Mensch-Maschine-Kooperation
— Impressionen, Überlegungen & Behauptungen zum „Requirements Engineering“ —

Hinrich E. G. Bonin[1]

Zusammenfassung:
Eine „Maschine“ bestehend aus Hard- und Softwarekomponenten ist ein formal logisches System und trotzdem undurchschaubar. In der Praxis reagiert sie häufig anders als vermutet und gewollt. Abhilfe soll ein solider Konstruktionsprozeß schaffen. Dieser basiert auf einer präzisen, vollständigen Spezifikation der Anforderung, dem Ergebnis der Phase „Requirements Engineering“ (RE).

Das ingenieurmäßige Paradigma unterstellt eine vorgegebene Realität, aufgrund deren Analyse die Anforderungen lösungsneutral spezifizierbar sind. Es verstellt jedoch die Einsicht von der prinzipiellen Undurchschaubarkeit der komplexen „Maschine“ in ihren Wechselwirkungen mit der Realität. Daher ist zum Verstehen, Entwerfen und Gestalten einer nützlichen, verantwortbaren Arbeitsteilung zwischen Menschen und Maschinen eine Ergänzung dieses Paradigmas notwendig.

Der Beitrag skizziert diese zusätzliche Perspektive. Einerseits sind Ursachen/-Wirkungsketten zu verstehen, andererseits ist mit der„Undurchschaubarkeit“ kompetent umzugehen. Als Folge davon nutzen wir Möglichkeiten zur Risikobegrenzung, kalkulieren Sicherheitszuschläge ein und versuchen Komplexität einzusparen.

logisches System

Undurchschaubarkeit

Stichwörter:
Mensch-Maschine-Kooperation, Softwarekonstruktion, Anforderungsanalyse, Requirements Engineering, Undurchschaubarkeit, Komplexitätseinsparung

[1] Prof. Dr. rer. publ. Dipl.-Ing. Dipl.-Wirtsch.-Ing. Hinrich E. G. Bonin, Fachhochschule Nordostniedersachsen, University of Applied Sciences in North-East-Lower-Saxony, Volgershall 1, D-21339 Lüneburg, Bonin’s Web Site: `http://as.fhnon.de/` Speicherort des Orignals auf 193.174.33.147: `D:\bonin\MMK\MMK.tex`

Inhaltsverzeichnis

1 Problemaufriß

Liebe Diplomierte, sehr geehrte Freunde, Förderer und amtlich hier Sitzende, sehr geehrte Damen und Herren!

Kampf mit Computern

Jeder von Ihnen hat vielfältige Erfahrungen mit Computern, sei es beispielsweise der hektische Kampf mit einem Fahrkartenautomaten und/oder die verzweifelten Bemühungen einem Buchungssystem mitzuteilen, dass man nur einen Teil der bestellten Lieferung akzeptiert. Dabei sei es hier gleichgültig, ob es nun eine Intel-CPU mit *Microsoft Windows2000*^{TM} oder eine RISC-CPU mit *IBM AIX*^{TM} und jeweils mit *Baan*^{TM}- oder *SAP*^{TM}-Standardsoftware waren. Wichtig ist, Sie können mit mir mitfühlen, wenn ich aus meiner Erfahrung mit Computern zwei Feststellungen treffe:

1. „Er, der Computer, tut nicht was er soll!“
2. „Er, der Computer, tut Dinge, die wir nicht packen!“

Terminologie der Informatik

Bevor beide Feststellungen vertieft werden, gestatten Sie mir eine kleine Anmerkung zur Terminologie. Sie haben schon bemerkt, Informatiker lieben englische Begriffe und viele Akronyme. Hier wurde ja beispielsweise das Aktronym *RISC* schon erwähnt. Es steht für *Reduced Instruction Set Computing*. Weder die vielen englischen Begriffe noch ihre Kurzform sollten Sie jedoch verwirren. Sie müssen sich nur an folgender Übersetzungsanalogie orientieren. Wenn ich von „Öko-Bio-Umwelt-Entsorgungs- und Recycling-Ingenieur“ spreche, dann übersetzen Sie dies in Ihre gewohnte Alltagssprache, also in „Müllmann“ (↪Abbildung[2] 1 Seite 4).

