

## **Informatikunterricht an Schulen -- Sicherheitstraining oder Schleuderkurs ?**

91. MNU Kongress  
Stuttgart, 18. April 2000

Prof. Dr. Erhard Plödereeder  
Universität Stuttgart  
ploedere@informatik.uni-stuttgart.de

### **Wieso dieser Titel ?**

#### **im ADAC-Kontext:**

**Schleuderkurs:** vermittelt handwerkliche Fähigkeit, sich aus einer Gefahrensituation zu retten

**Sicherheits-  
training:** vermittelt zusätzlich das Wissen, die Gefahrensituation nach Möglichkeit zu vermeiden

**im Kontext der Informatik:**

**Schleuderkurs:** vermittelt handwerkliche Fähigkeit, auch schwierige Softwareprobleme zu meistern

**Sicherheits-  
training:** vermittelt zusätzlich das Wissen, Gefahren und Schwierigkeiten bei der Programmgestaltung zu minimieren

- Allgemeine Betrachtungen
  - die "Kunden"
  - die Problemstellungen
- Programmiersprachliche Einflüsse auf "sichere" Software
- die hilfreichsten Mittel der Programmiersprachen
- kurzes Fazit

Gruppe 0:      Gruppe 1:              Gruppe 2:              Gruppe 3:  
                   "Privater Nutzer"    "Professioneller Nutzer"    "Informatiker"

<p><i>"ich habe nichts mit Computern zu tun"</i></p>	<ul style="list-style-type: none"> <li>• Information</li> <li>• Kommunikation</li> <li>• Unterhaltung</li> <li>• "E-Commerce"</li> </ul>	<ul style="list-style-type: none"> <li>• täglicher Umgang mit Anwendungen</li> <li>• programmier-ähnliche Anpassung der Anwendungen</li> </ul>	<ul style="list-style-type: none"> <li>• Systementwurf</li> <li>• Programmierung</li> <li>• Wartung</li> </ul>
--	--	--	--

Gruppe 0:      Gruppe 1:              Gruppe 2:              Gruppe 3:  
                   "Privater Nutzer"    "Professioneller Nutzer"    "Informatiker"

<ul style="list-style-type: none"> <li>• Sicherheit/Privacy</li> <li>• Schutz vor Viren u.ä.</li> <li>• Garantien</li> <li>• SW-Installation</li> <li>• (Copyright, Lizenzen)</li> </ul>	<ul style="list-style-type: none"> <li>• anwendungs-spezifische</li> </ul>	<ul style="list-style-type: none"> <li>• generische</li> </ul> <p>Maßnahmen zur</p> <ul style="list-style-type: none"> <li>• Fehlervermeidung</li> <li>• Fehlereindämmung</li> <li>• Fehlertoleranz</li> </ul>
--	--	--

---

Inform.techn. Grundbildung	AGs	Leistungskurs
Gruppe 0:	Gruppe 1:	Gruppe 2:
"Privater Nutzer"	"Professioneller Nutzer"	"Informatiker"



- Sicherheit/Privacy
  - Schutz vor Viren u.ä.
  - Garantien
  - SW-Installation
  - (Copyright, Lizenzen)
- anwendungs-  
spezifische  
Maßnahmen zur
    - Fehlervermeidung
    - Fehlereindämmung
    - Fehlertoleranz
- generische

- böswillige Softwareentwickler (z.B. Viren, Würmer, usw.)
- Bedienfehler bei der Softwareanwendung
- Entwurfs- und Codierfehler bei der Softwareentwicklung und -wartung  
*diese Ursachen lassen sich quantifizieren ...*

---

	Norm
Informationssysteme (kommerziell)	7,0
Informationssysteme (militärisch)	6,0
Kommerzielle Produkte	4,0
Telekommunikation (kommerziell)	3,1
Telekommunikation (militärisch)	2,5
Waffensysteme (Boden)	1,0
Waffensysteme (Luft/Raum)	1,0

