

# MATERIALSAMMLUNG - PROGRAMMING BY CONTRACT

Prof. Dr. Hans-Jürgen Buhl



Wintersemester 2005/2006

Bergische Universität Wuppertal  
Fachbereich C — Mathematik und Informatik



# Inhaltsverzeichnis

Vorbemerkungen – Softwarequalität heute . . . . .	3
Haftung . . . . .	3
Beispiele für Softwaredisfunktionalitäten . . . . .	5
Deep Impact . . . . .	5
USV-Software legt Server lahm . . . . .	5
Chaos an Hannovers Geldautomaten . . . . .	6
Therac 25 . . . . .	6
Berliner Magnetbahn . . . . .	7
Elektronik-Fehler führt zu Überhitzung bei Volvo-PKW . . . . .	7
The Patriot Missile . . . . .	8
Kontenabrufverfahren startet wegen Softwareproblemen als Provisorium . . . . .	9
Buffer Overflow im Linux-Kernel . . . . .	9
Auch Superhirne können irren - das Risiko Computer . . . . .	10
Neueste Risikoinformationen/Softwareprobleme . . . . .	11
Programming by Contract im Umfeld der Formalen Methoden . . . . .	13
Programmverifikation . . . . .	13
Programming by Contract . . . . .	14
Model based Specification . . . . .	16
Model Constraints . . . . .	22
<b>1 Qualitätsanforderungen an SW-Produkte</b> . . . . .	<b>25</b>
1.1 Prinzipien der ordnungsgemäßen Programmerstellung . . . . .	26
1.2 Spezifikation einer abstrakten Datenkapsel . . . . .	27
1.2.1 Axiomatische Spezifikation . . . . .	27
1.2.2 Beschreibende (denotationale) Spezifikation . . . . .	27
1.3 Prinzipien der Modularisierung: . . . . .	28
1.4 Typen der Modularisierung . . . . .	28
<b>2 Wiederverwendbarkeit</b> . . . . .	<b>29</b>
2.1 Begriffshierarchien . . . . .	31
2.2 Objekthierarchien als strukturierte Modulsammlungen: Beispiele aus Eiffel . . . . .	32
2.2.1 Vererbung und Erweiterung . . . . .	32
2.2.2 Vererbung und Abänderung . . . . .	33
2.2.3 Generizität . . . . .	34
2.2.4 Eingeschränkte Generizität . . . . .	34

2.2.5	Polymorphie und „late binding“ . . . . .	34
2.2.6	Aufgeschobene Feature-Implementierungen . . . . .	35
2.3	Resümee . . . . .	36

# Abbildungsverzeichnis

0.1	Design by Contract, by Example von Richard Mitchell und Jim McKim .	4
0.2	Bilder von Deep Impact . . . . .	5
0.3	<a href="http://catless.ncl.ac.uk/Risks/22.92.html">http://catless.ncl.ac.uk/Risks/22.92.html</a> . . . . .	11
2.1	Begriffshierarchien . . . . .	31



# Tabellenverzeichnis

0.1 Divergence in the Range Gate of a PATRIOT MISSILE . . . . . 8

## Programming by Contract

2 V Do 10 - 12 in D13.08

Einordnung: Diplom Mathematik/Nebenfach Informatik: Hauptstudium - Praktische und Technische Informatik; Bachelor IT: Praktische Informatik A - Programmiersprachen und Sprachkonzepte; Master Wirtschaftsmathematik: Wahlpflichtbereich Informatik; Wirtschaftswissenschaften: Modul I - Software- und Programmieretechnik; Studienschwerpunkte und Nebenfächer Informatik anderer Studiengänge

Vorkenntnisse: Einführung in die Informatik; Programmierkenntnisse in C++; erfolgreiche Teilnahme an „Einführung in die Benutzung der Ausbildungsrechner“

Inhalt: Die Programmiermethodik „Programming/Design by Contract“ klärt die Verantwortlichkeit von Diensteanbieter (Funktion/Methode) und Dienstenehmer (Aufrufer einer Funktion) durch genaue Vereinbarungen. Durch das Sprachmittel der Zusicherung werden Voraussetzungen, Diensteeerfüllung und Ausnahmebedingungen zur Laufzeit eines Programms (automatisch) überprüft und führen zu Code besserer Qualität.

Literatur: wird in der Veranstaltung bekannt gegeben.

# Vorbemerkungen – Softwarequalität heute

## Haftungsausschluß

Die Überlassung dieser Baupläne erfolgt ohne Gewähr. Der Plan gibt keine Garantie, Gewährleistung oder Zusicherung, daß diese Pläne für einen bestimmten Zweck geeignet sind, daß sie richtig sind oder daß ein Gebäude, das nach diesen Plänen gebaut wird, den Ansprüchen des jeweiligen Erwerbers genügt. Der Planer erklärt sich bereit, Ersatzkopien derjenigen Teile der Pläne zu liefern, die zum Zeitpunkt des Kaufs unleserlich sind. Darüber hinaus wird keinerlei Haftung übernommen. Der Erwerber dieser Pläne sollte beachten, daß in den entscheidenden Phasen des Baus und nach der Fertigstellung geeignete Tests durchzuführen sind und daß die üblichen Vorsichtsmaßnahmen zum Schutz des Lebens der Bauarbeiter zu treffen sind.

(Zitat: Robert L. Baber: Softwarereflexionen, Springer-Verlag)

und in der Praxis:

...

## **2. Haftung**

Wir werden immer bemüht sein, ihnen einwandfreie Software zu liefern. Wir können aber keine Gewähr dafür übernehmen, daß die Software unterbrechungs- und fehlerfrei läuft und daß die in der Software enthaltenen Funktionen in allen von Ihnen gewählten Kombinationen ausführbar sind. Für die Erreichung eines bestimmten Verwendungszweckes können wir ebenfalls keine Gewähr übernehmen. Die Haftung für unmittelbare Schäden, mittelbare Schäden, Folgeschäden und Drittschäden ist, soweit gesetzlich zulässig, ausgeschlossen. Die Haftung bei grober Fahrlässigkeit und Vorsatz bleibt hiervon unberührt, in jedem Fall ist jedoch die Haftung beschränkt auf den Kaufpreis.

Hauptgegenstand dieser Veranstaltung ist die konstruktive Methode

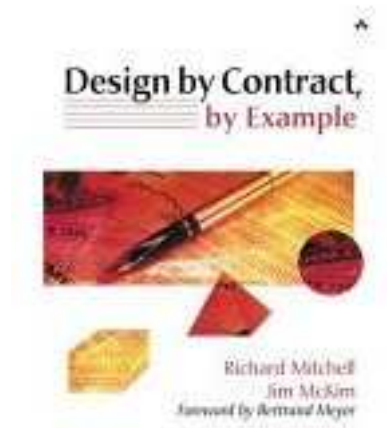


Abbildung 0.1: Design by Contract, by Example von Richard Mitchell und Jim McKim

zur Sicherung grundlegender Softwaregüte. In den ersten Kapiteln wird noch einmal Grundwissen zur Softwarequalität und -qualitätssicherung repetiert.

## Beispiele für Softwaredisfunktionalitäten

### Ein sahniger Brocken

(aus: *Die Zeit* vom 15.09.2005)

Begleitet von großem Werbeummel hat die NASA den Kometen Tempel1 beschossen. Nun zeigen die Daten: Getroffen hat sie gut, gelernt hat sie wenig.

Auch wenn in den offiziellen Mitteilungen der NASA keine Rede davon ist - unter den versammelten Astronomen hat sich längst herumgesprochen, dass der Erfolg von *Deep Impact* nicht nur von aufgewirbeltem Feinstaub verdunkelt wurde. Ein Softwarefehler hat dazu geführt, dass die ersten - und besten - Bilder des Zusammenpralls im Datenspeicher des Begleitsateliten von späteren Aufnahmen überschrieben wurden.

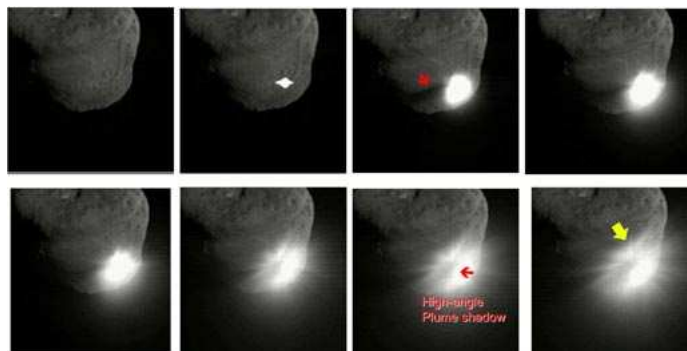


Abbildung 0.2: Bilder von Deep Impact

Der vollständige Artikel: [www.zeit.de/2005/38/komet](http://www.zeit.de/2005/38/komet)

### USV-Software legt Server lahm

**APC**, Hersteller von unterbrechungsfreien Stromversorgungssystemen (USV), rät in einem Knowledgebase-Artikel dazu, alte Versionen der **PowerChute Business Edition-Software 6.X** umgehend durch die Version 7.X zu ersetzen.

