Metriken sollen durch Zählung und Rechnung quantitative Aussagen über Software liefern. In den siebziger Jahren wurden zum Teil hochgesteckte algorithmische Metriken vorgeschlagen. Sie haben sich in der Praxis eigentlich nicht bewährt. Die simplen Metriken sind nach wie vor die wichtigsten.
Verschiedene Qualitätsaspekte werden sich wohl nie objektiv quantifizieren lassen. Das soll aber nicht davon abhalten, wenigstens geeignete subjektive Verfahren, also Schätzungen, anzuwenden.

Qualitäts- und Komplexitätsmetriken, subjektive Schätzungen

Wie gut ist die Software?

Von Jochen Ludewig

«The answer is: forty two!»
DOUGLAS ADAMS: *«The Hitchhiker's Guide to the Galaxy»*

Wenn wir Software beurteilen wollen, so geht es letztlich um die Frage:

Wie gut ist die Software?

Bedenken wir, daß sich auch die Qualität einfacherer Objekte kaum in einem einzigen Maß ausdrücken läßt, so kommen wir zu etwas differenzierteren Fragen, beispielsweise zu den folgenden:

Wie gut ist die Strukturierung der Software?
Wie verständlich ist die Software?
Wie effizient ist die Software?
Wie genau (oder eigentlich: wie ungenau) sind die Resultate?
Wie bedienfreundlich ist die Software?
Wie schwierig ist eine Änderung der Software?

Offenbar lassen sich auch diese Fragen nicht durch eine einzige Kennzahl beantworten; es handelt sich um Leistungs- und Qualitätsmerkmale, die zwar extrem wichtig, aber nicht präzise definiert sind.

Suche nach vernünftigen Kriterien

In sehr vielen praktischen Situationen wären aber Antworten auf solche Fragen *(«Bewertungen»)* außerordentlich nützlich, vorausgesetzt, sie sind

- differenziert
- vergleichbar und exakt
- reproduzierbar
- relevant
- verfügbar

Dabei bedeuten:

differenziert
Die Vielfalt möglicher Antworten ist groß genug, um Unterschiede, die auf andere Weise erkennbar werden, auch in der Bewertung sichtbar zu machen. Empfindet man zum Beispiel beim Durchlesen zweier Programme das eine als verständlicher, so sollten auch die entsprechenden Bewertungen unterschiedlich sein.

vergleichbar
Das Resultat der Bewertung liegt in einer geordneten Menge («Rangliste»), wir haben also eine **Ordinalskala** vor uns; ist dies nicht der Fall, wie beispielsweise bei der *Benennung* der verschiedenen Programmiersprachen, so haben wir nur eine **Nominalskala,** in der Vergleiche (zum Beispiel die alphabetische Ordnung) sinnlos sind. Noch besser als die Ordinalskala ist eine **Intervallskala,** in der auch *Differenzen* zwischen Bewertungen eine Bedeutung haben (Beispiel: Zeitpunkte, angegeben in Sekunden seit dem 1. Januar 1900, 12.00 Uhr GMT). Am besten ist eine **Rationalskala,** die einen *sinnvollen* Nullpunkt hat (zum Beispiel den Speicherbedarf in Byte) und darum auch dem *Verhältnis* der Bewertung Sinn gibt

exakt
Die Antwort bezeichnet auf der Skala einen bestimmten Punkt, nicht ein Gebiet oder eine «Wolke». Eine Bewertung sollte also beispielsweise «3,1» sein, nicht «zwischen 3 und 4» oder «ungefähr 3».

reproduzierbar
Wenn die Bewertung desselben Objekts unter sonst gleichen Bedingungen mehrmals vorgenommen wird, so soll sie stets das gleiche Resultat liefern. Reproduzierbarkeit ist nur zu erreichen, wenn die Bewertung auf wohldefinierten Größen und Algorithmen basiert, wenn sie also *mechanisch* vorgenommen werden kann.

relevant
Jede quantitative Aussage muß auf eine nachvollziehbare Weise mit relevanten Größen korreliert sein. Schneidet also Programm A bei der Bewertung der Wartungsfreundlichkeit wesentlich besser ab als die Programme B und C, deren Bewertungen sich kaum voneinander unterscheiden, so sollte dies – bei sonst gleichen Bedingungen – auch für die *tatsächlich* entstehenden Wartungskosten gelten.

verfügbar (und rentabel)
Die Bewertung muß vorliegen, wenn sie gebraucht wird, später hat sie keinen Nutzen mehr. Sie darf nicht mehr Aufwand erfordern, als nach ihrer Relevanz angemessen ist.

Exaktheit und Differenziertheit sind offenbar eng verbunden, wir fassen sie als *Schärfe* zusammen; Schärfe, Vergleichbarkeit, Reproduzierbarkeit, Relevanz und Verfügbarkeit sind dagegen konkurrierende Ziele. Jede praktische Lösung des Problems läuft also auf einen Kompromiß hinaus.
Heute anwendbare Ansätze lassen sich danach klassifizieren, welches Kriterium betont wird oder bei welchem die meisten Abstriche gemacht werden.
Traditionell dominiert die Verfügbarkeit. Man verzichtet auf Vergleichbarkeit und Schärfe und bekommt praktisch ohne Aufwand gewisse Klassifizierungen, zum Beispiel «Spezifikation liegt vor», «Programm ist im Test» oder «Installation ist durchgeführt». Im Sinne der gestellten Fragen sind diese Aussagen natürlich unergiebig. Da weder die verwendeten Begriffe noch der Kontext der Angaben definiert sind, liefern sie keine vergleichbaren Aussagen. Wenn ein Programm lange im Test war, kann man daraus auf schlechten Code schließen. Aber vielleicht war der Test in diesem Falle nur besonders sorgfältig vorbereitet; das Symptom wäre dann ein positives Merkmal.
Weitergehende Einschätzungen, selbst wenn sie grob und schwammig sind, geben die Reproduzierbarkeit preis, sind aber trotzdem nicht vergleichbar und für praktische Schlußfolgerungen weder ausreichend präzise noch differenziert.