\* kSLOC = 1000 Zeilen Quellcode

Quelle: Reifer 1996, Norm aus 1500 Projekten 1989-96

- "Korrektheit" der Software ist mit den heutigen technischen Möglichkeiten in der Praxis nicht zu erreichen; der Kunde muß mit fehlerhafter Software rechnen
- Aus pragmatischer Sicht ist seitens der Softwareproduktion daher vorrangig:
  - Minimieren der Fehlerquellen
  - Minimieren der Fehlerauswirkungen
  - Fehlertoleranz

- (primär) durch **Verbesserungen der Prozesse** in der Softwareentwicklung und -wartung
- durch ingenieurmäßige Verwendung geeigneter Technologien
  - Entwurfsmethodiken
  - Programmiersprachen und Werkzeugezur Vermeidung von Fehlern **vor** Einsatz des Programms.

*Programmiersprachen und ihr Umfeld (Werkzeuge, Ausbildung, induzierte "Programmierkultur") nehmen wesentlichen Einfluß auf die Fehlerhäufigkeit der Software.*

auch dieser Effekt läßt sich quantifizieren ...

	Norm
Informationssysteme (kommerziell)	7,0
Informationssysteme (militärisch)	6,0
Kommerzielle Produkte	4,0
Telekommunikation (kommerziell)	3,1
Telekommunikation (militärisch)	2,5
Waffensysteme (Boden)	1,0
Waffensysteme (Luft/Raum)	1,0

Quelle: Reifer 1996, Norm aus 1500 Projekten 1989-96

	Ada	C	C++	Norm
Informationssysteme (kom.)	4,0	7,0	5,1	7,0
Informationssysteme (mil.)	3,0	6,0	4,0	6,0
Kommerzielle Produkte	2,8	5,0	3,0	4,0
Telekommunikation (kom.)	1,6	2,0	1,7	3,1
Telekommunikation (mil.)	1,0	1,5	1,2	2,5
Waffensysteme (Boden)	0,5	0,8	0,7	1,0
Waffensysteme (Luft/Raum)	0,3	0,8	0,6	1,0

Quelle: Reifer 1996, 190 Projekte 1993-96, Norm aus 1500 Projekten 1989-96

	Ada	C	C++	Norm
Informationssysteme (kom.)	1,0	0,5	0,6	0,5
Informationssysteme (mil.)	0,8	0,5	k.A.	0,5
Kommerzielle Produkte	1,0	0,4	1,0	0,5
Telekommunikation (kom.)	3,0	1,0	2,0	1,8
Telekommunikation (mil.)	4,0	2,0	3,0	2,0
Waffensysteme (Boden)	6,0	3,0	k.A.	2,5
Waffensysteme (Luft/Raum)	8,0	3,0	k.A.	2,5

Quelle: Reifer 1996, 190 Projekte 1993-96, Norm aus 1500 Projekten 1989-96

- **Modularisierung und Kapselung**
  - Trennung von Spezifikation (Schnittstelle) und Implementierung
  - explizite Angabe (und Prüfung) von Modulabhängigkeiten
  - "information hiding"
  - abstrakte Typen
- **Betonung statischer Prüfungen**
  - strenge Typenbindung
  - Schnittstellenabgleich und Eingrenzung des Namensraums
- **Parametrierbarkeit von Modulen; Subsystemstrukturen**
- **Operationen mit komplexer Semantik**
- **Ausnahmebehandlung ("exceptions")**

- **Modularisierung und Kapselung**
  - Trennung von Spezifikation (Schnittstelle) und Implementierung
  - explizite Angabe (und Prüfung) von Modulabhängigkeiten
  - "information hiding"
  - abstrakte Typen
- **Betonung statischer Prüfungen**
  - strenge Typenbindung
  - Schnittstellenabgleich und Eingrenzung des Namensraums
- **Parametrierbarkeit von Modulen; Subsystemstrukturen**
- **Operationen mit komplexer Semantik**
- **Ausnahmebehandlung ("exceptions")**

