Software-Engineering 1

Softwaremetriken - Schätzverfahren

Inhalt

Artikel

Referenzen

Artikellizenzen

Schätzmethoden

Aufwandsschätzung (Softwaretechnik)

Aufwandsschätzung oder *-abschätzung* oder Kostenschätzung ist in der Softwaretechnik Bestandteil der Planung eines Softwareprojekts oder eines Arbeitspaketes. Dabei wird geschätzt, wie viele Personen und wie viel Zeit für die einzelnen Arbeitsschritte oder Programmteile notwendig sind, welche Ressourcen gebraucht werden und was es letztlich kostet. Kosten, Termine und benötigte Ressourcen sind dann Grundlage für ein Angebot oder für eine Entscheidung, ob und wie und wann ein Softwareprojekt oder Arbeitspaket davon gemacht wird.

Struktur der Kosten

Im Bereich der Softwareentwicklung sind die Hauptkosten die *Personalkosten*; die Schätzung bezieht sich daher hauptsächlich darauf.

Daneben gibt es noch *Sachkosten* (soweit nicht in den Personalkosten enthalten) wie z. B.

- benötigte Computer,
- Rechenzeiten und Netzwerkkosten,
- Lizenzen für Betriebssysteme und Tools,
- Testhardware,
- Kurse,
- Reisekosten.

Diese hängen oft von den *Personalkosten* ab, denn je länger ein Projekt dauert, je mehr Leute damit beschäftigt sind, desto mehr Sachkosten fallen auch an.

Für die zu erwartenden Gesamtkosten sind darauf noch erhebliche *kaufmännische Aufschläge* erforderlich, so für

- Realisierungsrisiko (Ein Großteil der IT-Projekte wird abgebrochen, ist nicht machbar etc.)
- Sicherheitsaufschlag für Fehleinschätzung (Eisberg-Faktor)
- Vorfinanzierungskosten
- Inflation, Personalkostensteigerung (bei länger laufenden Projekten)
- Wechselkursrisiken (bei Auslandsprojekten)

Personalaufwand

Die Schätzmethode ist abhängig von der Größe des Aufgabenumfangs (Also Schätzung vor der Schätzung).

Für die Schätzung kleinerer Aktivitäten, z. B. Arbeitspakete in einem laufenden Projekt oder Änderungen in einem bestehende System wird meistens die Schätzklausur oder Delphi-Methode benutzt, weil hier das Augenmaß der Beteiligten die besten und kostengünstigste Schätzung liefert.

Für die Schätzung sehr großer Projekte wird meist der Vergleich mit anderen sehr großen Projekten benutzt, mit Auf- und Abschlägen werden dann die Kosten für das aktuelle grob geschätzt. Durch Herunterbrechen auf einzelne Teilaufgaben wird dann der große Topf verteilt. Für diese Teilaufgaben werden dann Schätzungen erstellt. Diese können dann wieder zusammengerechnet werden.

Die Grundstruktur für die Schätzung des Personalaufwandes eines mittelgroßen Projektes ist:

```
Menge * Preis * Kompetenzfaktoren
```

Dies kann sowohl für einzelne Teile oder Phasen gemacht werden und die Kosten dann addiert werden, oder für das Gesamtprojekt.

Größenmaße (Mengen)

Gebräuchliche Größenmaße für Programme sind Lines of Code (LOC), Function Points (FP), DSI (Delivered Source Code Instructions), Dokumentseiten.

Nach Watts Humphrey (Autor von *The Personal Software Process*) sind die meisten Menschen nicht sehr gut darin, den Zeitaufwand direkt zu schätzen. Stattdessen kann man aber erstaunlich genau die Größe des Programmcodes vorhersagen. Aus den Größen der geplanten Softwarebausteine, Korrekturfaktoren für diverse Einflussgrößen und Erfahrungsdaten wird dann der zu erwartende Zeitaufwand ermittelt. Das Schätzverfahren COCOMO berücksichtigt besonders viele Einflussfaktoren.

LOC wird häufig kritisiert, da der Wert schon durch eine unterschiedliche Codeformatierung beeinflusst wird (bis zum Faktor 3). Ferner benötigen unterschiedliche Entwickler für die gleiche Funktionalität unterschiedlich viele Programmzeilen. Das LOC-Größenmaß stuft umständliche, unnötig lange Programme als aufwändiger ein als elegante kurze Lösungen. Schließlich beobachtet man in der Praxis dramatische Produktivitätsunterschiede der Entwickler. Sogenannte „Superprogrammierer“ können 10- bis 100-fach mehr LOC pro Tag erzeugen als der Durchschnitt. Es gibt Projekte, bei denen kaum eine Zeile Code geschrieben wird, sondern z. B. nur interaktive Eingaben in ein vorhandenes System erfolgen (z. B. Datenbank-Tuning). Bei Projekten, bei denen die Codierung des Quellcodes nur 7 % des Gesamtaufwands ausmacht, ist LOC kein geeignetes Maß. LOC als Schätzgrundlage muss heute eher als historisch angesehen werden.

Trotz aller Kritik ist das LOC-Größenmaß noch weit verbreitet, wahrscheinlich deshalb, weil es intuitiv und leicht zu messen ist. Insbesondere im Nachhinein werden mitunter LOC (wozu dann auch Scripts und Test-Code zählen) als Maß für den Umfang einer Software und als Maß für die Leistung der Entwickler herangezogen. Davor ist zu warnen: Wer LOC als Maß für Leistung nimmt, wird viele davon bekommen, mit allen negativen Auswirkungen auf Einfachheit, Wartbarkeit, Performance, Zuverlässigkeit.

Statt LOC wurden später oft auch DSI (delivered source instructions) genommen. Das scheint vernünftiger, aber die prinzipiellen Probleme bleiben. Kommentare, die erheblich zu Qualität beitragen, zählen nicht mit.

Die Function-Point-Methode (nach Allen J. Albrecht, IBM) geht nicht vom zu erwartenden Code aus, sondern von den Anforderungen und versucht, die Anzahl und Komplexität von Datenstrukturen und Funktionen/Methoden zu schätzen, indem diese als einfach/normal/schwierig klassifiziert und mit entsprechenden Aufwandsfaktoren versehen werden. Diese Mengenfaktoren werden dann für Erschwernisse korrigiert. Die Function-Point-Methode ist im kommerziellen Umfeld weit verbreitet. Es wurde versucht, sie an andere Umfelder anzupassen.

In Fällen, wo das Hauptergebnis in Textdokumenten besteht wird auch häufig die zu erwartenden Seiten Papier als Maß benutzt und mit 1PT/Seite bewertet. (Z. B. Studien, Analysen, Pflichtenhefte)

In anderen Fällen (z. B. Pflichtenheft) wird auch die Anzahl von Sitzungen eines Gremiums herangezogen und daraus der Zeitaufwand einschließlich Vorbereitung und Nacharbeiten aller Beteiligter errechnet.

Zerlegung in Komponenten

Für eine gute Aufwandsschätzung ist es erforderlich, dass man gut verstanden hat, was gefordert wird, dass der Leistungsumfang genau definiert, und Dinge, die zwar sinnvoll, nützlich oder nahe liegend aber nicht gefordert sind, explizite ausgeschlossen werden. Zwingende Notwendigkeit ist das Vorliegen eines Pflichtenheftes (Requirement Specification), das ggf. noch weiter präzisiert werden muss. Darauf basierend kann eine Liste der zu liefernden Komponenten und der im Projektverlauf zusätzlich zu erstellenden Hilfskomponenten erstellt werden. (Insbesondere Scripts, Tests, Diagnose-Hilfen) Es kann dann der Umfang oder Aufwand für jede Komponente einzeln geschätzt werden. Unter der Annahme, dass die jeweiligen Schätzfehler voneinander statistisch unabhängig sind, heben sich

die Schätzfehler für die einzelnen Komponenten teilweise gegenseitig auf.

Hierbei ist jedoch zu beachten, das systematische Schätzfehler, wie zum Beispiel ein generelles Unterschätzen von Aufwänden durch Komponenten-basiertes Schätzen, nicht behoben werden. Ferner stellt sich oft heraus, dass im Laufe des Projekts noch Komponenten benötigt werden, die gar nicht geschätzt wurden. Um solchen systematischen Schätzfehlern zu begegnen, kann man Schätzungen von alten Projekten mit den tatsächlichen Projektgrößen korrelieren und aus dieser Korrelation einen entsprechenden Korrekturfaktor für den systematischen Schätzfehler ableiten. (Eisbergfaktor)

Berücksichtigung von Erschwernissen und Restriktionen

Soweit nicht schon bei der Komponentenschätzung berücksichtigt müssen Erschwernisse oder Restriktionen durch Korrekturfaktoren berücksichtigt werden. Bei der COCOMO-Methode sind 14 solcher Faktoren angegeben, die aus statistischer Auswertung von vielen Projekten gewonnen wurden. Manche sind heute irrelevant (z. B. Turnaround-zeit im Closed-Shop-Betrieb), die meisten aber noch nützlich. Der wichtigste ist die Qualität der Entwickler. Er ist bei Böhm mit einer Spanne von 4.2 zwischen bestem und schlechtestem Team angegeben und dürfte heute noch höher sein.