Prof. Dr. JOCHEN LUDEWIG, Institut für Informatik der Universität Stuttgart, Azenbergstraße 12, D-7000 Stuttgart 1.

```
PROCEDURE DirektEinfuegen;

 VAR i, j : IndexTyp;
     Zwischenspeicher : ElementTyp;

BEGIN { DirektEinfuegen }
  FOR i := 2 TO MaxIndex DO BEGIN
    Zwischenspeicher  := Sortierfeld [i];
    Sortierfeld [0] := Zwischenspeicher;
    j := i;
    WHILE Zwischenspeicher < Sortierfeld [j-1] DO BEGIN
      Sortierfeld [j] := Sortierfeld [j-1];
      j := pred (j);
    END { WHILE };
    Sortierfeld [j] := Zwischenspeicher;
  END { FOR };
END { DirektEinfuegen };
```

Bild 1. Programmteil DE.

```
BEGIN { Sortieren }
PROCEDURE DirektEinfuegen;
  Generate (Kopie); Out (Kopie); Erster := TRUE;

  Prepare ('Direkt Aussuchen'); DirektAussuchen; Check;
  Prepare ('Direkt Einfuegen'); DirektEinfuegen; Check;
  Prepare ('Binaer Einfuegen'); BinaerEinfuegen; Check;
  Prepare ('Bubble Sort');      BubbleSort;      Check;
  Prepare ('HeapSort ');        HeapSort;        Check;
  Prepare ('QuickSort');        QuickSort;       Check;
END { Sortieren }.
```

Bild 2. Programmteil SN.

Metriken — wozu?

Die moderne Physik und alle anderen Wissenschaften und technischen Disziplinen sind dadurch gekennzeichnet, daß die vermuteten Gesetzmäßigkeiten nicht wie in der Antike und im Mittelalter durch philosophische Begründung, sondern durch Beobachtungen und Messungen erhärtet oder widerlegt werden.
In den siebziger Jahren wurde versucht, dieses Prinzip auch in die Informatik einzuführen; unter dem Schlagwort Metriken wurden Verfahren vorgeschlagen, die durch Zählung und Rechnung quantitative Aussagen über Software liefern sollten. Nach der ersten Euphorie flaute die Begeisterung allerdings rasch ab, denn die vorgeschlagenen Maße hatten sich überwiegend als willkürlich und nutzlos erwiesen.
Das Interesse an den Metriken hat jetzt wieder zugenommen. Die neue Blüte scheint sehr viel stabiler als die erste zu sein. Mehrere Voraussetzungen sind inzwischen deutlich besser geworden:

- An die Stelle der anfänglich überrissenen Erwartungen sind niedrigere, realistischere Anforderungen getreten.
- CASE (Computer-Aided Software Engineering) verbessert die Möglichkeiten, Werkzeuge zur Messung bereitzustellen.
- Jetzt ist einige brauchbare Literatur über Metriken verfügbar.
- Der wachsende Druck auf die Software-Entwickler, Termin-, Kosten- und Qualitätsziele einzuhalten, erhöht den Bedarf für Metriken.
- Ohne geeignete Metriken können Software-Produkte mit der notwendigen Objektivität weder verglichen noch zertifiziert werden.
- Nachdem sich die Forschung in den ersten vierzig Jahren der Informatik auf qualitative Aussagen konzentrieren konnte, wird nun das Defizit an quantitativen Aussagen immer hinderlicher.

Für einen an sich sinnvollen Einsatz der Metriken fehlen heute aber meist noch die Kenntnisse und die Erfahrungen. Hier setzte die «TR»-Werkstatt Software-Metriken inhaltlich an, die am 1. und 2. November 1990 in Gerzensee durchgeführt wurde. Drei Themenblöcke standen auf dem Programm:

- Qualitäts- und Komplexitätsmetriken sowie subjektive Schätzungen
- Prognostische Metriken
- Leistungs- und Kostenmetriken, Probleme bei der Einführung von Metriken im Unternehmen

Der vorliegende Beitrag von JOCHEN LUDEWIG befaßt sich mit dem ersten Themenkreis.
Wir werden in folgenden Ausgaben der «Technischen Rundschau» mit weiteren Beiträgen auch die anderen Themenblöcke der «TR»-Werkstatt vorstellen.

Redaktion

Beispiele sind jedem bekannt, man sagt von einem Stück Software, es sei «einfach ein Verhau» oder es sei «sehr raffiniert codiert», und hat damit eine relevante und – im großen und ganzen – reproduzierbare Aussage, die allerdings nicht viel nützt. Die Reproduzierbarkeit reicht auch nicht aus, um den Urheber zu überzeugen, denn dieser wird sich auf andere, ebenso vage Kriterien berufen («das Problem war eben besonders schwierig»). Unschärfe erzeugt also, wie überall im Leben, Unsicherheit und Streit. Darum ist dieser Ansatz unbefriedigend und wird hier nicht weiter betrachtet.

