallele Abarbeitung, Aufruf und Kontrolle, Benchmarking und Tuning