Die Software zur Steuerung unterbrechungsfreier Stromversorgungen und zum sicheren Server-Shutdown hat Probleme mit einem auslaufenden Java-Runtime-Zertifikat. Dies führt dazu, dass die Windows-Server, auf denen die alte Version läuft, zum Teil mehrere Stunden für eine Ab- beziehungsweise Anmeldung benötigen. Die Dienste des Servers wie zum Beispiel Netzwerkfreigaben funktionieren allerdings trotz der Anmeldeprobleme weiterhin.

(aus <http://www.heise.de/newsticker/meldung/62344>)

## **Chaos an Hannovers Geldautomaten (05.10.2003 13:00 Uhr)**

Computerprobleme haben am Samstag alle 240 Geldautomaten der Sparkasse in der Stadt und Region Hannover lahm gelegt. Die Fusion der Stadt- und Kreissparkasse sollte am Wochenende auch technisch umgesetzt werden, sagte der Sprecher des Geldinstituts, Stefan Becker. Beim Hochfahren eines Server habe sich ein Fehler eingeschlichen, so dass die Geldautomaten nicht mehr funktionierten. Die Sparkasse öffnete stattdessen fünf Filialen, damit Kunden etwa in Einkaufszonen Bargeld abheben können.

(aus: <http://www.heise.de/newsticker/meldung/40834>)

## **THERAC 25**

Selten sind solch schädliche Vorfälle so gut dokumentiert worden wie im Fall des „THERAC 25“, eines computergestützten Bestrahlungsgerätes. Dabei handelt es sich um ein Bestrahlungsgerät, welches in zwei „Modi“ arbeitet: im „X-Modus“ wird ein Elektronenstrahl von 25 Millionen Elektronen-Volt durch Beschuß einer Wolframscheibe in Röntgenstrahlen verwandelt; im „E-Modus“ werden die Elektronen selbst, allerdings „weicher“ mit erheblich reduzierter Energie als Korpuskelstrahlung erzeugt. Je nach therapeutischer Indikation wird die geeignete Strahlungsart eingestellt; in beiden Fällen kann der Bestrahlungsverlauf, nach Modus, Intensität und Bewegungskurve der Strahlungsquelle, mit einem Bildschirm-„Menü“ eingegeben werden.

Als mehrere Patienten berichteten, sie hätten bei Behandlungsbeginn das Gefühl gehabt, „ein heißer Strahl“ durchdringe sie, wurde dies vom Hersteller als „unmöglich“ zurückgewiesen. Erst nach dem Tod zweier Patienten sowie massiven Verbrennungen bei weiteren Personen kam heraus, daß neben dem X- sowie E-Modus mit niedriger Elektronenintensität infolge Programmierfehler ein unzulässiger dritter Zustand auftrat, nämlich direkt wirkende, 25 Millionen Elektronen-Volt „heiße“ Elektronen.

Dies geschah immer dann, wenn ein vorgegebenes „Behandlungsmenü“ mittels Curser-Taste modifiziert wurde. Um aufwendige Umprogrammierung zu vermeiden, wollte der kanadische Hersteller die Benutzung der Curser-Taste verbieten bzw. diese ausbauen und die Tastenlücke mit Klebeband abdichten lassen! Es ist zu befürchten, daß der Fall „THERAC 25“ kein Einzelfall ist. Zumeist ist es mangels entsprechender Vorsorge in computergesteuerten Medizingeräten schwerlich möglich, schädliches Systemverhalten später aufzuklären.

## Berliner Magnetbahn

Computer spielen in allen gesellschaftlichen Bereichen eine immer größere Rolle. Angesichts der von fehlerhafter Software ausgehenden Gefahr wird versucht, die Sicherheit von computergesteuerten Systemen so weit wie möglich zu garantieren.

### **Softwarefehler: Kleine Ursache, große Wirkung**

Fünf - Null, tippt der Operator in die Tastatur und erwartet, daß die Magnetschwebbahn auf 50 Stundenkilometer beschleunigen würde. Doch nichts geschah. Wieder tippt er fünf - null und vergaß diesmal nicht die „Enter“-Taste zu betätigen, mit der die Daten erst in den Rechner abgeschickt werden. Die insgesamt eingegebene Tastenfolge „fünf - null - fünf - null“ interpretiert der Rechner als Anweisung, auf unsinnige 5050 Stundenkilometer zu beschleunigen. Dies konnte die Bahn zwar nicht, aber immerhin wurde sie so schnell, daß sie nicht mehr rechtzeitig vor der Station gebremst werden konnte. Es kam zum Crasch mit Personenschaden – so geschehen vor zwei Jahren bei einer Probefahrt der Berliner M-Bahn.

Vernünftigerweise hätte die den Computer steuernde Software die Fehlerhaftigkeit der Eingabe „5050“ erkennen müssen. Schon dieses Beispiel mangelnder Software zeigt, von welcher Bedeutung das richtige Verhalten von Computerprogrammen sein kann. Nicht nur bei Astronauten, die mit softwaregesteuerten Raumfähren ins All starten, hängt heute Leben und Gesundheit von Software ab. Computerprogramme erfüllen mittlerweile in vielen Bereichen sicherheitsrelevante Aufgaben.

## Elektronik-Fehler führt zu Überhitzung bei Volvo-PKW

Kaum ein KFZ-Hersteller, der nicht mit Elektronik, Software und Hightech-Ausstattung das Autofahren komfortabler und die Wartung in der Werkstatt einfacher machen will. Doch die Tücken der Technik lassen für manchen Kunden den PKW zum IT-Sicherheitsrisiko werden. Nachdem vor kurzem erst Softwarefehler bei Mercedes-Dieseln für Aufsehen sorgten, können nun Defekte in der elektronischen Steuerung der Motorkühlung bei Volvo-Personenwagen zur Überhitzung führen.

Der Fehler tritt bei den Modellen S60, S80, V70 und XC70 aus den Baujahren 2000 und 2001 auf, erklärte Volvo, einzelne Modelle aus dem Jahr 1999 seien ebenfalls betroffen. Die fehlerhaft arbeitende Elektronik hat Bosch an Volvo geliefert – wer für den Fehler, der vor allem bei langsamer Fahrt bei hohen Außentemperaturen zur Überhitzung führen kann, verantwortlich ist, steht laut Volvo noch nicht fest. Insgesamt 460.000 Fahrzeuge weltweit ruft der schwedische Hersteller daher in die Werkstätten zurück. Laut dpa erhalten in Deutschland rund 40.000 Besitzer eines Volvo-PKW eine Aufforderung zum Werkstattbesuch – der für die Halter zumindest kostenlos bleibt.

(aus: <http://www.heise.de/newsticker/meldung/51019>)

## The Patriot Missile

The Patriot missile defense battery uses a 24 bit arithmetic which causes the representation of real time and velocities to incur roundoff errors; these errors became substantial when the patriot battery ran for 8 or more consecutive hours.

As part of the search and targeting procedure, the Patriot radar system computes a "Range Gate" that is used to track and attack the target. As the calculations of real time and velocities incur roundoff errors, the range gate shifts by substantial margins, especially after 8 or more hours of continuous run.

The following data on the effect of extended run time on patriot operations from Appendix II of the report would be of interest to numerical analysts anywhere.

HOURS	REAL TIME (seconds)	CALCULATED TIME (seconds)	INACCURACY (seconds)	APPROXIMATE SHIFT IN RANGE GATE (meters)
0	0	0	0	0
1	3600	3599.9966	.0034	7
8	28800	28799.9725	.0275	55
20a	72000	71999.9313	.0687	137
48	172800	172799.8352	.1648	330
72	259200	259199.7528	.2472	494
100b	360000	359999.6667	.3333*	687

Tabelle 0.1: Divergence in the Range Gate of a PATRIOT MISSILE

a: continuous operation exceeding 20 hours-target outside range gate

b: Alpha battery [at Dhahran] ran continuously for about 100 hours

\* corrected value [GAO report lists .3433]

On February 21, 1991 the Partiot Project Office send a message to all patriot sites stating that very long run times "could cause a shift in the range gate, resulting in the target being offset". However the message did not specify "what constitutes very long run times". According to the Army officials, they presumed that the users would not run the batteries for such extended periods of time that the Patriot would fail to track targets. "Therefore, they did not think that more detailed guidance was required".

