Materialsammlung
Programming by Contract

Prof. Dr. Hans-Jürgen Buhl

Wintersemester 2003/04

Bergische Universität Wuppertal
Fachbereich C — Mathematik

Inhaltsverzeichnis

1 Softwarequalität

Haftungsausschluß

Die Überlassung dieser Baupläne erfolgt ohne Gewähr. Der Planer 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 entscheidenen Phasen des Baus und nach Fertigstellung geeignete Tests durchzuführen sind und daß die üblichen Vorsichtsmaßnahmen zum Schutz des Lebens der Bauarbeiter zu treffen sind.

2. Haftung
Wir werden immer bemüht sein, ihnen einwandfreie Software zu liefern. Wir können aber keine Gewähr dafür übernehmen, daß die Software unterbrechungs- oder fehlerfrei läuft und daß die in der Software enthaltenen Funktionen in allen von Ihnen gewählten Kombinationen ausführbar sind. Für die Erreichung eines bestimmten Verwendungszwecks 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.

Fallbeispiele:

Chaos an Hannovers Geldautomaten

[05.10.2003 13:00]

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 Geldinstitutes, Stefan Becker. Beim Hochfahren eines Servers habe sich aber ein Fehler eingeschlichen, sodass die Geldautomaten nicht mehr funktionierten. Die Sparkasse öffnete stattdessen fünf Filialen, damit Kunden etwa in Einkaufszonen Bargeld abheben können. (*dpa*) / (**se[1]**/c't)

URL dieses Artikels:
http://www.heise.de/newsticker/data/se-05.10.03-003/

Links in diesem Artikel:
[1] mailto:se@ct.heise.de

EDV-Ausfall legt Post-Filialen lahm

[14.09.2002 15:16]

Ein kompletter Ausfall des EDV-Systems EPOS der **Deutschen Post[1]** und der **Postbank[2]** sorgte heute für jede Menge Ärger bei den Kunden. heise online erreichten etliche Hinweise, wonach wegen der Störung weder Pakete, Einschreiben noch Bankgeschäfte an den Schaltern erledigt werden konnten. Ein Sprecher der Post bestätigte gegenüber heise online, der Ausfall sei durch ein fehlerhaftes Update der Software zu Stande gekommen. Man arbeite mit Hochdruck daran, den Fehler zu beheben. Das System soll in den frühen Nachmittagsstunden wieder laufen. Die Postangestellten würden vermutlich bis dahin die anstehenden Vorgänge - etwa Einschreiben - von Hand aufnehmen und ins System einarbeiten, sobald es wieder zu Verfügung steht. Die zirka 7600 Geldautomaten der Postbank sollen von dem Ausfall nicht betroffen sein. (**tig[3]**/c't)

URL dieses Artikels:
http://www.heise.de/newsticker/data/tig-14.09.02-002/

Links in diesem Artikel:
[1] http://www.deutschepost.de/
[2] http://www.postbank.de
[3] mailto:tig@ct.heise.de

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, Dauer, Intensität und Bewegungskurve der Strahlungsquelle, mit einem Bildschirm-„Menü“ eingegeben werden.

Als mehere Patienten berichteten, sie hätten bei Behandlungsbeginn das Gefühl gehabt, „ein heißer Strahl durchdringe“ sie, wurde dies vom Hersteller als „unmöglich“ zurückgewiesen. Erst nach dem Tode 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 Cursor-Taste modifiziert wurde. Um aufwendige Umprogrammierung zu vermeiden, wollte der kanadische Hersteller die Benutzung der Cursor-Taste verbieten bzw. diese auszubauen 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:

Softwarefehler: Kleine Ursache, große Wirkung

Fünf – Null, tippte der Operator in die Tastatur und erwartete, daß die Magnetbahn auf 50 Stundenkilometer beschleunigen würde. Doch nichts geschah. Wieder tippte er fünf – null, und vergaß diesmal nicht die „Enter"-Taste zu betätigen, mit der die Daten erst in den Rechner abgeschickt werden. Die insgesammt eingegebene Tastenfolge „fünf – null – fünf – nul" interpretierte 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 Crash 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 mangelhafter 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.

... Fehler enthalten, und niemand kann ein Rezept anbieten, das eine absolut fehlerfreie Software garantiert. Auch bei der Software gibt es mithin ein nie völlig zu beseitigendes „Restrisiko", das es jedoch zu minimieren gilt. Auf einem kürzlich beim TÜV-Rheinland veranstalteten Symposium hatten 120 Experten diese Problematik diskutiert. Wie sich Fehler in einem Programm erkenne lassen, war dabei eine Kernfrage. Verschiedenste Prüfverfahren bis hin zu spezieller Analyse-Software, sollen die möglicherweise in einem Programm enthaltenen Fehler entdecken.

