Objektive Code-(Un-)Lesbarkeit

Zwei empirische Ansätze

Denis Washington

```
console.log([[]+1/!1][1^1][1>>1]+({}+[])[1<<1^11>>1]+([]+!!-
 [])[1<<1]+[/~/+{}][+!1][-~1<<1]+/\[[^1]+\]/[([]+![])[1<<1<<
 1]+(/|/[(1+{})[1+11>>>1]+[[]+{}][+!1][1]+([]+1/[])[1<<1>>1]
 +([1<1]+[])[1+11>>>1+1]+[[!!1]+1][+[]][1-1]+([]+!!/!/)[1|1]
 +(/1/[1]+[])[!1%1]+(-{}+{})[-1+1e1-1]+(1+[!!1])[1]+([]+1+{}
 )[1<<1]+[!!/!!/+[]][+[]][1&1]]+/=/)[1e1+(1<<1|1)+(([]+/-/[(
 !!1+[])[1>>1]+(!!1+[])[1<<1^1]+(!1+[])[1|1<<1]+(!!1+[])[1^1
 ]])[1^1]==+!1)]+(!![]+{})[1|1<<1]+[1+{}+1][!1+!1][(11>>1)+1
 ]]((([]+/-/[(!!1+[])[1>>1]+(!!1+[])[1<<1^1]+(!1+[])[1|1<<1]+
 (!!1+[])[1^1]]))[1&.1][11>>>1]+([,][~1]+[])[1-~1]+[[]+{}][!
 1.1%1][11111.1%11.1*111e11|!1]+(/1/+1/[1<1][1%1])[1^11]+[[]
,[]+{}][1<<1>>>1][1||1]+(/[<+>]/[1&1|1]+[1.1])[1/11.1&1.11])
```

```
console.log("I love you")
```

Meistens sind die Dinge nicht so klar

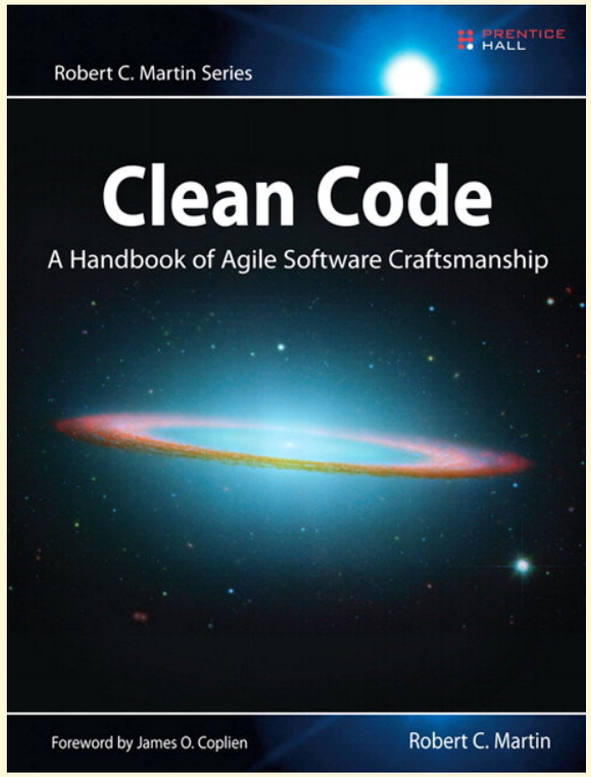
- Uneinigkeit darüber, was lesbar und verständlich ist
  - Formatierung
  - Kompakte vs. sprechende Namen
  - Lokale Variablen vs. Hilfsfunktionen
  - …
- Empfehlungen üblicherweise nicht empirisch belegt
  - Bücher wie *Clean Code* basieren auf Meinungen und anekdotischen Erfahrungen

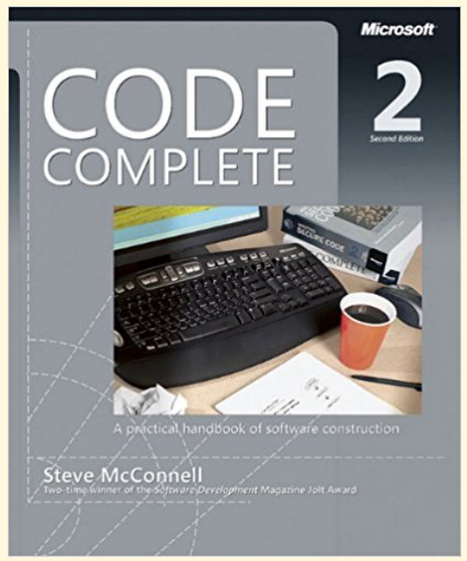
Brace placement	Style

`while (x == y) {`
`  something();`
`  somethingelse();`
`}`
	K&R and variants:
	1TBS, Stroustrup, Linu

`while (x == y)`
`{`
`  something();`
`  somethingelse();`
`}`
	Allman

`while (x == y)`
`  {`
`  something();`
`  somethingelse();`
`  }`
	GNU





Was, wenn Lesbarkeit messbar wäre?

- Man könnte Verständlichkeits- von Geschmacksfragen trennen
- Man könnte schwer lesbare Module in Codebasen identifizieren
- **Wir schauen uns zwei Arbeiten zum Thema an:**
  - Weimer & Buse:
    „Learning a Metric for Code Readability“
  - Gopstein et al.:
    „Understanding Misunderstandings in Source Code“

Westley R. Weimer & Raymond P.L. Buse

„Learning a Metric for Code Readability“

Überblick

Ziel:

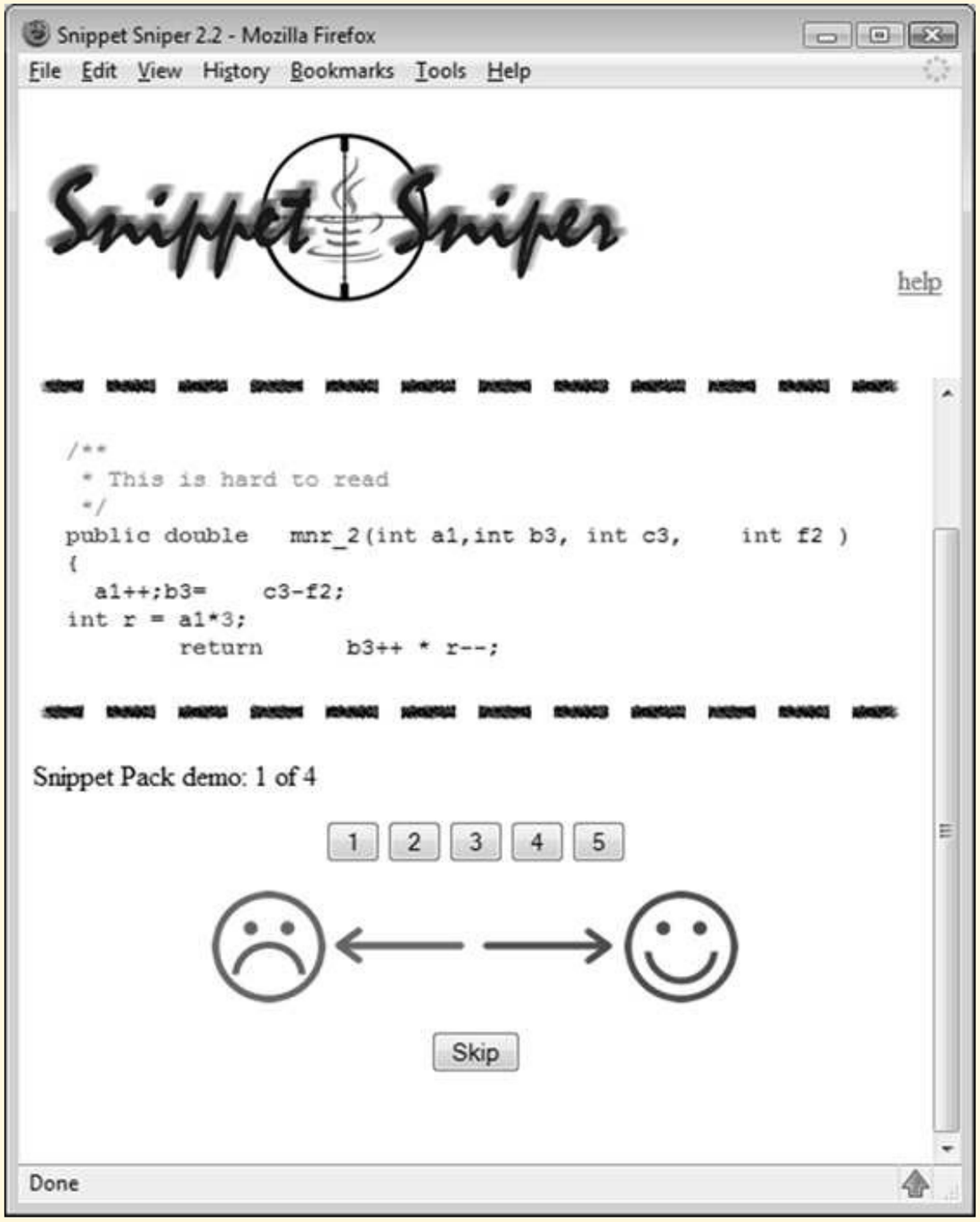
Automatisierte Metrik von Codelesbarkeit

