Programmieren lernen

Aller Anfang ist schwer… einige empirische Erkenntnisse

Programmieren lernen

Programme schreiben

Programme lesen

Fehler machen und beheben

Programmieren ist anspruchsvoll

Objektorientierung als Einstieg

Fundamentalkritik: Didaktik versus Pädagogik

Pair Programming: Zu zweit geht‘s besser

Wie gut können Studenten nach einem Jahr Informatik-Studium programmieren?

Aufgabe 1: RPN Ausdrücke berechnen
(z.B. 3 4 5 + * = 27)

Aufgabe 2: Infix Ausdrücke ohne Präzedenz berechnen (z.B. 2 + 3 * 4 = 20)

Aufgabe 3: Infix-Ausdrücke mit Klammerung berechnen (z.B. 2 + (3 * 4) = 24)

McCracken, Michael; Almstrum, Vicki; Diaz, Danny; Guzdial, Mark; Hagan, Dianne; Kolikant, Yifat Ben-David; Laxer, Cary; Thomas, Lynda; Utting, Ian; Wilusz, Tadeusz (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, Vol. 33, No. 4, pp. 125–180.

Wie gut können Studenten nach einem Jahr Informatik-Studium programmieren?

Bewertungsskala „Degree of Closeness“

5	Touchdown. The program should have compiled and worked. If it did not work, it could be that the student simply ran out of time.
4	Close but something missing. While the basic structure and functionality is apparent in the source code, the program is incomplete in some way. […]
3	Close but far away. In reading the source code, the outline of a viable solution was apparent, including meaningful comments, stub code, or a good start on the code.
2	Close but even farther away. The outline, comments, and stub code showed that the student had some idea about what was needed, but completed very little of the program.
1	Not even close. The source code shows that the student had no idea about how to approach the problem.

Wie gut können Studenten nach einem Jahr Informatik-Studium programmieren?

Bewertungsskala „Degree of Closeness“

5	Touchdown. The program should have compiled and worked. If it did not work, it could be that the student simply ran out of time.
4	Close but something missing. While the basic structure and functionality is apparent in the source code, the program is incomplete in some way. […]
3	Close but far away. In reading the source code, the outline of a viable solution was apparent, including meaningful comments, stub code, or a good start on the code.
2	Close but even farther away. The outline, comments, and stub code showed that the student had some idea about what was needed, but completed very little of the program.
1	Not even close. The source code shows that the student had no idea about how to approach the problem.

Wie gut können Studenten nach einem Jahr Informatik-Studium programmieren?

Der Kern einer iterativen Lösung unter Verwendung eines Stack (Auszug, ohne Fehlerbehandlung)

```
Stack stack = new Stack();
for (int i = 0; i < inputs.length; i++) {
        if (isNumber(inputs[i])) {
                stack.push(Float.parseFloat(inputs[i]));
        }
        else if ("+". inputs[i])) {
                checkStack(stack, 2, i);
                float result = stack.pop() + stack.pop();
                stack.push(result);
        } // ...
}
return stack.pop();
```

Wie gut können Studenten nach einem Jahr Informatik-Studium programmieren?

Warum sind die Resultate so schlecht?

- Aufgaben zu anspruchsvoll?
- Aufgaben zu wenig Bezug zu Studium?
- Bewertungsskala zu anspruchsvoll?
- Zu wenig Zeit für die Lösung der Aufgabe (1.5h)?
- Setting des Experiments unvorteilhaft (keine Hilfsmittel erlaubt)?
- Problemlösefähigkeiten der Studenten ungenügend?

Programmieren lernen

Programme schreiben

Programme lesen

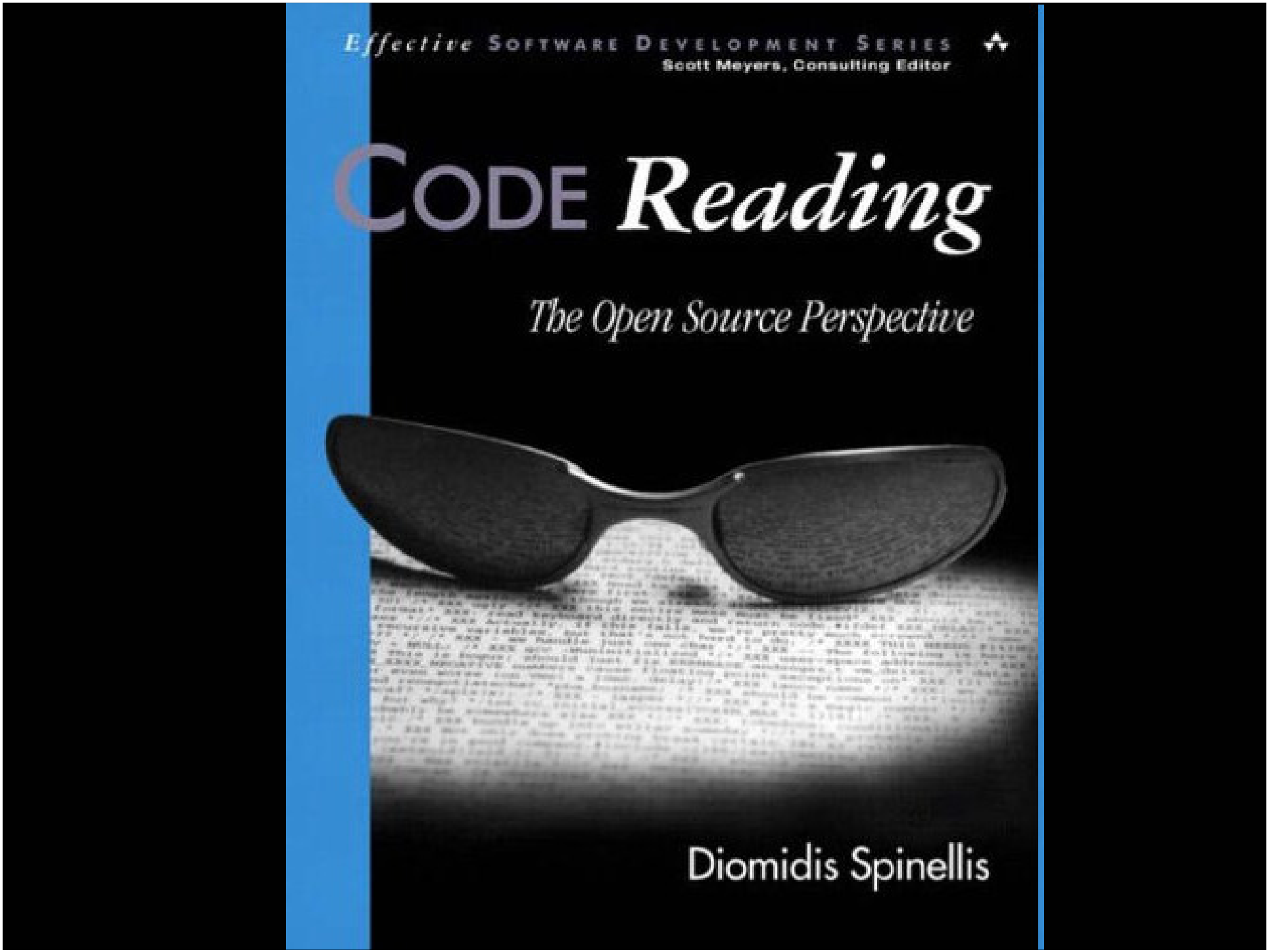
Fehler machen und beheben

Programmieren ist anspruchsvoll

Objektorientierung als Einstieg

Fundamentalkritik: Didaktik versus Pädagogik

Pair Programming: Zu zweit geht‘s besser


Effective Software Development Series
Scott Meyers, Consulting Editor
Code Reading
The Open Source Perspective
Diomidis Spinellis

Wie gut können Studenten nach einem Jahr Informatik-Studium Programme lesen?

Beispiel einer Multiple Choice Aufgabe der Studie

```
int[] x1 = {1, 2, 4, 7};        int[] x2 = {1, 2, 5, 7};
int i1 = x1.length-1;           int i2 = x2.length-1;
int count = 0;
while ((i1 > 0 ) && (i2 > 0 )) {
        if ( x1[i1] == x2[i2] ) {
                ++count; --i1; --i2;
        }
        else if (x1[i1] < x2[i2]) {
                --i2;
        }
        else { // x1[i1] > x2[i2]
                --i1;
        }
}
```

Welchen Wert hat „count“ nach Programmende?

a) 3
b) 2
c) 1
d) 0

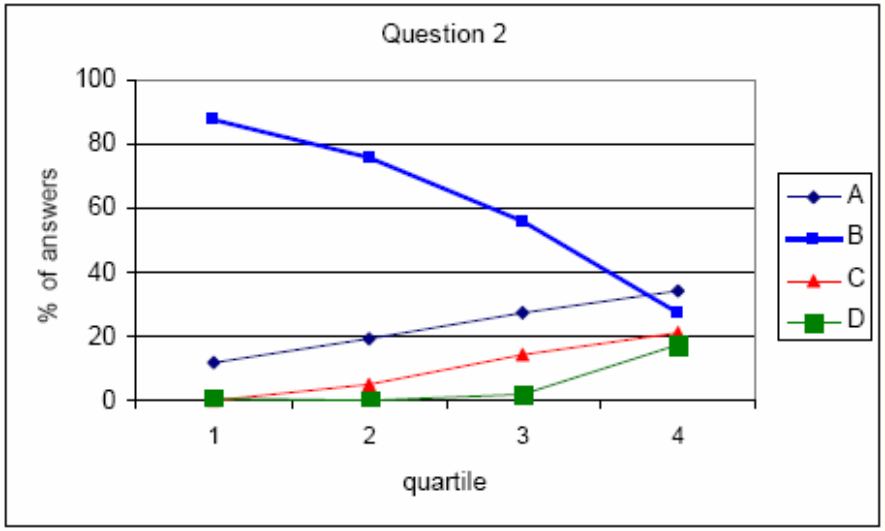
Lister, Raymond; Adams, Elizabeth S.; Fitzgerald, Sue; Fone, William; Hamer, John; Lindholm, Morten; McCartney, Robert; Moström, Jan Erik; Sanders, Kate; Seppälä, Otto; Simon, Beth; Thomas, Lynda (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, Vol. 36, No. 4, pp. 119–150.

Wie gut können Studenten nach einem Jahr Informatik-Studium Programme lesen?

Resultate des Experiments

- 556 Studenten haben 12 Multiple Choice Fragen beantwortet
- Clustering nach Leistung ergab vier fast gleiche grosse Gruppen von Studierenden
- Die Antworthäufigkeiten zur Frage auf der vorhergehenden Folie, aufgeschlüsselt nach Leistungsgruppen

Quartile	Score Range	No. of Students	Percent of Students
1st "top"	10 - 12	152	27%
2nd	8 - 9	135	24%
3rd	5 - 7	142	25%
4th "bottom"	0 - 4	127	23%



Wie gut können Studenten nach einem Jahr Informatik-Studium Programme lesen?

Voraussetzungen gegeben, um Probleme lösen zu können?

- Der leistungsschwächsten Gruppe (0–4 Fragen richtig beantwortet) fehlen wohl die Voraussetzungen zum Problemlösen.
- Die leistungsstärkste Gruppe (10–12 Fragen richtig beantwortet) erfüllt die Voraussetzungen bezüglich Programme lesen. Sollten Studenten aus dieser Gruppe Mühe haben, Programmieraufgaben zu lösen, könnte ihre Schwierigkeit tatsächlich das Problemlösen an sich sein.
- Hypothesen zu den 50% der Studenten aus den mittleren Gruppen (5–9 Aufgaben richtig beantwortet) sind schwieriger zu formulieren: Es könnte sein, dass sie prinzipiell Programme lesen können, ihnen aber gewisse Fertigkeiten für Problemlösungen fehlen.