Doch bei aller Kunst – heimtückische Fehler, die sich später nur bei ganz bestimmten Betriebsumständen bemerkbar machen, lassen sich so in der Regel nicht aufspüren. Eher hilft da schon das Austesten des Programms unter simulierten Betriebsbedingungen. ...

Patriot missile:

The Patriot missile defence 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 everywhere.

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

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 Patriot Project Office sent 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 26, 1991, Alpha battery had been in operation for over 100 consecutive hours. That's the day an incoming 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.

AEGIS:

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 tötliche Folgen. Das Vertrauen in das elektronische Superhirn ist angeschlagen.

Von HARALD WATERMANN

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. ...

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 könne 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 Abschuß 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.

1.1 Qualitätsanforderungen an Software-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 unautorisierten Änderungen)
9. Kompatibilität, Integrationsfähigkeit, Standards

B. **Projektorientiert**

1. Überprüfbarkeit
2. Verständlichkeit
3. Wartbarkeit
4. Änderbarkeit, Erweiterbarkeit
5. Portierbarkeit
6. Wiederverwendbarkeit[1]

[1] siehe Kapitel 2

1.2 Konstruktive Methoden zur Steigerung der Softwarequalität

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**

Die Prinzipien 2 bis 9 werden besonders durch die Verwendung von abstrakten Datentypen unterstützt.

1.3 Formale Spezifikation

Spezifikation einer abstrakten Datenkapsel:

TYPES
STACK[X]
FUNCTIONS

empty:	STACK[X]	$\rightarrow$	BOOLEAN
new:		$\rightarrow$	STACK[X]
push:	X$\times$STACK[X]	$\rightarrow$	STACK[X]
pop:	STACK[X]	$\tilde{\rightarrow}$	STACK[X]
top:	STACK[X]	$\tilde{\rightarrow}$	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

Vollständig + Widerspruchsfrei (+ Unabhängig)

oder denotational (etwa mit „**V**ienna **D**evelopement **M**ethod“)

Queue = Qelem*
$q_0 = [\,]$

1.0 *ENQUEUE* $(e : Qelem)$
.1 ext wr $q : Queue$
.2 post $q = \overleftarrow{q} \frown [e]$

2.0 *DEQUEUE* $()$ $e : Qelem$
.1 ext wr $q : Queue$
.2 pre $q \neq [\,]$
.3 post $\overleftarrow{q} = ...[e]...$

3.0 *ISEMPTY* $()$ $r : \mathbb{B}$
.1 ext rd $q : Queue$
.2 post $r \Leftrightarrow (\mathsf{len}\ q = 0)$

Aufgabe: Vervollständigen Sie die Nachbedingung von DEQUEUE.

1.4 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 **Datenaustausch** 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 expliziten Kennzeichnung als *public* oder einer gezielten Exportierung an möglichst wenige Nachbarmodule).
6. Ein Modul sollte **abgeschlossen** (z.B. vollspezifiziertes Bibliotheksmodul) **und offen** (d.h. erweiterungsfähig für weitere Attribute/Methoden ohne strukturgleiche Nachprogrammierung zu benötigen) sein.

2 Wiederverwendbarkeit

2.1 Wiederverwendbarkeit im Kleinen

Vermeide es, das Rad immer wieder neu zu erfinden

1. **Algorithmen** (Programme) lösen im allgemeinen eine Klasse von Problemen: Eingabewerte.

2. **Unterprogramme** (Funktionen, Prozeduren, Operatoren) lösen eine Klasse von Problemen: Gemäß dem Prinzip der methodischen Restriktion sind dabei heute die einzelnen Parameter jeweils Werte des Wertebereichs eines festen Typs.

3. **Unterprogramme mit konformen Feldparametern** (Pascal) beziehungsweise open-array-Parametern (Modula 2) erlauben es Parametern, einer Klasse von Feldern anzugehören (variable Dimension):

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

4. **Dynamische Felder/Teilfeld-Selektoren** erlauben einen der Dimension nach nicht festgelegten Feldtyp sowie Teilfelder als *first level* Objekte:

   ```
   TYPE vektor = ARRAY[*] OF REAL;
   a := t[*, 2];
      ... t[m:n, k:2] ...
   ```