Ein anderer, ebenfalls sehr populärer Ansatz besteht darin, auf die Relevanz zu verzichten. Man zählt, was sich problemlos zählen läßt, und ignoriert die wirklich bedeutsamen Fragen. So gilt der Speicherbedarf des übersetzten Programms vielerorts als wichtiges Maß, selbst dann, wenn die Programmierer den größten Schund verfassen und weitaus mehr Code schreiben, als in der betreffenden Situation angemessen wäre. Natürlich werden wirklich relevante Größen wie der Wartungsaufwand dadurch in die Höhe getrieben.
Schließlich kann man auf die Reproduzierbarkeit verzichten und Bewertungen «freihändig» schätzen. Natürlich führt auch dieser Ansatz zu Meinungsverschiedenheiten, denn Schätzungen sind unvermeidlich subjektiv.

Versuche, die beiden zuletzt charakterisierten Ansätze zu verbessern, sind Thema dieses Beitrags. Im ersten Teil geht es um Qualitäts- und Komplexitätsmetriken, also reproduzierbare Bewertungen, die – wenigstens dem Anspruch nach – Aussagen über relevante Eigenschaften der Software liefern. Solche Metriken wurden vor allem in den siebziger Jahren propagiert.
Der zweite Teil diskutiert Schätzverfahren, die so angewandt werden, daß die Streuung durch Subjektivität akzeptabel wird.
In diesem Beitrag wird nur in wenigen Fällen die Originalliteratur zitiert. Ausgezeichnete Bibliographien sind im Buch von CONTE, DUNSMORE, SHEN [3] und in der Übersicht von HÖCKER et al. [4] enthalten.

Komplexität der Software beschreiben

In den siebziger Jahren entstand die Idee, verschiedene Qualitätsaspekte,

also komplexe Merkmale der Software, durch Maßzahlen zu beschreiben. Damit war die Idee der Qualitätsmetrik geboren. Komplexitätsmetriken können je nach Sprachgebrauch als spezielle Qualitätsmetriken oder als nahe Verwandte betrachtet werden; sie sollen die Komplexität der Software beschreiben.
Allen diesen Ansätzen gemeinsam ist das Ziel, Maßzahlen zu definieren, die sich mechanisch berechnen lassen und die um so kleiner (oder bei anderer Definition: um so größer) sind, je besser die Software ist. Dabei wird a priori akzeptiert, daß die Problemstellung Grenzen setzt, daß also beispielsweise ein komplexes Problem auch komplexe Software erfordert. Trotzdem bleibt niedrige Komplexität, die durch einen niedrigen Wert des Komplexitätsmaßes angezeigt wird, ein zentrales Ziel.
Alle Bemühungen um reproduzierbare Qualitätsurteile begannen Anfang der siebziger Jahre mit der Einsicht, daß

- einerseits Aussagen über die Softwarequalität dringend gebraucht wurden
- anderseits «harte» Aussagen nur über elementare Aspekte, zum Beispiel den Umfang in Codezeilen, möglich waren

Es lag damit nahe, die Aussagen zu Qualität und Komplexität aus den elementaren Daten abzuleiten.
Vorreiter dieser Entwicklung waren McCabe und Halstead [5, 6], auf deren Arbeiten unten noch näher eingegangen wird.

Die simplen Metriken sind die wichtigsten

Algorithmische Metriken, wie sie in den siebziger Jahren vorgeschlagen wurden, haben bis heute keine nennenswerte Bedeutung erlangt, weil es nicht gelungen ist, geeignete Modelle zu finden. Zudem können Entwickler die algorithmischen Metriken unterlaufen. Darum sind noch immer die simplen Metriken die wichtigsten [1].
Sinnvoll ist dagegen die Sammlung von Basisdaten, die dann frei interpretiert werden können. Die Archivierung dieser Daten schafft die Chance, später Entwicklungen zu erkennen, die nicht sofort sichtbar waren.
Die subjektive Bewertung und ihre Umsetzung in Kennzahlen ist ein Bewertungsverfahren, das eng mit anderen Reviewtechniken verwandt ist. Es ist auch wie diese in jeder Phase anwendbar.
COCOMO [11] zeigt, wie sich subjektive Schätzungen (in diesem Fall für DLOC und Enflußfaktoren) mit einem Rechenverfahren kombinieren lassen. Die Resultate sind erstaunlich gut reproduzierbar und genau, wenn präzise Richtlinien bestehen und das Verfahren durch einige Anwendungen kalibriert ist; das können auch Nachkalkulationen sein. Denn es hat sich gezeigt, daß jede Umgebung einen spezifischen Korrekturfaktor benötigt. Die langsame Veränderung dieses Faktors zeigt an, wie sich die *Software-Kultur* des Unternehmens verändert.