Methodik:

- Umfrage zur subjektiven Lesbarkeit von Code-Snippets
- Ableitung einer automatisierten Lesbarkeitsmetrik mittels Machine Learning
- Untersuchung des Zusammenhangs zwischen der Metrik und *Softwarequalität* (und Code-Komplexität)

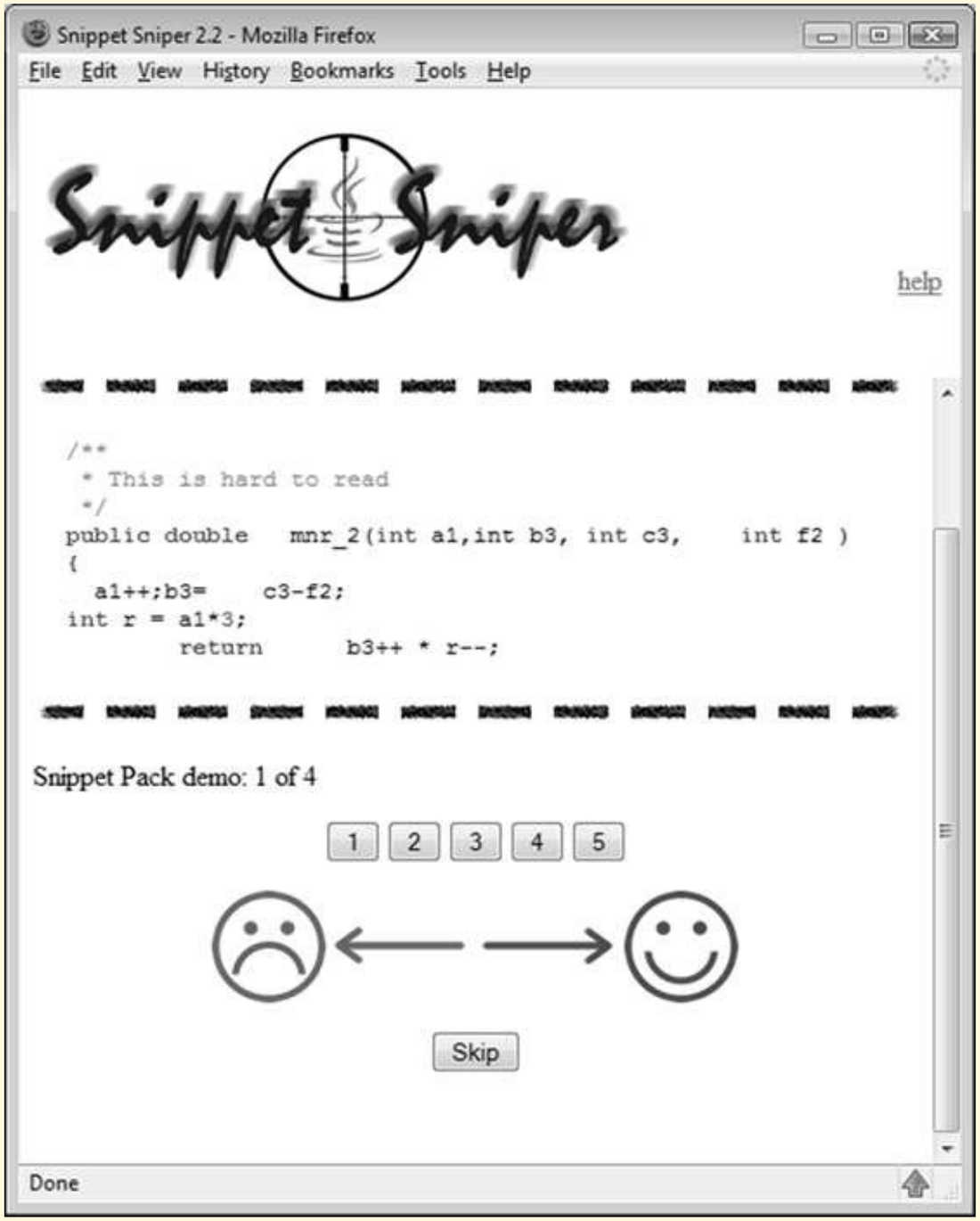
Umfrage

- 120 Probanden
  - University of Virginia
  - 17/63/30 in Studienjahr 1/2/3+
  - 10 Absolventen
- Bewertung von 100 Snippets hinsichtlich Lesbarkeit (1-5)
  - Durschnittlich 7,7 Zeilen Java
  - Syntax-Highlighting
  - Immer gleiche Reihenfolge
  - Lesbarkeit bewusst schwammig definiert → subjektive Einschätzung



Umfrage (2)

- Snippets aus 5 Open-Source-Projekten
  - Hibernate, jFreeChart, FreeCol, jEdit, JUnit
- Auswahlkriterien:
  - Logisch abgeschlossen
    → ohne Kontext verständlich
  - Genau 3 einfache Anweisungen
  - Nicht „trivial“ (z.B. nur Imports)



Umfrage: Ergebnisanalyse

Statistic	Avg—Humans
Cohen's κ	0.18 (p=0.0531)
Weighted κ	0.33 (p=0.0526)
Kendall's τ	0.44 (p=0.0090)
Pearson's r	0.56 (p=0.0075)
Spearman's ρ	0.55 (p=0.0089)

- Gibt es geteiltes Verständnis von Lesbarkeit zwischen Probanden?
  - Cohens κ (absolute Bewertungen gleich?)
  - Linear gewichtetes κ (Snippets gleich gerankt, wenn nach Bewertung sortiert)?
  - Kendall's τ (wie viele Bubble-Sort-Operationen zwischen zwei Rankings?)
  - Pearsons r (lineare Regression)
  - Spearmans ρ (Regression mit beliebiger monoton steigender Funktion)

Umfrage: Analyse (2)

Statistic	Avg—Humans
Cohen's κ	0.18 (p=0.0531)
Weighted κ	0.33 (p=0.0526)
Kendall's τ	0.44 (p=0.0090)
Pearson's r	0.56 (p=0.0075)
Spearman's ρ	0.55 (p=0.0089)

Buse & Weimer:

r oder ρ > 0,5 **„moderate to strong“**
für Umfragen dieser Art*

*L. L. Giventer, *Statistical Analysis in Public Administration*. Jones and Bartlett Publishers, 2007