The air fields and seaports of Dhahran were protected by six Patriot batteries. Alpha battery was to protect the Dhahran air base.

On February 25, 1991, Alpha battery had been in operation for over 100 consecutive hours. That was the day an incomming Scud struck an Army barracks and killed 28 American soldiers.

On February 26, the next day, the modified software, which compensated for the inaccurated time calculation, arrived in Dhahran.

## **Kontenabrufverfahren startet wegen Softwareproblemen als Provisorium**

Das automatische Kontenabrufverfahren nach dem „Gesetz zur Förderung der Steuerehrlichkeit“, das ab dem 1. April die Abfrage der Kontostammdaten für einige Behörden möglich macht, startet mit Anlaufproblemen. Sie liegen vor allem darin begründet, dass die entsprechende Abfragesoftware der Stammdaten, die ab November 2003 zum Zwecke der Terroristenfahndung entwickelt wurde, nicht richtig skaliert. Diese Software wurde auf ca. 2000 Abfragen pro Tag durch die Polizeifahnder ausgelegt. Mit mehr als täglichen 50.000 Abfragen, die von Finanzämtern, Bafög- oder Sozialämtern ab dem 1. April erwartet werden, ist die Software hoffnungslos überfordert. Für die 18 bis 20 Millionen Konten, die jährlich nach dem Willen des Gesetzgebers gesucht werden sollen, wird derzeit eine völlig neue Schnittstellenspezifikation entwickelt und ein komplett neues Programm geschrieben. Bis dieses Programm für die automatische Abfrage durch die Sachbearbeiter fertig ist, muss die Abfrage wie bisher manuell erfolgen.

Bei dieser manuellen Abfrage reichen Polizeibehörden und Strafverfolger ihre Anfragen auf Papier oder per Fax oder E-Mail bei der Bundesanstalt für Finanzdienstleistungsaufsicht (BaFin) ein und bekommen die gewünschten Kontodaten auf demselben Wege zurück. Dieses Verfahren soll durch eine Suchmaske ersetzt werden, die jede Behörde aufrufen kann – wenn die dahinter liegende Abfragesoftware die Datenmengen bewältigen kann.

(aus: <http://www.heise.de/newsticker/meldung/58096>)

## **Buffer Overflow im Linux-Kernel**

Paul Starzetz von isec hat Details zu einer neuen Lücke im Linux-Kernel veröffentlicht, mit der ein Angreifer Programme mit Root-Rechten ausführen kann. Anders als bei vergangenen Veröffentlichungen von Starzetz, wurden die Hersteller aber offenbar nicht vorab informiert, etwa über die geschlossene Mailing-Liste Vendor-Sec. Nach seinen Angaben würde die Linux-Community Veröffentlichungen ohne Embargos von Distributoren bevorzugen. Um aber die Regeln der so genannten Responsible Disclosure einzuhalten, veröffentlicht er diesmal keinen Exploit-Code.

Der Fehler findet sich wieder einmal im Linux ELF-Binary-Loader, in dem Starzetz in der Vergangenheit bereits mehrere Lücken aufdeckte. Diesmal ist ein Buffer Overflow in der Funktion `elf_core_dump` schuld, der beim Aufruf einer weiteren Funktion (`copy_from_user`) mit einer negativen Längenangabe auftritt. Starzetz hat nach eigenen Angaben die Lücke bereits durch ein präpariertes

ELF-Binary demonstrieren können, das mit Kernel-Privilegien lief. Ein Proof-of-Concept-Programm ist seinem Advisory beigefügt, das aber nur den Kern des Problems demonstriert.

(aus:<http://www.heise.de/newsticker/meldung/59498>)

## **Auch Superhirne können irren - das Risiko Computer**

Lenkwaffen, Flugsteuerungen, Diagnosegeräte, Verkehrsleitsysteme, Dateien, Produktions-Steuerung – überall hat der Computer das Kommando übernommen. Doch nicht überall gibt er die richtigen Befehle. Mancher Irrtum schon hatte tödliche Folgen. Das Vertrauen in das elektronische Superhirn ist angeschlagen.

Sollten US-Kriegsschiffe, die mit dem computergestützten Waffensystem „Aegis“ ausgerüstet sind, in Zukunft wieder in Spannungsgebieten kreuzen, werden die verantwortlichen Offiziere dort mit der Angst leben, daß sich die Ereignisse des 3. Juli 1988 wiederholen könnten: Damals folgte der Kapitän des Kreuzers „Vincennes“, von elektronischen Befehlen unter Entscheidungsdruck gesetzt, der Logik des Computers, dessen Abtastsystem ein Verkehrsflugzeug mit einer Kampfmaschine verwechselte. Er gab den verhängnisvollen Befehl zum Abfeuern der Raketen. Alle 290 Insassen des iranischen Airbus kamen dabei ums Leben. ...

Aus anderer Quelle:

Auch der erste KI-Unfall, bei dem das „künstlich intelligente“ AEGIS-System des US-Kreuzers „Vincennes“ im Sommer 1988 einen zivilen Airbus mit einem MIG-Militärjet verwechselte, dürfte bei heutigem Kenntnisstand durch einen Konzeptfehler mitverursacht worden sein. Aus der „Sicht“ des einzelnen AEGIS-Systems werden alle Signale, die auf einem Richtstrahl innerhalb einer 300 Meilen umfassenden Überwachungszone entdeckt werden, einem einzelnen Objekt zugeordnet. So können ein Militär- und ein Zivil-Jet nur durch ein räumlich getrenntes System unterschieden werden. Offenbar hat das AEGIS-System aber weder Inkonsistenzen der Daten (militärische und zivile Transponder-Kennung) noch die unvollständige räumliche Auflösung dem verantwortlichen Kommandeur übermittelt, der im Vertrauen auf die Datenqualität den Befehl zum Abschluß von fast 300 Zivilisten gab. Offensichtlich ist in Streßsituationen eine menschliche Plausibilitätskontrolle nicht nur bei derart komplexen Systemen erschwert. Aus einem bis dahin fehlerfreien Funktionieren wird induktiv auf korrektes Verhalten im Ernstfall geschlossen. Daher sind besondere Hinweise auf inkonsistente und unvollständige „Datenlagen“ und gegebenenfalls Sperren gegen automatische Prozeduren zwingend erforderlich.

## Neueste Risikoinformationen/Softwareprobleme

... findet man unter: <http://catless.ncl.ac.uk/Risks/>:

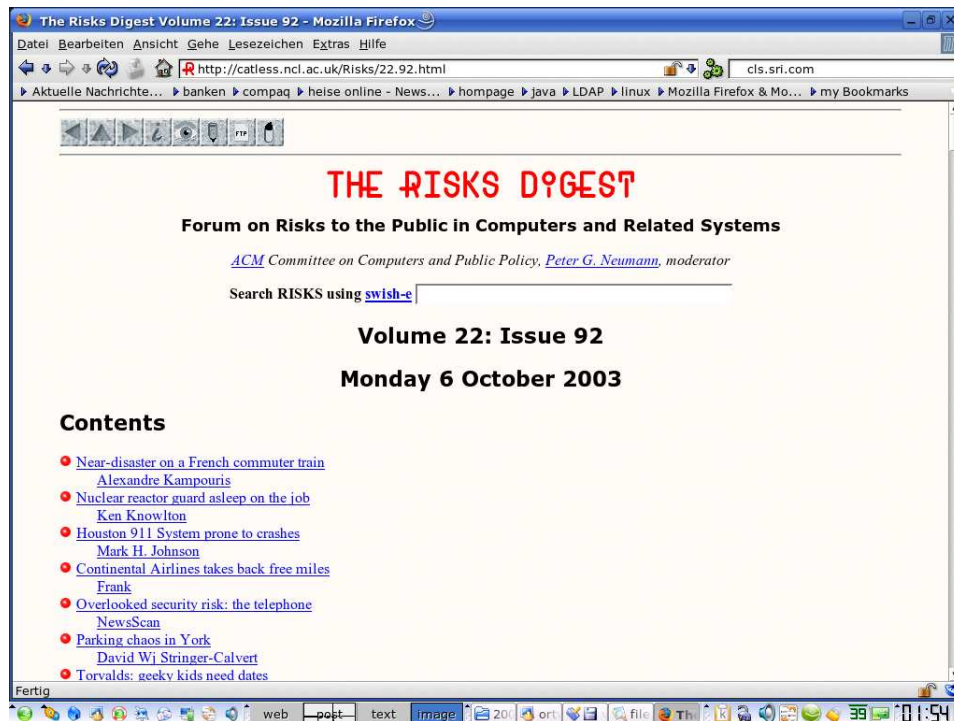


Abbildung 0.3: <http://catless.ncl.ac.uk/Risks/22.92.html>



# Programming by Contract im Umfeld der Formalen Methoden

## Programmverifikation

### Erläuterung

Programm-Verifikation ist ein systematischer Ansatz zum Nachweis der Fehlerfreiheit von Programmen. Dabei wird bewiesen, dass ein vorgegebenes Programm bestimmte wünschenswerte Eigenschaften besitzt. Bei sequentiellen Programmen geht es vor allem um die Ablieferung korrekter Ergebnisse und die Terminierung. Bei Programmen mit parallel ablaufenden Komponenten sind zusätzliche Eigenschaften wie Interferenzfreiheit, Deadlock-Freiheit und faires Ablaufverhalten wichtig.