5. **Polymorphismus** (d.h. Überladen) **von Funktionen/Operatoren** erlaubt 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;
                   sucsess              : boolean
                  ) : real;
```

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

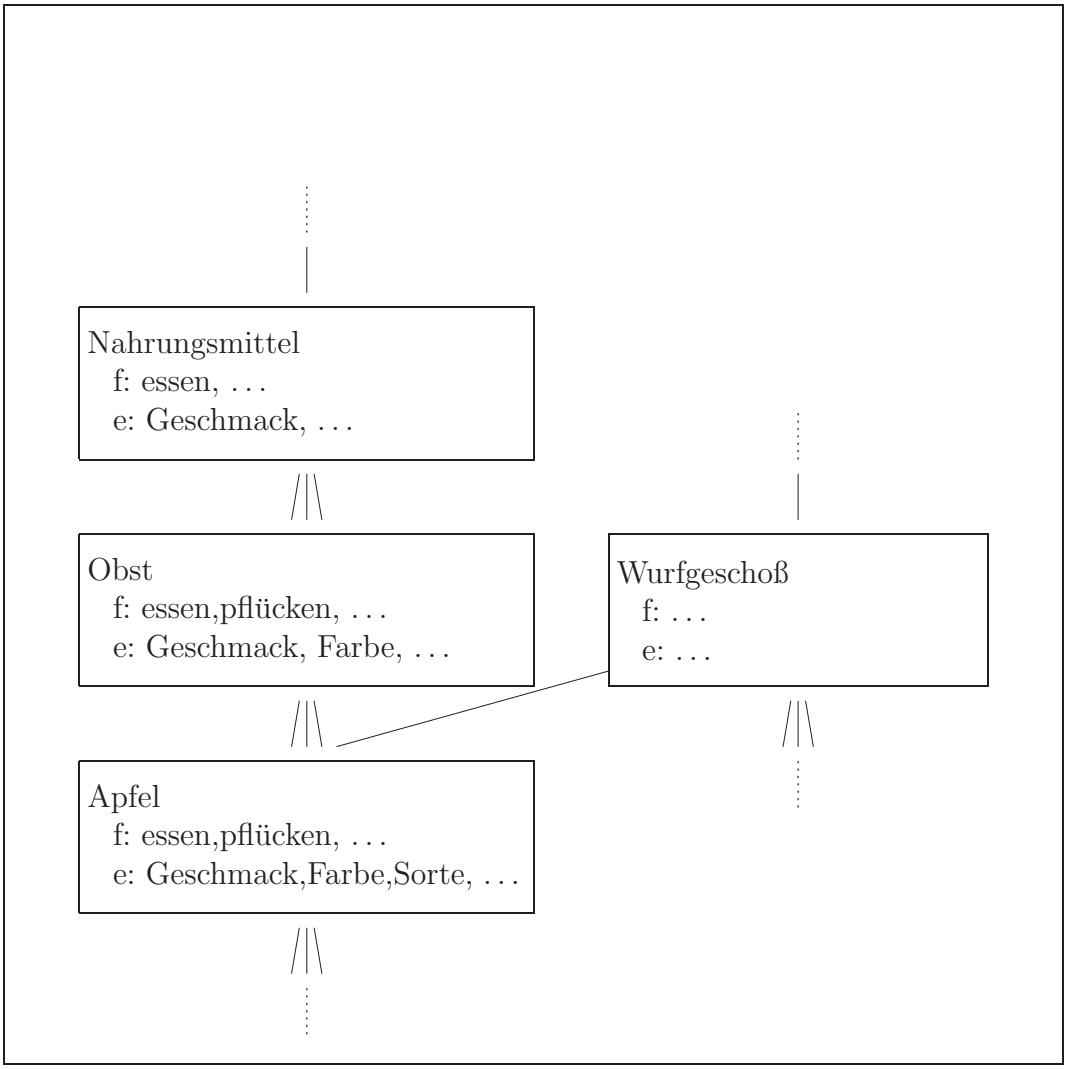
```
generic
  type T is private;
procedure swap(x,y : in out T) is
  t : T
begin
  t := x; x := y; y := t
end swap
⋮
procedure int_swap is new swap(INTEGER);
```

8. **Eingeschränkte Generizität:**