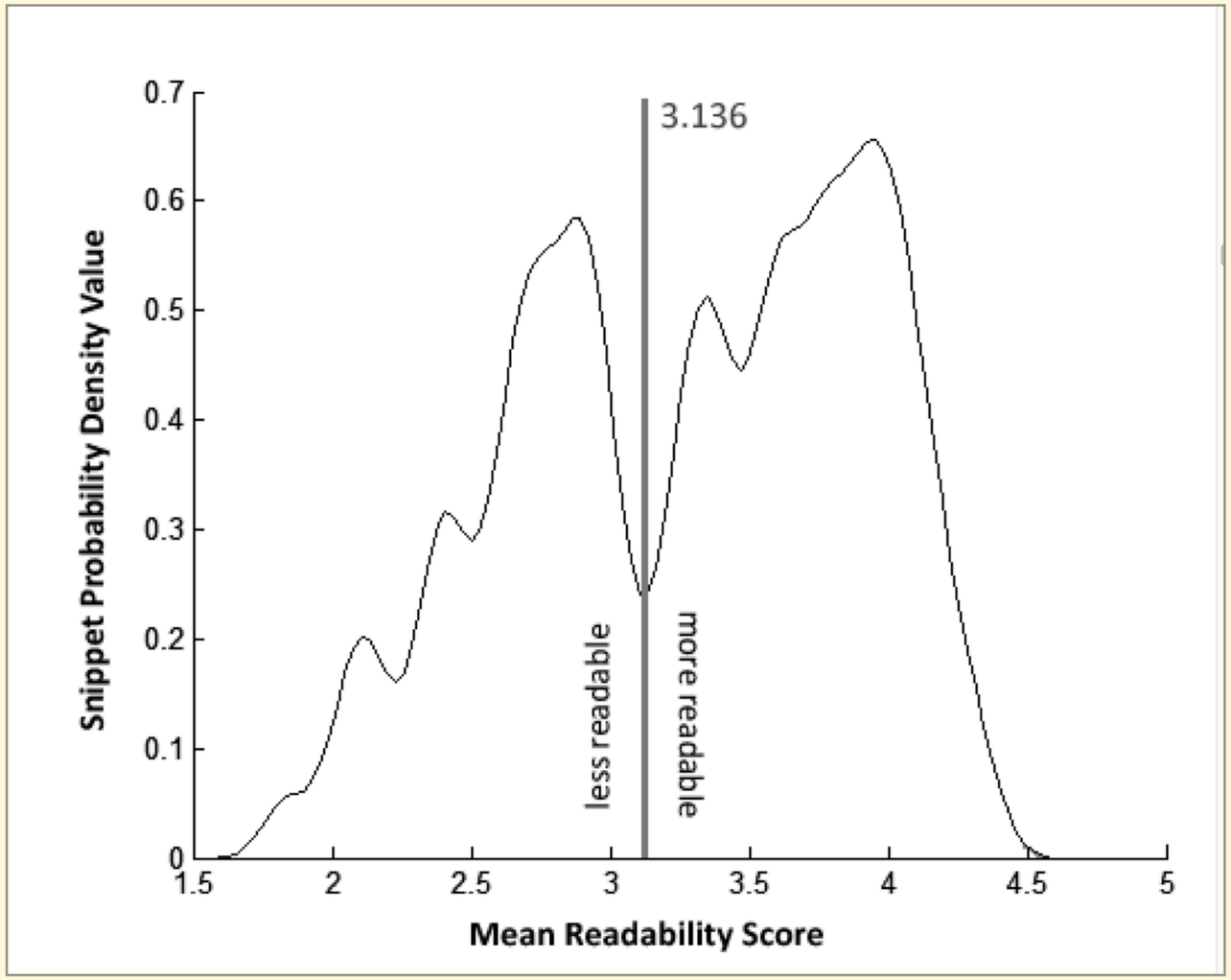
Lesbarkeitsmodell

- Snippets anhand von 19 Eigenschaften (**Features**) charakterisiert
  - Einfach messbar
  - Einfluss auf Lesbarkeit vermutet
  - **#** = Anzahl
  - **Avg.** = Durchschnitt pro Zeile
  - **Max.** = Maximum über Zeilen

Avg.	Max.	Feature Name
✓	✓	line length (# characters)
✓	✓	# identifiers
✓	✓	identifier length
✓	✓	indentation (preceding whitespace)
✓	✓	# keywords
✓	✓	# numbers
✓		# comments
✓		# periods
✓		# commas
✓		# spaces
✓		# parenthesis
✓		# arithmetic operators
✓		# comparison operators
✓		# assignments (=)
✓		# branches (if)
✓		# loops (for, while)
✓		# blank lines
	✓	# occurrences of any single character
	✓	# occurrences of any single identifier

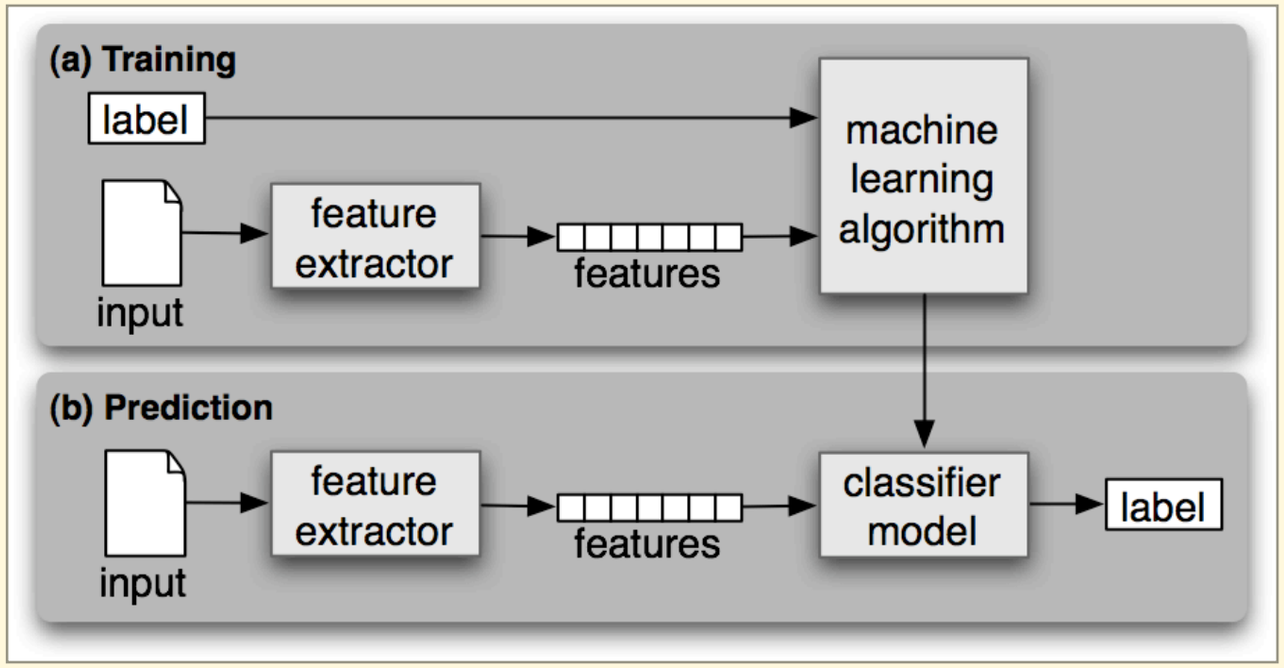
Lesbarkeitsmodell (2)

- Snippets als „lesbar“ oder „weniger lesbar“ klassifiziert
  - Durchschnittliche Bewertung > 3,136 → „lesbar“
  - Abgeleitet aus bimodaler Verteilung der Ergebnisse



Lesbarkeitsmodell (3)

- Ein **Machine-Learning**-Modell trainiert
  - Input = Snippets
  - Features = Snippet-Features
  - Labels = lesbar / unlesbar
  - 90% der Snippets zum Training, 10% zur Validierung