- **Modularisierung und Kapselung**
  - Trennung von Spezifikation (Schnittstelle) und Implementierung
  - explizite Angabe (und Prüfung) von Modulabhängigkeiten
  - "information hiding"
  - abstrakte Typen
- **Betonung statischer Prüfungen**
  - strenge Typenbindung
  - Schnittstellenabgleich und Eingrenzung des Namensraums
- **Parametrierbarkeit von Modulen; Subsystemstrukturen**
- **Operationen mit komplexer Semantik**
- **Ausnahmebehandlung ("exceptions")**

```
package Stack_Manager is
  type Stack is private;
  procedure push (S: Stack; Element: Integer);
  ...
end Stack_Manager;
```

Schnittstelle

```
package body Stack_Manager is
  procedure push (S: Stack; Element: Integer) is
  begin
    -- der Implementierungscode
  end push;
  ...
end Stack_Manager;
```

Implementierung

```
with Stack_Manager;
package body My_Code is
  My_Stack: Stack_Manager.Stack;
  procedure what_ever is
  begin
    Stack_Manager.Push(My_Stack, 7); -- ok
    ...My_Stack.contents... -- illegal, auch wenn Stack eine
    -- 'contents' Komponente besitzt
  end what_ever;
end My_Code;

anderswo: .... My_Code.My_Stack ... -- höchst illegal
```

```
package Stack_Manager is
  type Stack is private;
  procedure push (S: Stack; Element: Integer);
  ...
end Stack_Manager;
```

Schnittstelle

*"vielleicht mal in einem Monat"*

Implementierung

Entwurf und Implementierung von Software hat bereits heute eine fundamentale Wandlung von

*"Entwerfe und implementiere in einer Programmiersprache"*

hin zu

*"Entwerfe und kombiniere aus Bausteinen"*

erfahren. (Moderne Stichworte: APIs, Wiederverwendung, Komponentenfabrik) Dieser Trend wird immer stärker werden.

Die Grundausbildung in der Informatik hat diese Entwicklung bislang kaum berücksichtigt.

- **ausreichende Schnittstellenbeschreibung**

Die Funktionalität und die Vorbedingungen für die Verwendung der angebotenen Schnittstellen **muß ohne Inspektion der Implementierung** des Moduls aus der Schnittstellenbeschreibung hervorgehen.

(Diese Forderung wird in der Praxis oft grob verletzt !)

- **abgeschlossene Implementierung**

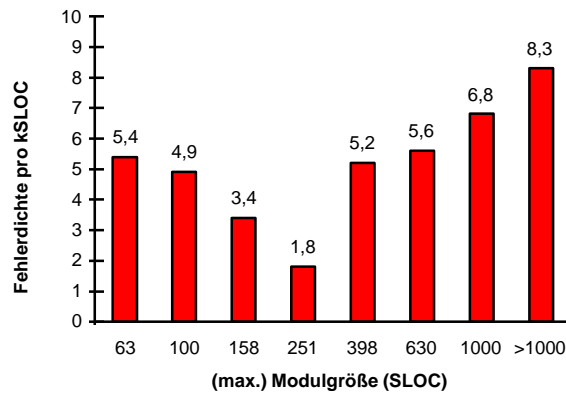
Mit Kenntnis der aus anderen Modulen verwendeten Schnittstellen sollte die Modulimplementierung verständlich und auf Korrektheit prüfbar sein. ("**Divide et impera**")

- **hohe Kohäsion** ("*high cohesion*")

In einem Modul sollten Einheiten zusammengefaßt werden, die einen hohen Grad an gegenseitiger funktionaler oder implementierungstechnischer Abhängigkeit besitzen.

- **niedrige Koppelung** ("*low coupling*")

Abhängigkeiten zwischen Modulen sollten auf ein Mindestmaß beschränkt bleiben.



Quelle: Withrow, IEEE Software, Jan. 1990; ähnliche Ergebnisse bei Shen, Basili, Perricone

© Erhard Plödereder

MNU Kongress 25 18.4.2000

... in der Ausbildung und in der Technologie der Programmiersprachen:

**ausreichend präzise Schnittstellenbeschreibungen**

Ohne diese Beschreibungen bleibt dem Programmierer nur der Weg der Inspektion der Implementierung. Eine Betonung auf Algorithmik mittels elementarer Programmschritte in der Ausbildung begünstigt dieses Verhalten (auch wenn die Schnittstellenbeschreibung vorliegt).