(vergleiche <http://www.software-kompetenz.de/servlet/is/22224/>)

### Beschreibung

Die Voraussetzung für die Programm-Verifikation ist eine Spezifikation der Vor- und Nachbedingungen (pre- und post conditions) des Programms bzw. von Teilen davon. Solche Bedingungen stellen logische Aussagen dar, die bei jedem möglichen Programmdurchlauf an den betreffenden Stellen den Wahrheitswert *True* liefern müssen. Aus den Vor- und Nachbedingungen und dem vorliegenden Quellcode lassen sich mit Hilfe eines Verifikationswerkzeugs sog. Verifikationsbedingungen (verification conditions) ableiten, von deren Erfüllung die Korrektheit des Programms abhängt.

Die Verifikation lässt sich aus Aufwandsgründen zumeist nicht *flächendeckend* auf mittlere bis große Programme anwenden, dennoch kann die Sicherheit eines Programms beträchtlich erhöht werden, wenn Kernalgorithmen formal verifiziert werden. ... (Zitat aus: <http://www.software-kompetenz.de/?22225>)

### Ein klassisches Beispiel:

```
begin {a > 0, b ≥ 0}
  x := a; y := b;
  while y ≠ 0 do {gcd(a,b) = gcd(x,y) }
    begin r := x mod y;
      x := y;
```

```

        y := r
    end
    {x = gcd(a, b)}
end

```

(*entnommen*: Suad Alagić/Michael A. Arbib: THE DESIGN OF WELL-STRUCTURED AND CORRECT PROGRAMS, Springer-Verlag, New York, 1978)

## Programming by Contract

- **Invarianten** einer Komponente sind allgemeine, unveränderliche Konsistenzbedingungen an den Zustand einer Komponente, die vor und nach jedem Aufruf eines Dienstes gelten. Formal sind Invarianten boolesche Ausdrücke über den Abfragen der Komponente; inhaltlich können sie z.B. Geschäftsregeln (business rules) ausdrücken.
- **Vorbedingungen** (preconditions) eines Dienstes sind Bedingungen, die vor dem Aufruf eines Dienstes erfüllt sein müssen, damit er ausführbar ist. Vorbedingungen sind boolesche Ausdrücke über den Abfragen der Komponente und den Parametern des Dienstes.
- **Nachbedingungen** (postconditions) eines Dienstes sind Bedingungen, die nach dem Aufruf eines Dienstes erfüllt sind; sie beschreiben, welches Ergebnis ein Dienstaufruf liefert oder welchen Effekt er erzielt. Nachbedingungen sind boolesche Ausdrücke über den Abfragen der Komponente und den Parametern des Dienstes, erweitert um ein Gedächtniskonstrukt, das die Werte von Ausdrücken vor dem Dienstaufruf liefert.

(*vergleiche*: <http://userserv.fh-reutlingen.de/~hug/artikel/ForumWI01%20SdV.pdf>)

### Ein Beispiel in C++ mit Hilfe von Nana:

```

#define EIFFEL_CHECK CHECK_ALL
#include <set>
#include <vector>
#include <eiffel.h>
#include <nana.h>
...
void quicksort(double v[], int l, int h)
{
    REQUIRE(l <= h+1);
    ...
    ENSURE(A(int k=l, k<h, k++, v[k]<=v[k+1]));
};

```

```

void quicksort(double v[], int n)
{
    REQUIRE(n>=1);
    ID(multiset<double> v_old_contents(&v[0],&v[n]));
    ...
    ENSURE(A(int k=0, k<n-1, k++, v[k]<=v[k+1]));
    ID(multiset<double> v_contents(&v[0],&v[n]));
    ENSURE(v_old_contents == v_contents);
};

```

```

class name_list{
    ...
void name_list::put(const string& a_name)    // Push a_name into list
DO
    REQUIRE(/* name not in list */    !has(a_name));
    ID(set<string> contents_old(begin(),end()));
    ID(int count_old = get_count());
    ID(bool not_in_list = !has(a_name));
    ...
    ENSURE(has(a_name));
    ENSURE( (!not_in_list) || (get_count() == count_old + 1));
    ID(set<string> contents(begin(),end()));
    ENSURE( (!not_in_list) || (contents == contents_old + a_name));
END;
...
}

```

Analoge Beispiele in C++ mit Nana siehe etwa:  
<http://www.math.uni-wuppertal.de/~buhl/teach/exercises/>.

# Model based Specification

Die Spezifikation wird durch die Angabe einer (die Semantik definierenden) Referenzimplementierung vorgenommen, vorzugsweise in einer abstrakten, der mathematischen Schreibweise nahekommenden Spezifikationssprache.

## Invarianten mit Hilfe von VDM:

Klasseninvarianten zeichnen die gültigen Stati der Klassenexemplare aus:

### 1.9.4 Records und Invarianten

```
49.0 Koordinaten = compose Polar of
.1     Arg : ℝ
.2     Mod : ℝ
.3     end
.4 inv mk-Polar(r, Θ)  $\triangleq (r \geq 0) \wedge (0 \leq \Theta) \wedge (\Theta < 2\pi)$ 
```

oder kürzer

```
50.0 Polar :: Arg : ℝ
.1     Mod : ℝ
.2 inv mk-Polar(r, Θ)  $\triangleq (r \geq 0) \wedge (0 \leq \Theta) \wedge (\Theta < 2\pi)$ 
```

Hier ein weiteres Beispiel:

```
51.0 Date :: Day : ℕ1
.1     Month : ℕ1
.2     Year : ℕ1
.3 inv mk-Date(d, m, y)  $\triangleq$ 
.4 (1901 ≤ y) ∧ (y ≤ 2200) ∧
.5 (m ≤ 12) ∧
.6 d ≤ cases m :
.7   9, 4, 6, 11 → 30,
.8   2 → if ((y rem 4 = 0) ∧
.9         (y rem 100 ≠ 0)) ∨
.10        (y rem 400 = 0) then 29 else 28,
.11   others → 31
.12 end
```

52

In VDM (VDM++) ist es möglich, „rapid prototypes“ innerhalb der Spezifikationssprache mit Hilfe von ausführbaren Anweisungen zu erstellen, z.B.:

```

module MergeSort
  parameters
1.0   types Item
2.0   functions  $\leq : Item \times Item \rightarrow \mathbb{B}$ 
      .1   — (Item,  $\leq$ ) is totally ordered

  exports
3.0   functions MergeSort : Item*  $\rightarrow$  Item*

  definitions
    functions
4.0   is-ordered : Item*  $\rightarrow$   $\mathbb{B}$ 
      .1   is-ordered (l)  $\triangleq$ 
      .2    $\forall i, j \in \text{inds } l \cdot i < j \Rightarrow l(i) \leq l(j)$ 
;

5.0   is-Permutation : Item*  $\times$  Item*  $\rightarrow$   $\mathbb{B}$ 
      .1   is-Permutation (l1, l2)  $\triangleq$ 
      .2    $\forall e \in (\text{elems } l_1 \cup \text{elems } l_2) \cdot$ 
      .3    $\text{card } \{i \in \text{inds } l_1 \mid l_1(i) = e\} = \text{card } \{i \in \text{inds } l_2 \mid l_2(i) = e\}$ 
;

6.0   Merge : Item*  $\times$  Item*  $\rightarrow$  Item*
      .1   Merge (l1, l2)  $\triangleq$ 
      .2   cases mk- (l1, l2) :
      .3     mk- ([], l), mk- (l, [])  $\rightarrow$  l,
      .4     others  $\rightarrow$  if hd l1  $\leq$  hd l2
      .5       then [hd l1]  $\curvearrowright$  Merge (tl l1, l2)
      .6       else [hd l2]  $\curvearrowright$  Merge (l1, tl l2)
      .7   end
      .8   pre is-ordered (l1)  $\wedge$  is-ordered (l2)
      .9   post is-ordered (Merge (l1, l2))  $\wedge$  is-Permutation (l1  $\curvearrowright$  l2, Merge (l1, l2))
;