- Mehrere Algorithmen, u.a.:
  - Neuronales Netzwerk
  - Bayes (bedingte Wahrscheinlichkeiten)
  - Voting Feature Interval

Lesbarkeitsmodell: Ergebnisse

- Modell **75 - 80% akkurat** (nach F-Maß)
  - Recall: % Probanden sagen „lesbar“ → Modell sagt „lesbar“
  - Precision: % Modell sagt „lesbar“ → Probanden sagen „lesbar“
  - Recall und Precision hier gleich gewichtet (F_1-Maß)

$$F_1 = 2 \cdot \frac{1}{\frac{1}{\mathrm{recall}} + \frac{1}{\mathrm{precision}}} = 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}}.$$

Lesbarkeitsmodell: Ergebnisse (2)

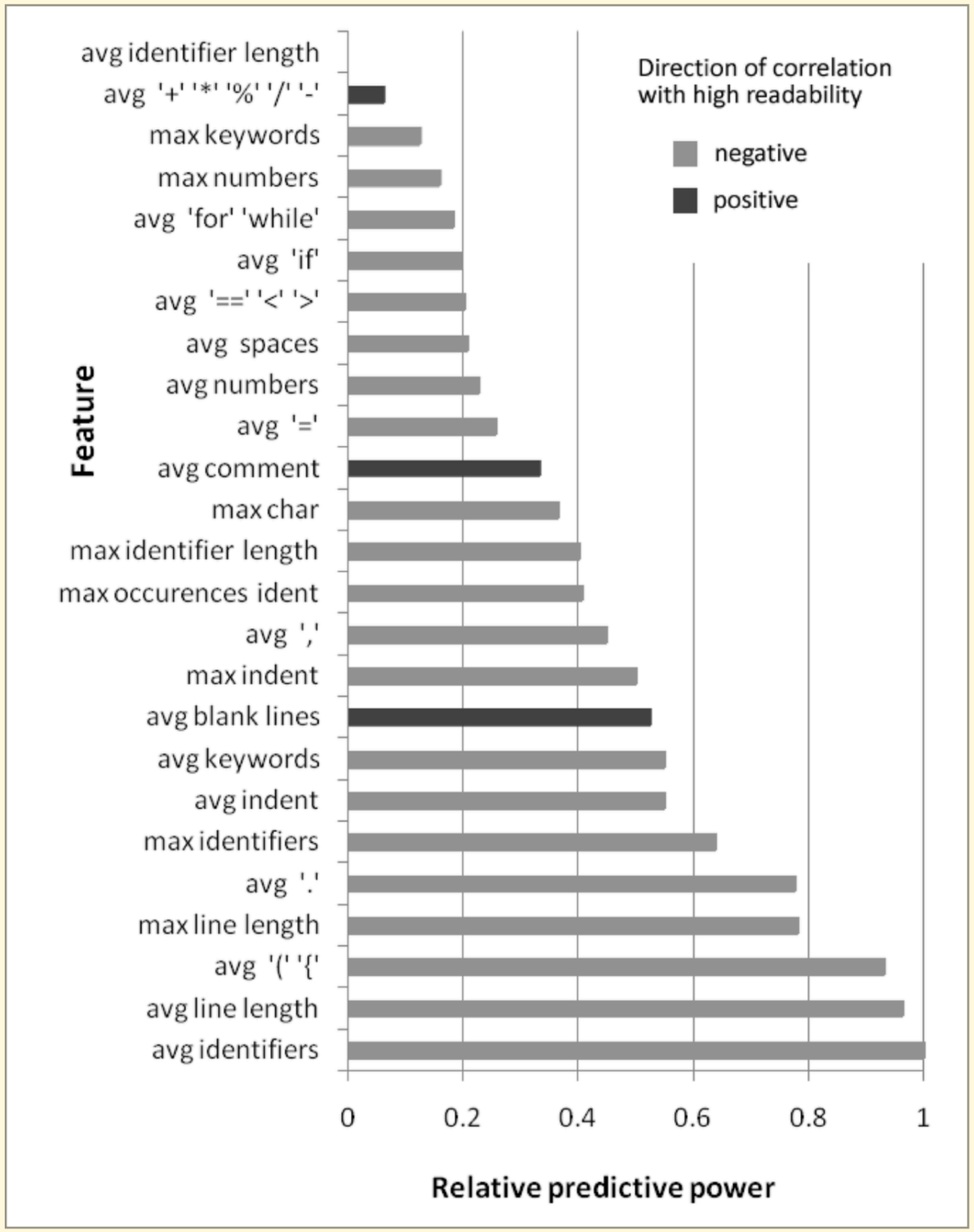
Statistic	Avg—Humans	Avg—Model
Cohen's κ	0.18 (p=0.0531)	0.07 (p=0.0308)
Weighted κ	0.33 (p=0.0526)	0.27 (p=0.0526)
Kendall's τ	0.44 (p=0.0090)	0.53 (p<0.0001)
Pearson's r	0.56 (p=0.0075)	0.63 (p<0.0001)
Spearman's ρ	0.55 (p=0.0089)	0.71 (p<0.0001)

Modell **stärker korreliert** mit Durchschnitt als Probanden

Ausgabe des Bayes-Klassifizierers (= bedingte Wahrscheinlichkeit für „lesbar“) auf Bewertungsskala übertragen

→ Modell gibt **repräsentativere Bewertung** ab als einzelner Proband

Lesbarkeitsmodell: Ergebnisse (3)



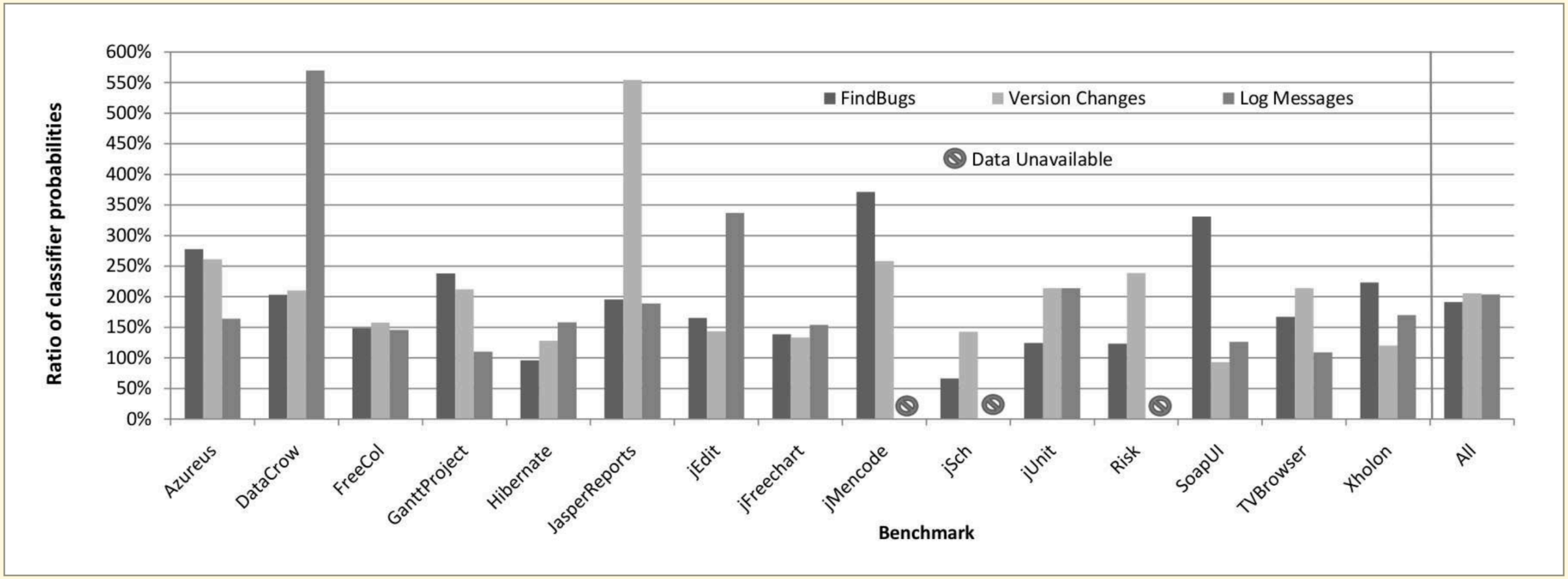
- Einfluss der einzelnen Features untersucht
  - Training mit jeweils nur einem Feature, F-Maße verglichen
- Einfluss der gewählten Features sehr verschieden
  - Einfluss groß u.a. bei Zeilenlänge, Bezeichner-Anzahl und Klammern-Anzahl
  - Einfluss klein u.a. bei Bezeichnerlänge und Operatoren-Anzahl
  - Stärkster positiver Einfluss bei Anzahl Leerzeilen