Schätzverfahren

Harry. W.Boehm, der Vater von COCOMO nennt auch noch einige andere Schätzverfahren:

Analogie

Man sucht nach ähnlichen Projekten und nimmt mit Auf- und Abschlägen für dies und jenes deren Kosten. Vorteil: Bezug auf reale Kosten. Nachteil: Unterschiede in Aufgabe, Systemumgebung, Personal. Geeignet für Gesamtsicht, wo man sich sonst in Details verlieren würde und für Plausibilisierung.

Parkinson-Verfahren

Parkinson sagt: „Arbeit dehnt sich aus so weit es geht:“ Wenn man Budget und den Endtermin kennt, ergibt sich daraus wie viele Leute man einsetzen kann und was es kostet. Dieses Verfahren ist aus der Praxis bekannt: Wenn man zu früh fertig ist, macht man Verschönerungen und testet mehr. Wenn man nicht fertig wird, aber das Budget erschöpft oder der Endtermin erreicht ist, wird der erreichte Zustand als fertig erklärt.

„Price-to-Win“

Aus diversen Quellen weiß man, was der Kunde bereit ist zu zahlen, oder was ein Konkurrent anbietet; meist weit weniger als eine solide Arbeit kosten würde. Die Schätzung wird solange frisiert, bis sie zur Erwartung passt. Boehm schreibt weiter dazu: „The price-to-win technique has won a large number of software contracts for a large number of companies. Almost all of them are out of business today.“ („Die Price-to-Win-Technik hat vielen Softwarefirmen eine große Zahl an Aufträgen verschaffen können. Die meisten dieser Firmen sind heute nicht mehr im Geschäft.“)

Delphi-Methode oder Schätzklausur

Alternativ oder zusätzlich kann die Aufwandsschätzung nach der Delphi-Methode verbessert werden. Dabei werden einfach mehrere Personen gebeten, unabhängig voneinander Schätzungen abzugeben. Die Hoffnung ist wieder, dass sich Schätzfehler bei der Mittelwertbildung gegenseitig ausgleichen. Zusätzlich besteht bei der Delphi-Methode die Möglichkeit, die Schätzer über stark voneinander abweichende Schätzungen diskutieren zu lassen. Dabei kann z. B. aufgedeckt werden, dass das Gros der Schätzer einen Problemaspekt übersehen und deswegen den Aufwand unterschätzt hat. Ebenso kann es sein, dass ein Schätzer eine Realisierungsidee hat, die einen wesentlich geringen Aufwand erfordert. Wichtig ist dabei, dass sich die Beteiligten über den Schätzgegenstand klar sind, also z. B.

Netto-Zeitaufwand für eine Programm-Änderung, Bruttozeitaufwand für Änderung samt Test, Dokumentationsänderung und Benutzerinformation. Die Schätzklausur arbeitet so ähnlich, nur weniger formalisiert. Die Experten schätzen hier nicht getrennt voneinander sondern im Kollektiv. Es werden die 3 Phasen Vorbereitung, Durchführung und Nachbereitung durchlaufen.

COCOMO

Die Grundidee der COCOMO-Methode „Constructive Cost Modell“ besteht darin, alle kostenrelevanten Elemente zu erfassen, zu bewerten und hochzurechnen. Die Bewertung stützt sich dabei auf Erfahrungswerte, die aus einer Erfahrungsdatenbank gewonnen wurde, in die eine Vielzahl von Projekten eingetragen wurden. Diese Erfahrungsdatenbank basiert auf DSI und einer Systemumgebung aus der Lochkartenzeit. Die Formeln für den Basisaufwand sind daher heute nicht mehr brauchbar, das Grundkonzept mit passenden Bezugsgrößen sehr wohl.

Function Point

Das Function-Point-Verfahren wurde von Allen J. Albrecht bei IBM entwickelt als Fortentwicklung des früheren Verfahrens. Es ist gedacht für kommerzielle Programme, die Eingabedaten bearbeiten unter Benutzung von Stammdaten und Referenzdaten und Ausgabedaten erstellen. Function Points gibt es für Datenstrukturen (je nach Komplexität), für Programme (je nach Schwierigkeit) für Referenzdaten. Ferner gibt es Korrekturfaktoren (Einflussfaktoren), sowie einen von der Projektgröße abhängigen Umrechnungsfaktor von FP in PM (Personenmonate). Die Function Point Methode basiert auf den funktionalen Anforderungen und ist im Prinzip unabhängig von der verwendeten Programmiersprache.

Es gibt eine internationale Benutzergruppe: http:// / www. ifpug. org Es gibt Bestrebungen, Function Point als ISO-Standard zu etablieren.

Beurteilung von Schätzverfahren

An Schätzverfahren werden neben Genauigkeit noch eine Reihe weiterer z. T. sich widersprechende Anforderungen gestellt:

- Es soll möglichst früh einsetzbar sein (Der Auftraggeber will am Anfang wissen, was es am Ende kostet)
- Änderungen in den Anforderungen sollen sich auf die Schätzung auswirken (Mehr-/ Minderkosten)
- Die Ergebnisse sollen einfach nachvollziehbar sein
- Außer den Kosten soll auch ein grober Terminplan herauskommen, der den Übergang zur Projektplanung ermöglicht
- Das Schätzverfahren selbst soll möglichst wenig kosten

Genauigkeit

Eine gute Aufwandsschätzung sollte auch immer mit Angaben zur Genauigkeit der Schätzung einhergehen. Solche Angaben können wieder aus der statistischen Betrachtung von früheren Schätzungen abgeleitet werden. Im Softwarebereich geht man bei einfachen Schätzansätzen von einer Genauigkeit zwischen Pi und 10 aus. Das heißt, bei einem geschätzten Aufwand von einem Personenjahr liegt der wahrscheinliche Aufwand zwischen 70 Tagen und 10 Jahren. Durch Komponentenschätzung und Delphi-Methode kann dies oft auf einen Schätzfehler in der Größenordnung des Faktor 2 verbessert werden. Humphrey berichtet in seiner Probe-Methode in sehr günstigen Fällen sogar von einem Schätzfehler von nur 15 % in 75 % Prozent der Projekte. Dies gelingt aber nur, wenn eine große Anzahl von ähnlich gelagerten Vergleichsprojekten zur Verfügung steht und wenn sich bei dem neuen Projekt kein entscheidender Einflussfaktor geändert hat. Neues Personal, neue technische Anforderungen, neue Entwicklungswerkzeuge, neue Laufzeitumgebungen, neue Kunden oder ähnliche Risiken können leicht wieder zu einem Schätzfehler von mehreren Größenordnungen führen.

Es gibt dabei auch ein Paradoxon: Je mehr „Faktoren“ (nicht Summanden) berücksichtigt werden, umso plausibler ist das Ergebnis, weil Faktoren, die offensichtlich einen großen Einfluss haben berücksichtigt werden. Allerdings wird die Schätzgenauigkeit dabei verschlechtert, denn wenn z. B. 10 Faktoren um einen Faktor 1.05 zu optimistisch oder pessimistisch geschätzt werden, so macht das einen Faktor 1.6 aus. Softwareentwickler tendieren stark zu Optimismus. Verantwortliche auch wegen des „price-to-win“.

Kosten des Schätzverfahrens

Aufwände entstehen

- für das Lesen und Verstehen der Anforderungen meist für mehrere Personen
- für die Definition von Leistungen, Einschränkungen, Restriktionen
- für die eigentliche Schätzung
- für das Verkaufen der Schätzung, Nacharbeiten, Pflege von Schätz-Datenbanken

Schätzen ist oft ein iteratives Verfahren, indem Leistungen reduziert oder ergänzt werden, Annahmen korrigiert werden.

Die Kosten für eine gute Schätzung betragen nach manchen Angaben bis zu 3 % des Projektumfangs, die für ein gutes Angebot bis zu 5 % des Projektumfangs.