1.1 „Er, der Computer, tut nicht was er soll!“

Diese kurze aber schwerwiegende Aussage trifft leider viel zu oft zu. Das kleine Beispiel einer Addition in Abbildung 3 Seite 5 soll dies verdeutlichen. Es zeigt, dass man sich auf die gewohnte Wirkung des Pluszeichens (+) offensichtlich nicht verlassen kann. Irgendwie kommen die Zahlen wohl „durcheinander“ (↪Abbildung 2 Seite 5).

1.2 „Er, der Computer, tut Dinge, die wir nicht packen!“

Rekursion

Diese Aussage ist trivial. Niemand bezweifelt heute, dass Computer „Dinge tun können“ zu denen Sie — höflicher gesagt ich — nicht in der Lage sind. Das kleine Beispiel der Berechnung der Fakultät 500 in Abbildung 5 Seite 7 soll dies

[2] Alle Textillustrationen mit *FUZZI* in diesem Beitrag sind von Herrn David Anderson, Berlin, gezeichnet. Sie entstammen dem Band 1 der Reihe „Programmierung komplexer Systeme“ ↪[Bonin91].

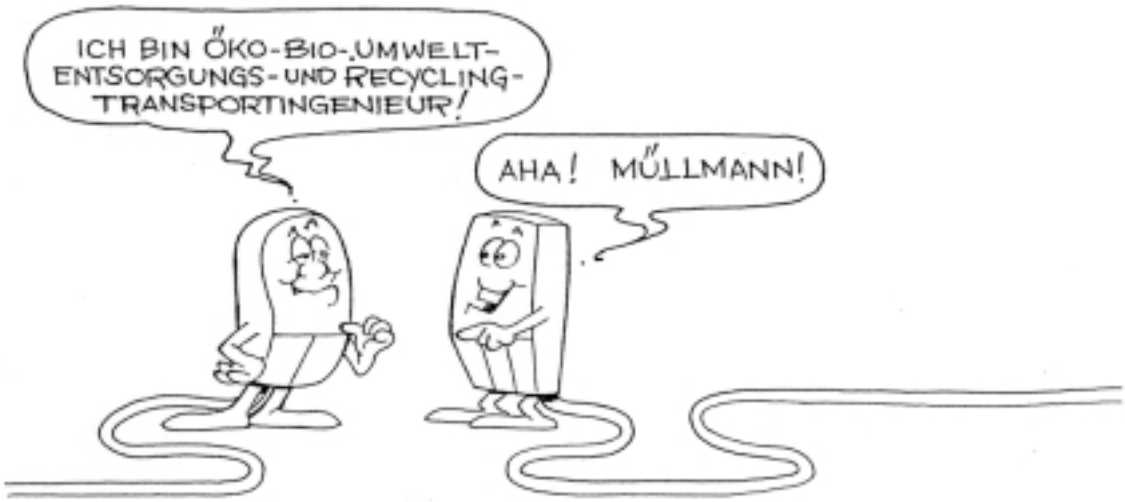


Abbildung 1: $\mathcal{FUZZI}$ übersetzt die Sprache der Informatik.

verdeutlichen. Sie erinnern sich, Fakultät war die unnütze Funktion mit der Sie das *Konzept der Rekursivität* (↪Abbildung 4 Seite 6) lernen sollten. Die Fakultät wurde daher notiert als: $n! = n * (n-1)!$ mit $0! = 1$.

Klar, Sie wissen es, das Problem ist nicht der Computer, sondern der *User*, hier insbesondere so ein Informatik-Fossil wie ich es bin. Ein Altlastinformatiker sollte doch erstmal eine moderne Softwareentwicklungsumgebung verwenden! Ohne eine farbige Dialogoberfläche mit vielen Fenstern und Animation kann man heute einen Computer nicht programmieren. Klar ist aber auch, der Fachbereich Wirtschaft der Fachhochschule Nordostniedersachsen hat Ihren Einwand längst aufgegriffen und setzt jetzt auf die aktuelle X-Technologie[3] mit Java^{TM}[4]. Zur heutigen Pflichtlektüre zählen zum Beispiel „Der Java-Coach“ (↪[Bo1998]) und „Der kleine XMLer“ (↪[Bo2000b]). Beide setzen auf die Handhabung einer bunten Fensterwelt. Die Abbildung 6 Seite 8 zeigt eine XML-Entwicklungsumgebung der IBM. Trotz alledem es bleibt offensichtlich ein Kernproblem:

Die Verständigung mit ihm — aber wie?