Korrelation zu Softwarequalität

- Modell auf 15 OSS-Projekte angewandt
  - Methoden einzeln klassifiziert
- Drei Indikatoren für Qualitätsdefizite
  - FindBugs: Wurde für die Methode ein Problem gemeldet?
  - Code Churn: Wurde die Methode zwischen den letzten zwei Releases geändert?
  - Versionskontrolle: Wurde laut Commit-Kommentar ein Defekt in der Methode behoben? (Letzte 500 Commits)

Project Name	KLOC	Maturity	Description
Azureus: Vuze 4.0.0.4	651	5	Internet File Sharing
JasperReports 2.04	269	6	Dynamic Content
Hibernate* 2.1.8	189	6	Database
jFreeChart* 1.0.9	181	5	Data representation
FreeCol* 0.7.3	167	3	Game
TV Browser 2.7.2	162	5	TV Guide
jEdit* 4.2	140	5	Text editor
Gantt Project 3.0	130	5	Scheduling
SoapUI 2.0.1	98	6	Web services
Data Crow 3.4.5	81	5	Data Management
Xholon 0.7	61	4	Simulation
Risk 1.0.9.2	34	4	Game
JSch 0.1.37	18	3	Security
jUnit* 4.4	7	5	Software development
jMencode 0.64	7	3	Video encoding

Softwarequalität: Ergebnisse



Indikatoren **signifikant häufiger** in „weniger lesbaren“ Methoden
(meist mindestens zweimal so häufig)

→ Zusammenhang zwischen
schlechter Lesbarkeit und **Qualitätsproblemen**

Diskussion

- Lesbarkeitsmodell ist **deskriptiv**, nicht **normativ**
  - Keine „Regeln“ für lesbaren Code ableitbar
  - Aber Signifikanzen der Features geben Hinweise für Folgestudien (z.B. geringe Relevanz von Kommentaren)
- Gefahren für (interne) Validität
  - Studenten als Probanden (andere Ergebnisse bei erfahrenen Entwicklern?)
  - Übungs-/Erschöpfungseffekte wegen immer gleicher Snippet-Reihenfolge (allerdings keine wesentlichen Tendenzen über Snippets hinweg gefunden)

Dan Gopstein et al.

„Understanding Misunderstandings in Source Code“

Überblick

Ziel:

Identifizierung minimaler verwirrender Muster in C-Code
(**„atoms of confusion“**, kurz **Atome**)

Methodik:

- Extraktion von Snippets und kleinen Programmen mit vermuteten Atomen aus Obfuscated-Code-Wettbewerben (= „verschleiertem“ Code)
- Validierung der Atome mittels Code-Verständnis-Experimenten
- Vergleich identifizierter Atome mit Code-Style-Guides

Atom Existence Experiment

- 73 Probanden
  - Hauptsächlich US-Studenten
  - ≥ 3 Monate C/C++-Erfahrung (Selbstauskunft)
- Zu C-Snippets nach Endausgabe gefragt
  - Ergebnis richtig = Code verstanden
  - Insgesamt 57 Snippetpaare, jeweils „verschleiert“ und „bereinigt“ (= 114 Snippets)
  - Pro Proband ⅔ der Paare abgefragt (mind. 11 Fragen zwischen zwei Snippets desselben Paares)

Design: Randomized Partial Counterbalanced
Sample: 73 programmers with ≥3 months C/C++ experience.
Control: Tiny program (~8 lines) containing a single atom.
Treatment: A version of the control code transformed to remove the atom of confusion.
Null Hypothesis H_0: Code from both control and treatment groups can be hand-evaluated with equal accuracy.
Alternative Hypothesis H_a: The existence of an atom of confusion causes more errors than other code in hand-evaluated outputs.

Figure 1: Summary of the Atom Existence Experiment

```
void main() {
  char V1 = 2["qwert"];

  printf("%c\n", V1);
}
```

(a) obfuscated

```
void main() {
  char V1 = "zxcvb"[4];

  printf("%c\n", V1);
}
```

(b) clarified

Figure 2: Example *Reversed Subscripts* snippet pair.

Atom Existence Experiment (2)

- Normalisierte Snippets aus International Obfuscated C Code Contest (**IOCCC**)
  - Formatiert
  - Kontextbereinigt (Variablennamen, Konstanten usw.)
  - Durchschnittlich 8 Zeilen
- Ein Atom-Kandidat pro Snippet
  - z.B. „Reversed Subscripts“
  - Jede vermutete Interpretation ein anderes Ergebnis
    → Rückschluss auf Missverständnis

```
#include <stdio.h>
#include <math.h>
#define E return
#define S for
char*J="LJFFF%7544x^H^XXHZZXHZ]]2#( #@@DA#(.@@%(0CAaIqDCI$IDEH%P@T@qL%PEaIpBJCA\
I%KBPBEP%CBPEaIqBAI%CAaIqBqDAI%U@PE%AAaIqBcDAI%ACaIaCqDCI%(aHCcIpBBH%E@aIqBAI%A\
AaIqB%AAaIqBEH%AAPBaIqB%PCDHxL%H@hIcBBI%E@qJBH#C@@D%aIBI@D%E@QB2P#E@'C@qJBHqJBH\
%C@qJBH%AAaIqBAI%C@cJ%"    "cJ"    "CH%C@qJ%aIqB1I%PCDI`I%BAaICH%KH+@'JH+@KP*@%S@\
3P%H@ABhIaBBI%P@S@PC#",      *j    ,*e;typedef float x;x U(x a){E a<0?0:a>1?1:a; }
typedef struct{x c,a,t;       }    y;y W={1,1,1},Z={0,0,0},B[99],P,C,M,N,K,p,s,d,h
;y G(x t,x a,x c){K.c=t    ;       K.t=c; K.a=a;E K;}int T=-1,b=0,r,F=-111,(*m)(i\
nt)=putchar,X=40,z=5,o,    a,      c,t=0    ,n,R;y A(y a,y b,x c){E G(a.c+b.c*c,a.a
+c*b.a,b.t*c+a.t);}x H=    .5,Y    =.66      ,I,l=0,q,w,u,i,g;x O(y a,y b){E q=a.t*
b.t+b.c*a.c+a.a*b.a;}x Q(){E A(P,M,T         ),O(K,K)<I?C=M,I=q:0;}y V(y a){E A(Z,
a,pow(O(a,a),-H));}x D(y p){S(I=X,P           =p,b=T; M=B[++b],p=B[M.c+=8.8-l*.45,
++b],b<=r;Q())M=p.t?q =M_PI*H,w=atan2(    P.a-M.a,P.c-M.c)  /q,o=p.c-2,a=p.a+1,t=
o+a,w=q*(w>t+H*a?o:     w>t?t:w<o-H*a?t    :w<o?o:w),A(       M,G(cos(w),sin(w),0),
1):A(M,p,U(O(A(P,M,T)     ,p)/O(p,p)));    M=P;M.a=-          .9;o=P.c/8+8;o^=a=P.t
/8+8; M=Q         ()?o&1     ?G(Y,0,0):W    :G(Y,Y,1);E        sqrt         (I)-.45;}
int main(    int     L,char     **k){ S(e    =L>1?1[z=     0,    k]:J    ;*e    &&l<24 ;
++e)S(o=a    =0,j    =J+9;(c=     *++j)&&    !(o&&c<     X&&(q=l+=w)    );o    ?o=*j++/
32,b++[B]    =G(q    +=*j/8&3,*     j&7,0    ),B[r     =b++]=G((c/8&          3)*( o<2?
T:1), (c&    7)+     1e-4,o>2),1:     (o    =(a     =(c-=X)<0?w=c+6    ,t=    a+1:c?(t
?0:m(c),a         ):*++j)==((*e|32          )     ^z)&&1[j]-X));S(z    =3*(    L<3);++
F<110;)S(L=-301;p=Z,++L<300;m(                p.c),m(p.a),m(p.t))S(c=T;++c<=z;)S(h
=G(-4,4.6,29),d=V(A(A(A(Z,V(G(5,0             ,2)),L+L+c/2),V(G(2,-73,0)),F+F+c%2),G
(30.75,-6,-75),20)),g=R=255-(n=z)*64;     R*n+R;g*=H){S(u=i=R=0;!R&&94>(u+=i=D(h=
A(h,d,i)));R=i<.01);S(N=V(A(P,C,                   T)),q=d.t*d.t,s=M,u=1;++i<6*R;u-=
U(i/3-D(A(h,N,i/3)))/pow(                               2,i));s=R?i=pow(U(O(N,V(A(
M=V(G(T,1,2)),d,T))))                                      ,X),p=A(p,W,g*i),u*=U(
O(N,M))*H*Y+Y,g*=                                             n--?Y-Y*i:1-i,s:G(
q,q,1); p=A(p,s                                                 ,g*u);h=A(h,N,.1
);d=A(d,N,-2*O                                                    (d,N));}E 0;}
```