Damit wird die Schätzung oft schon selbst zum Problem: Bei 10 Anbietern und 5 % Aufwand je Anbieter machen schon die Angebotskosten 50 % des Projektumfangs aus. Aus Sicht des Anbieters muss er aber bei einem erfolgreich akquirierten Projekt die Angebotskosten für alle 9 anderen wieder mit herein holen. Er müsste dazu ziemlich teuer sein, was es unwahrscheinlich macht, dass er den Auftrag bekommt. Dies legt es nahe, eine ehe grobe Schätzung mit hohem Aufschlag zu machen um Schätzaufwand (und dadurch auch Schätzkosten) gering zu halten.

Literatur

- Manfred Burghardt: *Projektmanagement: Leitfaden für die Planung, Überwachung und Steuerung von Entwicklungsprojekten.* 7. Auflage. Publicis Corporate Publishing, Erlangen 2006, ISBN 3-89578-274-2, S. 156ff.
- Siegfried Vollmann: *Aufwandsschätzung im Software Engineering*. IWT-Verlag, 1990, ISBN 3-88322-277-1.
- Barry W. Boehm: *Software Engineering Economics*. Prentice Hall, 1981, ISBN 0-13-822122-7.
- IBM Deutschland: *Die Funktion Point Methode, Schätzmethode für IS-Anwendungs-Projekte*. Formnr. E12–1618, 1985.
- F. P. Brooks: *Vom Mythos des Mann-Monats*. Addison-Wesley 1987, Übersetzung der Originalausgabe von 1975, ISBN 3-925118-09-8.
- Harry M. Sneed: *Software-Projektkalkulation.* Hanser, 2005.
- Steve McConnell: *Software Estimation: Demystifying the Black Art*. Microsoft Press, 2006, ISBN 978-0-7356-0535-0.

Weblinks

- Methodische Aufwandsschätzung aus Sicht eines agilen Projektmanagements [1] Hans-Jürgen Plewan, White Paper; publiziert in B. Oestereich (Hrsg.), Agiles Projektmanagement, dpunkt verlag 2006 S. 83–100

Siehe auch

- Softwaremetrik
- Testaufwand

Referenzen

[1] http://www.f-i-solutions-plus.de/spaw/uploads/files/whitepaperagilespm.pdf

Schätzmethode

Schätzmethoden dienen zur Ermittlung, welche Ressourcen in welchem Umfang für die Produkterstellung notwendig sind.

Wenn ein neues Produkt erstellt werden soll, wird in der Planungsphase die Produktbeschreibung erstellt. Zur Umsetzung müssen hierzu sogenannte Ressourcen bereitgestellt werden, die die Realisierung des oder der Produkte ermöglichen. Bei diesen Ressourcen kann es sich um Werkzeuge, Material, Finanzen, Maschinen oder Menschen handeln.

Im klassischen Arbeitsumfeld wie in projektorientierten Vorgehensweisen werden immer mehr sogenannte standardisierte Methoden zur Schätzung von Aufwänden verwendet. Sie

- sichern, dass die zu leistende Arbeit komplett erfasst wird
- ermitteln, welche Aufwände benötigt werden, um die zuvor festgelegten Ziele zu erreichen
- vermindern das Risiko von Fehleinschätzungen, bei denen die geplanten Mittel zu hoch oder zu gering dimensioniert werden.

Klassifizierung der Schätzmethoden

- Verfahren **ohne** ausdrückliche Angabe der Einflussgrößen:

  Methoden der Expertenbefragungen

  - Expertenschätzungen

    hilfreich bei Projekten mit hohem Neuheitsgrad ohne Erfahrungen aus ähnlichen Projekten

  - Einzelschätzung
  - Schätzklausur
  - Delphi-Methode
  - Zwei-Zeiten-Methode (Best Case - Worst Case)
  - Drei-Zeiten-Methode (auch "Beta-Methode")
  - Analogie-Methode (auch "Best-Practice-Methode")
- Verfahren **mit** ausdrücklicher Angabe der Einflussgrößen:

  Basieren auf systematischer Auswertung von Daten aus abgeschlossenen Projekten

  - Prozentsatzmethode
  - Kennziffern und Kennziffernsysteme auf Basis von Standardstrukturen (zum Beispiel Baubranche, IT-Bereich):

    Projekte oder bestimmte Komponenten ohne hohen Neuheitsgrad

- parametrische Schätzverfahren (zum Beispiel Rüstungsbereich, IT-Bereich):
- Modell COCOMO,
- Function-Point-Methode

Delphi-Methode

Die **Delphi-Methode** (auch Delphi-Studie, Delphi-Verfahren oder **Delphi-Befragung** genannt) ist ein systematisches, mehrstufiges Befragungsverfahren mit Rückkopplung bzw. eine Schätzmethode, die dazu dient, zukünftige Ereignisse, Trends, technische Entwicklungen und dergleichen möglichst gut einschätzen zu können.

Namensgeber der Methode ist das antike Orakel von Delphi, das seinen Zuhörern Ratschläge für die Zukunft erteilte.

Geschichte

Die Delphi-Methode wurde – nach Vorarbeiten Ende der 1950er Jahre – von der amerikanischen RAND-Corporation 1963 entwickelt[1] und wird seitdem häufig, wenn auch in variierter Form, für die Ermittlung von Prognosen/Trends sowie für andere Meinungsbildungen im Rahmen von Systemaufgaben angewendet. Mehr und mehr hat sich das Verfahren zu einem Bewertungsverfahren für Themen entwickelt, in dem festgestellt werden kann, ob es einen Konsens über das Thema gibt (bzw. ob dieser erreicht werden kann) oder nicht. In Deutschland war es in den 90er Jahren das damalige Bundesministerium für Forschung und Technologie (BMFT), das die ersten Delphi-Studien zur Entwicklung von Wissenschaft und Technik in Auftrag gab. Die Studien *Technologie am Beginn des 21. Jahrhunderts* (1991–1992) und *Deutscher Delphi-Bericht zur Entwicklung von Wissenschaft und Technik* (1993) wurden vom Fraunhofer-Institut für System- und Innovationsforschung (ISI) durchgeführt.

Vorgehensweise

Bei einer Delphi-Befragung wird einer Gruppe von Experten ein Fragen- oder besser: Thesenkatalog des betreffenden Fachgebiets vorgelegt. Die Experten (Expertise wird sehr breit definiert) haben in zwei oder mehreren sogenannten *Runden* die Möglichkeit, die Thesen einzuschätzen. Ab der zweiten Runde wird Feedback gegeben, wie andere Experten geantwortet haben, in der Regel anonym. Auf diese Weise wird versucht, den üblichen Gruppendynamiken mit sehr dominanten Personen entgegenzuwirken.

Die in der ersten Runde schriftlich erhaltenen Antworten, Schätzungen, Ergebnisse etc. werden daher aufgelistet und beispielsweise mit Hilfe einer speziellen Mittelwertbildung, Perzentilen oder Durchschnittswertberechnungen zusammengefasst und den Fachleuten anonymisiert erneut für eine weitere Diskussion, Klärung und Verfeinerung der Schätzungen vorgelegt. Dieser kontrollierte Prozess der Meinungsbildung erfolgt gewöhnlich über mehrere Stufen. Das Endergebnis ist eine aufbereitete Gruppenmeinung, die die Aussagen selbst und Angaben über die Bandbreite vorhandener Meinungen enthält.

Der Meinungsbildungsprozess enthält die Elemente: Generation, Korrektur bzw. teilweise Anpassung oder Verfeinerung, Mittelwertbildung bzw. Grenzwertbildung, oft auch offene Felder für Erläuterungen. Störende Einflüsse werden durch die Anonymisierung, den Zwang zur Schriftform und der Individualisierung eliminiert. Die Strategie der Delphi-Methode besteht aus: Konzentration auf das Wesentliche, mehrstufigem, teilweise rückgekoppeltem Editierprozess und sichereren, umfassenderen Aussagen durch Zulassen statistischer fuzzyartiger Ergebnisse. Ein häufiges Problem besteht darin, dass die Experten ihre einmal geäußerte Meinung in den folgenden Runden trotz Anonymität nicht ändern, so dass der Zusatznutzen weiterer Runden oft klein ist.

Als Ergänzung der Delphi Methode kann z. B. die Cross-Impact-Matrix-Methode verwendet werden. Auch in der D2-Methode finden sich Elemente der Delphi-Methode wieder. Kombinationen mit Szenarien werden inzwischen ebenfalls erprobt. Aus Delphi-Ergebnissen lassen sich einfache Roadmaps ableiten, so dass auch diese Kombination

sich zunehmender Beliebtheit erfreut.