[3] Dazu zählen beispielsweise XML (*Extensible Markup Language*), XSL (*Extensible Stylesheet Language*), XLink und XPointer.

[4] Kleine Anmerkung: Scheme gab es sowieso nicht im normalen Lehrprogramm. Da war COBOL als Klassiker angesagt.

Abbildung 2: $\mathcal{FUZZI}$ meistert unzählige Zahlen!

```
$ (define + +) ==> Value: +

$ (+ 3 4) ==> Value: 7

$ (define + -) ==> Value: +

$ (+ 3 4) ==> Value: -1

$
```

Legende: Scheme-Beispiel „Pluszeichen“
Scheme ist das Un*Common List Processing* (LISP); Konzepteure sind MIT-Wissenschaftler; Web-Site: http://www.swiss.ai.mit.edu/projects/scheme/ Zugriff: 12-Mar-2001

Abbildung 3: Subtraktion mit Additionszeichen

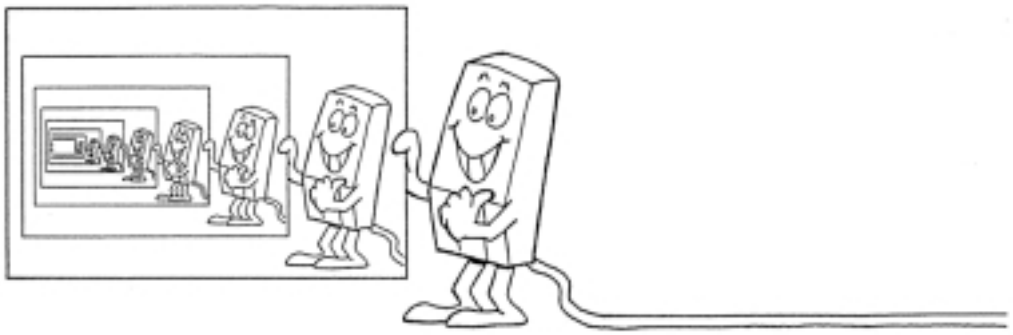


Abbildung 4: *FUZZI* sieht sich rekursiv!

Für diese Verständigung gibt es mindestens zwei intuitive Ansätze:

- **Striktes Kommandodenken!** — Mache erst dies, dann das, dann . . .!
  (Geht schlecht:Man müßte prinzipiell selbst die Lösung kennen!)
- **Pures Vertrauen**, das heißt er wird es schon machen! — Man beschreibt Regel A, Regel B,. . .,Regel X. Er möge diese dann irgendwie ausführen!
  (Geht auch nicht: „Zu tolle“ Ergebnisse sind die Folge.)

Beziehungskiste

Es ist daher fast wie bei einer „Beziehungskiste“ zwischen Menschen. Die Verständigung über: „Was gesollt werden soll!“ bleibt schwierig. Aus der strikten Ingenieursicht reduziert man das Problem auf die funktionalen Anforderungen, das heißt in der Informatiksprache formuliert, auf die Disziplin „Requirements Engineering“ (RE).