Das Modellierungsproblem

Betrachten wir als Beispiel die *Komplexität* eines Programms: Als Autoren oder Leser haben wir von der Komplexität eine vage Vorstellung, die aber nicht irrational ist: In vielen praktischen Fällen werden wir bei einem Vergleich verschiedener Programme auch ohne eigentliche Definition darin übereinstimmen, welches komplexer ist. Schlägt nun jemand ein Komplexitätsmaß vor, so erwarten wir, daß dessen Resultate plausibel sind, also unserer Einschätzung im wesentlichen entsprechen.
Die Beurteilung einer Komplexitätsmetrik erfolgt also durch Vergleich der Resultate, die wir auf zwei Wegen erzielen, nämlich subjektiv und durch Berechnung. Sind die beiden Resultate deutlich korreliert, so betrachten wir die Metrik als relevant.
Grundsätzlich könnte jeder beliebige Algorithmus auf der Grundlage irgendwelcher Daten, beispielsweise die Quersumme der Zeilenzahl, multipliziert mit dem Alter des Programmierers, ein relevantes Resultat liefern; praktisch ist aber nur dann eine relevante Größe zu erwarten, wenn ein plausibles Modell zugrunde liegt. Da die Begriffe, zu denen Metriken gesucht werden, selbst nur vage und uneinheitlich verstanden werden, hat dieses Modell dann auch *definierende* Wirkung.
Die Basisformel von Boehms COCOMO [11] kann dafür als Beispiel dienen:

$$A = k \cdot S^p$$

Der Aufwand A ist also proportional der mit dem Exponenten p potenzierten Größe S, wobei p etwas größer als 1 ist und die Einheiten durch Normierung beseitigt sind. Offenbar liegt das Modell zugrunde, daß der Aufwand mit der Größe steigt, und zwar etwas überproportional, weil bei steigendem Umfang ein wachsender Teil der Arbeit nur dazu dient, den Überblick zu behalten. Die Formel ist eine einfache Realisierung dieser – qualitativ plausiblen – Überlegung. Offenbar stimmt das Modell aber nicht, wenn die Software aus völlig entkoppelten Komponenten besteht oder wenn eine vorhandene Komponente eingebunden wird. Das Modell ist an bestimmte Randbedingungen geknüpft. Die Entdeckung geeigneter Modelle ist also der Schlüssel zu den algorithmischen Metriken. Dabei kann ein Modell verschieden eng an die Realität geknüpft sein: Im günstigsten Fall ist der Zusammenhang kausal, beispielsweise zwischen Aufwand in Entwicklertagen und Kosten. In diesem Fall kann das Modell nicht unterlaufen werden; wenn der Entwickler mit weniger Zeit auskommt, wird die Software wirklich billiger.
In den meisten Fällen beruhen die Modelle aber auf Größen, die mit den wirklich interessierenden nur korreliert sind. In aller Regel läßt sich das Programm im Hinblick auf eine bestimmte Metrik so «optimieren», daß das Resultat – bei unveränderter Qualität – günstiger ausfällt. So ist beispielsweise die Zahl der Kommentarzeilen in einem gutkommentierten Programm höher als in einem nahezu unkommentierten. Nimmt man aber diese Zahl als Qualitätskriterium, so kann der Codierer ohne nennenswerte Mühe beliebig viele Leerkommentare (oder solche mit sinnlosem Inhalt) einstreuen. Die Metrik ist dann wertlos.
Algorithmische Metriken sind darum kaum zur Beurteilung der Entwickler geeignet; dafür eignen sich objektive Kriterien (zum Beispiel Aufwandsangaben, Fehlerzahlen) und die folgenden beschriebenen subjektiven Bewertungen besser.

Komplexitätsmaß nach McCabe

Die Komplexität eines Programms sollte im Interesse der Wartbarkeit so klein wie möglich sein. Aber was ist Komplexität? Offenbar hat die Länge des Programms Einfluß, aber sie ist keineswegs entscheidend, wie die beiden Auszüge eines Programms zeigen, die verschiedene Sortieralgorithmen gegenüberstellen (Bilder 1 und 2).
Beide Programmteile, DE (Bild 1) und SN (Bild 2), sind nahezu gleich groß; DE hat 464 Zeichen, ohne Blanks und Kommentare noch 320, bei SN sind diese Werte 430 und 314. SN enthält 21 Anweisungen (gekennzeichnet durch «;» am Ende), DE nur acht. Trotzdem werden die meisten Leute deutlich mehr Mühe haben, DE zu verstehen.

Die *Zyklenkomplexität* («cyclomatic complexity») nach McCabe ist in Anlehnung an die Graphentheorie durch die Gleichung

$$v(G) = e - n + p$$

definiert, wobei e die Zahl der Kanten, n die Zahl der Knoten in einem Flußdiagramm ist. (Es ist dabei, wie sich zeigt, gleichgültig, wie das Flußdiagramm genau angelegt wird, ein zusätzlicher Knoten wird durch eine zusätzliche Kante kompensiert.) p ist die Zahl der Verbindungen nach außen, im Falle eines normalen Unterprogramms ist p also 2 (*ein* Eingang und *ein* Ausgang). Anschaulicher ist die Vorstellung, daß eine lineare Verkettung von Anweisungen, wie sie in SN vorliegt, die Komplexität 1 hat; jede Verzweigung oder Schleife erhöht diese um 1. In der Flußdiagrammdarstellung von DE in Bild 3 sieht man die beiden Schleifen, die zur Komplexität 3 führen.
Man kann also v(G) sehr leicht mechanisch bestimmen, man braucht nur ein Programm, das – hier auf PASCAL bezogen – die Zahl der IF-, FOR-, WHILE-, REPEAT- und CASE-Anweisungen feststellt. Jede erhöht v(G) um 1, nur CASE um die Zahl der Wege minus 1. PROGRAM, PROCEDURE und FUNCTION werden als Eingänge gezählt, AND und OR als verkappte Bedingungsschachtelung ebenfalls.