Man findet diverse Formen der Delphi-Methode, die das Verfahren der Schätzung etwas variieren: die Standard- und die Breitband-Methode sind einige der Varianten. Inzwischen werden die meisten Verfahren elektronisch durchgeführt. Realtime-Delphi-Verfahren (mit einer sofortigen Rückkopplung der Ergebnisse) sind eine Variante, die nur elektronisch möglich ist.

Standard-Delphi-Methode

Bei der Standard-Delphi-Methode werden mehrere Experten zur Schätzung eines Projektes – oder zur Prognostizierung – herangezogen, die sich nicht untereinander abstimmen dürfen. Der Prozess sieht wie folgt aus:

- Ein Projektleiter bereitet eine Projektbeschreibung vor, in der die einzelnen Teil-Produkte aufgelistet sind und bereitet sie in einem Arbeitsformular vor.
- Der Projektleiter stellt die Ziele des Gesamtprojektes vor und verteilt je ein Exemplar des Arbeitsformulars an jeden Experten. Es findet keine Diskussion der Schätzungen statt.
- Jeder Experte schätzt die im Arbeitsformular enthaltenen Arbeitspakete. Keiner der Experten arbeitet mit einem anderen Experten zusammen.
- Alle Arbeitsformulare werden vom Projektleiter gesammelt und ausgewertet.
- Ergeben sich gravierende Diskrepanzen, so werden diese vom Projektleiter einheitlich auf allen Arbeitsformularen in Bezug auf die Abweichung nach oben oder unten kommentiert. Jedes Arbeitsformular geht anschließend an seinen ursprünglichen Bearbeiter wieder zurück.
- Die Experten überdenken in Abhängigkeit von den Kommentaren ihre Schätzungen.
- Die beschriebene Schleife wiederholt sich so lange, bis sich in den Schätzungen unabhängig voneinander (in einem Toleranzbereich) Konsens einstellt.
- Von allen Schätzungen werden die Mittelwerte errechnet und als finale Schätzung präsentiert.

Das Fehlen jeglicher Diskussionen hat zwei Aspekte, die ein Projektleiter bewerten muss: Einerseits wird damit verhindert, dass sich aufgrund einer ungewollten Gruppendynamik Strömungen und Tendenzen in den Meinungen herausbilden, die unter Umständen gute Schätzungen verhindern. Auf der anderen Seite könnten Gruppendiskussionen dazu beitragen, Defizite im Know-how einzelner Experten und die damit verbundenen Fehleinschätzungen zu vermeiden.

Häufig werden Delphi-Umfragen schriftlich und getrennt durchgeführt, die Fragebogen werden also den Experten per Brief oder Mail gesandt. Die einzelnen Experten sehen sich nie und wissen auch erst nach Abschluss aller Umfragerunden die Namen der anderen Befragten. Dieses Vorgehen ist zuverlässiger als das Versammeln aller Experten in einem Raum. Liegt der Schlussbericht einmal vor, werden in der Regel alle Experten und andere Interessierte zu einem Symposium eingeladen.

Breitband-Delphi-Methode

Bei der Breitband-Delphi-Methode werden mehrere Experten zur Schätzung eines Projektes herangezogen, die sich untereinander abstimmen dürfen. Der Prozess sieht wie folgt aus:

- Ein Projektleiter bereitet eine Projektbeschreibung vor, in der die einzelnen Teil-Produkte aufgelistet sind und bereitet sie in einem Arbeitsformular vor.
- Der Projektleiter stellt die Ziele des Gesamtprojektes vor und verteilt je ein Exemplar des Arbeitsformulars an jeden Experten. Es findet eine Diskussion der Arbeitspakete unter den Experten statt, in der die Sicht der einzelnen Experten den anderen Teilnehmern in Bezug auf das Gesamtprojekt und die Teilaufgaben vermittelt werden.
- Anschließend schätzt jeder Experte die in seinem Arbeitsformular enthaltenen Arbeitspakete. Keiner der Experten arbeitet dabei mit einem anderen Experten zusammen.

- Der Projektleiter fasst die einzelnen Schätzaussagen zusammen, er begründet allerdings die Angaben und Unterschiede nicht. Die Ergebnisse werden an alle Experten verteilt.
- Der Projektleiter beruft ein neues Meeting mit den Experten zusammen und spricht die größten Diskrepanzen in den Schätzungen an. Jedes Arbeitsformular geht anschließend an seinen ursprünglichen Bearbeiter wieder zurück.
- Die Experten überdenken in Abhängigkeit von den angeführten Abweichungen ihre Schätzungen.
- Die beschriebene Schleife wiederholt sich so lange, bis sich in den Schätzungen unabhängig voneinander (in einem Toleranzbereich) Konsens einstellt.
- Von allen Schätzungen werden die Mittelwerte errechnet und als finale Schätzung präsentiert.

Durch die Wechselwirkungen der Experten untereinander werden unterschiedliche Ansichten vermittelt, was eine Konsens-Bildung beschleunigt. Vorteil dieser Methode ist zum einen die Anonymität der Schätzungen: Die Experten werden nicht mit ihren gravierenden Abweichungen der Schätzungen konfrontiert und können damit die Schätzaufwände in ihrem Sinne beeinflussen. Starke Abweichungen von Mittelwerten werden offengelegt. Nachteil dieser Methode ist die Gefahr einer Meinungsbildung durch die Gruppendynamik, in der eine unter Umständen notwendige gravierende Schätzabweichung dem Gruppenzwang unterliegt. Ein weiterer Nachteil ist, dass aufgrund mehrerer Iterations-Schleifen für die Meinungsbildung der gesamte Schätzaufwand recht umfangreich werden kann. Die Breitband-Delphi-Methode ist eine sinnvolle Technik für das Schätzen von großen Projekten, in denen komplexe Architekturen durch eine große Expertenrunde mit Hilfe der Interaktion der Experten untereinander zu realistischen Werten führen kann.

Kritik

Die Delphi-Methode versucht, durch das mehrstufige, manchmal auf Konsens angelegte Design, Fehleinschätzungen der Experten zu reduzieren. Dennoch lassen sich nicht alle Probleme der Expertenbefragung vermeiden; durch die Befragung mehrerer Personen entstehen weitere Einschränkungen.

- Themen bzw. Thesen müssen zunächst formuliert werden, bevor sie das zweistufige Verfahren durchlaufen können. In manchen Fällen werden die Thesen zwar im Verfahren selbst erarbeitet, in der Regel sind hierzu jedoch weitere Methoden notwendig.
- Die Thesen müssen kurz, prägnant, aber eindeutig formuliert sein. Dies kann ein Vorteil sein, zwingt es doch die Teilnehmer zur Konzentration auf das Wesentliche. Methodisch können aber nur bedingt komplexe Themenstellungen bewertet werden.
- Experten konzentrieren sich per Definition im Wesentlichen auf ihren Expertise-Bereich. Die Interdependenzen mit anderen Entwicklungen, die v. a. bei breit angelegten Studien wichtig sind, werden häufig vernachlässigt oder müssen nachgearbeitet werden.
- Werden relevante Rahmenbedingungen (z. B. soziale Entwicklungen bei der Prognose der technischen Entwicklung der Mobilkommunikation) beachtet, so kann man sich nicht darauf verlassen, dass die Befragten hierfür dieselbe zuverlässige Expertise besitzen wie in ihrem eigentlichen Expertisebereich.
- Experten neigen dazu, die Geschwindigkeit von Entwicklungen zu überschätzen. Vor allem die Diffusionsgeschwindigkeit einer Innovation in der Gesellschaft wird schnell überschätzt.
- Bei der Befragung einer Gruppe entsteht eine soziale Situation. Hierbei können durch Autorität, oder auch aufgrund persönlicher Grabenkämpfe, Verzerrungen entstehen. So ist nicht immer klar, ob ein Konsens (oder ein Dissens) tatsächlich nur auf dem intensiven Hinterfragen der eigenen Meinung beruht. Eine Anonymisierung in der Feedback-Runde kann diese Probleme i.A. nicht vollständig vermeiden. Bei der Verwendung von Fragebögen (E-Mail oder postalisch) wird das Delphi-Verfahren explizit dazu genutzt, diese Dominanzen zu umgehen. Bei Präsenzrunden ist dies nur bedingt möglich.