Noch schlimmer ist der Versuch, die Kenntnis der Funktionalität aus der Ausführung mit Testbeispielen herzuleiten.

© Erhard Plödereder

MNU Kongress 26 18.4.2000

### Minimalanforderungen:

- präzise Beschreibung der "Normalfunktionalität"
- Vorbedingungen, damit die Schnittstelle Normalfunktionalität besitzt
- Fehlerverhalten, falls Vorbedingungen verletzt sind
- Fehlerverhalten, falls interne Probleme auftreten

**type** Liste **is private**;

**function** Create **return** Liste;

**function** Recover(L: **in out** Liste);

**procedure** Insert(L: **in out** Liste; E: Element);

-- **Funktionalität:**

-- fügt das Element E am Anfang der Liste L ein; Eintrag von Duplikaten

-- ist zugelassen. Die Einfügung wird auf einer externen Datei für die Zwecke

-- von 'recover' protokolliert.

-- **Fehlerverhalten:**

-- Ausnahme 'Listen\_Fehler', falls L nicht initialisiert ist; die Liste L wird nicht

-- verändert.

-- Ausnahme 'Datei\_Fehler', falls EA-Probleme bei der Protokollierung

-- auftreten; die Liste L wird nicht verändert.

Unterstützung einer formal geprüften Schnittstellenbeschreibung beschränkt sich in den meisten Programmiersprachen auf die Signatur der Schnittstelle. (In einigen Sprachen, z.B. in C, wird m.E. auch die Signatur nicht geprüft.)

Manche Vorbedingungen lassen sich evtl. durch das Typmodell ausdrücken und automatisch prüfen. Z.B.

nicht: **function** Fakultae(N: Integer) **return** Integer;  
sondern: **function** Fakultae(N: Natural) **return** Natural;

- **Modularisierung und Kapselung**
  - Trennung von Spezifikation (Schnittstelle) und Implementierung
  - explizite Angabe (und Prüfung) von Modulabhängigkeiten
  - "information hiding"
  - abstrakte Typen
- **Betonung statischer Prüfungen**
  - strenge Typenbindung
  - Schnittstellenabgleich und Eingrenzung des Namensraums
- **Parametrierbarkeit von Modulen;  
Subsystemstrukturen**
- **Operationen mit komplexer Semantik**
- **Ausnahmebehandlung ("exceptions")**

- Parametrierung von Modulen für Zwecke der Mehrfach- und Wiederverwendung
- Verfeinerung des Schnittstellenbegriffs
- Subsysteme

```
package Stack_Manager is  
  type Stack is private;  
  procedure push (S: Stack; Element: Integer);  
  ...  
end Stack_Manager;
```

**Schnittstelle**

*wird kaum von der Tatsache abhängen,  
daß der Stack Integers enthält; der Element-  
typ muß nur eine Zuweisungsoperation haben.*

**Implementierung**

```
generic
  type T is private;
package Stack_Manager is
  type Stack is private;
  procedure push (S: Stack; Element: T);
  ...
end Stack_Manager;
```

generische  
Schnittstelle

```
package Int_Stack_Manager is new Stack_Manager(Integer);
  -- ein Paket, mit dem Stacks von Integern deklariert werden
```

```
package Matrix_Stack_Manager is new Stack_Manager(Matrix);
  -- ein Paket, mit dem Stacks von Matrices deklariert werden
```

Generik ist in einer Programmiersprache mit strenger statischer Typenbindung wichtig, damit Code, der nicht von bestimmten Typeigenschaften abhängig ist, für mehrere Typen ohne Codereplizierung eingesetzt werden kann.