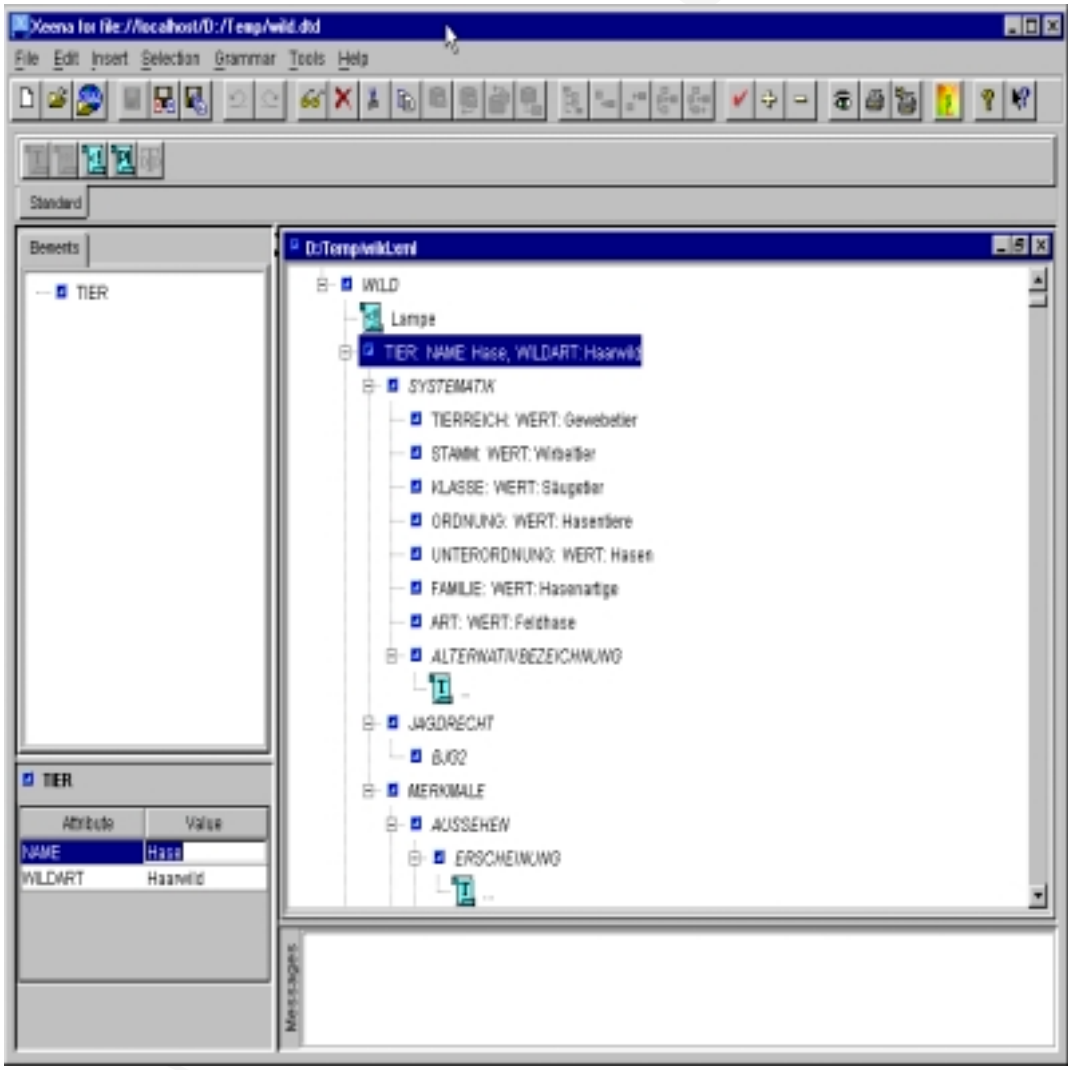
2 Lösungsansätze für die Verständigung

Sie wissen es, im Studium spielt das effektive Lernen eine Hauptrolle. Lassen Sie uns daher die Entwicklung der RE-Disziplin anhand von Lernphasen skizzieren. Für das RE können wir diese wie folgt charakterisieren:

```
$ (define fac (lambda (n)
    (cond ((< n 1) 1)
      (#T (* n (fac (- n 1))))))) ==>
 Value: fac
$ (fac 500) ==>
 Value: 1220136825991110068701238785423046926253574342803192842192413588385845373
15388199760549644750220328186301361647714820358416337872207817720048078520515932
92854779075719393306037729608590862704291745478824249127263443056701732707694610
62802310452644218878789465754777149863494367781037644274033827365397471386477878
49543848959553753799042324106127132698432774571554630997720278101456108118837370
95310163563244329870295638966289116589747695720879269288712817800702651745077684
10719624390394322536422605234945850129918571501248706961568141625359056693423813
00885624924689156412677565448188650659384795177536089400574523894033579847636394
49053130623237490664450488246650759467358620746379251842004593696929810222639719
52597190945217823331756934581508552332820762820023402626907898342451712006207714
64097945611612762914595123722991334016955236385094288559201872743379517301458635
75708283557801587354327688886801203998823847021514676054454076635359841744304801
28938313896881639487469658817504506926365338175055478128640000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000
$
```

Legende: Scheme-Beispiel „Fakultätsfunktion“
Scheme ↪Abbildung 3 Seite 5

Abbildung 5: Korrekte Berechnung der Fakultät 500?