Wie gut können Sie Programme lesen? Ein Beispiel aus der realen Welt: Vorher…

```
// Orginal-Code ohne jeden Kommentar, denn es ist ja offensichtlich, was er tun sollte…
public static String reFormatDate(String aStrDate) throws ParseException {
	int last = aStrDate.lastIndexOf(".");
	int first = aStrDate.indexOf(".");
	StringBuffer newDate= new StringBuffer();

	if ( last == -1 || first == -1 || first == last)
		throw new ParseException("wrong Date Format "+aStrDate,-1);

	int day = Integer.parseInt(aStrDate.substring(0,first));
	int month = Integer.parseInt(aStrDate.substring(first+1,last));

	String yearStr = aStrDate.substring(last+1, aStrDate.length());

	if (yearStr.length() == 2)
		yearStr = (new StringBuffer("20")).append(yearStr).toString();
	int year = Integer.parseInt(yearStr);
	if (yearStr.length() != 4)
		throw new ParseException("wrong Date Format "+aStrDate,-1);

	if ( day < 10)
		newDate.append('0');
	newDate.append(day).append('.');
	if( month < 10)
		newDate.append('0');
	newDate.append(month).append('.').append(year);
	return newDate.toString();
}
```

Wie gut können Sie Programme lesen? Ein Beispiel aus der realen Welt: … und nachher

```
/**
 * Reformats dates in either dd.mm.yyyy or dd.mm.yy format to dd.mm.yyyy
 * format.
 *
 * @param dateString date string to be validated and reformatted
 * @throws ParseException if dateString has illegal format
 */
public static String reFormatDate(String dateString) throws ParseException {
    Date date = new SimpleDateFormat("dd.MM.yy").parse(dateString);
    // if date is already in yyyy format, SimpleDateFormat will keep the
    // year as is; if data is in yy format, SimpleDateFormat will
    // map it to 2000+yy
    return new SimpleDateFormat("dd.MM.yyyy").format(date);
}
```

Programmlesbarkeit in der Praxis: What the fuck ?!

```
'=============================================
'    Name:                  ChangeSign
'    Description:           Changes the sign of a number
'    Date:                  8/02/2002
'    Revisions:
'=============================================

Public Function ChangeSign(ByVal dblAmount As Double) As Double
On Error GoTo ErrorHandler
    Dim dblMagnitude         As Double
    Dim dblTempValue         As Double
    Dim blnNegativeFlag      As Boolean

    'Bail if they pass zero
    If dblAmount = 0 Then
        ChangeSign = dblAmount
        Exit Function
    End If

    If dblAmount < 0 Then blnNegativeFlag = True
    dblMagnitude = Abs(dblAmount)

    If blnNegativeFlag Then
        'pos value equals magnitude
        dblTempValue = dblMagnitude
    Else
        'Subtract magnitude from zero to get neg value
        dblTempValue = dblTempValue - dblMagnitude
    End If

    ChangeSign = dblTempValue

 Exit Function
'=================================
```

thedailywtf.com

Interessante Beobachtung der Studie: Gekritzel hilft, Programme besser zu lesen

1. Consider the following code fragment:

```
int x[] = {2, 1, 4, 5, 7};
int length = 5;
int limit = 3;
int i = 0;
int sum = 0;

while ( (sum<limit) && (i< length) )
{
      ++i;
      sum += x[i];
}
```

What value is in the variable "i" after this code is executed?

a) 0
b) 1
c) 2
d) 3

sum=0 i=0

	sum	lim	i	len
1	0	3	0	5
			1	5
2		3		5
3		3		5
4		3		

Sum:

Sum: 1

Sum = 4 + 1 = 5

Die wirtschaftliche Bedeutung der Fähigkeit, Programme zu lesen

85% aller Entwicklungsarbeit entfällt auf für Wartung und Weiterentwicklung

60-90% der Fehler in einem Programm können mit Code Reviews gefunden werden

Glass, R. L. Facts and Fallacies of Software Engineering. Addison-Wesley, 2003.

50% der Zeit für Änderungen an bestehender Software werden für Code-Lektüre verwendet

Corbi, T. A. Program understanding: Challenge for the 1990s. IBM Systems Journal, 28(2), 1989.

Programmieren lernen

Programme schreiben

Programme lesen

Fehler machen und beheben

Programmieren ist anspruchsvoll

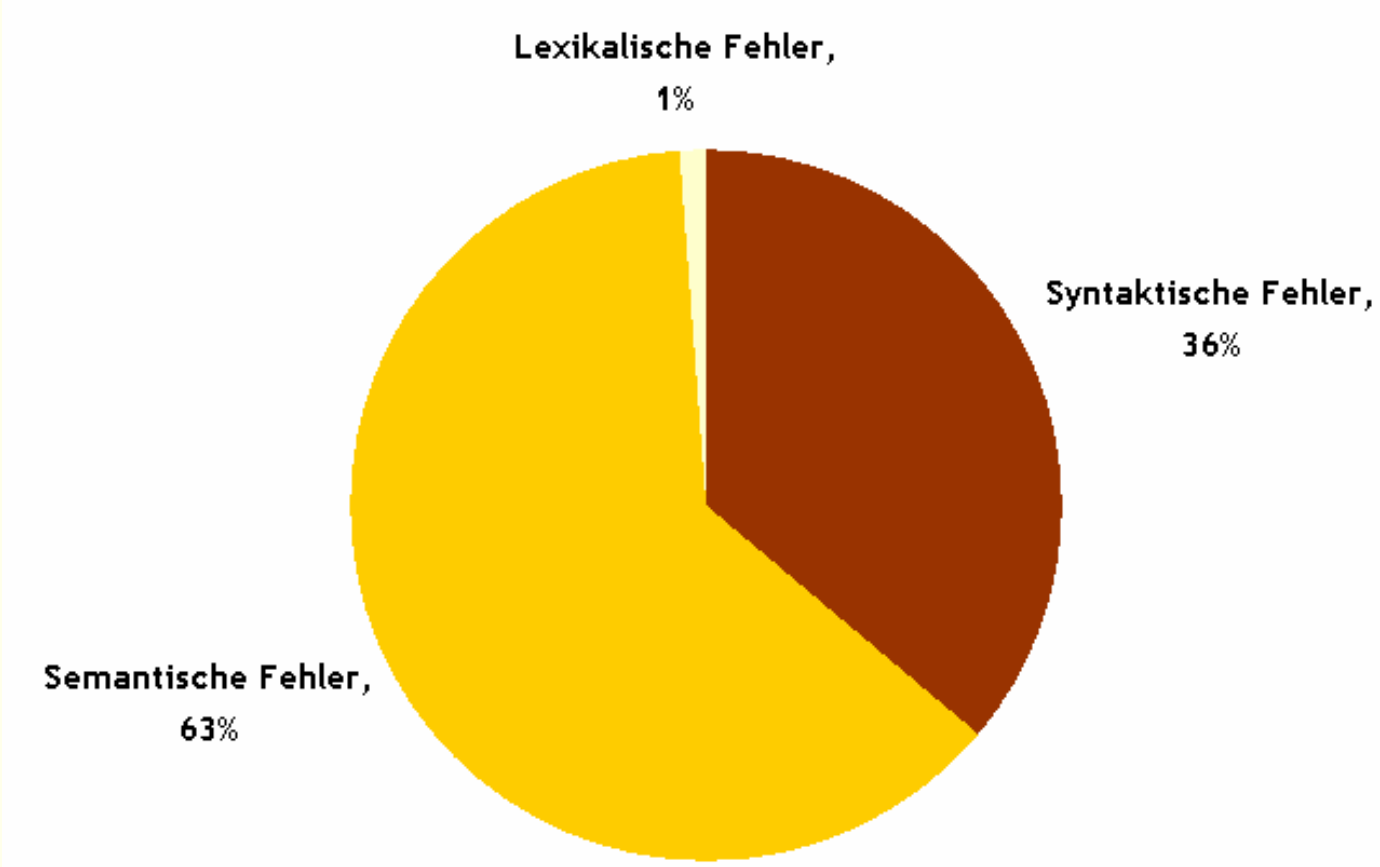
Objektorientierung als Einstieg

Fundamentalkritik: Didaktik versus Pädagogik

Pair Programming: Zu zweit geht‘s besser


CAN'T YOU DO ANYTHING RIGHT?!?
GLASBERGEN
© 1996 Randy Glasbergen.

Fehler machen gehört dazu: Fehlerarten 108'652 untersuchter Fehlern



Ahmadzadeh, Marzieh; Elliman, Dave; Higgins, Colin. An Analysis of Patterns of Debugging Among Novice Computer Science Students. *ITiCSE'05*, June 27–29, 2005.

Fehler machen gehört dazu

Drei Arten von Compiler-Fehlermeldungen

Syntaktische Fehler betreffen die Grammatik der Sprache, zum Beispiel “missing ;”.

Semantische Fehler treten auf, wenn die Bedeutung des Programmtextes unklar ist, zum Beispiel “unknown attribute name” oder “static methods can not be accessed in non-static way”.

Lexikalische Fehler bedeuten, der Compiler kann ein Symbol nicht erkennen, wenn zum Beispiel Parameter nicht Komma-getrennt werden

Ahmadzadeh, Marzieh; Elliman, Dave; Higgins, Colin. An Analysis of Patterns of Debugging Among Novice Computer Science Students. *ITiCSE’05*, June 27–29, 2005.

Fehler machen gehört dazu: Wenige Fehlerarten decken die meisten Fehler ab

6 von 226 Fehlermeldungen (Semantikfehler) des Compiler Jikes decken 50% der aufgetretenen Fehlern ab (Spalten = Übungen):

Compiler Error	Conditional	Loop	Method	Array	Class	File	String
Field Not Found	35	31	32	49	30	39	32
Use of Non-Static Variable Inside the Static Method	12		33	6	21	5	13
Type Mismatch	9	12		8			
Using a non-Initialised Variable		15	6				
Method Call with Wrong Arguments					6		10
Method Name Not Found						8	

Ahmadzadeh, Marzieh; Elliman, Dave; Higgins, Colin. An Analysis of Patterns of Debugging Among Novice Computer Science Students. *ITiCSE'05*, June 27–29, 2005.

Fehler machen gehört dazu: Welche Arten von Fehler machen die Lernenden?

158 kompilierfähige Programme (Lösungen dreier Aufgaben von unterschiedlichem Schwierigkeitsgrad), Total 8'831 Zeilen Pascal-Code:

Eine kleine Anzahl von Fehlerarten erklärt den Grossteil der Fehler: 32%–46% der Fehler auf 10% der Fehlerarten; 55% der Fehler auf 20%.