(Die Parametrierung geht üblicherweise weit über das gezeigte Beispiel hinaus: Typen, Klassen, Unterprogramme, Objekte als Parameter; mehrere auch voneinander abhängige Parameter)

Welcher Unterschied besteht  
zwischen ...

```
class stack {  
    object* v;  
    object* p;  
    int sz;  
    ...  
}
```

```
stack st;
```

und...

```
template<class T>  
    class stack {  
        T* v;  
        T* p;  
        int sz;  
        ...  
    }
```

```
stack(object) st;  
stack(window) window_st;
```

Eine Behälterklasse sollte sich nicht uneingeschränkt polymorpher  
Inhaltsklassen bedienen. Stattdessen sollte die Behälterklasse  
generisch sein, denn dies

- dokumentiert die erlaubten Klassen der Inhalte; eine Wurzelklasse  
'object' (sofern existent in der Sprache) tut dies nicht
- erzeugt (abhängig von der Sprache) Übersetzungs- oder Laufzeit-  
prüfungen, die eine typsichere Verwendung der Inhalte garantiert
- ist kombinierbar mit beschränkter Polymorphie, d.h., Angabe einer  
polymorphen Inhaltsklasse, wenn erwünscht.

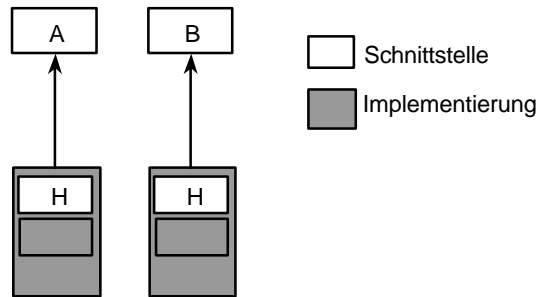
- eine weitere Differenzierung der Unterscheidung von Schnittstelle und Implementierung:



z.B. 'friend' Funktionen oder der 'protected' Begriff in C++ und Java;  
child units in Ada 95

```
class stack{
    private:    // nach außen nicht sichtbare Deklarationen
        int top;
        int list[100];
    protected: // nur für Objekte aus Unterklassen sichtbar
        int peek(int position) {..};
    public:    // sichtbare Deklarationen
        int empty() {..};
        void push (int number) {..};
        int pop() {..};
        stack() {..}; // constructor
        ~stack() {..}; // destructor
}
```

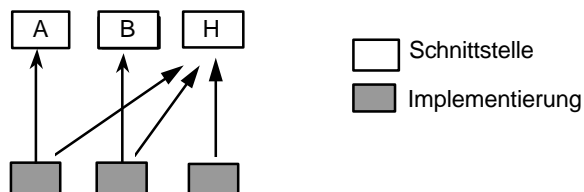
A und B bieten allgemeine Dienste an, die mit Hilfe des nicht allgemein zugänglichen Moduls H implementiert werden.



Wegen der Duplizierung von H eine schlechte Architektur, trotz hoher Kohäsion und geringer Koppelung

© Erhard Plödereder

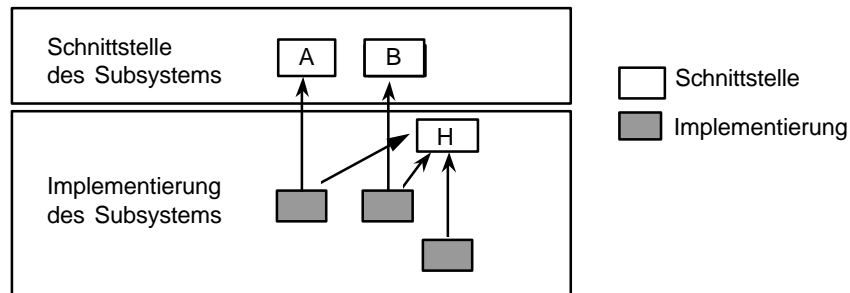
MNU Kongress 39 18.4.2000



Diese (gängige) Architektur vermeidet die Duplizierung, verliert aber den wichtigen Aspekt, daß nur A und B auf H zugreifen dürfen.

© Erhard Plödereder

MNU Kongress 40 18.4.2000



Ein Konzept für Subsysteme ist nötig, um diesen Gesichtspunkten Rechnung zu tragen.

© Erhard Plödereder

MNU Kongress 41 18.4.2000

```

package My_System is .... -- gemeinsame Spezifikationen

package My_System.A is ... -- das gezeigte Paket A
package My_System.B is ... -- das gezeigte Paket B

private package My_System.H is ... -- das gezeigte Paket H
    -- dieses Paket ist wegen der Designation 'private' nur für
    -- Implementierungen von Paketen 'My_System.*', d.h., im
    -- gleichen Untersystem, sichtbar
    
```

Dieses Prinzip ist in Ada rekursiv anwendbar. (Analoges existiert in Java.)