Atom Existence: Kandidaten

Atom Name	Description	Atom Example	Transformed
Change of Literal Encoding	All numbers are stored as binary, but for convenience we represent numbers in decimal, hex, or octal. Depending on the circumstance, certain representations are more understandable.	`printf("%d",013)`	`printf("%d",11)`

Preprocessor in Statement	The preprocessor replaces directives with whitespace. Consequently, preprocessor directives may be placed inside a statement. Since the preprocessor directive and the source code are compiled in different phases, they are processed independently.	`int V1 = 1`
		`#define M1 1`
		`+ 1;`
			`#define M1 1`
			`int V1 = 1 + 1;`

Macro Operator Precedence	Macro references are impossible to distinguish from other identifiers and can act in ways that variables and functions can not, such as lax parameter binding.	`#define M1 64-1` `2*M1`	`2*64-1`
Assignment as Value	The assignment expression changes the state of the program when it executes, however it also returns a value. When reading an assignment expression people may forget one of the two effects of the expression.	`V1 = V2 = 3;`	`V2 = 3;` `V1 = V2;`
Logic as Control Flow	Traditionally, the && and || operators are used for logical conjunction and disjunction, respectively. Due to short-circuiting, they can also be used for conditional execution.	`V1 && F2();`	`if (V1) F2();`
Post-Increment / Decrement	The post-increment and decrement operators change the value of their operands by 1 and return the original value. Confusion arises because the value of the expression is different from the value of the variable.	`V1 = V2++;`	`V1 = V2;` `V2 += 1;`
Type Conversion	The C compiler will often implicitly convert types when there is a mismatch. Sometimes this conversion also results in a different outcome than a reader expects.	`(double)(3/2)`	`trunc(3.0/2.0)`
Reversed Subscripts	Arrays can be indexed using the subscript operator, but underneath “E1[E2] is identical to (*((E1)+(E2)))” [1]. Since addition is commutative, so too is the subscript operator.	`1["abc"]`	`"abc"[1]`
Conditional Operator	The conditional operator is the only ternary operator in C, and functions similarly to an if/else block. However, it is an expression for which the value is that of the executed branch.	`V2 = (V1==3)?2:V2`	`if (V1 == 3)` `V2 = 2;`
Operator Precedence	C has nearly 50 operators each in one of 15 precedence classes with either right or left associativity. Most programmers know only a functional subset of these rules.	`0 && 1 || 2`	`(0 && 1) || 2`
Comma Operator	The comma operator is used to sequence series of computations. Whether due to its eccentricity, or its odd precedence, the comma operator is commonly misinterpreted.	`V3 = (V1 += 1, V1)`	`V1 += 1;` `V3 = V1;`
Pre-Increment / Decrement	Similar to post-increment/decrement, these operators change a variable’s value by one. In contrast to the other operators, pre-increment/decrement first update the variable then return the new value.	`V1 = ++V2;`	`V2 += 1;` `V1 = V2;`
Implicit Predicate	The semantics of a predicate are easily mistaken. The most common example happens when assignment is used inside a predicate or when a success state is represented as 0.	`if (4 % 2)`	`if (4 % 2 != 0)`
Repurposed Variables	When a variable is used in different roles in a program, its current meaning can be difficult to follow.	`argc = 7;`	`int V1 = 7;`
Omitted Curly Braces	When control statements omit curly braces, the scope of their influence can be difficult to discern.	`if(V) F(); G();`	`if(V){F();}G();`
	Candidates which eventually failed to meet statistical significance		
Dead, Unreachable, Repeated	Redundant code is executed to no functional effect, but its appearance may imply that meaningful changes are being made.	`V1 = 1;` `V1 = 2;`	`V1 = 2;`
Arithmetic as Logic	Arithmetic operators are capable of mimicking any predicate formulated with logical operators. Arithmetic, however, implies a non-boolean range, which may be confusing to a reader.	`(V1-3) * (V2-4)`	`V1!=3 && V2!=4`
Pointer Arithmetic	Pointers admit several operations like integer addition/subtraction, but, in many cases, these operations are interpreted by the reader to effect the target data instead of the pointer data.	`"abcdef"+3`	`&"abcdef"[3]`
Constant Variables	Constant variables are a layer of abstraction that emphasize a concept rather than a particular value itself. When trying to hand evaluate a piece of code having a layer of indirection can obscure the value of the data.	`int V1 = 5;` `printf("%d", V1);`	`printf("%d", 5);`