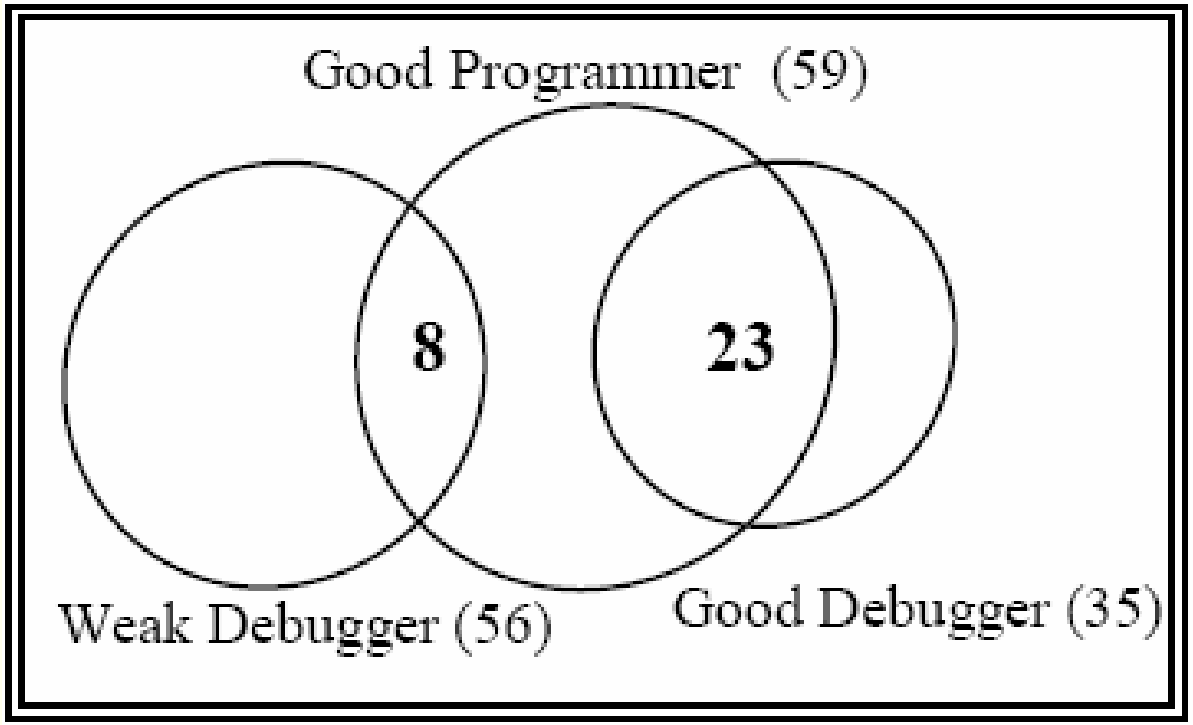
Sprachkonstrukte sind nicht das primäre Problem: 52% der Fehler sind nicht Sprachkonstrukt-basiert, 9% Sprachkonstrukt-basiert, 42% entweder-oder.

Spohrer, James C.; Soloway, Elliot (1986). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, Vol. 29, No. 7, pp. 624–632.

Fehler machen gehört dazu: Debuggen und Fehler beheben ebenso

66% der “guten Debugger” sind “gute Programmierer”

Nur 39% der “guten Programmierer” sind “gute Debugger”



Ahmadzadeh, Marzieh; Elliman, Dave; Higgins, Colin. An Analysis of Patterns of Debugging Among Novice Computer Science Students. *ITiCSE’05*, June 27–29, 2005.

Fehler machen gehört dazu: Debuggen und Fehler beheben ebenso

“weak debuggers […] do not have the ability to use the debugging knowledge in all situations […] have a limited understanding of the scope and applicability of debugging practices”

“**good programmers, but weak debuggers** […] two thirds of them were able to isolate the bug but could not correct it (24 out of 36) […] **did not have the necessary understanding of the actual program implementation**”

Das heisst: Programme schreiben # Programme lesen …

Ahmadzadeh, Marzieh; Elliman, Dave; Higgins, Colin. An Analysis of Patterns of Debugging Among Novice Computer Science Students. *ITiCSE’05*, June 27–29, 2005.

Wie entstehen die Fehler?

Problem: Write a program to read in a person's marital status ('m' = married, or 's' = single) and income. Guard against typos by verifying that the input data is valid, and if it is not, give the person a second chance to enter the data. (Assume the second time, the data will always be typed in correctly.)

Correct Fragment: Unmerged Goals

```
writeln('Enter marital status ( m / s ) : ' ) ;
readln(status); . . . . . .        INPUT FOR STATUS
if (status <> ' m' ) and (status <> 's') then begin . . . . . .      GUARD FUR STATUS
  writeln('Try again: m or s');
  readln(status); . . . . . .      RETRY FOR STATUS
end;
writeln('Enter income (e.g. 175OO.O0):');
readln(income);  . . . . . .       INPUT FOR INCOME
if (income < 0) then begin . . . . . .      GUARD FOR INCOME
  writeln('Try again: income greater-than 0');
  readln(income)  . . . . . .      RETRY FOR INCONE
end;
```

James C. Spohrer, Elliot Soloway, Edgar Pope. Where The Bugs Are. Proceedings of the SIGCHI conference on Human factors in computing systems, 1985, 47-53.

Wie entstehen die Fehler?

Fehlerhaftes Fragment: Zwei Ziele (Eingaben) werden auf parallel verfolgt statt sequentiell, was in einem komplexeren Boole'schen Ausdruck resultiert

```
writeln('Enter marital status (s/m) and income (e.g., 17500.00:');
readln(status, income); . . . . . INPUT FOR STATUS AND INCOME
if (income < 0) or (status <> 's' ) or (status <> 'a') then begin . . GUARD
  writeln('Try again: status s or m, income greater-than 0');
  readln(status, income)  . . . . . RETRY FOR STATUS AND INCOME
end;
```

James C. Spohrer, Elliot Soloway, Edgar Pope. Where The Bugs Are. Proceedings of the SIGCHI conference on Human factors in computing systems, 1985, 47-53.

Wie entstehen die Fehler?

Korrigiertes Fragment: Zwei Ziele (Eingaben) werden auf parallel verfolgt statt sequentiell, mit korrigierter Bedingung

```
writeln('Enter marital status (s/m) and income (e.g., 17500.00:');
readln(status, income); . . . . . INPUT FOR STATUS AND INCOME
if (income < 0) or ((status <> 's' ) and (status <> 'a')) then begin . GUARD
  writeln('Try again: status s or m, income greater-than 0');
  readln(status, income)  . . . . . RETRY FOR STATUS AND INCOME
end;
```

James C. Spohrer, Elliot Soloway, Edgar Pope. Where The Bugs Are. Proceedings of the SIGCHI conference on Human factors in computing systems, 1985, 47-53.

Wie entstehen die Fehler?

Die zentrale Beobachtung der Studie: “**students tend to write code with merged goals […] and they tend to write buggier code than those who do not.**”

Was die Frage aufwirft: Warum verfolgen Lernende mehrere Ziele gleichzeitig?

Die Hypothese der Autoren: “Merging” entspricht unseren Alltagsstrategien, welche die Lernenden unbewusst auf den für sie neuen Bereich der Programmierung anwenden.

James C. Spohrer, Elliot Soloway, Edgar Pope. Where The Bugs Are. Proceedings of the SIGCHI conference on Human factors in computing systems, 1985, 47-53.

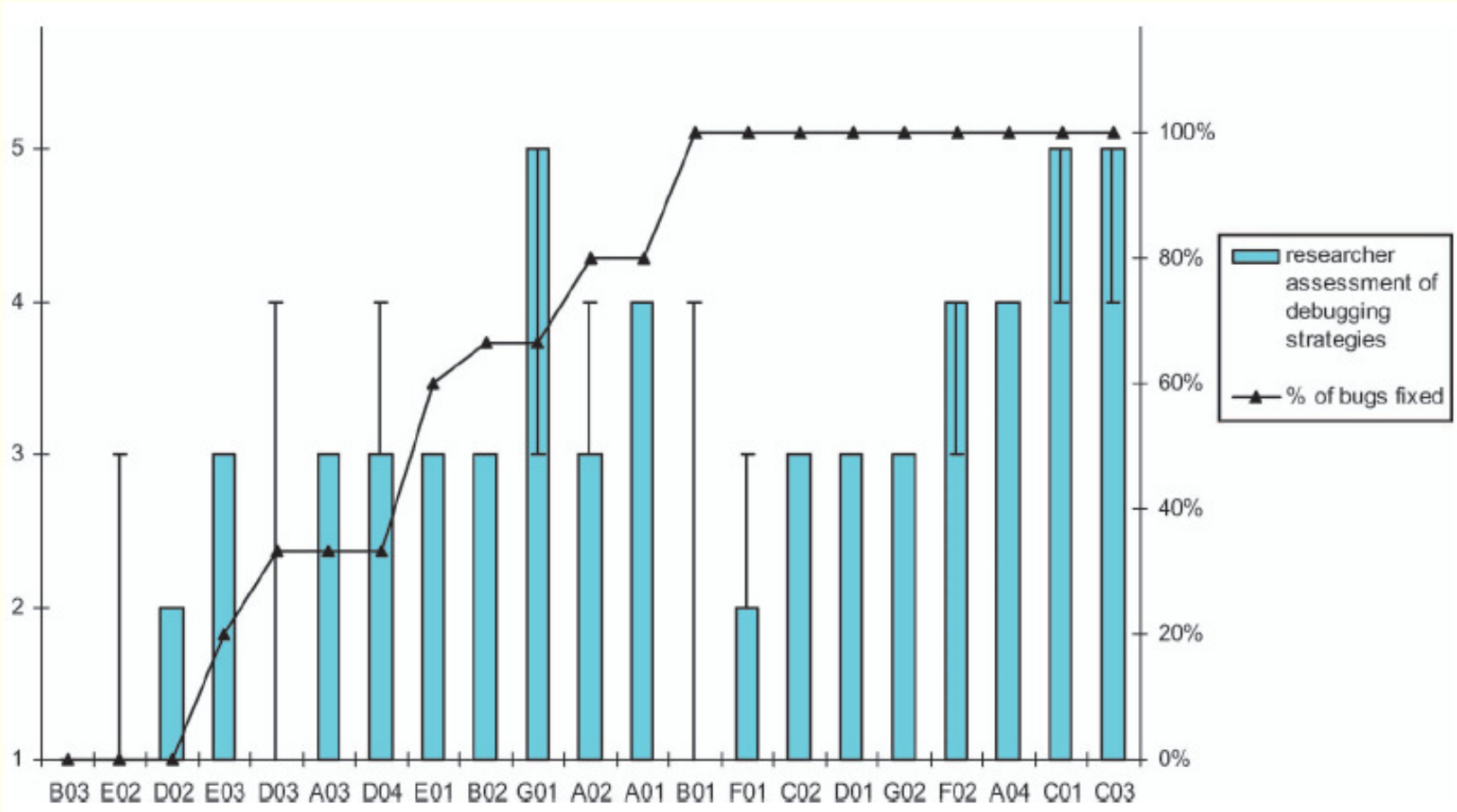
Fehler finden ist schwieriger als Fehler zu beheben

„Students employed a variety of strategies to find 70% of all bugs and of the bugs they found they were able to fix 97% of them.“

„They had the most difficulty with malformed statements, such as arithmetic errors and incorrect loop conditions.“

Fitzgerald, Sue; Lewandowski, Gary; McCauley, Renée; Murphy, Laurie; Simon, Beth; Thomas, Lynda; Zander, Carol. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, Vol. 18, No. 2, June 2008, 93–116.

Debugging-Fähigkeiten: Selbsteinschätzung vs. Lehrereinschätzung vs. Leistung



Fitzgerald, Sue; Lewandowski, Gary; McCauley, Renée; Murphy, Laurie; Simon, Beth; Thomas, Lynda; Zander, Carol. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, Vol. 18, No. 2, June 2008, 93–116.