```
generic
  type T is private;
  with function "<=" (a,b : T) return BOOLEAN is <>;
function minimum(x,y : T) return T is
begin
  if x <= y then return x;
  else return y
  end if
end minimum.
```

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

2.2 Begriffshierarchien



f = Operationen, Funktionen
e = Eigenschaften, Attribute

2.3 Objekthierarchien als strukturierte Modulsammlungen

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

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

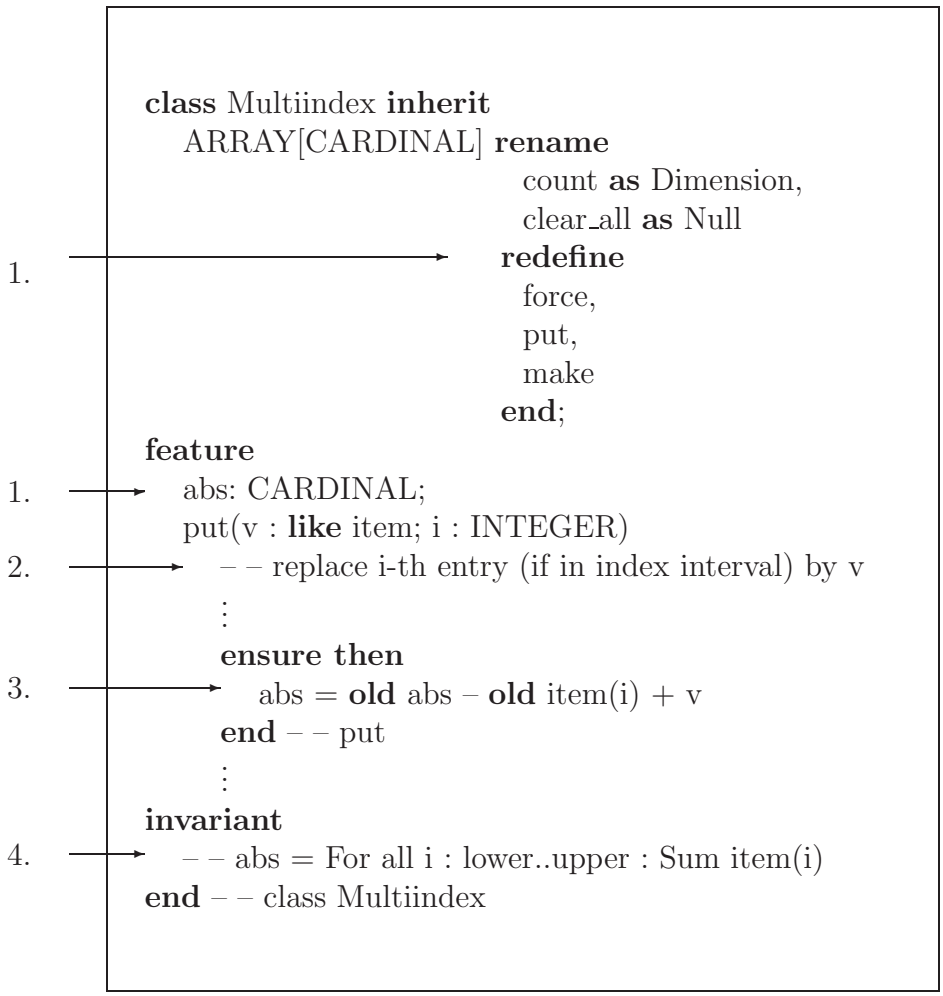
Vererbung und Erweiterung:

Namesänderung →

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
```

Vererbung und Abänderung:



1. alternative Implementierung
2. auch die Vorbedingung wird vererbt (und eventuell schwächer gemacht)
3. zusätzliche Nachbedingung (insgesamt also stärker als in der Elternklasse)
4. alte Invariante wird vererbt (und eventuell verstärkt)

Wenn aus Effektivitätsgründen **redundante Daten** angelegt werden:
Redundanz spezifizieren!

Generizität:

```
class STACK[T]
feature
   ⋮
end – – class STACK[T]
```

Eingeschränkte Generizität:

```
class VECTOR[T → ADDABLE]
feature
   ⋮
end – – class VECTOR
```

Siehe auch Abschnitt 2.1 Punkt 8.

Polymorphismus und *Late binding* (zur Laufzeit):

```
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 σ(Rectangle) $\subset$ σ(POLYGON) und f(Rectangle) $\supset$ f(POLYGON) gilt in der Anwendung[1]:

[1] σ steht für die Menge der Werte von ...

```
⋮
p : POLYGON;
r : Rectangle;
⋮
!! p; – Konstruktion eines Exemplars der Klasse POLYGON
!! r; – Konstruktion eines Exemplars der Klasse Rectangle
⋮
print (p.perimeter); – – perimeter aus POLYGON aufrufen
⋮
p := r;
print (p.perimeter); – – perimeter aus Rectangle aufrufen
⋮
```

Aufgeschobene Feature-Implementierungen:

```
deferred class STACK[T]
feature
   nb_elems : INTEGER is
   deferred
   end – – nb_elems

empty : BOOLEAN is
   do
      Result := (nb_elems = 0)
   ensure Result = (nb_elems = 0)
   end – – empty
   ⋮
end – – class STACK[T]
```

Sie dienen der partiellen Implementierung einer Gruppe möglicher Implementierungen (FIXED_STACK[T], LINKED_STACK[T], ...), um gemeinsame Strukturen deutlich herauszustellen oder um mit zusätzlicher Hilfe der Mehrfachvererbung ein *generisches* Konzept zu realisieren:

```
class X_STACK [T] inherit
   STACK [T];
   X[T] rename x_feature as nb_elems
        end
   ⋮
end – – class X_STACK
```

Zusammenfassung:

→ *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 (Methoden)
2. mehr Daten (Attribute)
3. eingeschränkte Wertebereiche der Daten
4. alternative Implementierungen

→ 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 ist *nur* ein Browser zur Betrachtung der Quellen der Klassen vorhanden und der Programmierer muß sich selbst den Durchblick durch die Konzeption der Klassenbibliothek erkämpfen.

→ 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 Informatik/ Mathematik/ Anwendungswissenschaften fehlen geeignete Klassenbibliotheken gänzlich:

- Eine sprachübergreifende **Standardisierung** einer Sammlung von grundlegenden Klassenhierarchien (zu Datenstrukturen, Zahlen, Vektor- und Punkträumen, Funktionsräumen, ...) ist **dringend notwendig**.
- Das Studium derselben sollte eine Grundlage für eine Ausbildung von *Software-Konstrukteuren* sein.

  [Ansätze existieren etwa in:

  Lins: Modula-2 Software Component Library,
  Musser/Stepanov: The ADA Generic Library,
  Eiffel-, Smalltalk-, ... , -Klassenbibliotheken,
  NIHCL, LEDA, STL,
  Interview, ...]

→ Geordnete *evolutionäre* objektorientierte Entwicklung im Team erfordert ein richtiges Management (open-closed-Phasen, *Versions*-Management, ...).

→ 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 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.

Das Forschungsministerium fördert Standard für IT-Sicherheit:
http://www.heise.de/newsticker/data/dab-01.10.03-002/

3 Softwarequalitätssteigerung mit normalen C/C++-Sprachmitteln

3.1 Umgangssprachliche Spezifikation

„Informelle Beschreibung“: Auf einem Parkplatz stehen PKW's und Motorräder. Zusammen seien es n Fahrzeuge mit insgesamt m Rädern. Bestimme die Anzahl P der PKW's.

„Lösung“: Sei

P := Anzahl der PKW's
M := Anzahl der Motorräder

$$\left\{ \begin{array}{rcl} P + M & = & n \\ 4P + 2M & = & m \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{rcl} M & = & n \text{ - } P \\ P & = & \frac{m \text{ - } 2n}{2} \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{rcl} M & = & \frac{4n \text{ - } m}{2} \\ P & = & \frac{m \text{ - } 2n}{2} \end{array} \right\}$$

„Algorithmus“:

```
        ⋮

M := (4 * n - m) / 2;
P := (m - 2 * n) / 2;
write (M,P);

        ⋮
```

Problem:

**** Problemfall: Null-Mark-Rechnung, Null-Mark-Mahnung,...

$(m, n) = (9, 3) \Rightarrow P = 1\frac{1}{2}$
$(m, n) = (2, 5) \Rightarrow P =$ -4

Vor der Entwicklung eines Algorithmus ist zunächst für das Problem eine *funktionale Spezifikation* bestehend aus

1. Definitionsbereich,
2. Wertebereich *und*
3. für die Lösung wichtigen Eigenschaften (insbesondere funktionaler Zusammenhang zwischen Eingabe- und Ausgabegrößen)

anzufertigen!

Besser ist also:

Eingabe : $m, n \in \{0, 1, \ldots, \text{INT\_MAX}\}$

Vorbedingung : m gerade, $2n \leq m \leq 4n$

Ausgabe : $P \in \{0, 1, \ldots, \text{INT\_MAX}\}$, falls die Nachbedingung erfüllbar ist (sonst „keine Lösung“)

Nachbedingung : Ein $(P, M) \in \{0, 1, \ldots, \text{INT\_MAX}\}$ mit

$$\begin{array}{rcrcl} P & + & M & = & n \\ 4P & + & 2M & = & m \end{array}$$

Aufgabe: Beweisen Sie, dass bei den spezifizierten Vorbedingungen jeweils genau eine Lösung gemäß Spezifikation existiert (das Problem also lösbar ist).

3.2 Unbeachtete Under-/Overflows in C, C++ ...

Gute Programme sollten die Wertebereiche von Variablen möglichst eng fassen:

– „Spezifikation“ von Variablen so genau wie in der Programmiersprache möglich.

```
const
      ErrorResult = -1;
 ⋮

function fakultaet(n : integer {n >= 0}): integer;
var
        zaehl : 2..maxint;
        teilres : 1..maxint;   {Teilresultat}
begin
        teilres := 1;
        for zaehl := 2 to n do begin
              teilres := teilres * zaehl
        end;
        if (n >= 0) then fakultaet := teilres
                    else fakultaet := ErrorResult;
end;
```

– Verifikation:

Endlichkeit: Die Schleife wird genau (n-1)-mal durchlaufen, falls $n \geq 2$. Ansonsten wird sie kein mal durchlaufen.

Schleifenvariante: Zerlegung des gewünschten Ergebnisses $\boxed{n!}$ in ein schon berechnetes Teilergebnis `teilres` und einen noch zu berechnenden Rest:

$$\boxed{n! \;=\; \texttt{teilres} \;*\; \prod_{i=zaehl}^{n} i}$$

(bei Schleifenbegin: teilres $= 1, zaehl = 2$).

Korrektheit bei Abbruch:

```
begin
                {n in integer; zaehl, teilres undef}
    teilres := 1;
                {n! = teilres * n!}
    for zaehl := 2 to n do begin        {n >= 2}
                {n! = teilres * ∏_{i=zaehl}^{n} i, n >= zaehl}
        teilres := teilres * zaehl
                {n! = teilres * ∏_{i=zaehl+1}^{n} i, n >= zaehl}
    end;
                {n ≥ 2: n!=teilres·1, zaehl undef}