Legende: Xeena-Web-Site: *MENSCH ⇔ MASCHINE* © bonin@fhnon.de
http://www.alphaWorks.ibm.com/tech/xeena Zugriff: 12-Mar-2001

Abbildung 6: Xeena — eine XML Entwicklungsumgebung der IBM

1. Lernfokus: Absturzfreies Programm (Code)
2. Lernfokus: Modifizierbare Software (Menge von Programmen)
3. Lernfokus: Akzeptierbare Lösung („Benutzer im Mittelpunkt“)
4. Lernfokus: Zusammenwirken von Lösungen (Evolution, Infrastuktur)

2.1 Fokus: Absturzfreiheit

Sie erinnern sich, bei den notwendigen Präsentationen Ihrer kleinen Programme im Rahmen der Leistungsnachweise waren Sie — und natürlich auch die Prüfer — stets erfreut, wenn das System nicht mittendrin abstürzte. In der Praxis ist die Absturzfreiheit, die erste Anforderung bei einer systematischen Softwarekonstruktion. Diese Anforderung läßt sich pointiert in drei Aspekten beschreiben:

1. Die Software „läuft“ überhaupt!
2. Man kann mit der Software tatsächlich arbeiten!
3. Die Ergebnisse sind durchaus nützlich!

2.2 Fokus: Wartbarkeit

In einem System ist der abgebildete Ausschnit einer „Realität“ fest codiert. Die Realität ist jedoch dynamisch, sie ändert sich laufend. Die Software muß sich also möglichst problemlos anpassen! Wartung/Pflege und Fortentwicklung rücken in den Fokus.

Pflege?

Sie kennen die alte Wahrheit vom Unterschied zwischen dem schlechten Programmierer und dem guten: Unwissende nehmen an, der gute macht keine Fehler. Ganz falsch! **Der bessere Programmierer macht nur bessere Fehler!**

Die Fehlersuche und Fehlerbereinigung sind mühsam, zeit- und kostenintensiv. Jeder der durch ein fremdes oder ein älteres eigenes Programm sich durchquälen muß, weiß es. Durch zweckmäßige Strukturierung, Modularisierung (↪Abbildung 7 Seite 10) verbunden mit einer gründlichen Qualitätskontrolle versucht man Wartungs- und Weiterentwicklungskosten in den Griff zu bekommen.

Software-entsorgung

Trotz aller Bemühungen: Nach einer gewissen Nutzungszeit ist das Programm derartig kaputtgepflegt, dass nur noch Software-Sanierung oder Softwareentsorgung bleibt. Wenn bloß nicht die verhexten Umstellungsprobleme wären! Manch altes Stapelprogramm (vielleicht geschrieben in SAP R/3 ABAP4) wäre längst erfolgreich entsorgt.

Abbildung 7: *FUZZI* zeigt wie die Systemkonstruktion geht!

2.3 Fokus: Akzeptanz

Es ist eine Binsenwahrheit: „Die Realität ist viel komplexer als sie der Informatiker erfaßt und abbildet!“. Für den Erfolg eines Projektes müssen daher möglichst alle Benutzer, Betroffenen und Verantwortlichen einbezogen werden. Von ihnen erhofft man sich die Ideen und damit letztlich die präzisen Anforderungen an das System. Unstrittig geht es daher längst nicht mehr um die Partizipation der Anwender, sondern um die Art und Weise einer erfolgversprechenden Teamarbeit, das heißt um die Partizipation der unterschiedlichen Spezialisten.

Partizipation

Damit das System akzeptiert wird, ist besonders die Meinung der Benutzer gefragt. Alle, Unternehmensleitung, Betriebs- bzw. Personalräte, Softwareingenieure reden davon, dass die Systemgestaltung sich nach dem Menschen zu richten hat. **Der Mensch rückt in den Mittelpunkt. Wir sollten uns daran erinnern, dass dies auch die Kanibalen sagen.** Die Partizipation reicht nicht, der Zweck und die Interessen sind ausschlaggebend.