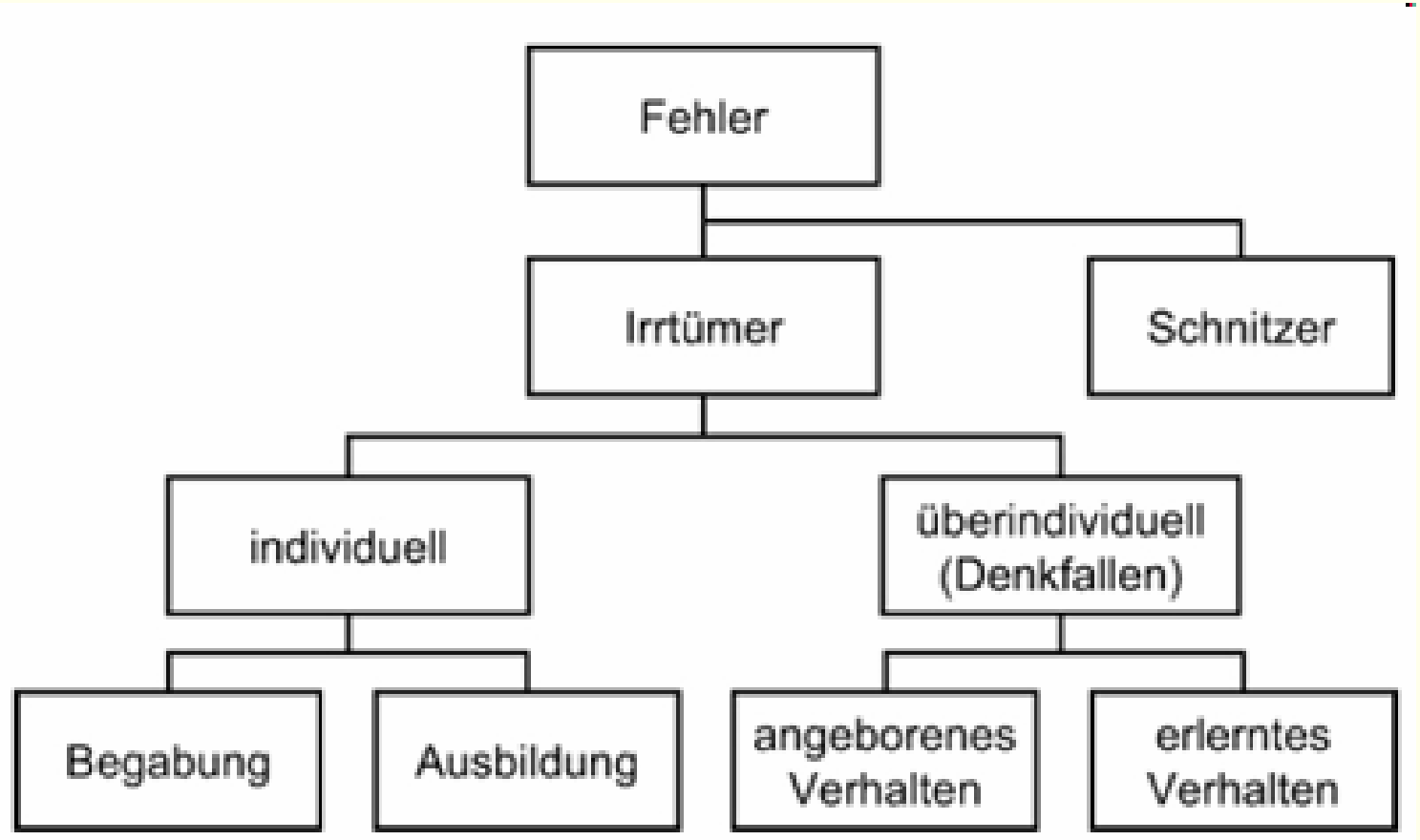
Debugging-Fähigkeiten: Selbsteinschätzung vs. Lehrereinschätzung vs. Leistung

„We see that these two ratings are generally aligned for the middle and upper ability students, although none of the students responded to the question „Do you consider yourself a good debugger?“ with a rating of 5. […] One surprising aspect […] **students who had just found and fixed all bugs in a program in a 20 minute window frequently (5/9) scored their debugging ability as average**. None of the students rated themselves at the highest debugging skill ranking.“

„In contrast, for those whose strategy assessments were poor, all but one responded to the question with a self-rating of 3 or 4.“

Fitzgerald, Sue; Lewandowski, Gary; McCauley, Renée; Murphy, Laurie; Simon, Beth; Thomas, Lynda; Zander, Carol. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, Vol. 18, No. 2, June 2008, 93–116.

Welche grundsätzlichen Arten von Fehlern gibt es?



Grams, Timm (1990). Denkfallen und Programmierfehler. Springer-Verlag.


Der unbekannte Retter
der Menschheit

Apropos Fehler, und apropos „worst case“…

„[…] 26. September 1983 meldete der Computer eine auf die Sowjetunion anfliegende amerikanische Atomrakete. […] Kurze Zeit später meldete das Computersystem eine zweite, dritte, vierte und fünfte abgefeuerte Rakete. Petrow glaubte weiterhin an einen Fehlalarm, hatte jedoch keinerlei andere Quellen […] Kurz danach an diesem Morgen stellte sich heraus, dass […] das satellitengestützte sowjetische Frühwarnsystem hatte Sonnenreflexionen auf Wolken in der Nähe der *Malmstrom Air Force Base* in Montana, wo auch amerikanische Interkontinentalraketen stationiert waren, als Raketenstarts fehlinterpretiert.“

http://de.wikipedia.org/wiki/Stanislaw_Jewgrafowitsch_Petrow
http://www.20min.ch/news/wissen/story/Der-unbekannte-Retter-der-Menschheit-11616307