    {n ≥ 2: n! = teilres; n in [0,1]:teilres = 1 = n!}
    {n < 0: n! undef }
      ⋮
```

– Informelle Spezifikation:

Zweck: „`fakultaet`“ berechnet die Fakultät von n:

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n\text{-}1) \cdot n,$$
$$0! = 1$$

für alle n in [0,1, … ,maxint], für die $n!$ auch in `INTEGER` liegt.

Interface:

n in Argument, dessen Fakultät berechnet werden soll.

Result out Fakultät von n oder Fehlerkennzeichnung „ErrorResult“

Result $\in$ [ErrorResult]$\cup$[1, … ,maxint].

Vorbedingungen:

n in integer.

Nachbedingung:

`fakultaet` liefert:

$n!$, falls $n \geq 0$ und $n! \leq$ maxint;

ErrorResult, falls $n < 0$;

undefiniertes Verhalten bei $n! >$ maxint.

Beachte: Diese Spezifikation ist für die Praxis ungeeignet! „Verhalten undefiniert“ hängt bei diesem Pascalbeispiel vom Compiler und der Laufzeitumgebung ab (Overflow-Exception ja/ nein?).

– Verbesserungen:

Ziel: Berechne `teilres := teilres * zaehl` nur dann, wenn das innerhalb des Intervalls [1, ... ,maxint] möglich ist:

```
          {n!=teilres* ∏_{i=zaehl}^{n} i, n ≥ zaehl,}
          {teilres <= maxint }
if (maxint div teilres >= zaehl) then
          {teilres * zaehl <= maxint}
     teilres := teilres * zaehl
          {n!=teilres* ∏_{i=zaehl+1}^{n} i, n ≥ zaehl,}
else begin
          {n! ≥ teilres * zaehl > maxint}
     ⋮
```

Ein ähnliches Problem tritt zum Beispiel bei Effizienzverbesserungen von Algorithmen auf. Bei einer verbesserten Version von „Power“ unter Zugrundelegung von

$$x^{2i} = (x^2)^i,\ x^{2i+1} = x^{2i} \cdot x$$

statt von

$$x^i = x^{i\text{-}1} \cdot x.$$

```
⋮
begin
     Teilerg := 1.0;
     while (n > 0) do
          if odd(n) then begin
               Teilerg := Teilerg * x; n := n - 1;
          end else begin
               x := sqr(x); n := n div 2;
          end;
          ⋮
```

versucht folgende Implementierung nach *O.J. Dahl/E.W. Dijkstra/C.A.R. Hoare: Structured Programming, Seite 14* einige Fallunterscheidungen zu vermeiden:

```
⋮
begin
     Teilerg := 1.0;
     while (n > 0) do begin
          if odd(n) then begin
               Teilerg := Teilerg * x; n := n - 1; ←
          end;                                          { n gerade}
          x := sqr(x); n := n div 2; ←
     end;
     ⋮
```

Obwohl beide Varianten in $(\mathbb{R}, \mathbb{Z}_0^+)$ das richtige Ergebnis x^n lieferten, wird in der zweiten Variante am Schluß eine unnötige Quadrierung von x vorgenommen (while-Statement mit $n = 1$). Das kann zu einem Overflow führen, obwohl der durch das entprechende `x := sqr(x)` berechnete x-Wert für die gesuchte Potenz gar nicht mehr benötigt wird! Es wird also für die Variante eine strengere Vorbedingung als nötig und wünschenswert gefordert (d.h. der Bereich der Argumente, für die x^n berechnet wird, ist unnötigerweise verkleinert worden):

- **Vorbedingung des Originals:** x^n ist (im Rahmen der begrenzten Rechengenauigkgeit in IEEE_real) kleiner oder gleich `FLT_MAX`.
- **Vorbedingung der Variante:** x^n und $x^{2^{\lfloor log_2(n) \rfloor + 1}}$ ist (im Rahmen der begrenzten Rechengenauigkgeit in IEEE_real) kleiner oder gleich `FLT_MAX`.

Es muß also für die Variante nicht nur x^n, sondern auch die Potenz von x zum Exponenten der zu n nächsthöheren Zweierpotenz kleiner oder gleich `FLT_MAX` sein.

Aufgabe: Wie sehen die Fallunterscheidungen zur Vermeidung von `INTEGER`-Overflows und -Underflows bei den vier Grundrechenarten, bei `sqrt()` und bei `sqr()` aus?

3.3 Vergessene Problematik: Unordered real values

```
y1 = 0;
y2 = 0;
for (i=0;i<ITERATIONS;i++)
{
     x1 = y1;
     x2 = y2;

     y1 = x1 * x1 - x2 * x2 +c1;
     y2 = 2.0 * x1 * x2 + c2;

     n2 = y1 * y1 + y2 * y2;
     if (n2 > 4.0) then
          /* skip */

     else
          plot(y1,y2);