«Software Science» nach Halstead

Relativ einfache Maße wie die Zyklenkomplexität nach McCABE sind anschaulich und können – vor allem in Relation zu anderen Maßen – extreme Situationen anzeigen, beispielsweise durch die Kombination einer hohen Komplexität und einer großen Zeilenzahl einen Strukturierungsfehler. (Eine große Zeilenzahl ist höchstens dann zulässig, wenn es sich um strikt sequentiellen Code handelt, beispielsweise zur Vorbesetzung vieler Variablen.)
Weitergehende Schlüsse lassen sich daraus nicht ziehen, wenigstens nicht mechanisch. HALSTEAD hat daher Mitte der siebziger Jahre versucht, komplexere Metriken zu definieren. Sein Ansatz war ausgesprochen ehrgeizig, und entsprechend hochgestochen war die Bezeichnung «Software Science».
HALSTEADS Modell beruht auf der Vorstellung, daß Operatoren (wie «+», «/», «AND») und Operanden (Literale, vordefinierte und neue Bezeichner) die Software prägen und die Daten dieser beiden Mengen die Software insgesamt charakterisieren. Er arbeitet daher mit vier Basiskennzahlen:

N_1 ist die Zahl der Operatoren im Programm insgesamt
η_1 ist die Zahl der *verschiedenen* Operatoren
N_2 ist die Zahl der Operanden im Programm insgesamt
η_2 ist die Zahl der *verschiedenen* Operanden

$N = N_1 + N_2$ ist die *Länge* des Programms (die Zahl der Wörter).

HALSTEAD postuliert nun zwischen den vier Basisgrößen und mit anderen Größen alle möglichen Zusammenhänge, beispielsweise für die Länge

$$N = \eta_1 \, ld\eta_1 + \eta_2 \, ld\eta_2$$

worin «ld» der Logarithmus zur Basis 2 ist.
Beispielsweise besteht das Programm DE aus 13 verschiedenen Operatoren – nämlich «()», «,», «-», «:», «:=», «;», «<», «[]», «BEGIN ... END», «FOR ... TO ... DO», «pred», «PROCEDURE», «WHILE ... DO» –, die insgesamt 39mal vorkommen, und 11 verschiedenen Operanden («0», «1», «2», «DirektEinfuegen», «ElementTyp», «i», «IndexTyp», «j», «MaxIndex», «Sortierfeld», «Zwischenspeicher»), die 31mal vorkommen. Unsere Basisdaten sind damit

$$\eta_1 = 13 \quad \eta_2 = 11 \quad N_1 = 39$$
$$N_2 = 31 \quad N = 70$$

Diese Länge ist nicht wesentlich verschieden von der theoretischen nach HALSTEAD, denn 13 ld 13 + 11 ld 11 ergibt etwa 86. Die Abweichung läßt sich durch die Separierung eines Teilprogramms erklären, denn einige Vorkommen (zum Beispiel zur Initialisierung oder zur Ausgabe) fehlen in der Prozedur.

Ein anderes Beispiel ist der Programmieraufwand, den HALSTEAD aus der Zahl der Elementarentscheidungen ableitet. Diese gibt er an mit

$$E = \frac{\eta_1 \, N_2 \, N \, ld\,(\eta_1 + \eta_2)}{2\eta_2}$$

$$= \frac{13 \cdot 31 \cdot 70 \cdot ld\,24}{22}$$

$$= 5879$$

Zusammen mit der Zahl der Elementarentscheidungen, die ein Programmierer pro Sekunde fällen kann (etwa 18), ergibt sich daraus ein Zeitbedarf von 327 s oder etwa 5,5 min. Das dürfte wohl nur für einen guten Programmierer zu schaffen sein.

Schwierigkeiten mit algorithmischen Metriken

Jeder Metrik wird zunächst vorgeworfen, daß sie die wirklichen Probleme nicht erfaßt, daß also die Beurteilung realitätsfremd sei. Diese Kritik hat verschiedene Aspekte.

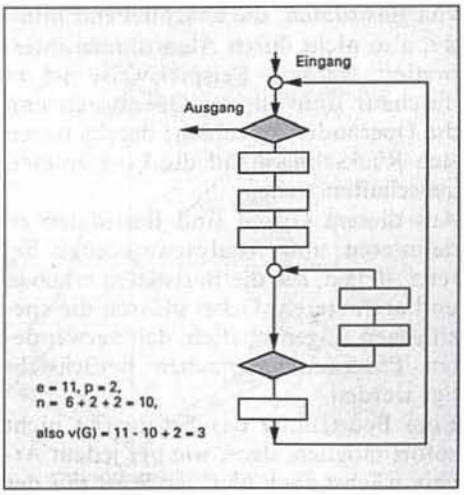

Bild 3. Flußdiagramm für DE.

Was ist schwerer, 1 kg Blei oder 1 kg Federn?

Die Scherzfrage ist nicht ganz so banal, wie sie auf den ersten Blick erscheint. Sie zeigt, daß wir mit einem gespaltenen Gewichtsbegriff leben, nämlich dem exakt physikalischen und dem intuitiven, der gelegentlich mit dem exakten kollidiert. Ähnliches gilt für alle Begriffe, die älter sind als die physikalischen Definitionen, zum Beispiel Kraft oder Zeit. Der Kontext sorgt in der Regel dafür, daß wir den richtigen Begriff verwenden.
Im Software-Engineering werden wir in Zukunft das gleiche Problem haben, es sei denn, wir führen völlig neue Wörter ein. Wir werden also beispielsweise einen Komplexitätsbegriff haben, der dem intuitiven nicht genau entspricht, dafür aber exakt definiert ist. Eine Kritik neuer Vorschläge mit dem Hinweis auf diesen Widerspruch ist also unangemessen.
Bei den algorithmischen Metriken ist Skepsis allerdings leider sehr begründet, denn bislang sind keine Maße in Sicht, die wenigstens minimalen Ansprüchen genügen. Das hat der Artikel von LIND, VAIRAVAN [7] gezeigt: Einfachste Metriken, vor allem die DLOC (Zahl der ausgelieferten Codezeilen), haben sich darin den algorithmischen als überlegen erwiesen.