Programmieren lernen

Programme schreiben

Programme lesen

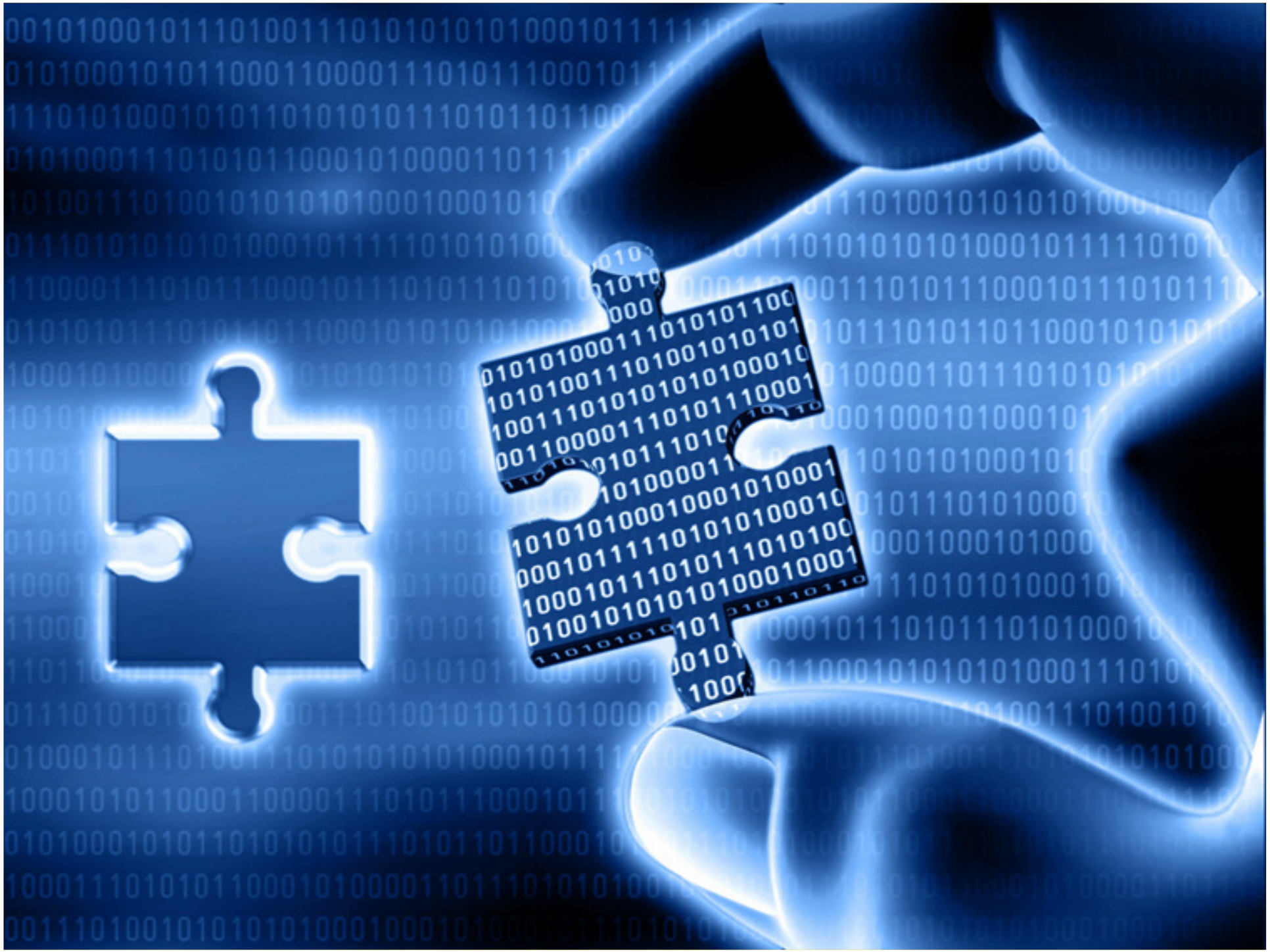
Fehler machen und beheben

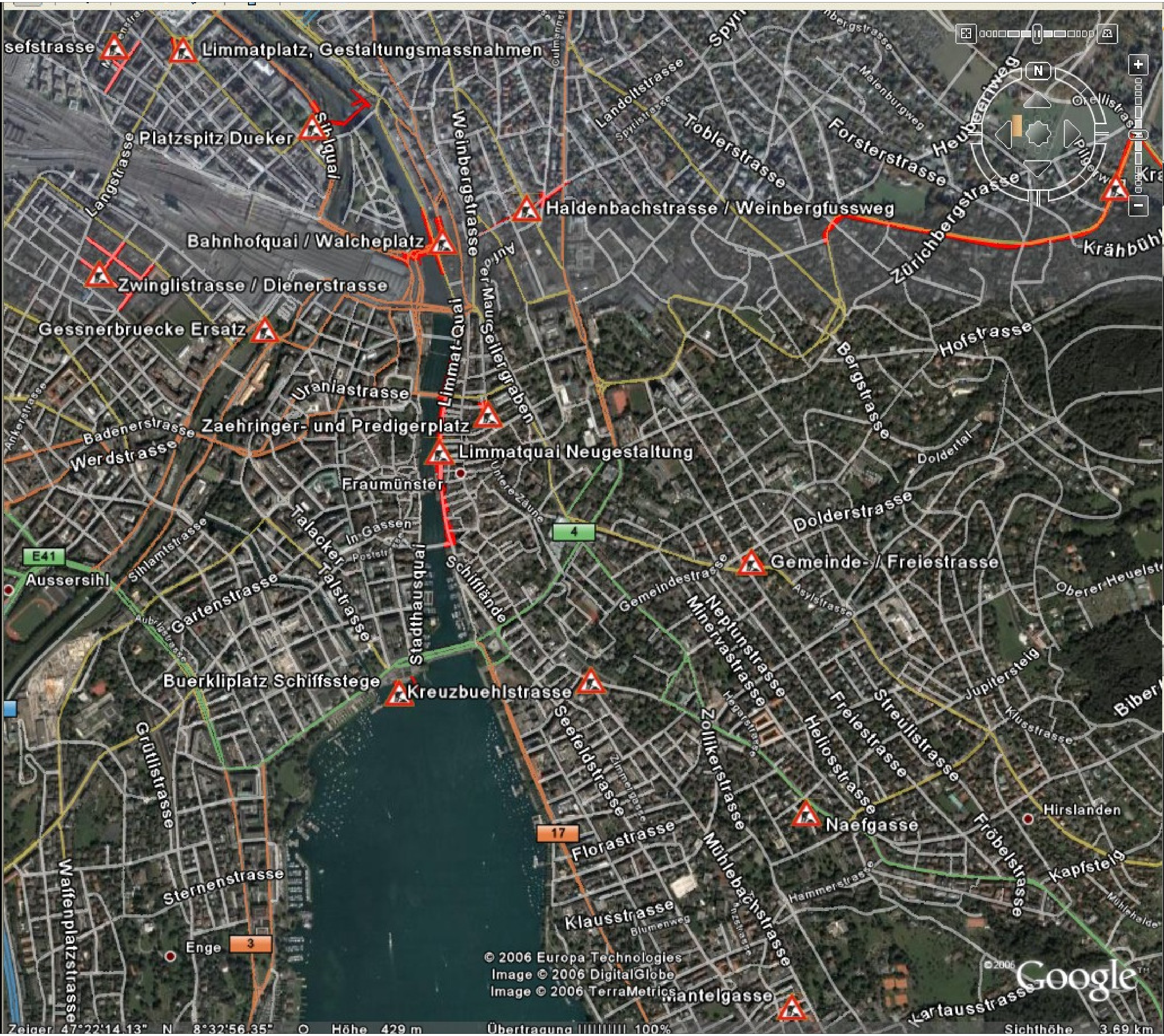
Programmieren ist anspruchsvoll

Objektorientierung als Einstieg

Fundamentalkritik: Didaktik versus Pädagogik

Pair Programming: Zu zweit geht‘s besser





Limmatplatz, Gestaltungsmassnahmen
Platzspitz Dueker
Haldenbachstrasse / Weinbergfussweg
Bahnhofquai / Walcheplatz
Zwinglistrasse / Dienerstrasse
Gessnerbruecke Ersatz
Zaehringer- und Predigerplatz
Limmatquai Neugestaltung
Fraumünster
Gemeinde- / Freiestrasse
Buerkliplatz Schiffsstege
Kreuzbuehlstrasse
Naefgasse
Mantelgasse
Sihlquai
Langstrasse
Weinbergstrasse
Limmat-Quai
Seilergraben
Uraniastrasse
Badenerstrasse
Werdstrasse
Talacker
Talstrasse
Stadthausquai
Schifflände
Gartenstrasse
Aussersihl
Grütlistrasse
Sternenstrasse
Enge
Waffenplatzstrasse
Seefeldstrasse
Florastrasse
Klausstrasse
Zollikerstrasse
Mühlebachstrasse
Neptunstrasse
Minervastrasse
Freiestrasse
Heliosstrasse
Streulistrasse
Fröbelstrasse
Hirslanden
Kapfsteig
Jupitersteig
Dolderstrasse
Bergstrasse
Hofstrasse
Zürichbergstrasse
Tobelerstrasse
Forsterstrasse
Landoltstrasse
Gemeindestrasse
Kartausstrasse
© 2006 Europa Technologies
Image © 2006 DigitalGlobe
Image © 2006 TerraMetrics
Google
Zeiger 47°22'14.13" N 8°32'56.35" O Höhe 429 m
Übertragung 100%
Sichthöhe 3.69 km

Programmieren lernen ist schwierig

Programmieren heisst, fünf Themenbereiche zu meistern

1. Ein Verständnis dafür, was programmierbar ist,
2. das zugrundeliegende Modell des zu programmierenden Computers ("notional machine"),
3. Syntax und Semantik der verwendeten Programmiersprache,
4. Strukturen von Lösungsansätzen (kleine algorithmische Schemata, oder auch Patterns),
5. praktische Vorgehensweisen wie Programmentwurf, Entwicklung, Testen, Debuggen.

⇒ Diese Bereiche sind voneinander abhängig und müssen daher miteinander erlernt werden.

⇒ Kognitionspsychologisch betrachtet ist Programmieren ein komplexer explorativer und inkrementeller Prozess und kein Prozess, der einer einfachen linearen Logik gehorcht.

Robins, Anthony; Rountree, Janet; Rountree, Nathan (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, Vol. 13, No. 2, pp. 137–172.

Programmieren lernen ist schwierig

Modelle und Prozesse

1. Bevor ein Programm geschrieben werden kann, muss die Problemdomäne verstanden werden.
2. Zudem braucht es ein Verständnis des Computermodells ("notational machine"), das durch die gegebenen Programmiersprache impliziert wird.
3. Eine weitere Herausforderung ist das Verständnis des Zusammenhangs zwischen der statischen Beschreibung eines Programms, also dem Programmquellcode, und der dynamischen Ausführung dieses Programms, also sein Verhalten zur Laufzeit.
4. Laufzeitverhalten beinhaltet Gesichtspunkte wie Kontrollfluss, Datenfluss, Datenstrukturen etc., wobei insbesondere der Datenfluss für Anfänger schwierig zu verstehen ist.

Programmieren lernen: Schach spielen lernen

"Master and novice chess players were shown a **meaningful chess board**, and asked to recall the pieces (once the board was taken away). The masters recalled more of the pieces than did the novices.

Next, the masters and novices were shown a board in which the chess pieces were **placed randomly** on the board. What happened? The performance of the masters was the same as that of the novices."

Experten erkennen Stellungen und Muster ("information chunks") auf dem Spielbrett, die Neulinge nicht sehen.

Chase, W.C., and Simon, H. Perception in chess. *Cognitive Psychology* 4 (1973), 55-81, zitiert nach Soloway, Elliot (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, Vol. 29, No. 9, pp. 850–858.

Programmieren lernen: „Goals and Plans“erlernen

Beispiel “Averaging problem: Write a program that will read in integers and output their average. Stop reading when the value 99999 is input.”

Eine Lösung dafür bedingt die Kombination von zwei “prototypischen Plänen” (oder neudeutscher: Patterns):

1. Summe Berechnen

```
initialize a running total
ask user for a value
if input is not the sentinel value then
    add new value into running total
    loop back to input
end
```

2. Anzahl Inputs zählen

```
initialize a counter
ask user for a value
if input is not the sentinel value then
    increment counter
    loop back to input
end
```

Soloway, Elliot (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, Vol. 29, No. 9, pp. 850–858.

Programmieren lernen: „Goals and Plans“ erlernen

Beispiel “Averaging problem: Write a program that will read in integers and output their average. Stop reading when the value 99999 is input.”

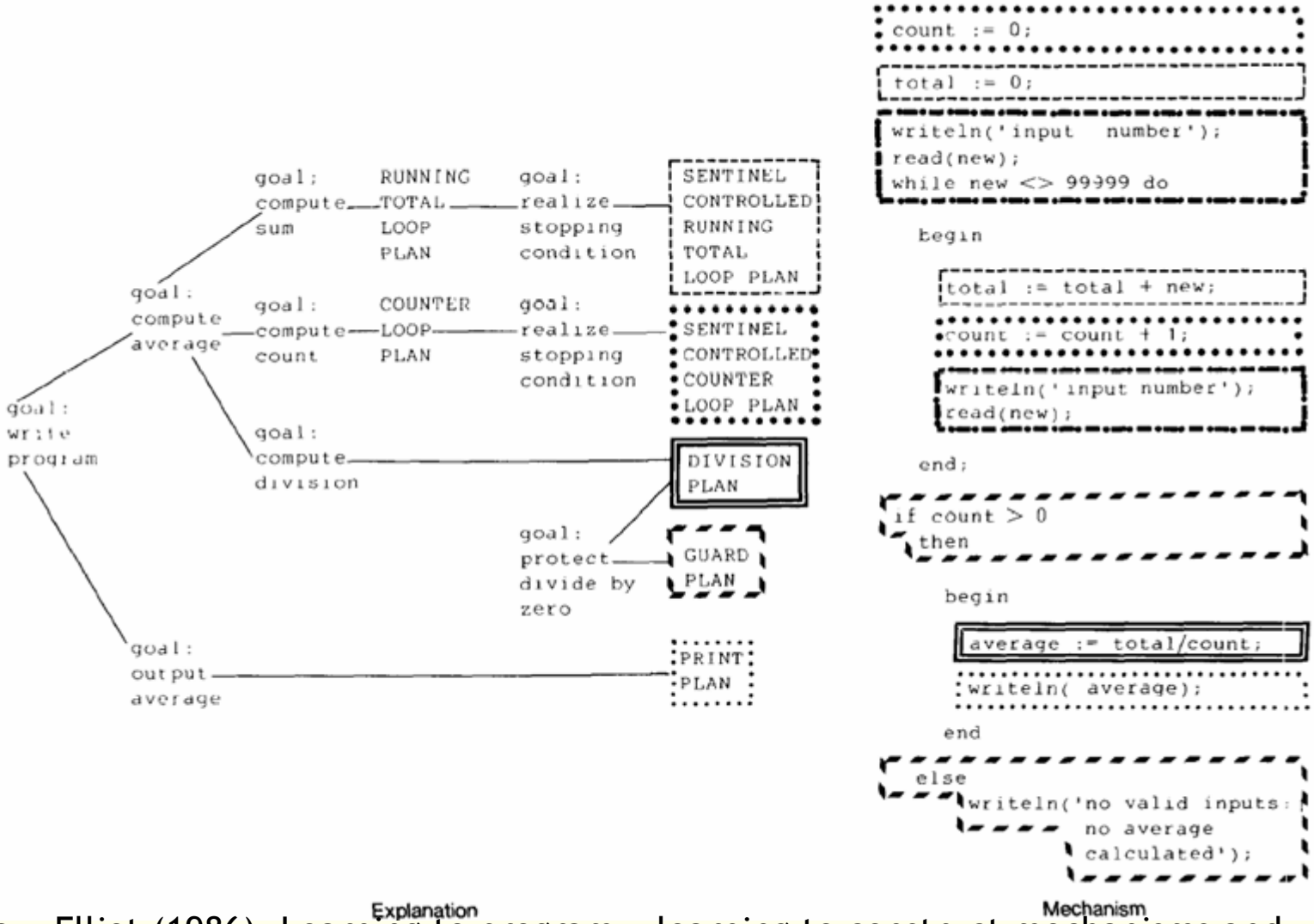
Die Lösung mit beiden Plänen kombiniert sowie mit dem Plan für Durchschnittsberechnung:

```
initialize a running total
initialize a counter
ask user for a value
if input is not the sentinel value then
    add new value into running total
    increment counter
    loop back to input
end
```

```
if count is greater than 0 then
    do the calculation
else
    report the problem to the user
```

Soloway, Elliot (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, Vol. 29, No. 9, pp. 850–858.

Programmieren lernen: Programme auf „Goals and Plans“ analysieren



Soloway, Elliot (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, Vol. 29, No. 9, pp. 850–858.