```

```

7.0      MergeSort : Item* → Item*
.1      MergeSort (l)  $\triangleq$ 
.2      cases l :
.3          [] → l,
.4          [e] → l,
.5      others → let  $l_1 \overset{\curvearrowright}{\sim} l_2 = l$  be st abs (len  $l_1$  - len  $l_2$ ) < 2 in
.6              let  $l_l = \text{MergeSort } (l_1)$ ,
.7                   $l_r = \text{MergeSort } (l_2)$ 
.8              in Merge ( $l_l, l_r$ )
.9      end
.10     pre true
.11     post is-ordered (MergeSort (l))  $\wedge$  is-Permutation (l, MergeSort (l))

end MergeSort

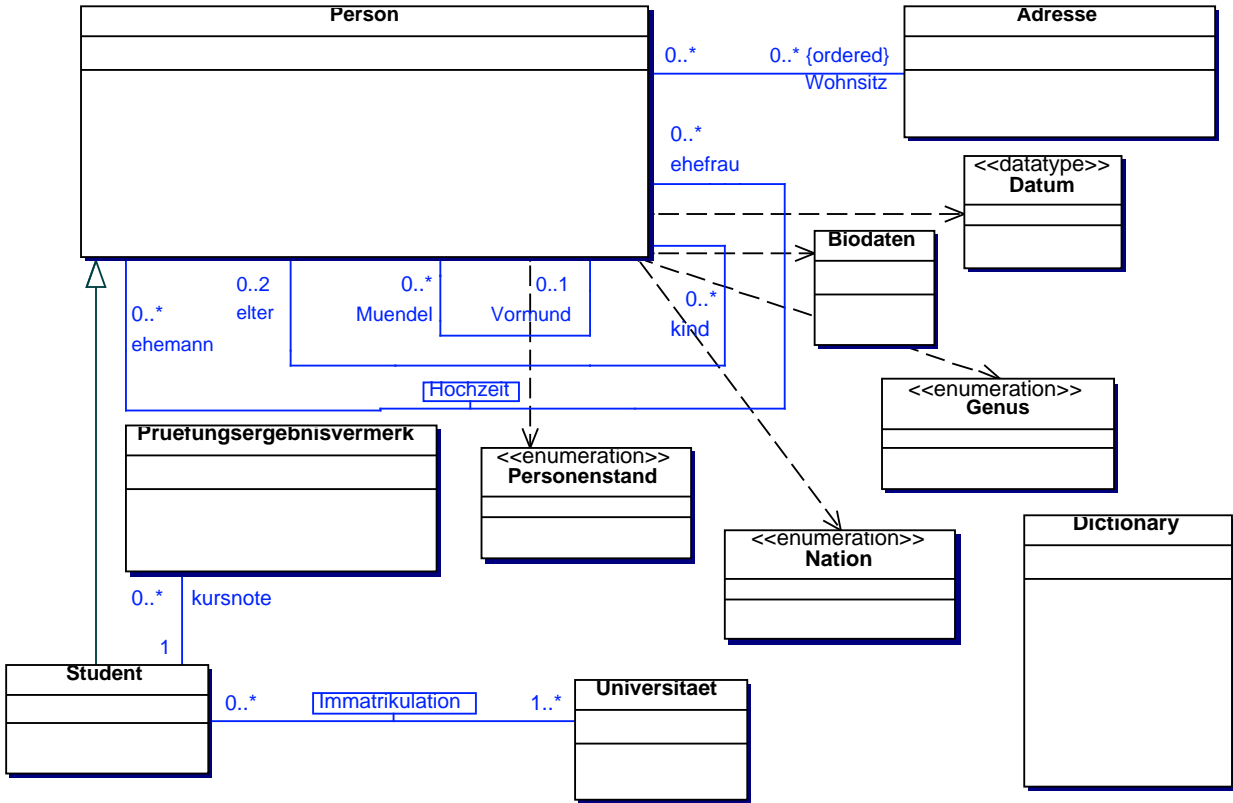
```

Da die Mathematik-nahe Schreibweise bei Programmierern nicht sehr beliebt ist und auch innerhalb von Quelldateien nicht direkt benutzt werden kann, setzt sich eine Programmiersprachen-ähnliche Schreibweise durch:

```

class Datum
    -- <<datatype>>
    functions
        static private gueltigesDatum : nat1 * nat1 * nat1 -> bool
        gueltigesDatum(t, m, j ) ==
            1920 <= j and
            j <= 2100 and
            m <= 12 and
            t <=
                cases m :
                    4, 6, 9, 11 -> 30,
                    2          -> if((j rem 4 = 0) and
                                   not (j rem 100 = 0))
                                   or (j rem 400 = 0)
                                   then 29
                                   else 28,
                    others      -> 31
            end;
    instance variables
        Tag : nat1;
        Monat : nat1;
        Jahr : nat1;
        inv gueltigesDatum(Tag, Monat, Jahr);
    values
        -- Konstanten
        public static invalidDatum = Datum(31, 12, 2100);
    operations
        -- Konstruktor ausführbar
        static Datum : nat1 * nat1 * nat1 ==> Datum
        Datum(t, m, j) ==
            ( Tag := t;
              Monat := m;
              Jahr := j;
            )    pre gueltigesDatum(t, m, j);
end Datum

```



und zugehörige formale Spezifikationen:

```
private Cousins : () ==> set of Person
Cousins() ==
  let Grosseltern = dunion { e.elter | e in set elter } in
  let OnkelUndTanten =
    dunion { g.kind | g in set Grosseltern } \
    elter in
  let CousinsUndCousinen=dunion {o.kind|o in set
    OnkelUndTanten} in
  return {c|c in set CousinsUndCousinen&c.Geschlecht=
    <maennlich>};
```

instance variables

```
Hochzeitsort : seq of char;
inv Hochzeitsort <> "";

public Hochzeitsdatum : Datum;

public Familienname : seq of char := ehemann.Nachname;
inv Familienname <> "";

public Scheidungsdatum : Datum := Datum'invalidDatum;
inv Scheidungsdatum.groesserGleich(Hochzeitsdatum);

public ehemann : Person;
inv ehemann.Alter(Hochzeitsdatum) >= 14;
inv ehemann.Geschlecht = <maennlich>;
inv self in set ehemann.hochzeit;

public ehfrau : Person;
inv ehfrau.Alter(Hochzeitsdatum) >= 14;
inv ehfrau.Geschlecht = <weiblich>;
inv self in set ehfrau.hochzeit;
```

**Literatur:** John Fitzgerald u.a.: Validated Designs for Object-oriented Systems, Springer-Vlg., London, 2005.

# Model Constraints

... an UML-Modelle:

```
context Dictionary::put( k : OclAny, v : OclAny )
pre : not has(k)
post : has(k)
post : value_for(k) = v
post : keys@pre->asSet()->including(k) = keys->asSet()
post : keys@pre->forall( k | value_for@pre(k) = value_for(k) )
```

-- In OCL keine ausführbare Spezifikation der Operation möglich!

```
context Dictionary::remove( k : OclAny )
pre : has(k)
post : not has(k)
post : keys@pre->asSet() = keys->asSet()->including(k)
post : keys->forall( k | value_for@pre(k) = value_for(k) )
```

und in der Fallstudie Personenstandsdaten:

```
context Person
def: keineZeitweiseBigamie( s : Set(Hochzeit) ) : Boolean =
  s->forall( h1, h2 | h1 <> h2 implies
    (/* h1.Scheidungsdatum >= */ h1.Hochzeitsdatum
      >= h2.Scheidungsdatum
      /* >= h2.Hochzeitsdatum */ or
      /* h2 Scheidungsdatum >= */ h2.Hochzeitsdatum
      >= h1.Scheidungsdatum
      /* >= h1.Hochzeitsdatum */ )
  )
```

```
context Person
inv : Geschlecht = Genus::maennlich implies
      istMaennlicherVorname(Vorname)
```

```
inv : Geschlecht = Genus::maennlich implies
      Hochzeit[ehemann]->size() > 0 implies
      keineZeitweiseBigamie(Hochzeit[ehemann])
...
inv : Status = Personenstand::tot implies
      Sterbedatum <> Datum::invalidDatum
...
```

**Zu OCL2 vergleiche:** Jos Warmer u.a.: The Object Constraint Language, second edition, Addison-Wesley, 2003



# 1 Qualitätsanforderungen an SW-Produkte

## A. Produktorientiert:

1. funktionale Korrektheit
2. funktionale Vollständigkeit
3. Robustheit gegenüber dem Benutzer
4. Benutzerfreundlichkeit
5. Effizienz in Laufzeit
6. Effizienz im Arbeitsspeicherbedarf
7. Effizienz im Plattenspeicherbedarf
8. Integrität (gegenüber unauthorisierten Änderungen)
9. Kompatibilität, Integrationsfähigkeit, Standards