Siehe auch

- Delphi-Effekt
- Cooke-Methode, ähnlich der Delphi-Methode, jedoch werden die Expertenmeinungen mit dem Fähigkeitsgrad der Experten gewichtet, um einen andere Schätzung zu erhalten und um die Gefahr der Tendenz zum Mittelwert auszuschließen.[2]
- Strategische Frühaufklärung
- Zukunftsforschung

Literatur

- Michael Häder (Hrsg.): *Delphi-Befragungen. Ein Arbeitsbuch*. Westdt. Verlag, Wiesbaden 2002, ISBN 3-531-13748-4.
- Ammon, Ursula. (2005). *Delphi-Befragung. Quantitative Organisationsforschung.* Qualitative-Research.net, Online-Portal für qualitative Sozialforschung, Freie Universität Berlin. Online verfügbar auf [3]
- von der Gracht, H. A./ Darkow, I.-L.: *Scenarios for the Logistics Service Industry: A Delphi-based analysis for 2025*. In: International Journal of Production Economics, Vol. 127, No. 1, 2010, 46-59. - Ein Beispiel für wissenschaftlich fundierte Delphibasierte Szenarioentwicklung
- Steinmüller, Karlheinz. (1997): *Grundlagen und Methoden der Zukunftsforschung: Szenarien, Delphi, Technikvorausschau.* Werkstattbereicht 21, SFZ. Online verfügbar bei Freie Universität Berlin, Arbeitsbereich Erziehungswissenschaftliche Zukunftsforschung, auf [4]
- Thomas Seeger: *Die Delphi-Methode. Expertenbefragungen zwischen Prognose und Gruppenmeinungsbildungsprozessen. Überprüft am Beispiel von Delphi-Befragungen im Gegenstandsbereich Information und Dokumentation*. Hochschulverlag, Freiburg i. Br. 1979, ISBN 3-8107-2024-0 (=Dissertation, Freie Universität Berlin).
- USAF Project RAND Report *Delphi Assessment: Expert Opinion, Forecasting and Group Process* (pdf) [5]

Sekundärliteratur zur Delphi-Methode:

Delphi-Studien:

- Höller, A.: *Das 21. Jahrhundert – Das Problem von Voraussagen durch Delphi-Studien und dynamische langfristige Betrachtung.* Kassel 2002
- Keller, A.: *Elektronische Zeitschriften im Wandel – eine Delphi-Studie.* Wiesbaden 2001
- Krauß-Leichert, U.: *Einsatz neuer Technologien im Bibliothekswesen – eine Expertenbefragung.* München, K. G. Sauer, 1990
- Linstone, H. A.: *The Delphi-Method – Techniques & Applications.* Massachusetts 1975
- Rauch, W. und Wersig G.: *Delphi-Prognosen in Information und Dokumentation.* München 1978
- Ullrich, K. und Wenger, C.: *Vision 2017 – Was Menschen morgen bewegt.* Heidelberg 2008, ISBN 978-3-636-01582-2
- Wissen, D.: *Bibliographie der Zukunft – Zukunft der Bibliographie – eine Expertenbefragung mittels Delphi-Technik in Archiven und Bibliotheken in Deutschland, Österreich und der Schweiz.* Berlin 2007

Weblinks

- Hintergrund des BMBF-Foresight-Prozesses [6]
- Foresight-Studien des Forschungsprojekts FAZIT [7]

Referenzen

[1] vgl. Hüttner, Manfred:Markt- und Absatzprognosen (1982) S.29
[2] Für eine beispielhafte Anwendung der Cooke-Methode mit Erklärung siehe Nature, Ausgabe 463, Seiten 294-295.
[3] http://www.qualitative-research.net/organizations/2/or-db-d.htm
[4] http://www.institutfutur.de/_service/download/methoden-zukunftsforschung_sfz-wb21.pdf
[5] http://www.rand.org/pubs/reports/2006/R1283.pdf
[6] http://www.bmbf.de/_search/searchresult.php?URL=http%3A%2F%2Fwww.bmbf.de%2Fde%2F12687.php&QUERY=delphi
[7] http://fazit-forschung.de/fazit-zukunft0.html

Zwei-Zeiten-Methode

Die **Zwei-Zeiten-Methode** ist eine rudimentäre Schätzmethode im Projektmanagement.

Die Abschätzung von Zeit und Aufwand für ein Arbeitspaket in einem Projekt ist häufig problematisch:

- Da meist noch unzureichendes Wissen über den Inhalt des Arbeitspaketes existiert ist die Schätzung zwangsläufig mit einer Unsicherheit behaftet
- Diese Unsicherheit ist umso größer, je größer der Entwicklungs- und Forschungsanteil des Paketes ist (s. Cone of Uncertainty)
- Dennoch wird diese Schätzung anschließend häufig als feste "Deadline" betrachtet
- Projektmitarbeiter sind meist nur widerstrebend dazu bereit, eine Schätzung von Zeit und Aufwand abzugeben, da sie sich über den genauen Umfang noch nicht im Klaren sind und auch nicht sein können
- Bei neuartigen Aufgaben tendiert man dazu, den nötigen Aufwand zu positiv zu schätzen

Diese Probleme können durch die Zwei-Zeiten-Methode gemildert werden, indem für ein Arbeitspaket jeweils ein Best- und ein Worst-Case geschätzt werden. Die "wahre" Schätzung wird sich mit hoher Wahrscheinlichkeit innerhalb der geschätzten Unschärfe befinden. Da die Unsicherheit im Projektverlauf kleiner wird (s. Cone of Uncertainty) müssen die Schätzungen vom Projektleiter in regelmäßigen Abständen überarbeitet werden. Da die meisten PM-Tools keine Unsicherheiten verarbeiten können, kann für die Planung der Mittelwert

verwendet werden. Dieses sehr einfache Verfahren wird in der Drei-Zeiten-Methode durch eine zusätzliche Gewichtung verfeinert.

Drei-Zeiten-Methode

Auch 3-Punkt-Schätzung genannt.

Die Drei-Zeiten-Methode ist eine Methode zur Schätzung von Zeit oder Aufwand im Projektmanagement. Sie erweitert die Zwei-Zeiten-Methode durch eine zusätzliche Gewichtung der Schätzung von Best-Case und Worst-Case. Dies ist insbesondere dann sinnvoll, wenn davon auszugehen ist, dass die tatsächliche Zeit näher an der minimalen oder der maximalen Schätzung liegen wird. Die Ursachen hierfür können zum Beispiel sein:

- Das Projekt ist extrem risikobehaftet, sodass das Eintreten des Best-Case nur sehr unwahrscheinlich ist
- Der schätzende Mitarbeiter ist erfahrungsgemäß sehr pessimistisch oder optimistisch

In diesen Fällen empfiehlt sich die zusätzliche Schätzung des wahrscheinlichsten Falles (Likely Case) zusätzlich zur Minimalschätzung (Best-Case) und Maximalschätzung (Worst-Case). Der Planwert kann über die Formel

$$\frac{BestCase + 4 * LikelyCase + WorstCase}{6}$$

errechnet werden. Der wahrscheinliche Fall erhält auf diese Weise eine starke Gewichtung. Auch bei dieser Methode ist es jedoch unabdingbar, dass der Projektleiter die Schätzungen regelmäßig aktualisiert, da die Unsicherheit im Projektverlauf sinkt (s. Cone of Uncertainty)

Best Practice

Der Begriff **Best Practice** (wörtlich: *bestes Verfahren*, freier: *Erfolgsrezept*), auch **Erfolgsmethode** genannt, stammt aus der angloamerikanischen Betriebswirtschaft. Wenn ein Unternehmen nach *best practice* vorgeht, setzt es bewährte und kostengünstige Verfahren, technische Systeme und Geschäftsprozesse ein, die es zumindest auf wesentlichen Arbeitsfeldern zum Musterbetrieb für andere machen.

Orientierung am Besten

Diese Aussage ist nach einem Benchmarking möglich, wenn sich mehrere vergleichbare Unternehmen ausgetauscht haben, um den oder die Besten dieser Gruppe herauszufinden. Mit der Orientierung an „Best Practice“ wollen die schwächeren Unternehmen die eigenen Dienstleistungen, Produkte, Projekte, Methoden und Systeme mit der praxistauglichen Elle der anderen messen, bewerten und gegebenenfalls durch neue Zielsetzungen verbessern. Voraussetzung des Erfolgs ist, die Prozessstruktur aus dem Best-Practice-Unternehmen vollständig zu übertragen. Daran scheitern halbherzige Veränderungen.

Um ein Umkrempeln ihres Betriebes zu vermeiden, geben sich Manager neuerdings mit **Good Practice** zufrieden. Dies bedeutet die Realisierung punktueller Maßnahmen, die den Unternehmenserfolg wenigstens in Teilgebieten deutlich verbessern und einen maßvollen Umbau mit dem Verzicht auf das Anstreben einer Spitzenleistung um jeden Preis verbinden.

Wer sich als „Best-Practice-Unternehmen“ rühmt, will auf seine von externer Seite festgestellte vorbildliche Arbeitsweise und -qualität hinweisen.