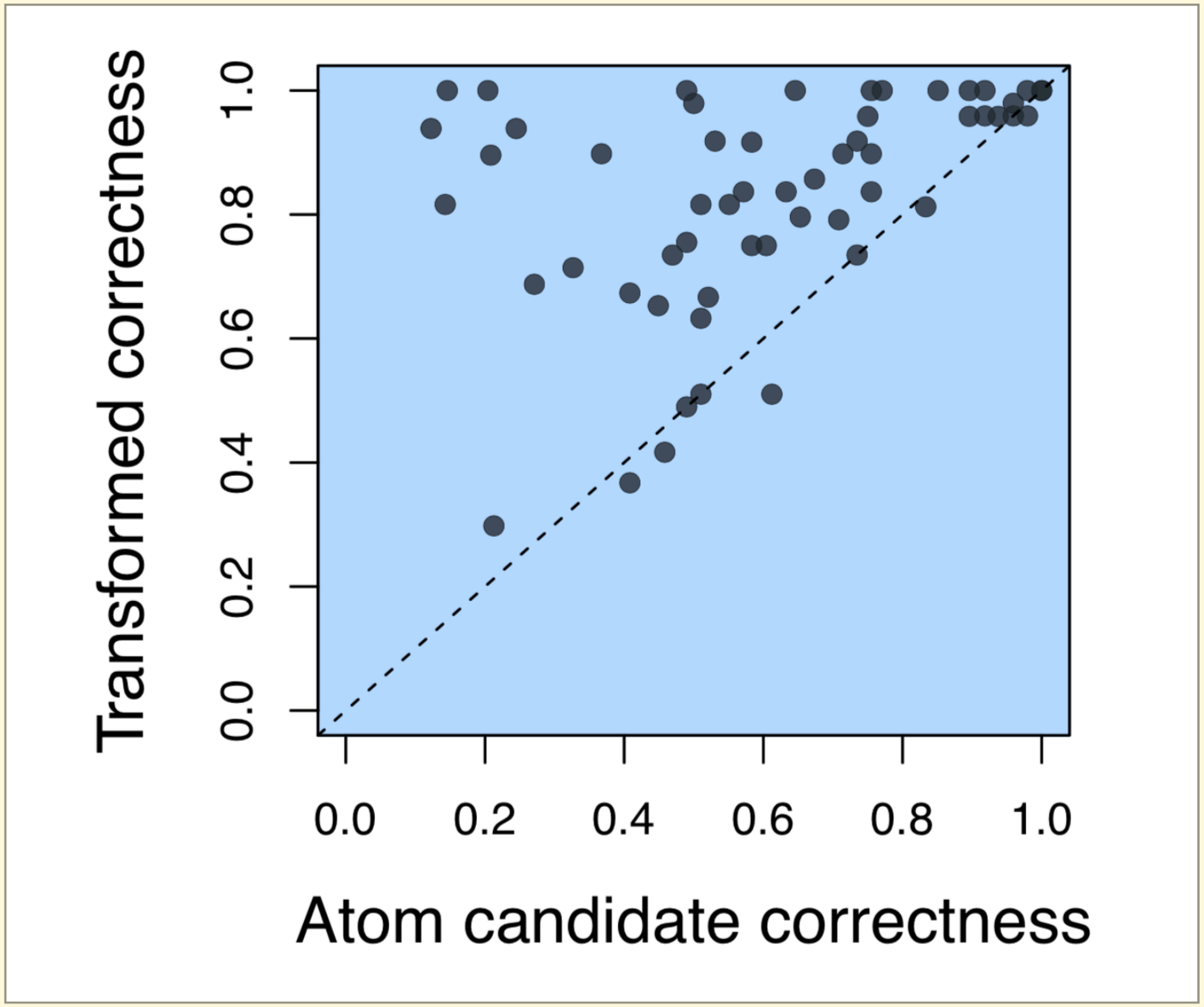
Atom Existence: Analyse

	Test 2 positive	Test 2 negative	Row total
Test 1 positive	*a*	*b*	*a* + *b*
Test 1 negative	*c*	*d*	*c* + *d*
Column total	*a* + *c*	*b* + *d*	*n*

$$\chi^2 = \frac{(b-c)^2}{b+c}$$

- Bewertung der Atom-Kandidaten mit **McNemars Test**
  - Test 1 = verschleiertes Snippet
  - Test 2 = bereinigtes Snippet
  - Positiv = Proband wurde verwirrt (falsches Ergebnis)
  - Negativ = richtiges Ergebnis
  - Intuitiv: Wenn b (verschleiert verwirrend + bereinigt verständlich) wesentlich höher als c (verschleiert verständlich + bereinigt verwirrend), zeigt das eine Korrelation zwischen Atom und Verwirrung

Atom Existence: Ergebnisse



Snippets mit Atom-Kandidatin signifikant verwirrender

Analyse mit McNemars Test. $p = 3{,}68^{-78}$, $\varphi = 0{,}36$

Atom Existence: Ergebnisse (2)

Table 2: Statistical properties of atom candidates

	Dispersion		Δ	Effect	
Atom Name	Obfs'd	Clar'd	Correct	Size	p-value
Change of Literal Encoding	1.65	0.75	0.60	0.63	**2.93e-14**
Preprocessor in Statement	1.01	0.46	0.47	0.54	**8.53e-11**
Macro Operator Precedence	0.50	0.32	0.36	0.53	**1.77e-07**
Assignment as Value	0.99	0.45	0.42	0.52	**3.78e-10**
Logic as Control Flow	1.58	0.90	0.41	0.48	**5.62e-09**
Post-Increment/Decrement	1.31	0.52	0.34	0.45	**6.98e-08**
Type Conversion	0.85	0.50	0.29	0.42	**5.17e-07**
Reversed Subscripts	1.71	0.93	0.23	0.40	**1.52e-06**
Conditional Operator	0.91	0.07	0.23	0.36	**1.74e-05**
Operator Precedence	0.59	0.30	0.14	0.33	**5.90e-05**
Comma Operator	2.02	0.76	0.23	0.30	**2.46e-04**
Pre-Increment/Decrement	0.95	0.82	0.16	0.28	**6.89e-04**
Implicit Predicate	0.61	0.34	0.10	0.24	**4.27e-03**
Repurposed Variables	1.78	1.50	0.12	0.22	**6.66e-03**
Omitted Curly Braces	1.22	0.94	0.14	0.22	**8.64e-03**
Dead, Unreachable, Repeated	0.19	0.06	0.03	0.16	0.059
Arithmetic as Logic	0.23	0.15	0.03	0.10	0.248
Pointer Arithmetic	1.54	1.06	0.01	0.03	0.752
Constant Variables	0.28	0.29	0.00	0.00	1.000

15 signifikante Atome identifiziert

Kriterium: $p < 0{,}05$. Effektstärke klein: 0,1; mittel: 0,3; groß: 0,5

Atom Impact Experiment

- 43 Probanden
  - ≥ 6 Monate C/C++-Erfahrung
- Händische Evaluation ganzer C-Programme
  - 14 - 84 Zeilen
  - Aus IOCCC, normalisiert, mit Print-Anweisungen für Variablenwerte
  - Aufgabe: vermutete Print-Ausgaben aufschreiben
  - 4 Programmpaare (verschleiert & bereinigt); eine Variante pro Proband

Design: Randomized Partial Counterbalanced
Sample: 43 programmers with ≥ 6 months C/C++ experience.
Control: Small programs (between 14-84 lines) containing several atoms of confusion.
Treatment: A version of the control code transformed to remove the atoms of confusion.
Null Hypothesis H_0: Code from both control and treatment groups can be hand-evaluated with equal accuracy.
Alternative Hypothesis H_a: Multiple atoms of confusion cause more errors in hand-evaluated outputs than other code.

Figure 4: Summary of the Atom Impact Experiment

```
printf("foo: %i %i\n", V1, V2);
```
→ foo: 23 42

Atom Impact: Analyse

- Bewertung jeder Antwort mit Punktzahl
  - 1 Punkt pro richtiger Ausgabezeile
  - Keine Folgefehler (falsche Variablenwerte nachfolgend als richtig angenommen)

- Leistungen verschleiert vs. bereinigt mit **t-Test** verglichen
  - Frage: Statistisch signifikant?