## B. Projektorientiert:

1. Überprüfbarkeit
2. Verständlichkeit
3. Wartbarkeit
4. Änderbarkeit, Erweiterbarkeit
5. Portierbarkeit
6. Wiederverwertbarkeit

## 1.1 Prinzipien der ordnungsgemäßen Programmerstellung

1. Konstruktive Voraussicht und methodische Restriktion
2. Strukturierung
3. Modularisierung
4. Lokalität
5. Integrierte Dokumentation
6. Standardisierung
7. Funktionale und informelle Bindung
8. Schmale Datenkopplung
9. Vollständige Schnittstellenspezifikation
10. Lineare Kontrollstrukturen
11. Verbalisierung

## 1.2 Spezifikation einer abstrakten Datenkapsel

### 1.2.1 Axiomatische Spezifikation

TYPES	
STACK[X]	
FUNCTIONS	
empty:	STACK[X] → BOOLEAN
new:	→ STACK[X]
push:	X x STACK[X] → STACK[X]
pop:	STACK[X] ↦ STACK[X]
top:	STACK[X] ↦ X
PRECONDITIONS	
pre pop (s: STACK[X])	= (not empty(s))
pre top (s: STACK[X])	= (not empty(s))
AXIOMS	
for all x:X, S : STACK[X]:	empty(new())
	not empty (push(x,S))
	top (push(x,S))=x
	pop (push(x,S))=S
Vollständigkeit + Widerspruchsfreiheit (+ Unabhängigkeit)	

### 1.2.2 Beschreibende (denotationale) Spezifikation

$Queue = Qelem^*$ $q_0 = [ ]$  <b>ENQUEUE</b> ( $e : Qelem$ ) <b>ext wr</b> $q : Queue$ <b>post</b> $q = \overleftarrow{q} \frown [e]$  <b>DEQUEUE</b> ( $e : Qelem$ ) <b>ext wr</b> $q : Queue$ <b>pre</b> $q \neq [ ]$ <b>post</b> $\overleftarrow{q} = [e] \frown q$  <b>ISEMPTY</b> ( $r : \mathbb{B}$ ) <b>ext rd</b> $q : Queue$ <b>post</b> $r \Leftrightarrow (len\ q = 0)$	„mathematische“ Modellierung mit Hilfe von Folgen, Mengen, ...
---	--

## 1.3 Prinzipien der Modularisierung:

1. Module sollten **syntaktischen Einheiten** der Programmiersprache entsprechen.
2. Module sollten **mit möglichst wenigen anderen Modulen** „kommunizieren“.
3. „Kommunizierende“ Module sollten so **wenig** wie möglich **Informationen (Daten) austauschen**.
4. Jeder **Datenausch** zweier „kommunizierender“ Module muß **offensichtlich** in der Modulspezifikation (und nicht indirekt) kenntlich gemacht werden.
5. Alle **Daten** eines Moduls sollten **nur diesem bekannt** sein (außer im Falle einer gezielten Exportierung an möglichst wenige Nachbarmodule).
6. Ein Modul sollte **abgeschlossen und offen** sein.

## 1.4 Typen der Modularisierung

1. modulare **Zerlegbarkeit** (z.B. Top-Down-Design)
2. modulare **Zusammenfügbarkeit** (z.B. UNIX-Filter)
3. modulare **Verständlichkeit** (d.h. jede Modulbeschreibung selbsterklärend)
4. modulare „**Stetigkeit**“

Kleine Spezifikationsänderungen wirken sich nur in **wenigen** Modulen aus. (Z.B. dyn. Felder, symbolische Konstanten, ...)

5. modularer „**Schutz**“

Fehler/Ausnahmebedingungen bleiben in ihrer Auswirkung auf nur **wenige** Module beschränkt. (Z.B. direkte Konsistenzüberprüfung von Tastatureingaben, ...)

## 2 Wiederverwendbarkeit

Vermeide es, das Rad immer wieder neu zu erfinden!

1. **Algorithmen (Programme)** lösen i. allg. eine Klasse von Problemen, die durch Eingabewerte parametrisiert sind.
2. **Unterprogramme** (Funktionen, Prozeduren, Operatoren) lösen eine Klasse von Problemen: Gemäß dem Prinzip der methodischen Restriktion sind dabei die einzelnen Parameter jeweils Werte des Wertebereiches eines festen Typs.
3. **Unterprogramme mit konformen Feldparametern** (in Pascal bzw. open-array-Parameter in Modula2) erlauben es Parametern, einer Klasse von Feldern anzugehören (variable Dimension);

```
PROCEDURE EuklNorm (v:ARRAY OF REAL): REAL;
```

4. **Dynamische Felder / Teilfeld-Selektoren** erlauben einen in der Dimension noch nicht festgelegten Feldtyp:

```
TYPE vector = ARRAY[*] OF REAL;  
a := t[*],2];  
... t[min, k:l] ...
```

5. **Polymorphismen** (d.h. Überladen) **von Funktionen/Operatoren** erlauben die Benutzung einer mit demselben Namen versehenen Klasse von Funktionen, in denen jeder Parameter aus einer (disjunkten) Vereinigung von Typen stammen darf:

```
writeln(x : real);           k := i * j;  
writeln(i : integer);       z := x * y;  
...
```

6. **Unterprogramme als Parameter** anderer Unterprogramme erlauben Algorithmen für eine Klasse von Unterprogrammen gleicher Signatur:

```

function Bisection (function f(x : real) : real;
                   xLeft, xRight      : real;
                   success             : boolean
                   ) : real;

```

7. **Generizität** ermöglicht Parametrisierung nach Typen:

```

generic
  type T is private;
  procedure swap (x, y : in out T) is t : T
  begin
    t := x; x := y; y := t
  end swap
  :
  procedure int_swap is new swap (INTEGER);

```

**Eingeschränkte Generizität** schränkt die aktuellen Typ-Parameter ein:

```

generic
  type T is private;
  with funktion " $\leq$ " (a, b : T) return BOOLEAN is <>;
  funktion minimum (x, y : T) return T is
  begin
    if  $x \leq y$  then return x;
    else return y
    end if
  end minimum

```

(Ähnliches kann durch Textprozessoren oder die typunsichere Benutzung des typungebundenen Zeigers ADDRESS erreicht werden.)