}
```

Einsetzen von

```
else if unordered (n2,4.0) then
     /* skip */
else
     plot(y1,y2);
```

beachtet „unordered“ Argumente! (Konvergente Iterationspunkte sollen geplottet werden. Ein Wert größer als 4.0 kann als Divergenzkriterium gewertet werden. Im Falle ∞-$\infty = NaN$ ist jedoch der Fall der Divergenz ohne Erfüllung von $n2 > 4.0$ gegeben.)

3.4 assert in C und C++

```
# include <assert.h>
     ⋮
int fakultaet(int i)
{
     assert (i>=0);     /* Vorbedingung */
          ⋮
     assert(result>0);     /* Nachbedingung */
}
```

„Assertion failed: file ass.c, line 15“ bei fakultaet(17) statt 17! zum Beispiel das Ergebnis -288522240.

Nach dem Austesten: eventuell Abstellen der Überprüfungen durch

```
# define NDEBUG
# include <assert.h>
     ⋮
```

und erneute Compilation oder Benutzung der Compiler-Option `-DNDEBUG`.

3.5 Vermeide `enum` als `INTEGER`-Typ zur Modellierung von nicht-`INTEGER`-Objekten

Wenn einige `INTEGER`-Operationen auf durch `enum`-Typen zu modellierenden Datentypen nicht sinnvoll sind, so benutze diese nur als `private`-Implementierung in einer eigenen Klasse:

```
#include        <iostream>
#include        <sstream>
#include        <string>

using namespace std;

class Day{
private:
        enum DayType {_Montag, _Dienstag, _Mittwoch, _Donnerstag,
                      _Freitag, _Samstag, _Sonntag};

// nicht als member zugelassen; nur Konstruktoren dürfen als
//    Initialisierer von Membern in Klassen dienen!!!
//
// static const string DayTable[] = {"Montag", "Dienstag", "Mittwoch",
//                      "Donnerstag", "Freitag", "Samstag", "Sonntag"};


        static const string DayTable[7];

        DayType t;

        Day(const DayType& dt): t(dt) {};

public:

        static const Day Montag;
        static const Day Dienstag;
        static const Day Mittwoch;
        static const Day Donnerstag;
        static const Day Freitag;
        static const Day Samstag;
        static const Day Sonntag;

        Day(const Day& d = Montag): t(d.t) {};

        Day& operator++();
```

```
        const Day operator++(int);

        friend istream& operator>>(istream&, Day&);
        friend ostream& operator<<(ostream&, const Day&);

};

const Day Day::Montag(_Montag);
const Day Day::Dienstag(_Dienstag);
const Day Day::Mittwoch(_Mittwoch);
const Day Day::Donnerstag(_Donnerstag);
const Day Day::Freitag(_Freitag);
const Day Day::Samstag(_Samstag);
const Day Day::Sonntag(_Sonntag);

const string Day::DayTable[] = {"Montag", "Dienstag", "Mittwoch", "Donnerstag",
                                "Freitag", "Samstag", "Sonntag"};

Day& Day::operator++()
{
        (*this).t = (_Sonntag == t) ? _Montag : DayType((*this).t + 1);
        return *this;
}

const Day Day::operator++(int)
{
        Day old_value(*this);
        ++(*this);
        return old_value;
}

istream& operator>>(istream& is, Day& d)
{
        string s;
        is >> s;
        for (int i = 0; i < 7; i++)
                if (s == Day::DayTable[i]) {
                        d.t =  Day::DayType(i);
                        return is;
                }

        is.clear(ios_base::badbit);
        return is;
}
```

```
ostream& operator<<(ostream& os, const Day& d)
{
        os << Day::DayTable[int(d.t)];
        return os;
}


int main()
{
        Day d1;
Day d2(Day::Sonntag);
        Day d3;

        cout << d1 << endl;
        cout << d2 << endl;

        d1 = Day::Montag;
        for (int i = 0; i < 15; i++)
                cout << ++d1 << endl;
        cout << endl;

        // ...
}
```

3.6 Compiletime Assertions

Wollen Sie Ihr Programm nur auf solchen Compilern übersetzbar machen, auf denen der Typ `int` mindestens 16 Bit Genauigkeit anbietet, so kann das dann mittels

```
CT_ASSERT(sizeof(int) * CHAR_BIT >= 16, INT_TO_SMALL);
```

geschehen.
Ähnlich kann in einem rekursiven Template `FAKULTAET` die Überprüfung der Vorbedingung

```
CT_ASSERT(k >= 0, DOMAIN_NEGATIVE);
```

dafür sorgen, dass bei Aufruf dieses Templates mit negativem statischem Argument eine interpretierbare Fehlermeldung der Art