Ansatz für die praktische Arbeit

Wie im Buch von CONTE, DUNSMORE und SHEN dargelegt, hat HALSTEAD seine hochgesteckten Ziele nicht erreicht. Bewährt hat sich dagegen die Erhebung

von Basisdaten, die anschließend intuitiv, also nicht durch Algorithmen interpretiert werden. Beispielsweise ist es durchaus sinnvoll, die Operatoren und die Operanden zu zählen; daraus lassen sich Rückschlüsse auf die Programmeigenschaften ziehen.
Aus diesem Grund sind Basisdaten zu definieren und Analysewerkzeuge bereitzustellen, die die Basisdaten erheben und archivieren. Dabei müssen die spezifischen Eigenschaften der verwendeten Programmiersprachen berücksichtigt werden.
Eine Beurteilung des Erfolgs ist nicht sofort möglich, denn wie bei jedem Archiv wächst auch hier der Wert mit der Zeit. Daher sollten die Daten so archiviert werden, wie sie erhoben wurden; die Auswertungen sind spekulativ, die Basisdaten sind, soweit die Erhebung seriös war, zuverlässig.

Zwischen klaren Aussagen und Schweigen — Schätzungen

LUDWIG WITTGENSTEIN hat (im «tractatus logico-philosophicus») behauptet, alles lasse sich klar sagen, oder man solle besser schweigen. Dieses *Tertium non datur* ist in den exakten Wissenschaften bis heute als Dogma anerkannt.
Wo aber die volle Exaktheit nicht erreichbar ist, führt es in die *Wittgenstein-Falle*, weil es unsere vielen Möglichkeiten auf eine simple Alternative zwischen klar sprechen und schweigen reduziert. Tatsächlich gibt es ja viele Aussagen, die vage oder schwammig, aber dennoch nützlich sind. Die Einsicht, daß sich manche Qualitätsmerkmale vermutlich auch in Zukunft nicht quantifizieren lassen («transzendente Qualitäten» nach KAPOSI, KITCHENHAM [8]), braucht uns darum nicht von allen Versuchen abzubringen, Quantisierungen vorzunehmen, notfalls auf der Basis subjektiver Einstufungen. Schließlich ist mit diesem Prinzip auch die Schule seit langem (mäßig) erfolgreich. Solche Bewertungen, weder objektiv noch exakt reproduzierbar, sind Thema dieses Abschnitts.

Qualitätsschätzungen: Stand der Technik

Qualitätsschätzungen werden in vielen Firmen und Organisationen auf verschiedene Art angewandt; vermutlich waren wie so oft die Informatiker im militärischen Bereich die ersten, die dazu gedrängt wurden (vgl. MCCALL, MATSUMOTO [9]). Leider ist wenig darüber publiziert, so daß man sich auf mündliche oder vertrauliche Informationen stützen muß.
Das nachfolgende Beispiel ist – übersetzt und vereinfacht – einem Vortrag von AZUMA [10] entnommen, dessen Thema «Software Quality Measurement and Assurance Technology» (SQMAT) war. Das ist eine Technik der Firma NEC.
Die angestrebten Qualitäten sind definiert und durch verschiedene Kriterien beschrieben, beispielsweise die Organisation der Fehlerbehandlung in einer Softwarekomponente (Tabelle 1). Das zugehörige Formular ist in Bild 4 dargestellt.
Wie man im Formular sieht, werden die Antworten summiert und auf den Bereich 0 bis 1 normiert. Bei einfacheren Varianten, in denen statt einer Note nur *ja* oder *nein* als Antwort zugelassen ist, wird der Anteil der positiven Antworten als Gesamtresultat bestimmt.
Die Resultate für einzelne Kriterien werden dann, unter Berücksichtigung von Gewichtungen, zu komplexeren Qualitätswerten zusammengefügt. Schließlich entsteht eine relativ anschauliche Darstellung, die auf den ersten Blick zeigt, wieweit die angestrebten Ziele erreicht wurden (Bild 5).
Das Verfahren wird bei NEC dazu verwendet, die besten Entwickler zu identifizieren; sie erhalten Prämien.

Ansatz für die praktische Arbeit

Wie das Beispiel von AZUMA zeigt, ist das Prinzip der Qualitätsschätzung die Punktebewertung durch *Gutachter* nach einem vorgegebenen Schema. Damit ergibt sich zunächst eine Menge von Einzelbewertungen, die anschließend nach dem Prinzip der algorithmischen Metriken zu Komplexen zusammengefaßt werden.
Erste Voraussetzung für den Einsatz dieses Verfahrens sind *Kriterien*, wohldefiniert und mit Gewichten versehen. Je klarer die Definitionen sind, um so kleiner wird die Streuung bei der Bewertung durch verschiedene Gutachter sein. Aber erst die Routine, also die wiederholte Anwendung der Kriterien und die Diskussion der Ergebnisse, schafft

Tabelle 1. Definition einer Prüfung.