© Erhard Plödereder

MNU Kongress 42 18.4.2000

- **Modularisierung und Kapselung**
  - Trennung von Spezifikation (Schnittstelle) und Implementierung
  - explizite Angabe (und Prüfung) von Modulabhängigkeiten
  - "information hiding"
  - abstrakte Typen
- **Betonung statischer Prüfungen**
  - strenge Typenbindung
  - Schnittstellenabgleich und Eingrenzung des Namensraums
- **Parametrierbarkeit von Modulen; Subsystemstrukturen**
- **Operationen mit komplexer Semantik**
- **Ausnahmebehandlung ("exceptions")**

Ein einfaches Beispiel:

**type** Matrix **is** array(1..10, 1..10) **of** Integer;

A,B: Matrix;

A := B;      *oder*

```
for i in 1..10 loop  
  for j in 1..10 loop  
    A(i,j) := B(i,j);  
  end loop;  
end loop;
```

**Nota bene:** diese Äquivalenz gilt nur für Sprachen, deren Zuweisung Wertesemantik hat; bei Referenzsemantik (z.B. Java) entsteht ein gravierender semantischer Unterschied

Eindrucksvollere Beispiele von Operationen mit komplexer Semantik sind insbesondere im Bereich der Realzeit- und Parallelitätsunterstützung zu finden, die aber im Kontext "Informatik an Schulen" kaum relevant sein dürften.

- **Modularisierung und Kapselung**
  - Trennung von Spezifikation (Schnittstelle) und Implementierung
  - explizite Angabe (und Prüfung) von Modulabhängigkeiten
  - "information hiding"
  - abstrakte Typen
- **Betonung statischer Prüfungen**
  - strenge Typenbindung
  - Schnittstellenabgleich und Eingrenzung des Namensraums
- **Parametrierbarkeit von Modulen; Subsystemstrukturen**
- **Operationen mit komplexer Semantik**
- **Ausnahmebehandlung ("exceptions")**

*... und was soll das Programm tun, wenn trotz aller Vorsicht im Entwurf und der Programmierung etwas bei der Ausführung schiefgeht ?*

**Beispiele:**

- $A(I)$ ;  $I$  außerhalb der Indexgrenzen von  $A$
- $A^{\wedge}.\text{Feld}$ ; mit Zeiger  $A = \text{null}$
- $A + B$ ; das Ergebnis außerhalb darstellbarer Bereiche
- $\text{OPEN}(\text{"xyz"})$ ; die Datei existiert nicht
- usw....

❄ *Programmabsturz ??* ❄

**Programmabstürze sind archaisch. Sie sind (fast) vollständig vermeidbar.**

Moderne Sprachen bieten hierzu Ausnahmeerhebung und -behandlung als Mechanismen an, die eine Recovery ermöglichen.

Oft ist vollständige Recovery möglich ("full fault-tolerance"). Im schlimmsten Fall kann man die Programmbeendigung kontrolliert und unter Ausführung von Notmaßnahmen ("last wishes") durchführen.

In einer ersten Näherung unterscheiden sich die Ausnahmenmodelle der gängigen Sprachen (C++, Java, Ada, ...) kaum....

- implizite Ausnahmeerhebung bei den zuvor zitierten Situationen, die auf Verletzungen von sprachdefinierten Vorbedingungen beruhen
- explizite Erhebung benutzerdeklarerter Ausnahmen:  
"raise Listen\_Fehler;" (Ada) ("throw ListenFehler();" in Java, C++)
- Bei Ausnahmeerhebung Abbruch der momentanen Anweisungsfolge und Fortsetzung der Ausführung in der Ausnahmebehandlung eines **dynamisch** umschließenden Konstrukts (**try**-block in C++, Java, beliebige Blöcke in Ada), sofern für die erhobene Ausnahme vorhanden (**catch**-Klausel in C++, Java; **exception ... when** in Ada).
- Nach der Behandlung der Ausnahme wird die normale Ausführung am Ende dieses umschließenden Konstrukts fortgesetzt.

```
...  
begin  
  A := B + C;  
exception  
  when Constraint_Error =>  
    A := Integer'Last;  
end;
```