# 2.1 Begriffshierarchien

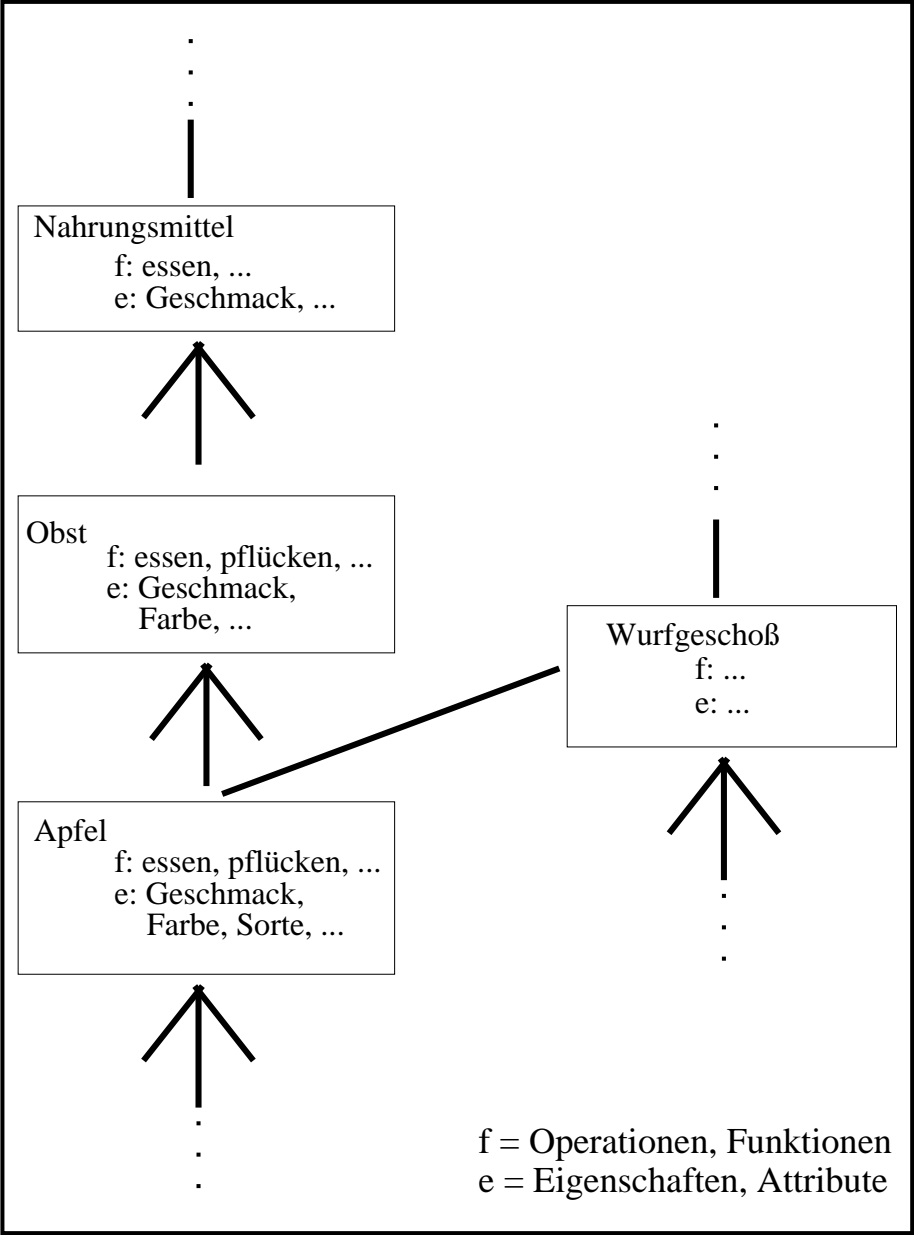


Abbildung 2.1: Begriffshierarchien

## 2.2 Objekthierarchien als strukturierte Modulsammlungen: Beispiele aus Eiffel

*Objektorientierte* Programmiersprachen ermöglichen die **Datenkapselung** und eine **evolutionäre** Programmerstellung:

Nutze vorhandene Objektklassen (Typen) oder erzeuge neue Objektklassen, wobei **bei Teilstrukturgleichheit** möglichst viel durch **Vererbung** existierender Klassen realisiert wird.

### 2.2.1 Vererbung und Erweiterung

Namensänderung

Neues *feature*

```
class Multiindex inherit
  ARRAY[CARDINAL] rename
                        count as Dimension,
                        clear_all as Null
  end;

feature
  abs: CARDINAL is
    require not empty
    do ...
    ensure
      - - abs = For all i:lower..upper:
        SUM item(i)
    end - - abs
end - - class Multiindex
```

## 2.2.2 Vererbung und Abänderung

alternative Implementierung →

alternative Implementierung →

auch die Vorbedingung wird geerbt →

zusätzliche Nachbedingung →

alte Invariante wird geerbt →

```

class Multiindex inherit
  ARRAY[CARDINAL]rename
    count as Dimension,
    clear_all as Null
  redefine
    abs,
    put,
    make,
  end;

feature
  abs: CARDINAL;
  put(v:like item; i:INTEGER)
    - - replace i-th entry, if in index interval, by v
    :
  ensure then
    abs = old abs - old item(i) + v
  end - - put
  :
invariant
  - - abs = For all i:lower..upper:Sum item(i)
end - - class Multiindex

```

Wenn aus Effektivitätsgründen **redundante Daten** angelegt werden, so sollten diese Redundanzen spezifiziert werden!

Es gelten folgende Regeln bei der Vererbung (von is-a-Methoden):

- Vorbedingungen können in einer Kindklasse abgeschwächt werden.
- Nachbedingungen in einer Kindklasse müssen stärker sein als diejenigen der Elterklasse.
- Invarianten in der Kindklasse müssen ebenfalls stärker als in der Elterklasse sein.

Dann ist ein echtes *Subcontracting* realisiert.

Bemerkung: Es reicht die Kindnachbedingung im Falle des Eintreffens der Eltervorbedingung stärker als die Elternachbedingung zu realisieren. Im Falle „Kindvorbedingung **and not** Eltervorbedingung“ darf die Kindnachbedingung frei gewählt werden.

### 2.2.3 Generizität

```
class STACK[T]
feature
  :
end - - class STACK[T]
```

### 2.2.4 Eingeschränkte Generizität

```
class VECTOR[T -> ADDABLE]
feature
  :
end - - class VECTOR
```

### 2.2.5 Polymorphie und „late binding“

```
class Rectangle inherit
  POLYGON redefine perimeter
  end
feature {NONE}
  side1: REAL;
  side2: REAL;
feature {ANY}
  perimeter: REAL is
  do
    Result := 2 * (side1 + side2)
  end - - perimeter
  :
end - - class Rectangle
```

Wegen des Zusammenhangs  $\text{Rectangle} \subset \text{POLYGON}$  und  $\text{MethodenVon}(\text{Rectangle}) \supset \text{MethodenVon}(\text{POLYGON})$  gilt in der Anwendung:

*perimeter*  
für die Menge  
aller Erben von  
POLYGON  
verfügbar  
unter gleichem  
Namen.

```

:
p : POLYGON;
r : Rectangle;
:
!!p; !!r;
:
print (p.perimeter);      - - perimeter aus
:                          - - POLYGON
p := r
print (p.perimeter);      - - perimeter aus
:                          - - Rectangle

```

## 2.2.6 Aufgeschobene Feature-Implementierungen

```

deferred class Stack[T]
feature
  nb_elements : INTEGER is
    deferred
  end - - nb_elements
  empty : BOOLEAN is
  do
    Result := (nb_elements = 0)
  ensure Result = (nb_elements = 0)
  end - - empty
:
end - - class STACK[T]

```

... dienen der partiellen Implementierung einer Gruppe möglicher Implementierungen (Schablone). Sie stehen somit in Konkurrenz und ergänzen generische Klassen.

- "Objektorientiertes" Programmieren (als Alternative zum funktionalen Top-Down-Entwurf und zum datengesteuerten Entwurf nach Jackson) ist die Softwarekonstruktion mit Hilfe der **Adaption** von Sammlungen abstrakter Datentyp-Implementierungen.
- Unterklassen können sich von ihren Basisklassen unterscheiden durch:
  - 1) mehr Operationen
  - 2) mehr Daten (Attribute)
  - 3) eingeschränkte Wertebereiche der Daten
  - 4) alternative Implementierungen

## 2.3 Resümee

1. Objektorientiertes Programmieren setzt eine gute Kenntnis der vorhandenen Klassenhierarchien voraus! Diese sind heute jedoch häufig nicht ausreichend dokumentiert (fehlende Spezifikation, fehlende Fixierung der Design-Ideen, ...). Häufig steht nur ein Browser zur Betrachtung der Quellen der Klassen zur Verfügung, und der Programmierer muß sich selbst den Durchblick durch die Konzeption der Klassenbibliotheken erkämpfen.
2. Einige **objektorientierte** Sprachen bieten gar keine mitgelieferten Klassenbibliotheken an. Andere haben sprachspezifisch bzw. sogar herstellerspezifisch eigene — zwar häufig an Smalltalk angelehnte, aber dennoch in wichtigen Details abweichende — Klassenhierarchien. Für viele Gebiete in der Informatik/Mathematik/Anwendungswissenschaft fehlen geeignete Klassenbibliotheken gänzlich.
3. Geordnete **evolutionäre** objektorientierte Entwicklung im Team erfordert ein richtiges Management (open-close-Phasen, Versions-Management, ...)
4. Objektorientierte Programmiersprachen sollten syntaktische Sprachmittel für **Zusicherungen** besitzen (mindestens Aussagenlogik, besser **Prädikatenlogik**). Diese sollten **in** den geforderten **Klassenhierarchien** (zumindest in Kommentarform) **intensiv genutzt** werden. Eine etwa VDM ähnliche Syntax wäre gewiß interessant.

Wir müssen anspruchsvoller werden im Hinblick auf die Verlässlichkeit und die Qualität unserer Software. Die Benutzer müssen kritischer werden und weniger bereit, Softwareerzeugnisse geringer Qualität zu akzeptieren.

(Zitat: Robert L. Baber: Softwarereflexionen, Springer-Verlag)

## Forschungsministerium fördert Standard für IT-Sicherheit

Trotz des flächendeckenden Einsatzes von Computersystemen in sicherheitsrelevanten Bereichen fehlt bislang eine standardisierte Methode, die das fehlerfreie Funktionieren solcher Systeme garantiert. Das **Bundesministerium für Bildung und Forschung** (BMBF) will nun Arbeiten fördern, bei denen mit Methoden der Verifikation der so genannte geschlossene integrierte Korrektheitsbeweis erbracht werden kann. Damit sollen sich Fehler bereits im Entwurf von autonomen oder integrierten Computersystemen erkennen und korrigieren lassen – eine sorgfältige Spezifikation vorausgesetzt. Alle möglichen Fehlersituationen könnten aber nur dann abgefangen werden, wenn bereits in der Planung die entsprechenden Einsatzszenarien definiert wurden, betonte Projektleiter Prof. Dr. Wolfgang Paul gegenüber heise Security.