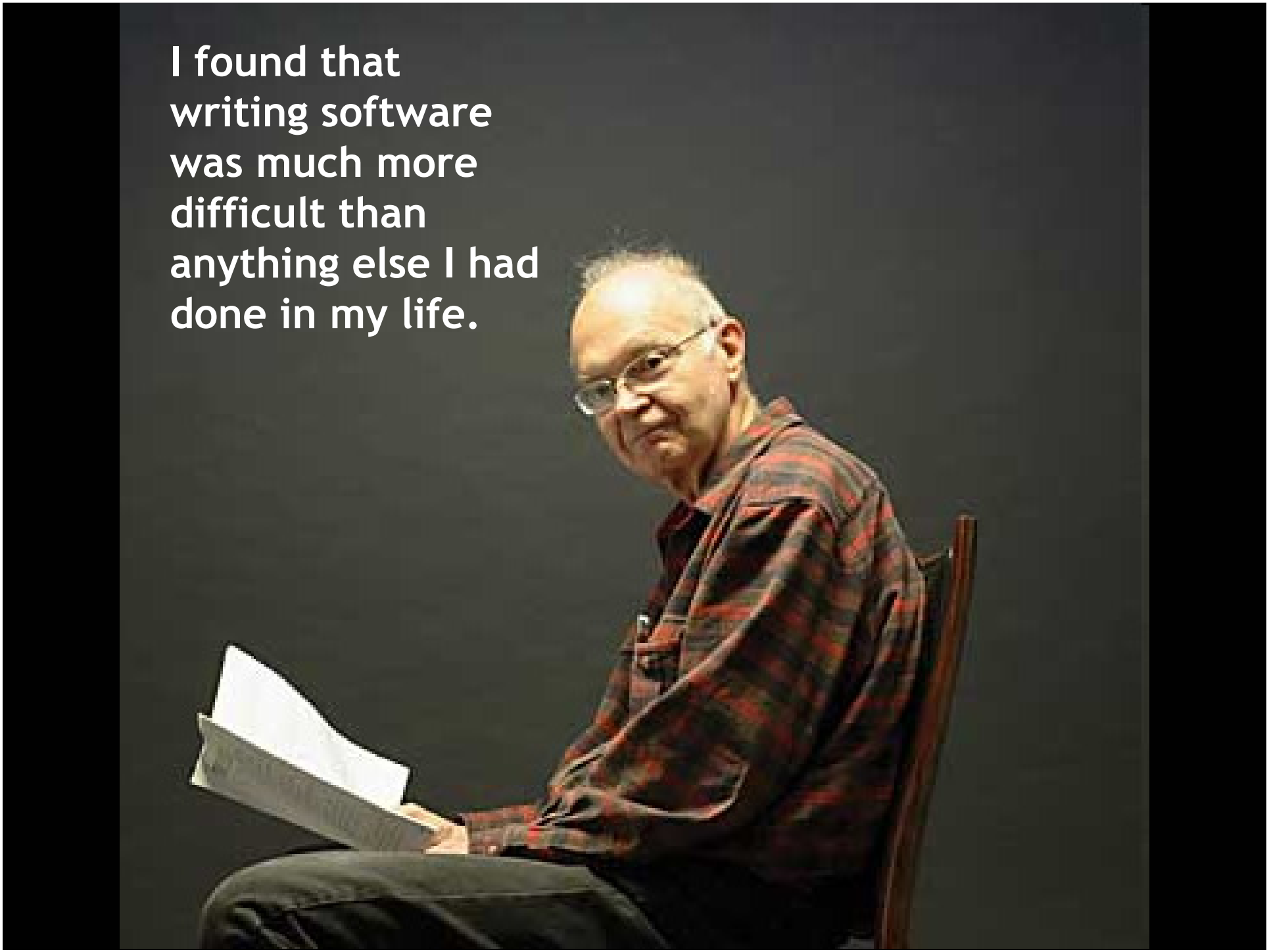
Programmieren lernen: „Goals and Plans“ erlernen

“Clevere” Programme verwirren auch Experten: “the performance of advanced programmers was reduced to that of novice programmers when the advanced programmers were asked to deal with programs that violated various rules of discourse”

```
Program Average
VAR    count : INTEGER;
       sum, average, number : REAL;
BEGIN
    sum := -99999; . . . . . .
    count := -1;       . . . . . .
    REPEAT
        writeln('please input a number');
        read (number)
        sum := sum + number;
        count := count + 1;
    UNTIL (number = 99999);
    average := sum/count;
    writeln('the average is: ',average);
END.
```

Wert hängt ab von Schlaufenendbedingung
Funktioniert nur wg. nicht-abweisender Schlaufe

Soloway, Elliot (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, Vol. 29, No. 9, pp. 850-858.


I found that
writing software
was much more
difficult than
anything else I had
done in my life.

Programmieren ist anspruchsvoller, als man meinen könnte…

I found that writing software was much more difficult than anything else I had done in my life. I had to keep so many things in my head at once. I couldn't just put it down and start something else. It really took over my life during this period. I used to think there were different kinds of tasks: writing a paper, writing a book, teaching a class, things like that. I could juggle all of those simultaneously. **But software was an order of magnitude harder.** I couldn't do that and still teach a good Stanford class.

Donald Knuth (2008) im Interview „Donald Knuth: A Life's Work Interrupted", Communications of the ACM, Vol. 51, No. 8, pp. 31-35.

… anspruchsvoller, als man meinen könnte… und wird häufig unterschätzt…

„I did sincerely believe that [LaTex] was only going to take me a year to do it. […] “Okay, implement this, please, this summer. That’s your summer job.” I thought I had specified a language. To my amazement, the students, who were outstanding students, did not complete it. They had a system that was able to do only about three lines of TeX. I thought, “My goodness, what’s going on? I thought these were good students.” Later I changed my attitude, saying, “Boy, they accomplished a miracle.” Because going from my specification, which I thought was complete, they really had an impossible task, and they had succeeded wonderfully with it.“ - **„it was to be a 10-year project.”**

Donald Knuth (2008) im Interview „Donald Knuth: A Life’s Work Interrupted”, Communications of the ACM, Vol. 51, No. 8, pp. 31-35.

Programmieren lernen

Programme schreiben

Programme lesen

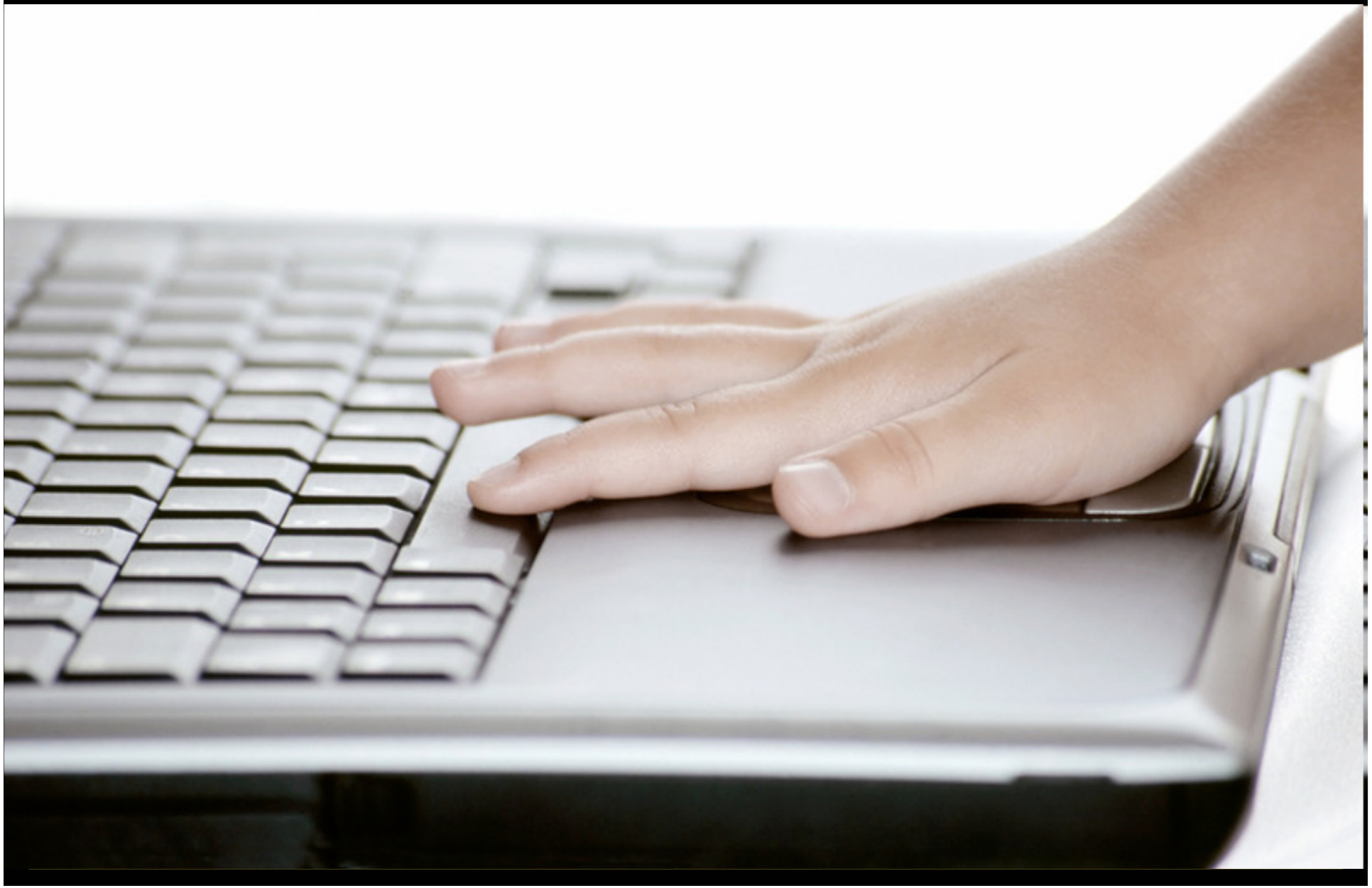
Fehler machen und beheben

Programmieren ist anspruchsvoll

Objektorientierung als Einstieg

Fundamentalkritik: Didaktik versus Pädagogik

Pair Programming: Zu zweit geht‘s besser



Angenommen, objektorientierte Programmierung sei intuitiv…

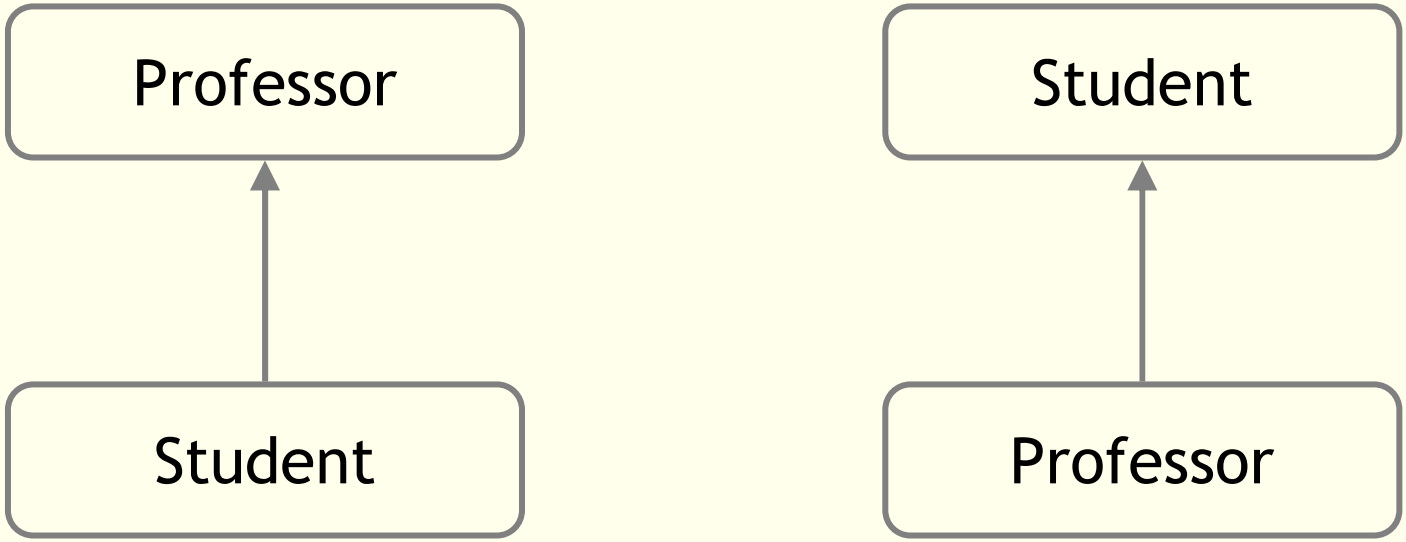
Eine kleine Rechenaufgabe

> Ein Baseball-Schläger und ein Ball kosten zusammen einen Dollar und 10 Cents. Der Schläger kostet einen Dollar mehr als der Ball. Wieviel kostet der Ball?

Wie hilfreich ist Intuition bei analytischen Aufgaben?

Hadar, Irit; Leron, Uri (2008). How intuitive is object-oriented design? *Communications of the ACM*, Vol. 51, No. 5, pp. 41–46.

Objektorientierte Programmierung: Wer erbt hier von wem?



Hadar, Irit; Leron, Uri (2008). How intuitive is object-oriented design? *Communications of the ACM*, Vol. 51, No. 5, pp. 41–46.