DC040	Regeln zur Fehlerbehandlung	**Konsistenz**
Definition	Ist die Fehlerbehandlung definiert und eingehalten?	
Erklärung	Die Fehlerbehandlungsstrategie soll vor dem Entwurf festgelegt sein und dann konsequent verfolgt werden, damit es im Betrieb nicht zu inkonsistenten oder unverständlichen Reaktionen des Systems kommt.	
Einheit	Modul	
Vorgehen	1. Namen aller Module auf Formblatt notieren. 2. Jedes Modul auf einer Skala von 1 bis 4 (sehr gut) bewerten. 3. Resultate eintragen und auswerten.	
Bemerkungen		

Tabelle 2. Beispiel einer Qualitätsschätzung.

	Einzelnote n_i	Gewicht g_i	Produkt $n_i \cdot g_i$	Gruppennote q_g
Normenkonformität		**5**		
Größe der Einheiten (Prozeduren usw.)	10	2	20	
Kennzeichnung	8	1	8	
Bezeichnerwahl	5	4	20	
Gestaltung (Layout)	8	2	16	
Separierung der Literale	0	1	0	
Kommentierung	3	4	12	
Insgesamt		**14**	**76**	**5,42**
Lesbarkeit		**3**		
Datentype	8	1	8	
Ablaufstrukturen	10	1	10	
Kommentierung	2	1	2	
Insgesamt		**3**	**20**	**6,67**

Autor (Gutachter):				
Qualitätsaspekt **Regelung der Fehlerbehandlung**	Datum	Version	Visa	Seite
Kriterium **Konsistenz**	**Phase**		**Auswertung**	
	o Definition		6 mal Note 4 = 24	
			1 mal Note 3 = 3	
Name, Kennzeichen			1 mal Note 2 = 2	
des Systems bzw.	x Entwurf		2 mal Note 1 = 2	
Programms				
	o Codierung		m = 10 $n = \sum_{i=1}^{m} n_i$ = 31	
			Gesamtnote n/(4·m) = 0.77	

Einheit Modul	Bewertung 4	3	2	1	Kennzeichen des SPR	Bemerkung
1 A	x					
1 B	x					
1 C				x	*DC040-1-1*	
1 D		x				
1 E	x					
2 A	x					
2 B	x					
2 C			x			
2 D				x	*DC040-2-1*	
2 E	x					

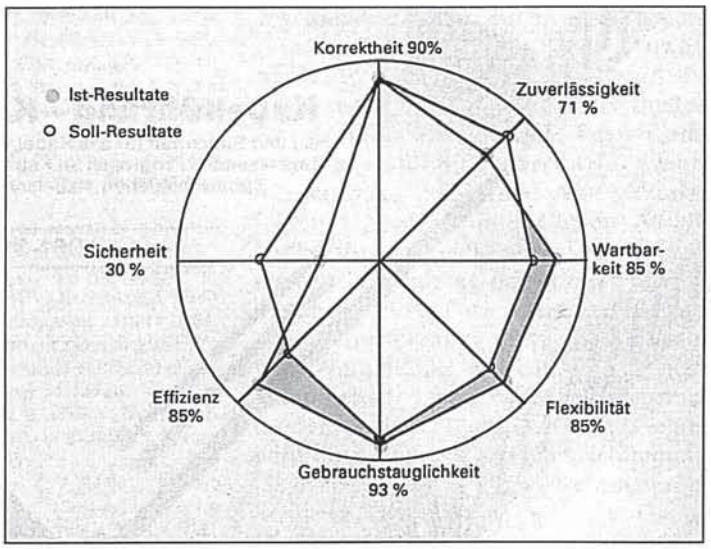

Bild 5. Radardiagramm nach Azuma.

◀ Bild 4. Ausschnitt aus dem Formular zur Qualitätsbewertung nach Azuma.

für alle Beteiligten Sicherheit und Vertrauen.

Nicht immer sind dieselben Kriterien und Gewichte sinnvoll, sie müssen projektabhängig bestimmt werden. Dabei sollten allerdings die Kriterien selbst unverändert bleiben, es geht also nur darum, das Gewicht festzusetzen, eventuell auch auf Null, falls das Kriterium keine Rolle spielt.

Kriterien sollten untereinander möglichst *orthogonal* sein, sich also gegenseitig so wenig wie möglich beeinflussen.

Die ganze Literatur über Softwarequalitätssicherung, insbesondere der Bericht von McCall und Matsumoto [9], enthält Hinweise auf geeignete Kriterien.

Beispiel für die Anwendung von Qualitätsschätzungen

Die Bewertung der Wartungsfreundlichkeit könnte beispielsweise anhand der folgenden Kriterien durchgeführt werden: Normenkonformität, Lesbarkeit, Lokalität, Typkontrolle, Testbarkeit.

Diese Kriterien müssen weiter differenziert werden, damit die wichtigsten Aspekte berücksichtigt und fair gewichtet werden:

Normenkonformität
Größe der Einheiten (Prozeduren usw.)
Kennzeichnung
Bezeichnerwahl
Gestaltung (Layout)
Separierung der Literale
Kommentierung

Lesbarkeit
Datentypen
Ablaufstrukturen
Kommentierung

Typkontrolle
Typdifferenzierung
Typbeschränkung

Lokalität
Parameterverwendung
«Information Hiding»
Ablauflokalität
Gestaltung der Außenschnittstelle(n)

Testbarkeit
Testrahmen
Testdaten
Vorbereitungen zur Testauswertung
Diagnosekomponenten
dynamische Konsistenzprüfungen

Für alle Einzelpunkte ist eine Skala anzugeben, beispielsweise 0 (extrem schlecht) bis 10 (extrem gut). Die Teilbenotungen ergeben sich dann als Mittel der Einzelbenotungen, die Gesamtbenotung als Mittel der Teilbenotungen.