Passende deutsche Übersetzung

Der Begriff *Best Practice* wird mit „optimaler Geschäftsablauf“, „beste Methode“, „beste Praxis“, „beste Vorgehensweise“ oder „bestes Verfahren“ ins Deutsche übersetzt.[1]

Ableitung des Begriffes *worst practice*

Im Gegensatz zu *best practice* stehen Beispiele, die als *worst practice* bezeichnet werden müssen, weil sie ineffizient und unproduktiv waren und schlimmstenfalls zum Scheitern einer Aufgabe geführt haben. Darin drückt sich eine a-posteriori-Perspektive aus, d. h. es wurden identifizierbare Fehler begangen.

Ein Vorteil von *worst practices* ist, dass eindeutige Fehler erkannt und in der Folge potentiell vermieden werden können. Ein Lernen aus Fehlern kann wiederum zu *best practice* führen. Es wird eine Dynamik aus Scheitern und Erfolg deutlich. Allerdings werden *worst practices* nicht annähernd so öffentlich kommuniziert wie *best practices*. Die Gründe dafür können vielfältig sein. Eine psychologische Erklärung wäre das Negativbild vom Scheitern, das vermieden werden soll.

Weblinks

- Best Practice Handbook, Campus Steyr, FH Oberösterreich [2]

Referenzen

[1] Übersetzung in www.dict.cc (http://www.dict.cc/?s=best+practice), abgefragt am 4. Februar 2009
[2] http://docs.google.com/View?docid=dfg8n7rr_4hpf9qh

COCOMO

COCOMO (COnstructive COst MOdel) ist ein algorithmisches Kostenmodell, das in der Softwareentwicklung zur Kosten- bzw. Aufwandsschätzung verwendet wird. Mit Hilfe von mathematischen Funktionen wird ein Zusammenhang zwischen gewissen Softwaremetriken und den Kosten eines Projekts dargestellt.

Es fließen mehrere firmenspezifische Parameter in die Berechnung hinein, die feststellt, wie viele Personenmonate bzw. Personenjahre notwendig sind, um ein Softwareprojekt zu realisieren. Weiterhin kann die Gesamtdauer des Projekts abgeschätzt werden. COCOMO beruht auf vielfältiger Erfahrung, die in der Großindustrie, z. B. bei Boeing, bei der Softwareentwicklung gemacht worden ist. Das Verfahren wurde bereits 1981 durch Barry W. Boehm, Softwareingenieur bei Boeing, entwickelt.

Verfahren

Definitionen und Annahmen

- Der primäre Kostenfaktor (Cost Driver) für das Modell sind die Delivered Source Instructions (DSI) des Projekts.
- Die Entwicklungsperiode beginnt mit dem Start des Produktdesigns und endet mit dem Abschluss der Produktintegration und der Akzeptanztests. Die Aufwände der anderen Phasen werden separat ermittelt.
- Ein COCOMO-Personenmonat oder auch Staff Month (SM) besteht aus 152 Arbeitsstunden.
- COCOMO geht von gutem Management von Seiten der Entwickler als auch der Kunden aus und setzt voraus, dass unproduktive Zeiten möglichst gering gehalten werden.
- COCOMO setzt voraus, dass der Anforderungsspezifikation nach der Anforderungsphase keine grundlegende Änderung widerfährt. Eine signifikante Änderung in der Spezifikation zieht auch eine Änderung der Aufwandsschätzung nach sich.

Delivered Source Instructions (DSI)

Als Basis für die Berechnung muss die Anzahl von auszuliefernden Codezeilen in **KDSI** (1000 (K) delivered source instructions [1 KDSI = 1000 Instruktionen]) ermittelt werden. Als *Delivered* wird nur Software bezeichnet, welche dem Kunden als Teil des Produkts auch übergeben wird. Diese Definition schließt Code, der für Support-Software oder zum Testen geschrieben wurde, aus. Die Abschätzung dieser Größe ist von vielen Faktoren (z. B. Programmiersprache) abhängig und wird von COCOMO nicht behandelt. *Source Instructions* entsprechen den von Entwicklern geschriebenen und ausführbaren Computeranweisungen. Neben den Kommentaren schließt diese Definition also auch noch generierten Code aus. Instructions beruhen größtenteils noch auf den damals gängigen Lochkarten. So definiert Boehm Instructions in seinem Werk[1] als Card Images. Delivered Source Instructions sowie Codezeilen oder Function Points sind Softwaremetriken zur Messung der Größe der Software.

Komplexität bestimmen

Dann muss man entscheiden, ob man an einem einfachen („organic mode“), mittelschweren („semi-detached“) oder einem komplexen („embedded“) Projekt arbeitet. Diese Projektmodi sind zentrale Punkte in COCOMO 81, welche die Unterschiede im Arbeitsprozess in den verschiedenen Arbeitsdomänen repräsentieren. Die Wahl des Projektmodus wirkt sich maßgeblich auf das Ergebnis der Berechnung aus - wobei die Formel der Berechnung gleich bleibt und sich nur die Koeffizienten ändern.

Organic Mode

Der Organic Mode entspricht kleinen bis mittelgroßen Softwareprojekten. Die meisten am Projekt beteiligten Mitarbeiter haben bereits eingehende Erfahrungen mit ähnlichen Projekten in diesem Unternehmen sowie der verwendeten Soft- und Hardware. Dies gewährleistet einen geringen Overhead an Kommunikation, da die Beteiligten schon eine genaue Vorstellung von dem zu erstellenden Produkt, haben. Dokumentation von Spezifikationen und Schnittstellen wird nicht streng gehandhabt, wodurch diesbezügliche Verhandlungen schneller abgeschlossen werden und der dadurch entstehende Mehraufwand (diseconomies of scale) gering gehalten wird. Weitere Merkmale des Organic Modes sind stabile Entwicklungsumgebungen mit wenig neuen Technologien, minimaler Bedarf an neuen Innovationen und wenig Zeitdruck.

Semidetached Mode

Der Semidetached Mode ist für Projekte gedacht, deren Größe und Komplexität zwischen Organic und Embedded Mode anzusiedeln sind. Es handelt sich um mittelgroße Projekte (zwischen 50 und 300 KDSI), deren Beteiligte bereits ein mittleres Maß an Erfahrung in der Entwicklung solcher Systeme haben oder in denen das Team aus erfahrenen sowie unerfahrenen Kollegen besteht oder das Team nur auf einem Teilgebiet Erfahrung besitzt. Projekte, welche diesem Modus entsprechen, sind komplexer, benötigen anspruchsvollere Interaktionsroutinen und flexible Schnittstellen.

Embedded Mode

Der Embedded Mode ist durch seine straffen, unflexiblen Strukturen und Richtlinien gekennzeichnet. Dies stellt auch den größten Unterschied zu den beiden anderen, eher locker geführten Modi, dar. Es zielt größtenteils auf sicherheitsrelevante Projekte (z. B. Flugassistenzsysteme, Systeme für Banken) ab, welche dadurch sehr unflexibel bezüglich Änderungen in Spezifikationen und Schnittstellen sind. Des Weiteren sind Projekte im Embedded Mode in der Regel Neuentwicklungen mit wenig bis keinen vergleichbaren Vorgängerprojekten. Aus diesem Grund zeichnen sich diese Projekte auch dadurch aus, dass sie mit einer relativ langen Analyse- und Design-Phase beginnen. Sind diese Phasen abgeschlossen, werden möglichst viele Entwickler parallel damit beauftragt, das System zu implementieren und zu testen. Hier entspricht bereits der Testaufwand von intermediate Projekten (im Umfang von 8000 DSI) dem von großen Projekten (128000 DSI) im Organic Mode.

Aufwand errechnen

Der Aufwand PM in Personenmonaten wird dann errechnet als ein Faktor m multipliziert mit einer Potenz n der Metrikzahl.

$PM = m \cdot \mathbf{KDSI}^{\,n}$

- einfach: $PM = 2,4 \cdot \mathbf{KDSI}^{\,1,05}$
- mittelschwer: $PM = 3 \cdot \mathbf{KDSI}^{\,1,12}$
- komplex: $PM = 3,6 \cdot \mathbf{KDSI}^{\,1,20}$

Beispiel: Bei 100 KDSI betragen die Personenmonate PM etwa 300 für ein einfaches Projekt, etwa 500 für ein mittelschweres und etwa 900 für ein komplexes.