Objektorientierte Programmierung: Objekte identifizieren ist schwierig

Ein kleines Beispiel aus der Studie

“design an authorization system that routes users as follows:

1. An existing user will login into the system.
2. A new user will register and receive authorization.”

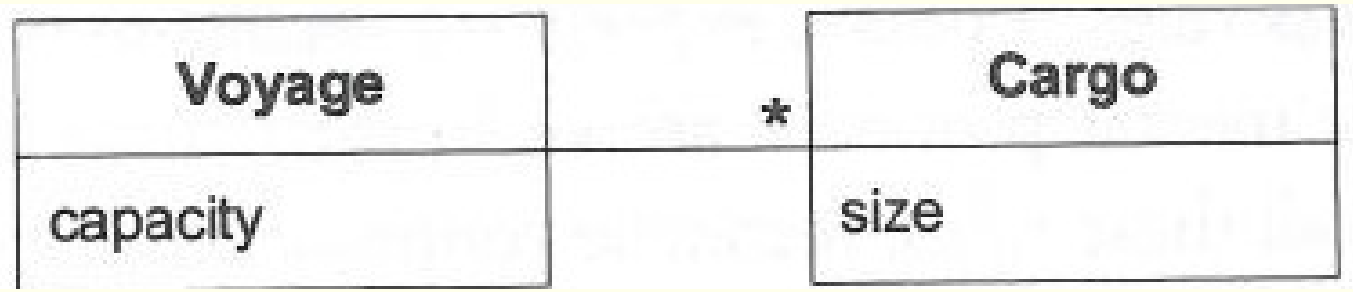
Was sind hier die Objekte? User? Login? Register? Authorization? Für Anfänger schwierige Fragen!

Hadar, Irit; Leron, Uri (2008). How intuitive is object-oriented design? *Communications of the ACM*, Vol. 51, No. 5, pp. 41–46.

Objektorientierte Programmierung: Objekte identifizieren ist schwierig

Problemstellung: Eine Fahrt (Voyage) eines Frachtschiffs hat mehrere Frachten (Cargo). Das Schiff hat eine gewissen Kapazität, jede Fracht eine Grösse. Es soll 10% Überbuchung zulässig sein.

Lösungsansatz:

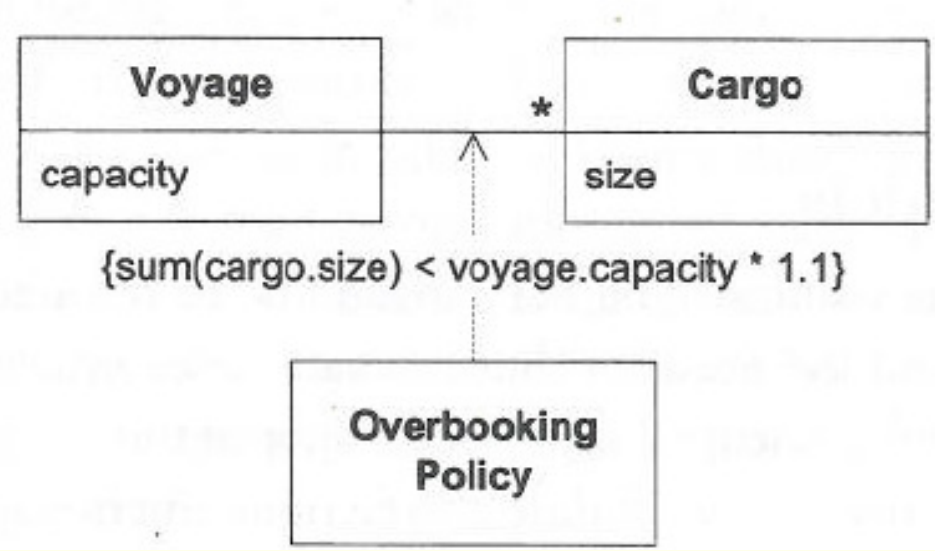


Evans, Eric (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional.

Objektorientierte Programmierung: Objekte identifizieren ist schwierig

Problemstellung: Eine Fahrt (Voyage) eines Frachtschiffs hat mehrere Frachten (Cargo). Das Schiff hat eine gewissen Kapazität, jede Fracht eine Grösse. Es soll 10% Überbuchung zulässig sein.

Alternativer Lösungsansatz mit expliziter Überbuchungsstrategie:



Evans, Eric (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional.

Objektorientierte Programmierung: Design-Entscheide aufgrund der Zugänglichkeit

“Highly accessible features will influence decisions, while features of low accessibility will be largely ignored.

Unfortunately, there is no reason to believe that the most accessible features are also the most relevant to a good decision.”

Kahneman, D. (Nobel Prize Lecture). Maps of bounded rationality: A perspective on intuitive judgment and choice. In Les Prix Nobel, T. Frangsmyr, Ed. (2002), 416-499.

Objektorientierte Programmierung: Wie naheliegend sind OO-Modelle?

„[…] claims regarding the **‘‘naturalness, ease of use, and power’’** of the OO approach. […]

The papers reviewed do not support this position. They show that **identifying objects is not an easy process,**

that **objects identified in the problem domain are not necessarily useful in the program domain,**

that the mapping between domains is not straightforward, and that novices need to construct a model of the procedural aspects of a solution in order to properly design objects/classes.

Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and Teaching Programming: A Review and Discussion, *Computer Science Education,* 2003, Vol. 13, No. 2, pp. 137–172.

Objektorientierte Programmierung

Was ist eine Klasse?

Objektorientierung: „A Class Defines a Data Type”

“A class [...] defines a data type. Yet this fact is typically not mentioned in CS1 textbooks. Lacking an accurate description of a data type, the best a textbook (or an instructor) can do is say something like: A class is a template, blueprint, or pattern of an object. These characterizations suggest how a class can be viewed, not what a class is, much less what a class can do, thus making it difficult to learn the true nature of a user-defined data type.”

Chenglie Hu. Just Say 'A Class Defines a Data Type'. ACM Communications, Vol.51, No.3, March 2008, p. 19–21.


Objects First
with
Java™
A Practical Introduction Using BlueJ
Fourth Edition
CD-ROM Included
David Barnes & Michael Kölling

?

Objektorientierung: Objects first?

Der Ansatz “objects first” gerät in einer zunehmenden Anzahl von Publikationen in die Kritik. Wie wurde der Ansatz überhaupt begründet?

“It is now almost consensus among OO teachers that object orientation is best taught by teaching about objects from the start, rather than starting with a small scale, structured programming approach and adding objects later. While there is very little scientific evidence to support this, the anecdotal evidence is so strong that the great majority of teachers and textbooks now follow this approach.“

Kölling, Michael; Rosenberg, John (2001). Guidelines for Teaching Object Orientation with Java. In *Conference on Information Technology in Computer Science Education*, pp. 33–36.

Programmieren lernen

Programme schreiben

Programme lesen

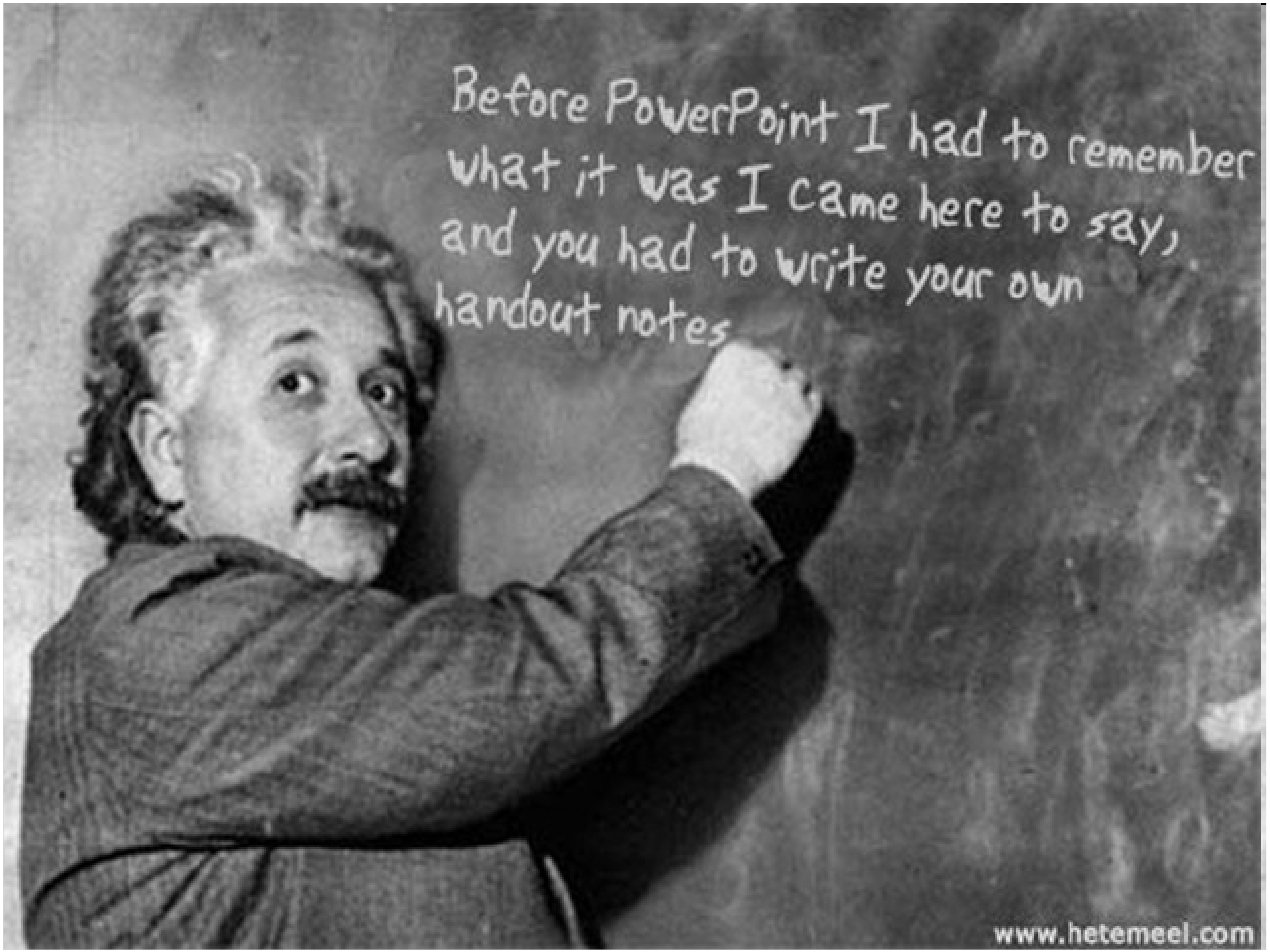
Fehler machen und beheben

Programmieren ist anspruchsvoll

Objektorientierung als Einstieg

Fundamentalkritik: Didaktik versus Pädagogik

Pair Programming: Zu zweit geht‘s besser


Before PowerPoint I had to remember
what it was I came here to say,
and you had to write your own
handout notes
www.hetemeel.com

Minimal Guidance During Instruction: Constructivist, Discovery, Problem-Based Learning Methods

First, they challenge students to solve "authentic" problems or acquire complex knowledge in information-rich settings based on the assumption that **having learners construct their own solutions** leads to the most effective learning experience.

Second, they appear to assume that **knowledge can best be acquired through experience based on the procedures of the discipline** (i.e., seeing the pedagogic content of the learning experience as identical to the methods and processes or epistemology of the discipline being studied)

Kirschner, P. A., Sweller, J., and Clark, R. E. (2006) Why minimal guidance during instruction does not work: an analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. Educational Psychologist 41 (2) 75-86

Minimal Guidance During Instruction: Constructivist, Discovery, Problem-Based Learning Methods

After a half-century of advocacy associated with instruction using minimal guidance, it appears that there is no body of research supporting the technique.