Für die erste zweijährige Forschungsphase werde das BMBF 7,2 Millionen Euro zur Verfügung stellen, teilte das Ministerium am heutigen Mittwoch in Berlin mit. An dem Projekt beteiligen sich neben der **Universität Saarland** unter anderen auch die TUs Darmstadt, Karlsruhe, München sowie Infineon, T-Systems und BMW.

Die Entwicklung eines integrierten Korrektheitsbeweises gilt zurzeit als eine der größten Herausforderungen der Informatik. Er soll die Funktionen bei der Entwicklung von Hard- und Systemsoftware bis zur Netzwerk- und Anwendungsebene laufend überprüfen. Zunächst sollen die mathematischen Grundlagen entwickelt, vollständig formalisiert und für Informatikanwendungen in den Bereichen Embedded Systems, Kommunikation und Anwendungssoftware erschlossen werden. Darauf aufbauend sollen die Projektpartner Demonstratoren entwickeln und mit ihnen Computersysteme für Chipkarten, Telekommunikation und Automobilelektronik von der Hardware bis zur Anwendungssoftware überprüfen. Im Rahmen des Projektes werden auch Softwaretools entwickelt, die den Verifikationsprozess unterstützen. (dab/c't)

**Link:** <http://www.heise.de/newsticker/data/dab-01.10.03-002/>

**Restinhaltsübersicht:** (siehe Vorlesungsmitschrift)

3. Softwarequalitätssteigerung mit normalen C/C++-Sprachmitteln

- 3.1 Umgangssprachliche (informelle) Spezifikation
- 3.2 Unbeachteter Integer-Over-/Underflow
- 3.3 assert() in C/C++
- 3.4 Sicher nutzbare Klasse statt C-enum
- 3.5 Compile Time Assertions
- 3.6 Exceptions in C++
- 3.7 Ausnahmebedingungen in VDM/VDM++
- 3.8 Weitere Möglichkeiten von Constraints (... mit ANNA-Beispielen)
- 3.9 Contracting und Subcontracting

4. Sprachabstraktionen zur guten Spezifikation/  
Nutzung abstrakter Bibliotheken (z.B. newmat10)

5. Nana - Contracts in C/C++-Programmen  
Installation, Modifikation und erste Anwendungen

6. Richard Mitchell/Jim McKim: Design by Contract, by Example

... hier mit Beispielen in C++ mit Nana (statt mit Eiffel-Beispielen):

6.1 Vorüberlegungen zu PbC

6.2 Elementare Prinzipien (Regeln) des PbC  
Contracts für die Klasse Day

6.3 Anwendungsbeispiel der sechs PbC-Regeln: Klasse mydictionary  
Erste (Ausführungs-)Richtlinien

6.4 nana-Quantoren und Rahmenbedingungen (Frame-Regeln)

6.4.1 Q.h

6.4.2 Qstl.h

- Quantoren und implementierungsspezifischer Komponentenzugriff
- Kopierkonstruktor und Komponentenzugriff mit den grundlegenden Observatoren
- STL-Container-Hilfsobjekte und STL-Methoden im Contract
- neue grundlegende STL-Container-wertige grundlegende Observatoren

## 6.5 Weitere Richtlinien

```
// Exemplar vollständig festgelegt durch die Werte von
// ... (grundlegende Observatoren)
// ...
```

## 6.6 Constraints bei der Vererbung

### 6.6.1 Invarianten

### 6.6.2 geschützte Nachbedingungen der Elternklassen

## 6.7 Iteratoren

- Iteratoren und der Operator\* als grundlegender Observator

## 7. Ausblick: PbC/DbC im neuen C++-Standard?

### Links:

<http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2004/n1613.pdf>

[http://www.codeguru.com/Csharp/.NET/net\\_general/article.php/c4677/](http://www.codeguru.com/Csharp/.NET/net_general/article.php/c4677/)

<http://www.codeproject.com/cpp/DesignByContract.asp>

<http://dotnet.sys-con.com/read/38959.htm>

<http://www.awprofessional.com/bookstore/product.asp?isbn=0201634600>  
name\_list9.cc

[http://www.sgi.com/tech/stl/table\\_of\\_contents.html](http://www.sgi.com/tech/stl/table_of_contents.html)

<http://cppreference.com/cppstl.html>

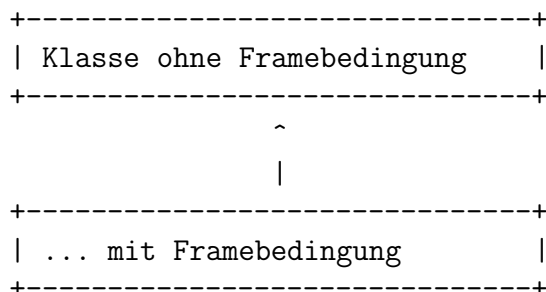
[http://en.wikipedia.org/wiki/Standard\\_Template\\_Library](http://en.wikipedia.org/wiki/Standard_Template_Library)

### PbC-Regeln:

- Vermeide statusändernde Methoden, die einen Wert liefern! (Observator **oder** Modifikator)
- Unterscheide grundlegende von abgeleiteten (redundanten) Observatoren.
- Schreibe für jeden abgeleiteten Observator eine Nachbedingung mit Hilfe der (aller) grundlegenden Observatoren.
- Schreibe für jeden Modifikator Nachbedingungen, die mit Hilfe der (aller) grundlegenden Observatoren den Inhalt des Klassenexemplars nach Methodenende in seiner Relation zum Exemplarinhalt bei Methodenbeginn **exakt** beschreiben. Nutze implizite (als Kommentar) oder explizite Frame-Bedingungen.
- Schreibe für alle Methoden Vorbedingungen (an Parameter bzw. Exemplarinhalt).
- Schreibe und benutze Invarianten, die gültige von ungültigen Exemplaren trennen.

### Leitlinien:

- Nutze technische Einschränkungen, wo immer erforderlich: z.B. Zeiger != 0, nicht-leere Container, nichtidentische Exemplare, ...
- In Vorbedingungen genutzte Observatoren sollten effizient berechnet werden. Notfalls führe neue effiziente abgeleitete Observatoren ein und benutze sie in den Vorbedingungen. Die neuen effizienten Observatoren benötigen Nachbedingungen, die ihre Konsistenz zu den grundlegenden Observatoren sicherstellen.
- Attribute haben keine Nachbedingungen. Benutze deshalb die Klasseninvariante für Contracts (oder Nachbedingungen mit get-Methoden).
- Nachbedingungen von virtuellen Methoden sollten die Form `ENSURE (!Vorbedingung || Nachbedingung)` haben; Invarianten sollten `protected als virtual bool invariant() const` deklariert werden.
- Nutze die Vererbung:



um dem wiederverwendenden Nutzer die Wahl zwischen der Verwendung der effizienten oberen oder sicheren unteren Klasse zu überlassen.

- Nutze die Vererbung analog z.B. für:

Klasse ohne Framebedingungen

Klasse mit Framebedingungen

... mit unzugänglichen (`private`) Observatoren,  
die weiter oben lediglich für die Contracts  
genutzt werden

... mit benutzerfreundlichen Methodenvarianten  
ohne Vorbedingungen

...

### Reformulierte Regeln:

1. Formuliere *grundlegende Observatoren*, die den Zustand eines Exemplars vollständig beschreiben können und eine *Klasseninvariante*, die gültige von ungültigen Exemplaren trennt. Falls grundlegende Observatoren mit Parametern existieren, so gib den zur vollständigen Exemplarbeschreibung nötigen Parameter-Wertebereich an. Im Contract sollte kein Bezug auf Implementierungsdetails sondern lediglich auf die grundlegenden Observatoren genommen werden! Nachbedingungen von grundlegenden Observatoren spezifizieren deshalb lediglich Konsistenzbedingungen zwischen den Methoden. Observatoren sind `const`-Methoden.

2. *Abgeleitete Observatoren* sind i.a. besser lesbar als eine (komplizierte) Kombination grundlegender Observatoren, können evtl. effizienter implementiert sein und sollten dann in Vorbedingungen unbedingt statt der grundlegenden Observatoren benutzt werden. Sie sind `const`-Methoden. Ihre Nachbedingungen sollten die Return-Werte mit Hilfe der grundlegenden Observatoren vollständig spezifizieren.

3. Konstruktoren (default, Kopier-): ...

4. Zuweisungsoperator: ...

5. `operator==`, `operator!=`: ...

6. Destruktor: ...

7. Modifikatoren: ...

8. `friend`-Methoden und Operatoren: ...