Atom Impact: Ergebnisse

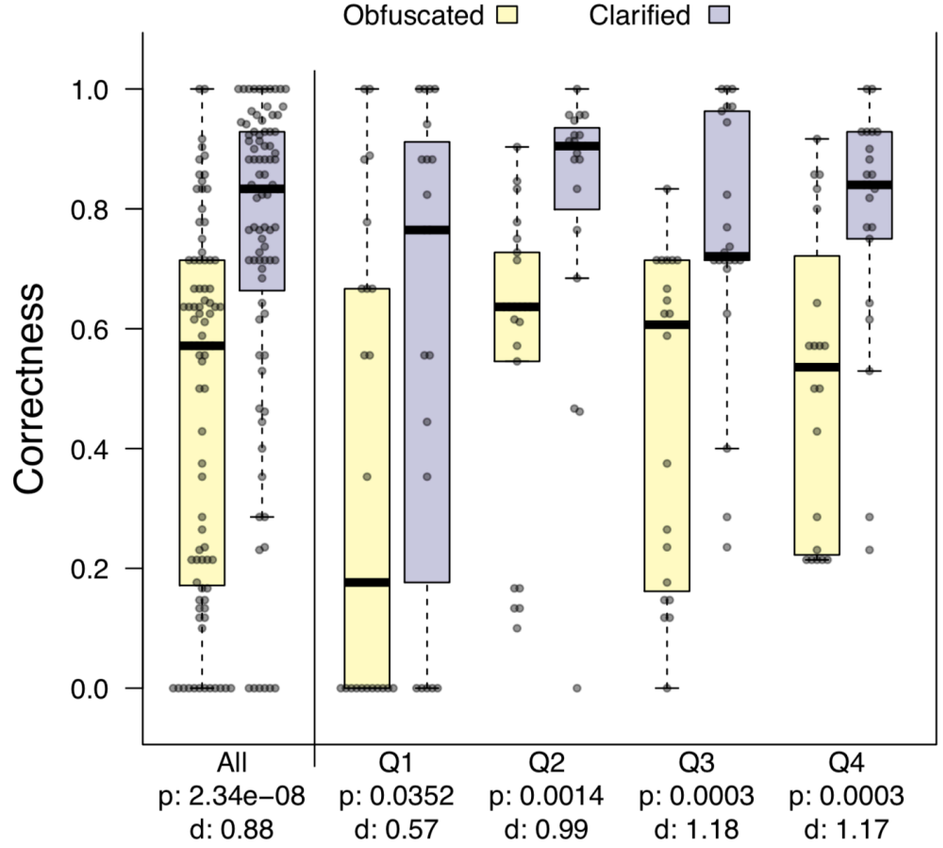


Figure 5: Average score by question type. Rate of correct answers for each program type, and for all obfuscated and clarified programs combined. *p* denotes p-value and *d* denotes Cohen's d effect size.

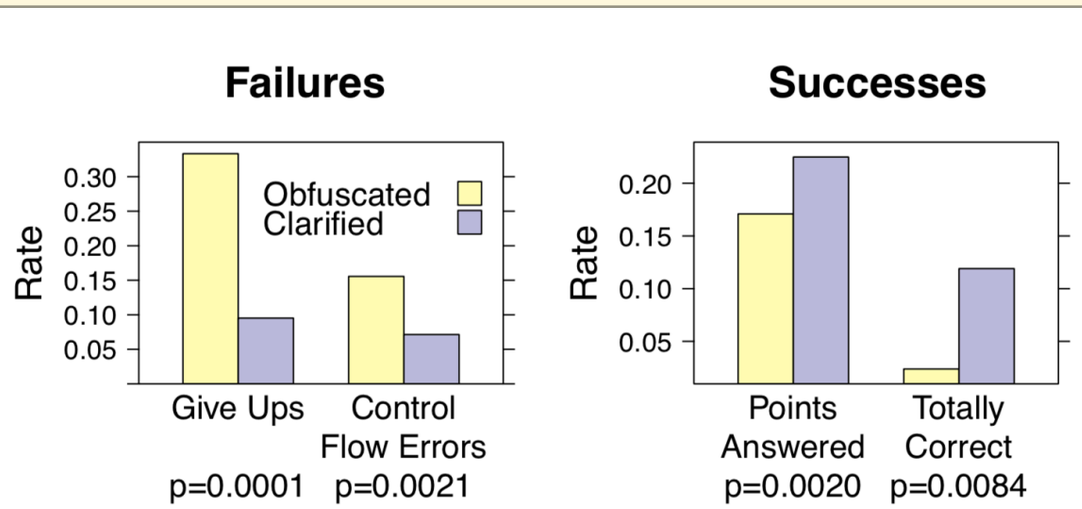


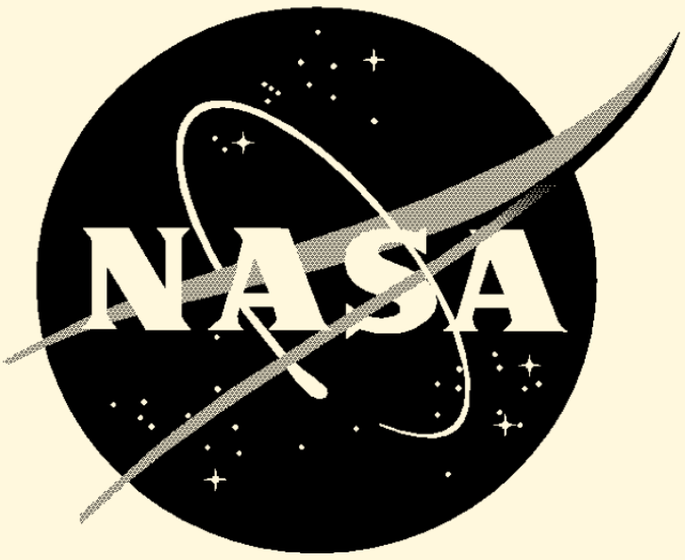
Figure 6: Secondary statistics regarding subjects' behavior

Signifikant schlechtere Leistung bei Programmen mit Atomen

Dreimal öfter aufgegeben und mehr als 90% weniger alles richtig bei Programmen mit Atomen

Vergleich mit Style Guides

- Mehrere Style-Guides nach identifizierten Atomen durchsucht
  - GNU Coding Standards, NASA C Style Guide, …
- 15 von 19 Atomen gefunden
  - *Reversed Subscripts* und *Preprocessor in Statement* nicht genannt
  - 2 andere waren nicht statistisch signifikant
- Einige Empfehlungen widersprechen Atomen
  - z.B. Klammern weglassen bei einzeiligem `if`



Diskussion

- Gefahren für **interne** Validität
  - Bereinigte Programme noch verwirrend (verfälscht möglicherweise Effektstärken der getesteten Atome)
  - Atom-Kandidaten subjektiv ausgewählt (möglicherweise effektstärkere Atome übersehen)
  - „Monate C/C++-Erfahrung“ ungenau zum Messen des Kenntnisstands
- Gefahren für **externe** Validität
  - Probanden sind Studenten
  - Obfucated-C-Code nicht repräsentativ für Produktivcode (aber z.B. hunderte von Atome wie *Preprocessor in Statement* im Linux-Kernel gefunden)
  - Normalisierung erzeugt untypischen Code (kontextfrei, generische Variablennamen usw.)

Vergleich & Fazit

Vergleich

- Untersuchung von Lesbarkeit als Metrik
- Subjektive Messung mit Lesbarkeitsbewertungen
- Nutzung kurzer, normalisierter Code-Snippets aus Produktivcode
- Ergebnis ist beschreibend (Metrik hilft nicht direkt, Code lesbarer zu machen)

- Untersuchung von Hindernissen für Lesbarkeit
- Objektive Messung mit Verständnisfragen
- Nutzung kurzer, normalisierter Code-Snippets und kleiner Programme aus Obfuscated-C-Code-Contest
- Ergebnis ist handlungsweisend (Vermeidung von Atomen erhöht Lesbarkeit)

Kritik

- Sehr schmale Definition von „Lesbarkeit“
  - Lokal (kleine Snippets, kurze Programme)
  - Kontextfrei (z.B. gute Benennung außer Acht gelassen)
- Keine professionellen Entwickler befragt
  - Möglicherweise höhere Toleranz für Code, den weniger unerfahrene Entwickler (Studenten) „schwer lesbar“ nennen würden
  - → Praktische Relevanz von Lesbarkeit wie hier definiert ist unklar

Vielen Dank

Westley R. Weimer and Raymond P.L. Buse. Learning a Metric for Code Readability. In IEEE Transactions on Software Engineering, vol. 36, no. , pp. 546-558, July/August 2010, doi:10.1109/TSE.2009.70

Dan Gopstein, Jake Iannacone, Yu Yan, Lois Anne Delong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. 2017. Understanding Misunderstandings in Source Code. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM. Accepted at ESEC/FSE'17

https://atomsofconfusion.com/