2.4 Fokus: Infrastruktur

Selten finden wir für das zu konzipierende System eine„grüne Wiese“ vor, auf der wir ohne Bauauflagen unsere Anforderungen entwickeln und realisieren können.

Bauauflagen

Softwareentwicklung findet im bebauten Gebiet statt. Sie wissen es, wenn wir heute ein System so und nicht anders errichten, dass wir uns viele Alternativen damit verbauen. Es ist klar: **Große Projekte, wie Straßen, Kanalisation, Hochhäuser determinieren die kleinen.**

„Ehe“ mit einem System?

Anders und holzschnittartig formuliert: Die Entscheidung für eine Basissoftware, wie zum Beispiel ein Betriebssystem, eine Programmiersprache oder eine Produktionshilfe (euphorisch auch Werkzeug genannt) gleicht einer **ehelichen Bindung**. Scheidung ist schmerzlich und kostenintensiv. Wer heute zum Beispiel auf das Datenbankmanagementsystem *Oracle*[5] setzt, kann nicht morgen sich schmerzfrei das objektiv bessere System XYZ (vielleicht *POET*[6]) anlachen. Die Wahl der langfristig zweckmäßigen Infrastruktur ist bedeutsam und risikoreich. Halt! Kein Grund für unsere COBOL-Freunde sich beruhigt zurück zu legen. In der Bundesrepublik sind im letzten Jahr einige tausend unglückliche Ehen geschieden worden.

3 Undurchschaubarkeit

Im Problemaufriß wurden nicht sofort nachvollziehbare „Ungereimtheiten“ demonstriert. Unsere „Maschine“ bestehend aus Hard- und Softwarekomponenten ist bekanntermaßen ein formal logisches System und trotzdem undurchschaubar. In der Praxis reagiert sie häufig anders als vermutet und gewollt. Abhilfe soll ein solider ingenieurmäßiger Konstruktionsprozeß schaffen.

Üblicherweise hat RE die Aufgabe, die Anforderungen vollständig zu sammeln, zu Visualisieren und zu Prüfen. Dieses klassische RE-Denkmodell unterstellt eine vorgegebene Realität, aufgrund deren gründlicher Analyse man die Anforderungen lösungsneutral spezifizieren könnte. Es verstellt jedoch die Einsicht von der prinzipiellen Undurchschaubarkeit der komplexen„Maschine“. Zum Verstehen, Entwerfen und Gestalten einer nützlichen, verantwortbaren Arbeitsteilung zwischen Menschen und Maschinen ist eine Ergänzung des klassischen RE-Denkmodells notwendig.

Eigenleben

Das erweiterte RE-Denkmodell akzeptiert ein undurchschaubares „Eigenleben“ der Maschine. Trotz aller Spezifikationsbemühungen kalkulieren wir nun quasi gleichwertig das „Nicht-Denkbare“ ein.

Komplexität meistern

Einerseits bemühen wir uns intensiv um das Verstehen von Ursachen/Wirkungsketten, wohl wissend, dass wir relevante Zusammenhänge nicht erkennen. Das Grundübel hierbei ist unser Unvermögen, mehr als einen dynamischen Vorgang bewußt verfolgen zu können.

Andererseits versuchen wir mit der „Undurchschaubarkeit“ kompetent umzugehen. Als Folge davon beachten wir zum Beispiel Möglichkeiten zur Risiko-

[5] Web-Site: `http://www.oracle.com` Zugriff: 13-Mar-2001.
[6] Web-Site: `http://www.poet.de` Zugriff: 13-Mar-2001.

begrenzung und arbeiten mit Sicherheitszuschlägen. Leitidee ist das Einsparen von Komplexität, das heißt Komplexität, bedingt durch eine zu komplizierte Konstruktion, ist zu vermeiden.

Anders formuliert: RE klärt nicht mehr nur die Fragen nach dem „WAS“. Es integriert unentwirrbar die Fragen nach dem „WIE“, zumindest in der Form:

„Wie bewältigen wir das *Eigenleben*?“

Die klassische Vorgehensweise: „Der Auftraggeber bestimmt die Anforderungen, Softwareingenieure vollziehen diese ordnungsgemäß“ ist für unser RE-Denkmodell nicht hinreichend.