```
"filename.cc", line 21: Error:
    Cannot cast from ERROR_DOMAIN_NEGATIVE to CompileTimeChecker<0>.
1 Error(s) detected.
```

erzeugt wird.

`CT_ASSERT` ist dabei etwa wie folgt zu definieren:

```
template<bool> struct CompileTimeChecker
{
CompileTimeChecker(...);
};

template<> struct CompileTimeChecker<false> {};

#define CT_ASSERT(expr, msg)\
 {\
    class ERROR_##msg{};\
    (void)sizeof(CompileTimeChecker< (expr) != 0  > ((ERROR_##msg())));\
 }
```

3.7 Ausnahmebedingungen: `Exceptions`, `Traps`

Das Codestück

Listing 3.1: Exceptions

```
...

double power1(double x,int exp)
{
  double erg(1.0);
  if ( exp < 0 ) throw(exp);

  ...

}
```

erzeugt bei Nichterfüllen der Vorbedingung eine (abfangbare) Ausnahmebedingung (Exception) des Typs `int`. Exceptions können über `catch`-Anweisungen abgefangen werden:

Listing 3.2: Abfangsequenz

```
try{
    // ...
} catch (const char* err){
    cerr << endl <<"### Fehler: " << err << endl;
    exit(1);
} catch (const string& err){
    // ...
} catch (const int& i_err){
    // ...
} catch (...){
    cerr << endl <<"### Fehler: unbekannte Exception" << endl;
    exit(2);
}
```

Beachten Sie, dass bei der Übergabe des Exception-Objekts **keine** Typkonversion stattfindet, eine `const char*`-Exception also nicht von einer `const string&`-Catchanweisung abgefangen wird.
Die standardmäßig vorhandene `exception`-Hierarchie lautet:

```
- exception
-- bad_alloc
-- bad_exception
-- bad_cast
-- bad_typeid
-- ios::failure
```

```
-- runtime_error
---- range_error
---- overflow
---- underflow
-- logic_error
---- length_error
---- domain_error
---- out_of_range
---- invalid_argument
```

Wollen Sie eine eigene Exception-Hierarchie, etwa

```
- Matherr
-- Overflow
-- ZeroDivide
```

aufbauen, so kann das folgendermaßen geschehen:

Listing 3.3: eigene Exception-Hierarchie

```
// ...
class Matherr {};
class Overflow : public Matherr {};
class ZeroDivide : public Matherr {};
// ...
try{
  // ...
  if (Bed1) throw ZeroDivide();
  // ...
}catch(const ZeroDivide& e){
    // ...
}catch(const Matherr&){
    // ...
}
```

(Empfehlenswert ist jedoch, auch seine eigene Exceptionhierarchie in den Hierarchiebaum von `exception` einzugliedern.)

Wird in einer Funktion eine erzeugte Exception nicht abgefangen, so wird diese Funktion abgebrochen und ein Handler für die Exception in der sie aufrufenden Programmeinheit gesucht ...

Ist schließlich auch in `main()` kein Handler (zutreffende `catch`-Anweisung) aufzufinden, wird das gesamte Programm abgebrochen.

4 Spezifikation mit sprachexternen Mitteln: VDM, ANNA, OCL

4.1 Invarianten mit Hilfe von VDM

```
4.0  Date ::
 .1          DAY : ℕ1
 .2          MONTH : ℕ1
 .3          YEAR : ℕ1

 .4  inv mk-Date(d, m, y) ≜
 .5      (1901 ≤ y) ∧
 .6      (m ≤ 12) ∧
 .7      (d ≤
 .8      cases m :
 .9          4, 6, 9, 11 → 30,
 .10         2 → if ((y rem 4 = 0) ∧ (y rem 100 ≠ 0)) ∨
 .11                 (y rem 400 = 0)
 .12                  then 29 else 28,
 .13         others → 31
 .14     end
 .15     )
```

4.2 Ausnahmebedingungen in VDM

5.0 $DEQUEUE\,()$ e : [Qelement]
.1 pre $q \neq [\,]$
.2 post $\overleftarrow{q} = [e] \curvearrowright q$
.3 errs $QUEUEEMPTY : q = [\,] \rightarrow q = \overleftarrow{q} \wedge e =$ nil

4.3 Module in VDM

module $Queue$

parameters

6.0 types $Qelem$

7.0 values ...

8.0 functions ...

9.0 operations ...

imports

10.0 types ...

11.0 values ...

12.0 functions ...

13.0 operations ...

instantiation

14.0 types ...

15.0 values ...

16.0 functions ...

17.0 operations ...

exports

18.0 types ...

19.0 values ...

20.0 functions ...

21.0 operations ...

definitions

types . . .

22.0 state *attributes* of

.1 $QueueContents : Qelem^*$

.2 inv *pattern* $\triangleq$ *expression*

.3 init $QueueContents \triangleq [\,]$

.4 end

values . . .

functions . . .

operations . . .

end $Queue$

4.4 Klassen in VDM++