In so far as there is any **evidence** from controlled studies, it almost uniformly **supports direct, strong instructional guidance** rather than constructivist-based minimal guidance **during the instruction of novice to intermediate learners**.

Kirschner, P. A., Sweller, J., and Clark, R. E. (2006) Why minimal guidance during instruction does not work: an analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. Educational Psychologist 41 (2) 75-86

Lesen von ausgearbeiteten Beispielen besser als Aufgaben selber lösen

There are two groups of students, each of which is shown two worked-out algebra problems. Our experimental group then gets eight more algebra problems, completely worked out. Our control group solves those eight more problems. As you might imagine, the control group takes five times as long to complete the eight problems than the experiment group takes to simply read them. Both groups then get new problems to solve. **The experimental group solves the problems in half the time and with fewer errors than the control group. Not problem-solving leads to better problem-solving skills than those doing problem-solving.**

Sweller, J., & Cooper, G. A. (1985). "The use of worked examples as a substitute for problem solving in learning algebra". Cognition and Instruction 2 (1): 59–89.

Lesen von ausgearbeiteten Beispielen besser als Aufgaben selber lösen

Kontroll-Gruppe	Experiment-Gruppe
2 Musterlösungen studieren	2 Musterlösungen studieren
8 Aufgaben selber lösen 5x Minuten	8 weitere Musterlösungen x Minuten
Weitere Aufgaben lösen <wieviele?> Minuten	Weitere Aufgaben lösen x Minuten

Sweller, J., & Cooper, G. A. (1985). "The use of worked examples as a substitute for problem solving in learning algebra". Cognition and Instruction 2 (1): 59–89.

Lesen von ausgearbeiteten Beispielen besser als Aufgaben selber lösen

Kontroll-Gruppe	Experiment-Gruppe
2 Musterlösungen studieren	2 Musterlösungen studieren
8 Aufgaben selber lösen 5x Minuten	8 weitere Musterlösungen x Minuten
Weitere Aufgaben lösen **2x Minuten**	Weitere Aufgaben lösen **x Minuten** **Weniger Fehler**

Sweller, J., & Cooper, G. A. (1985). "The use of worked examples as a substitute for problem solving in learning algebra". Cognition and Instruction 2 (1): 59–89.

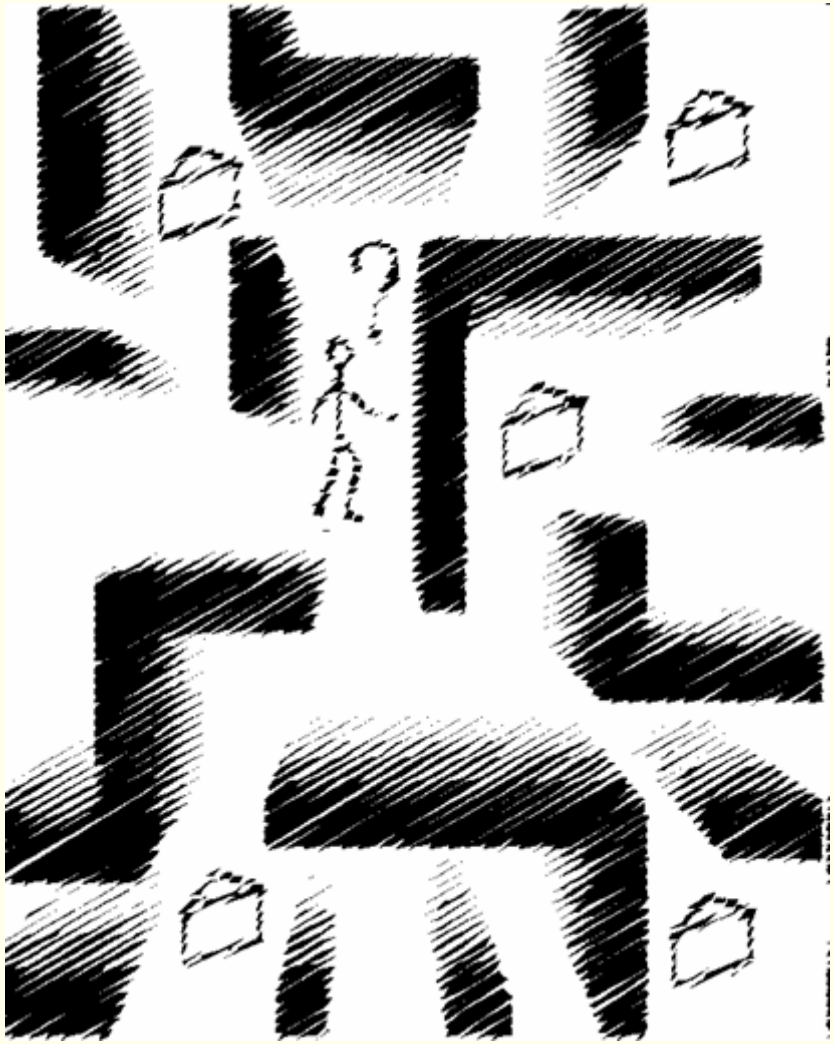
Warum funktioniert die „althergebrachte“ Didaktik besser?

Cognitive Overload

This literature is not saying never program. **Rather, it's a bad way to start.** Students need the opportunity to gain knowledge first, before programming, just as with reading.

Later, there is a expertise reversal effect, where the worked example effect disappears then reverses. Intermediate students do learn better with real programming, real problem-solving. There is a place for minimally guided student activity, including programming. It's just not at the beginning.

http://computinged.wordpress.com/



Programmieren lernen

Programme schreiben

Programme lesen

Fehler machen und beheben

Programmieren ist anspruchsvoll

Objektorientierung als Einstieg

Fundamentalkritik: Didaktik versus Pädagogik

Pair Programming: Zu zweit geht‘s besser

Pair Programming

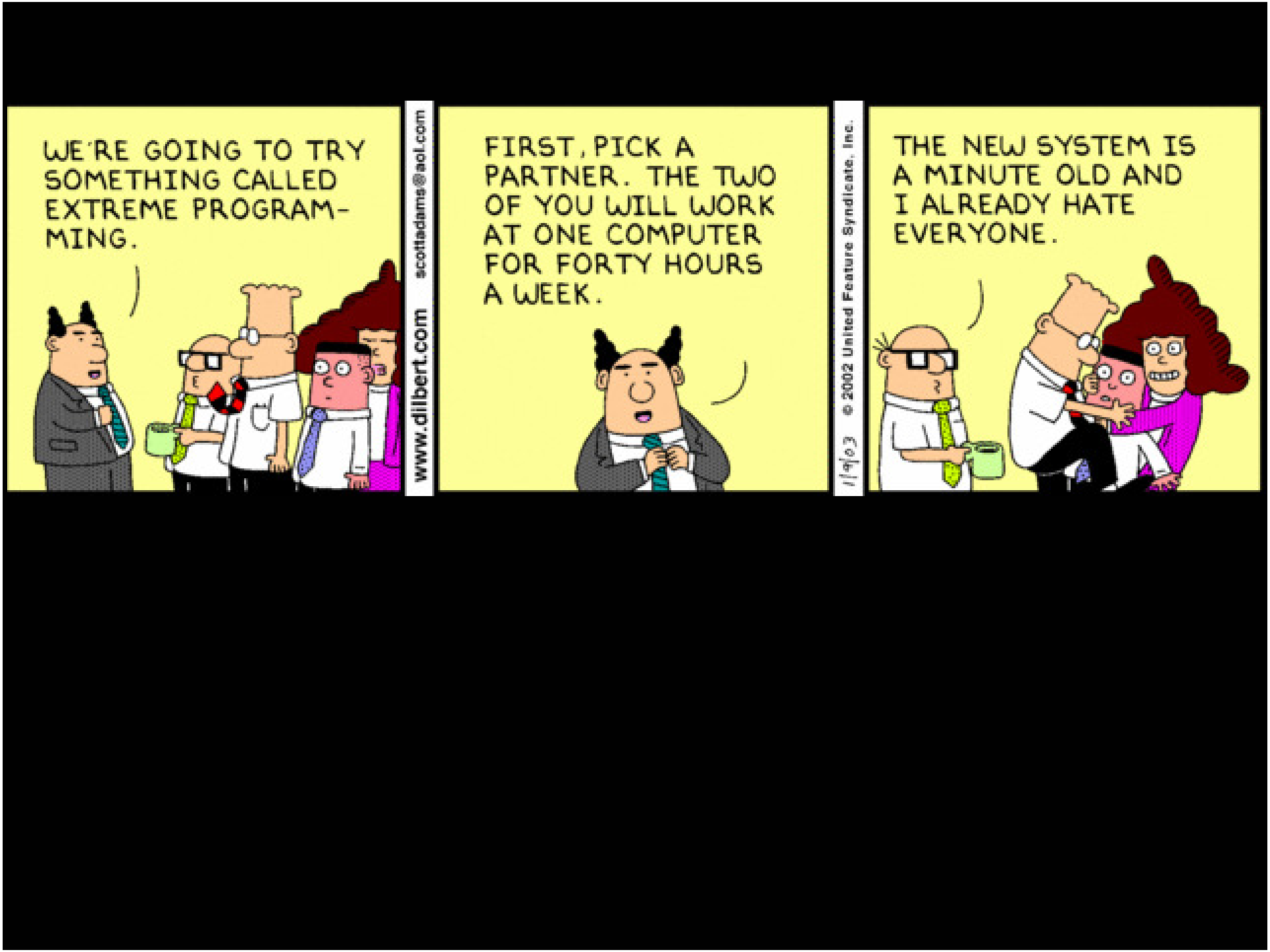
- Zwei Entwickler an einem PC.
- Der Driver programmiert.
- Der Observer überprüft und denkt mit.
- Es wird regelmässig abgewechselt.

- Überprüfen heisst hinterfragen: Machen wir das Richtige? Machen wir es richtig? Ist das Design angemessen? Sind die Testfälle sinnvoll? …

Pair Programming: Vorteile

- Höhere Disziplin. Paare entwickeln viel eher an der richtigen Stelle und machen kürzere Pausen.
- Besserer Code. Beim Pair Programming entwickelt man sich weniger leicht in Sackgassen und erreicht so eine höhere Qualität.
- Belastbarerer Flow. Pair Programming führt zwar zu einer anderen Art von Flow, ermöglicht diesen aber eher als der konventionelle Ansatz: Ein Programmierer kann seinen Partner jederzeit nach dem aktuellen Stand fragen und dort anknüpfen. Unterbrechungen werden auf diese Art besser abgewehrt.
- Höhere Moral. Pair Programming ist oft spannender und interessanter als alleine zu arbeiten.
- Collective Code Ownership. Wenn das gesamte Projektteam mit der Methode Pair Programming arbeitet und die jeweiligen Partner oft wechseln, erlangen alle Wissen über die gesamte Codebasis.
- Mentoring. Jeder hat Wissen, das andere nicht haben. Pair Programming ist eine bequeme Möglichkeit, dieses Wissen zu verteilen.
- Teambildung. Die Leute lernen sich gegenseitig schneller kennen, wodurch die Zusammenarbeit verbessert werden kann.
- Weniger Unterbrechungen. Paare werden seltener unterbrochen als jemand, der alleine arbeitet.

http://de.wikipedia.org/wiki/Paarprogrammierung


WE'RE GOING TO TRY SOMETHING CALLED EXTREME PROGRAM-MING.
www.dilbert.com scottadams@aol.com
FIRST, PICK A PARTNER. THE TWO OF YOU WILL WORK AT ONE COMPUTER FOR FORTY HOURS A WEEK.
1/9/03 © 2002 United Feature Syndicate, Inc.
THE NEW SYSTEM IS A MINUTE OLD AND I ALREADY HATE EVERYONE.