Natürlich können auch durch Gewichtungen Prioritäten gesetzt werden, die Mittelung muß dann unter Berücksichtigung dieser Gewichte erfolgen.

Sind m Einzelnoten n_i vergeben worden, so ergibt sich die Durchschnittsnote q als

$$q = \frac{1}{m} \sum_{i=1}^{m} n_i$$

Werden zusätzlich Gewichtungen berücksichtigt, so tritt an die Stelle von m die Summe der Gewichte.

$$q_g = \frac{1}{M} \sum_{i=1}^{m} n_i \cdot g_i \quad \text{mit} \quad M = \sum_{i=1}^{m} g_i$$

E die beiden ersten
K r Tabelle 2 ausge-
fi

Für die Gesamtbewertung folgt aus diesen beiden Aspekten ein Beitrag von

$$\frac{5{,}42 \cdot 5 + 6{,}67 \cdot 3}{5 + 3} = 5{,}88$$

Probleme mit Schätzungen

Die naheliegende Kritik an diesem Verfahren geht von der Subjektivität aus;

dagegen hilft nur die Erfahrung. Es ist aber sicher wichtig, die Resultate der ersten Versuche nicht zu hoch zu bewerten und das Verfahren zu verbessern, bevor die Entwickler mit handfesten Konsequenzen konfrontiert werden. Das Problem wird wie bei anderen Bewertungsverfahren (Reviews) entschärft, wenn die Spielregeln allen Beteiligten klar sind und eine koordinierende Stelle, die Softwarequalitätssicherung, Mißbräuche verhindert. Die erfahrenen Entwickler sollten auch Gelegenheit haben, selbst als Gutachter zu wirken.

Voraussetzungen für die Einführung

Das beschriebene Schätzverfahren läßt sich einführen, wenn folgende Vorbedingungen erfüllt sind:

- Die Ziele und ihre Prioritäten müssen festgelegt sein.
- Die Konsequenzen der Bewertung müssen klar und bekannt sein (zum Beispiel Auszahlung von Prämien, Integration in die Mitarbeiterbewertung, Verwendung zur SQS).
- Allgemein akzeptierte Gutachter müssen bestimmt sein.
- Die Gutachter müssen die Bewertung an Spielmaterial geübt haben (Kalibrierung der Bewertungen).
- Für die Kriterienkataloge und die Gewichte muß eine regelmäßige Überprüfung eingeplant sein (Feststellung der Streuung, Diskussionen zwischen den Gutachtern, Vergleich mit anderen Daten, zum Beispiel dem tatsächlich entstehenden Wartungsaufwand).

Da es sich um einen für die meisten Leute völlig neuen Ansatz handelt, sollte man mit einem sehr bescheidenen Katalog beginnen und diesen nach den ersten Erfahrungen korrigieren und erweitern. Später sollten die *Kriterien* allerdings möglichst stabil bleiben, damit auch die Entwicklung über die Zeit festgestellt werden kann. 13 TR

Literatur

1 Boehm B. W. (1987): Industrial software metrics top 10 list. IEEE Software, September 1987, 84–85.

2 Grady R. B., Caswell D. L. (1987): Software Metrics. Establishing a company-wide program. Prentice Hall. Über die Einführung simpler Basismetriken bei Hewlett-Packard. Sehr gute Bibliographie!

3 Conte S. D., Dunsmore H. E., Shen V. Y. (1986): Software Engineering Metrics and Models. Benjamin/Cummings, Menlo Park, California.

4 Höcker H., Itzfeld W. D., Schmidt M., Timm M. (1984): Cooperative Descriptions of Software Quality Measures. GMD-Studien Nr. 81, GMD, Pf. 1240, D-5205 St. Augustin 1, Febr. 1984. Systematische Beschreibung von 50 Metriken.

5 McCabe T. J. (1976): A complexity measure. IEEE Transactions on Software Engineering, SE-2, 308–320.

6 Halstead M. H. (1977): Elements of Software Science. Elsevier North Holland, New York.

7 Lind R. K., Vairavan K. (1989): An experimental investigation of SW Metrics and their relationship to SW development effort. IEEE Transactions on Software Engineering, SE-15, 5, 649–653. Desillusionierende Studie über die Aussagekraft verschiedener Metriken.

8 Kaposi A., Kitchenham B. (1987): The architecture of system quality. IEE/BCS Software Engineering Journal 2 (1), 2–8.

9 McCall J. A., Matsumoto E. T. (1980): Report RADC-TR-80-109, vol. I: (Software Quality Metrics Enhancements) und vol. II (Software Quality Measurement Manual). Rome Air Development Center (eine sehr ergiebige, aber leider kaum zugängliche Quelle).

10 Azuma M. (1987): SQMAT – Folien eines Vortrags an der ETH Zürich über Software Quality Measurement and Assurance Technology. Unveröffentlicht.

11 Boehm B. W. (1981): Software Engineering Economics. Prentice Hall, Englewood Cliffs, N. J.