Projektdauer

Man kann die Personenmonate jedoch nicht durch eine beliebige Anzahl von Personen teilen, um das Produkt schneller fertig zu stellen. Als Beispiel dient oft das Gebären eines Kindes – dies kann nicht durch den Einsatz von neun Frauen auf einen Monat abgekürzt werden. Es gibt gewisse Prozesse, die sequentiell ablaufen müssen, und je mehr Menschen mit einem Projekt betraut sind, umso mehr muss in die Kommunikation investiert werden.

COCOMO spricht von TDEV, *time to develop* (Entwicklungszeit). Auch hier werden drei Komplexitätsarten unterschieden:

- einfach: $TDEV = 2,5 \cdot \mathbf{PM}^{\,0,32}$
- mittelschwer: $TDEV = 2,5 \cdot \mathbf{PM}^{\,0,35}$
- komplex: $TDEV = 2,5 \cdot \mathbf{PM}^{\,0,38}$

Für ein einfaches Projekt mit 32 KDSI liefert die COCOMO-Abschätzung 91 PM und ein TDEV von 14 Monaten.

Bei der COCOMO-Berechnung von TDEV ist die Mindestdauer acht Monate. Wenn man diese Werte andersherum auf die Anzahl von Codezeilen berechnet, die eine Person pro Tag produziert, bekommt man für ein einfaches Projekt 16 und für ein komplexes 4 DSI/Person/Tag.

Kostentreiberfaktoren

Das erweiterte COCOMO-Verfahren (Intermediate COCOMO) berücksichtigt weitere sogenannte Kostentreiberfaktoren, die den errechneten Basiswert entweder verringern oder erhöhen. Diese basieren auf vielen Erfahrungen, die bei großen Firmen gemessen worden sind. Solche Faktoren sind unter anderem:

- Zuverlässigkeit des gelieferten Systems – ist ein Fehler nur störend oder gefährdet er Menschenleben?
- Wie groß ist die Datenbank, die erstellt werden muss?
- Wie komplex sind die Ein-/Ausgabeverarbeitung und die Datenstrukturen?
- Wie schnell muss das System Ergebnisse liefern?
- Wie viel Speicherbedarf hat das System?
- Wie oft erwartet man, dass das System an äußere Rahmenbedienungen angepasst werden muss? Hier schwankt die Bandbreite zwischen einmal im Jahr bis monatlich.
- Teamfaktoren – was für Erfahrung haben die Teammitglieder in der Analyse, in der verwendeten Programmiersprache, mit Software-Werkzeugen, mit dieser besonderen Hardware?
- Wie eng ist der Zeitplan?

Als Beispiel dafür, wie sehr diese Faktoren das Ergebnis beeinflussen, dient folgende Berechnung:

Komplex, 128 KDSI, entspricht 1216 PM (Basisberechnung nach COCOMO).

Faktor	von	bis
Zuverlässigkeit	sehr hoch = 1,4	sehr niedrig = 0,75
Komplexität	sehr hoch = 1,3	sehr niedrig = 0,70
Speicherbedarf	hoch = 1,2	kaum = 1,0
Werkzeugverwendung	niedrig = 1,1	hoch = 0,90
Zeitplan	schnell = 1,23	normal = 1,0
angepasst	**3593 PM**	**575 PM**

Erklärung: Die Einzelfaktoren werden zu einem "Gesamtfaktor" aufmultipliziert und mit dem Basiswert für den Aufwand multipliziert.
Formel: Angepasster Wert = Basiswert * (Zuverlässigkeit * Komplexität * Speicherbedarf * Werkzeugverwendung * Zeitplan)

Diese Werte sind jedoch nur grobe Erfahrungswerte, jede Firma muss für sich selbst die eigenen Faktoren durch Kostenüberwachung und Analyse von bisher erstellten Projekten bestimmen.

Weiterentwicklung

Boehm weist darauf hin, dieses Modell nicht leichtfertig anzuwenden: „*Basic COCOMO is good for rough order of magnitude estimates of software costs, but its accuracy is necessarily limited because of its lack of factors to account for differences in hardware constraints, personnel quality and experience, use of modern tools and techniques, and other project attributes known to have a significant influence on costs.*“[1] (Das COCOMO-Basismodell ist gut geeignet für eine grobe Abschätzung der Größenordnung der Softwarekosten. Die Genauigkeit des Modells ist notwendigerweise eingeschränkt, weil es ihm an Faktoren für Unterschiede bei der verwendeten Hardware, der Personalqualität und -erfahrung, dem Einsatz moderner Werkzeuge und Technologien und anderen Merkmalen fehlt, die bekanntermaßen einen signifikanten Einfluss auf die Kosten haben.). Ein relativ genaues Ergebnis bekommt man erst unter Berücksichtigung mehrerer, auf das Projekt einwirkender Faktoren (siehe Intermediate und Detailed COCOMO).

ADA COCOMO

ADA COCOMO ist eine Weiterentwicklung von COCOMO 81 - welches sehr stark vom Batch-Processing und dem Wasserfall-Prozessmodell geprägt ist - zur Anpassung an die TRW-Implementierung des ADA-Prozessmodells. Dieses Modell beinhaltet Risk Management, Architektur Skeletons, Inkrementelles Implementieren und Testen und einheitliche Softwaremetriken. Hauptaugenmerk des Modells sind die Verringerung des Kommunikationsoverheads, Vermeidung späten Überarbeitens aufgrund instabiler Anforderungen. Die Änderungen zu COCOMO 81 können in drei Kategorien eingeteilt werden:

1. Allgemeine Verbesserungen von COCOMO: Diese beinhalten größtenteils Anpassungen der vorhandenen sowie zusätzlicher Kostenfaktoren. Neue Faktoren sind z. B. Security und Development for software reusability.
2. Ada-spezifische Effekte: Neue Regeln zum Abzählen der Instruktionen (DSI) für die Programmiersprache ADA. Zusätzliche Kostenfaktoren bezüglich Programming Language Experience.
3. Effekte bedingt durch das Ada-Prozessmodell: Diese Effekte wirken sich vor allem in den Exponenten der Regressionsgleichungen aus und leiten sich aus den Eigenschaften der Ada-Prozessmodells her. Hierfür wurden vier Skalierungsfaktoren eingeführt (Experience with the Ada Process Model, Design Thoroughness at PDR (Preliminary Design Review), Risks Eliminated at PDR, Requirements Volatility). Zusätzlich wurde ein Methode bereitgestellt, um den Aufwand von inkrementellen Projekten zu schätzen.

Der Rest von COCOMO 81 blieb unverändert - die generelle Form mit den verschiedenen Modi, etc.

COCOMO II

COCOMO II wurde ebenfalls, wie seine beiden Vorgänger, von Barry W. Boehm entwickelt und das erste Mal 1997 publiziert. Die offiziell bekannte Version ist jedoch 2000 mit dem Werk[2] veröffentlicht worden. COCOMO II repräsentiert die Änderungen in der 'modernen' Softwareentwicklung, mit Anpassungen an neue Softwareentwicklungs-Prozessmodelle sowie neuen Entwicklungsmethodiken (z. B. Objektorientiertes Programmieren). Es werden wieder, wie in COCOMO 81, drei verschiedene Schätzarten unterschieden, mit dem Unterschied jedoch, dass sich diese vermehrt auf den Entwicklungsstand des Projekts beziehen. Auf die Unterteilung in verschiedene Projektmodi wurde hier verzichtet.

Kritik

Das COCOMO Modell ist nur sehr beschränkt für die Abschätzung des Aufwandes eines Projektes geeignet, da es schwer ist die zugrundeliegenden Delivered Source Instructions auf Basis einer Anforderungsspezifikation zu schätzen. Die Ungenauigkeit dieser Schätzung macht diese Methode für die Aufwandsschätzung unbrauchbar.

Literatur

- Die Beispiele sind dem Standard-Lehrbuch von Ian Sommerville, *Software Engineering*, Addison-Wesley, ISBN 0-321-21026-3 entnommen.

Siehe auch

- Function-Point-Verfahren

Weblinks

- Biographie – Barry W. Boehm [3]
- Sammlung von COCOMO-Nachfolgern und -Software [4]
- Verfahren – COCOMO-Methode [5]

Referenzen

[1] Barry Boehm. *Software engineering economics*. Englewood Cliffs, NJ:Prentice-Hall, 1981, ISBN 0-13-822122-7

[2] Barry Boehm, et al. *Software cost estimation with COCOMO II* (with CD-ROM). Englewood Cliffs, NJ:Prentice-Hall, 2000, ISBN 0-13-026692-2

[3] http://www.sei.cmu.edu/about/bov/boehm.html

[4] http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html

[5] http://www.software-kompetenz.de/?4774

Function-Point-Verfahren

Die **Function-Point-Analyse** (engl. *Function Point Analysis*) ist ein Verfahren zur Bestimmung des fachlich-funktionalen Umfangs einer EDV-Anwendung bzw. eines EDV-Projektes.