Lokal behobenes Problem, sofern die höchste Integerzahl laut Spezifikation für diesen Fall akzeptabel ist.

```
for Attempt in 1..4 loop
  begin
    Get(Sensor, Data); -- may propagate exceptions
    exit;             -- if reading is successful
  exception
    when SENSOR_INPUT_ERROR =>
      if Attempt < 4 then Reset(Sensor); -- restoring state
      else raise FATAL_SENSOR_ERROR;
      end if;
  end;
end loop;
```

Vierfacher "Retry" zum Tolerieren transienter Sensorfehler

```
begin
  Find_Very_Accurate_Solution;
exception
  when Numeric_Instability | Not_Enough_Time =>
    Find_Sufficiently_Accurate_Solution;
end;
```

Recovery durch alternative Berechnung, meist mit degenierendem  
Systemverhalten verbunden

```
while not Shut_Down loop
  begin
    Get_Sensor_Data (Data);
    Control_Aircraft(Data);
  exceptions
    when others => null;
      -- "laß den Zyklus aus", aber KEEP FLYING !!
  end;
end loop;
```

Ein Verzweiflungsidiom, das aber Menschenleben retten kann.

```
begin
  Open(Valve); -- can raise Valve_Did_Not_Open
  while Sensor_Indicates_Flow loop
    Measured_Quantity := Sensor_Reading;
    Communicate(Monitor, Measured_Quantity);
    -- may propagate a number of communication exceptions
    delay 0.01;
  end loop;
  Close(Valve);
exception
  when Valve_Did_Not_Open =>
    Communicate(Monitor, 0); raise;
  when others => Close(Valve); raise;
end;
"Fail-Safe": Das Ventil wird garantiert geschlossen.
```

- Eine frühzeitige Auseinandersetzung in der Ausbildung mit der **Erstellung und Verwendung** von Komponenten und deren **Schnittstellenbeschreibungen** wäre sehr angebracht.
- Ein "Trial-and-Error"-Ansatz, sich über Funktion und Funktionieren von Software zu informieren, ist im Sinne "sicherer Software" sowohl eine faktische als auch eine didaktische Katastrophe.
- Durch präzise Schnittstellenbeschreibungen und gute Modularisierung lassen sich viele Fehler vermeiden.
- Fehlertoleranzmechanismen sind vorhanden und sollten - zumindest rudimentär - in der fortgeschrittenen Ausbildung vermittelt werden.

## Informatikunterricht an Schulen -- Sicherheitstraining oder Schleuderkurs ?

91. MNU Kongress  
Stuttgart, 18. April 2000

Prof. Dr. Erhard Plödereder  
Universität Stuttgart  
ploedere@informatik.uni-stuttgart.de

*"Backup-Folien (nicht gezeigt) zum Thema OOP folgen ..."*

- ist eine ausdrucksstarke Methode in der Hand des guten und disziplinierten Designers
- hat deutliche Vorteile beim Prototyping, bei der Erweiterung existierender Systeme und bei der Wiederverwendung von Methoden existierender Klassen

*aber...*

- ist eine Katastrophe in der Hand des im Design weniger erfahrenen Programmierers ...

- Ausnutzung der Vererbung, insbesondere der Mehrfachvererbung, zur bloßen Wiederverwendung existierender Codes ohne Berücksichtigung der Bedeutung der Unterklassenbildung führt schnell zu chaotischen Klassenhierarchien.
- OOP stellt eine große Herausforderung an bestehende QA Methoden dar, da Ergänzungen in bereits validierten Systemen dazu führen können, daß unveränderter (Binär-)Code drastisch unterschiedliches Verhalten aufweist.
- Die Wartung in gängigen OOP-Sprachen ist noch mit vielen ungelösten Fragen konfrontiert. Die Vorteile für die Entwicklung scheinen sich dort eher in gravierende Nachteile zu wandeln.