4 Paradigma „Kooperation“

Partnerrolle

Als Analogie könnte eher das Bild eines Kooperationsvertrag zwischen (Be-) Nutzer und Maschine dienen. Die Maschine rückt in eine „Partner“-Rolle. Mehr im Sinne einer zu regelnden „Beziehungskiste“. Das Bild einer „Mensch-Maschine-Kooperation“ zielt auf ein zweckorientiertes Verstehen. Dabei ist Undurchschaubarkeit prinzipiell kein Hinderungsgrund für eine nützliche Arbeitsteilung zwischen Menschen und Maschinen. Kooperation heißt hier keinesfalls „Gleichwertigkeit“. Dies zu thematisieren, ist Unsinn. Obwohl dem Partner „Maschine“ jedes eigene „Interesse“ fehlt, vermittelt das Kooperationsbild diese hilfreiche Assoziation.

Partner Computer

Der Nutzer formuliert die erwarteten Leistungen vom Kooperationspartners, stets mit dem Bewußtsein, dass der „Partner“ nur im gewissen Umfang, dass tut was man hätte sich wünschen sollen. Ihm wichtige und möglicherweise strittige Aspekte spezifiziert er daher genau, andere subsumiert er unter „Treu und Glauben“ und formuliert sie daher nicht explizit. Wie in jeder Kooperation, so unterstellt der Auftraggeber, dass sein Partner einen selbständigen Aktionsbereich hat, den er nicht durchschaut. Aus positiven und negativen Erfahrungen heraus, entwickeln sich die Aspekte, die im Vertrag präzise zu spezifizieren sind.

adäquate Vorgabe

Wir streben daher keine vollständige Spezifikation an, sondern eine adäquate, im Hinblick auf einen existierenden *Partner*. Dabei berücksichtigen wir die bisher gemachten Erfahrungen. Kurz und plakativ formuliert: Requirements Engineering ist maschinenabhängig. Für Sie, nun als trainierter Akronym-Fan, formuliert:

> RE gegen/mit einem *UNIX-Oracle-SAP-R/3-ABAP*-Partner ist signifikant anders als gegen/mit einem *NT-Java-Poet*-Partner.

5 Fazit: Undurchschaubarkeit meistern!

Wenn es es richtig ist, dass wir die „Maschine“ bei unserem Kooperationsmodell selbst bei einem gelungenen Konstruktionsprozeß und einem ordnungsgemäßen

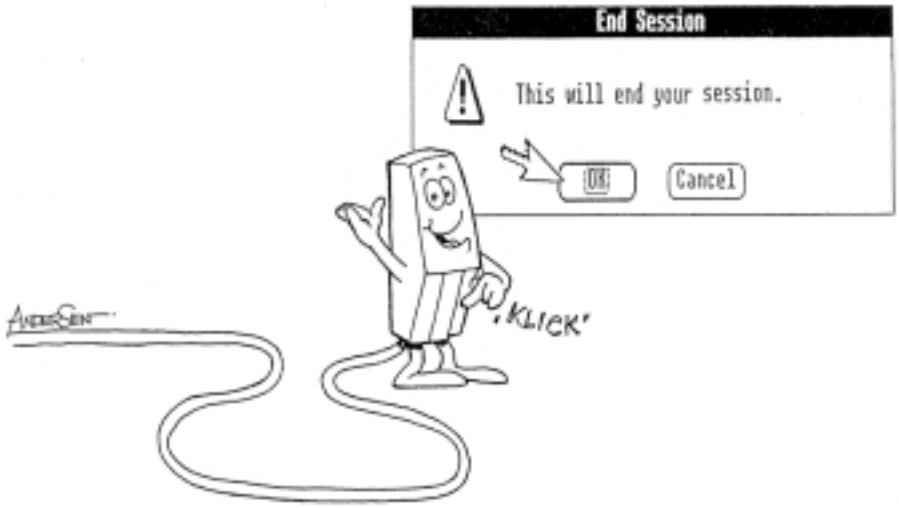


Abbildung 8: $\mathcal{FUZZI}$'s wichtiger Tip.

Betrieb nicht mehr völlig durchschauen, dann sollten wir lernen, diese Undurchschaubarkeit zu akzeptieren und mit ihr umzugehen. Ich bitte Sie daher, als zukünftige Konstrukteure und Manager von Computersystemen, vertrauen Sie nicht blind auf „ihn“, den Partner Computer. Meistern Sie die Undurchschaubarkeit indem Sie beispielsweise dafür sorgen:

- **Risiken zu minimieren**,
- eher zuviel als zugeringe **Sicherheitzuschläge einzukalkulieren** und
- soweit wie möglich **auf Reversibilität zu achten**.

Nach diesem Wunsch bleibt mir hier nur noch den wichtigen Tip von $\mathcal{FUZZI}$ (↪Abbildung 8 Seite 13) zu vollziehen. OK?

Literatur

[Bo1988] Hinrich Bonin; Die Planung komplexer Vorhaben der Verwaltungsautomation, Heidelberg (R. v. Decker & C. F. Müller), 1988 (Schriftenreihe Verwaltungsinformatik; Bd. 3). {Hinweis: Eine Kritik an der Softwareentwicklung nach dem Phasenmodell.}

[Bonin91] Hinrich E.G. Bonin; Software-Konstruktion mit LISP, Berlin New York (Walter de Gruyter), 1991, ISBN 3-11-01786-X. {Hinweis: Ein LISP-Buch mit den Schwerpunkten Rekursion und Objektorientierung.}

[Bo1993] Hinrich E. G. Bonin; The Joy of Computer Science, — Skript zur Vorlesung EDV —, Unvollständige Vorabfassung (4. Auflage März 1995), FINAL, 3. Jahrgang Heft 5, 20. September 1993, [FINAL]. {Hinweis: Ein Vorlesungsskript.}

[Bo1998] Hinrich E. G. Bonin; Der Java-Coach, FINAL, 8. Jahrgang Heft 1, Oktober 1998, [FINAL], aktuelle Fassung unter:
`http://as.fhnon.de/publikation/anwdall.pdf` (Adobe PDF-Format) {Hinweis: Erläutert die Java-Konstrukte undschließt beispielsweise die Themen „objektorientierte Datenbank“ (POET) und „interne Klassen“ mit ein.}

[Bo2000a] Hinrich E. G. Bonin; WI>DATA — Eine Einführung in die Wirtschaftsinformatik auf der Basis der Webtechnologie, FINAL, 10. Jahrgang Heft 2, September 2000, [FINAL], aktuelle Fassung unter:
`http://as.fhnon.de/publikation/widata.pdf` (Adobe PDF-Format) {Hinweis: Befaßt sich mit der Konstruktion, dem Einsatz und den Wirkungen von computergestützten Systemen, die für Unternehmen potentiell nützlich sind.}

[Bo2000b] Hinrich E. G. Bonin; Der kleine XMLer — Eine Einführung in das Programmieren mit XML, FINAL, 10. Jahrgang Heft 4, Dezember 2000, [FINAL], aktuelle Fassung unter:
`http://as.fhnon.de/publikation/xmler.pdf` (Adobe PDF-Format) {Hinweis: Befaßt sich mit der Web-Technologie; insbesondere mit XML, SGML, DTD. HTML, CSS und XSL.}

[FINAL] Fachhochschule Nordostniedersachsen, Informatik, Arbeitsberichte, Lüneburg (FINAL) herausgegeben von Hinrich E. G. Bonin, ISSN 0939-8821, ab 7. Jahrgang (1997) auch auf CD-ROM, beziehbar: FHNON, Volgershall 1, D-21339 Lüneburg, Germany. {Hinweis: Eine Reihe, die primär von Lehrenden und Studierenden der FH Nordostniedersachsen getragen wird.}

[Floyd89] Christiane Floyd; Softwareentwicklung als Realitätskonstruktion, in: [Lippe89], S. 1–20 {Hinweis: Ein Klassiker für die Kritik am klassischen Phasenmodell.}

[Lippe89] W.-M. Lippe (Hrsg.); Software-Entwicklungs-Konzepte, Erfahrungen, Perspektiven; Fachtagung, veranstaltet vom GI FA 2.1, Marburg 21.-23. Juni 1989, Berlin Heidelberg u.a. (Springer-Verlag), Informatik-Fachberichte 212.

Index