Function-Points werden in der Softwareentwicklung als Basis für Aufwandsschätzung und Benchmarking herangezogen. Die Function-Point-Analyse allein stellt allerdings kein Verfahren zur Aufwandsschätzung dar, sondern dient der reinen Größenbestimmung von EDV-Anforderungen.

Der Fokus der Analyse liegt auf den fachlichen Anforderungen des zu erstellenden Systems. Dadurch kann diese Methode schon recht früh (bei Vorlage eines Lastenhefts) eingesetzt werden. Das Verfahren besteht aus mehreren Phasen:

1. Die zu verarbeitenden Daten und die Funktionen werden nach festgelegten Regeln gezählt. Daraus erhält man den unjustierten Function-Point-Wert. Dieser kann als objektives Maß für den funktionalen Umfang, den das System dem Anwender anbietet, angesehen werden.
2. In dieser Phase, die unabhängig von der Ersten durchlaufen werden kann, werden bestimmte Systemeigenschaften bewertet. Diese spiegeln grundlegende, nicht-funktionale Anforderungen wider. Aus der Verknüpfung des hier ermittelten und des unjustierten Function-Point-Wertes erhält man den justierten Function-Point-Wert. Dieser kann als objektives Maß für den zu erwartenden Entwicklungsaufwand angesehen werden.
3. Der justierte Function-Point-Wert kann nun herangezogen werden, um eine Aufwandsabschätzung durchzuführen. Hierfür wird er anhand von Referenzdaten bewertet. Diese Referenzdaten stammen idealerweise aus dem eigenen Unternehmen und werden ständig gepflegt. Es handelt sich dabei um die Daten abgeschlossener Projekte, von denen der Zeitaufwand und der justierte Function-Point-Wert bekannt sind. Über die Referenzdaten fließen weitere Randbedingungen in die Aufwandsabschätzung mit ein, die umso genauer ausfällt, je spezifischer die Referenzdaten dem gewählten System entsprechen. Randbedingungen können beispielsweise die Entwicklungsplattform, die Programmiersprache, -methode, Einsatz von Werkzeugen/Off-the-Shelf-Produkten, die Qualifikation, die durchschnittliche Team-Größe, der Entwicklungsprozess usw. sein.

Function-Points werden ermittelt, indem man Anwendungssysteme in Elementarprozesse und logische Datenbestände zerlegt. Anschließend ordnet man diesen Komplexitätsgrade (einfach, mittel, komplex) zu und vergibt die Werte in Function-Points. Die Addition der einzelnen Function-Point-Werte ergibt ungewichtete Function-Points. Als letztes gewichtet man diese mit Bewertungsfaktoren, die technische Einflussfaktoren widerspiegeln.

Siehe auch

- COCOMO
- Delphi-Methode
- Testaufwand

Literatur

- Christof Ebert, Reiner Dumke, Manfred Bundschuh, Andreas Schmietendorf: *Best Practices in Software Measurement*, Springer Verlag 2005, ISBN 3540208674
- Robert Hürten: *Function-Point Analysis Theorie und Praxis, 2. erweiterte Auflage*, expert verlag 2005, ISBN 3-8169-2398-4
- Manfred Bundschuh, Axel Fabry: *Aufwandschätzung von IT-Projekten*, MITP Verlag 2004, ISBN 382660864X
- Benjamin Poensgen, Bertram Bock: *Die Function-Point-Analyse*, Dpunkt Verlag 2005, ISBN 3898643328
- David Garmus, David Herron: *Function Point Analysis*, Addison-Wesley 2000, ISBN 0201699443

- H. Balzert: *Lehrbuch der Software Technik, Bd. 1 und 2, Spektrum Akademischer Verlag, 2001/1998*

Function Point Analyse

Function Point Analyse oder auch Function-Point-Verfahren ist eine funktionale Metrik, die aus Anwendersicht den Funktionsumfang von Anwendungssystemen unabhängig von der zu Grunde liegenden Technologie misst. Ein Anwendungssystem kann man sich dreidimensional vorstellen: fachliche Funktionalität, technische Funktionalität und die Qualität, die die technische und fachliche Funktionalität zu liefern hat.

FPA ist ein Messverfahren für die fachlichen Funktionalität eines Anwendungssystems. Es muss festgelegt werden, ob man Anwendungssysteme insgesamt betrachtet bzw. misst oder ob man nur Projekte misst. FPA betrachtet fachliche Funktionalität in Form der Elementarprozesse, so genannte transaktionale Funktionen.

Elementarprozess: Ein Elementarprozess ist definiert als die für den Anwender sinnvollste, kleinste Aktivität, die das System nach ihrer Ausführung in einem konsistenten Zustand lässt. Elementarprozesse unterteilen sich in drei Gruppen:

- Eingabe
- Ausgabe
- Anfrage

Datenbestand: Neben den Elementarprozessen gibt es Datenbestände, die so genannten Data Functions. Auch die Datenbestände sind aus Anwendersicht zu betrachten. Sie lassen sich in die folgenden Gruppen unterteilen:

- externe Datenbestände
- interne Datenbestände

Da jeder Elementarprozess bzw. Datenbestand aus der Sicht der FPA unterschiedlich zu bewerten ist, gibt es die Komplexitätsberechnung von Elementarprozessen und Datenbeständen. Die Komplexität wird anhand von folgenden Typen berechnet.

- Record Element Types (RET),
- Data Element Types (DET) oder
- File Type Referenced (FTR)

Unadjusted Function Point: Der FP-Wert ändert sich entsprechend der Komplexität. Es wurden drei verschiedene Komplexitätsgrade definiert: niedrig, durchschnittlich und hoch. Diesen so berechneten FP-Wert nennt man Unadjusted Function Point Count (UFPC).

General System Characteristics: Der Entwickler der FPA, Allen J. Albrecht, wollte das Verfahren aber auch zur Aufwandschätzung einsetzen. Daher musste er die anderen beiden Dimensionen (technische Funktionalität und Qualität eines Anwendungssystems) zur Aufwandschätzung ebenfalls betrachten. Hierfür wurden 14 General System Characteristics (GSC) definiert, die die technischen Anforderungen und die Qualität eines Softwaresystems berücksichtigen.

Adjusted Function Point: Jedes der GSC-Kriterien ist von 0 bis 5 Punkten gewichtet. Eine solche Gewichtung heißt Degree of Influence (DI). Die Summe aller 14 DI ist der Total Degree of Influence (TDI). Dieser fließt in eine Formel ein, die den FP-Wert nochmals verändert; man bezeichnet diesen als Adjusted Function Point - Wert (AFP).

Lines of Code

Lines of Code (abgekürzt **LOC** oder auch **LoC**, im Englischen auch "SLOC - Source Lines of Code") ist ein Begriff aus der Informationstechnik beziehungsweise dem Programmiererjargon. Er kommt aus dem Englischen und heißt übersetzt so viel wie „Anzahl an Programmzeilen“.

Die Messung der „Lines of Code“ kann als Metrik für die Größe oder für das Wachstum eines Programms verwendet werden.

Die Metrik darf nicht dazu missbraucht werden, um die Effizienz eines Programmierers an der Anzahl von Programmzeilen zu messen. Da die Anzahl von unterschiedlichsten Faktoren abhängt (gewählte Architektur, Erfahrung des Programmierers, gewählte Lösung, Formatierung des Quellcodes, verwendete Programmiersprache, ...), besitzt diese Metrik keinerlei Aussagekraft im Bezug zur Leistung des Programmierers. Zudem besagt die 80/20-Regel, dass 80 % der Zeilen in 20 % der Zeit geschrieben werden. Das Testen eines Programms kann, sofern es seriös gemacht wird, sehr viel Zeit in Anspruch nehmen, während die Anzahl geänderter oder ergänzter Programmzeilen nur sehr gering ist.

> “Measuring programming progress by lines of code is like measuring aircraft building progress by weight.”
>
> „Den Programmierfortschritt durch die Anzahl der Programmzeilen zu messen ist, wie wenn man den Fortschritt beim Bau eines Flugzeugs an dessen Gewicht misst.“
>
> – Bill Gates

Verwendung

Die meisten Vergleiche von Programmen über die LOC betrachten nur die Größenordnungen der Anzahl Zeilen verschiedener Projekte. Computerprogramme können aus nur wenigen Dutzend bis zu hunderten von Millionen Programmzeilen bestehen. Der Umfangs eines Programmes muss nicht zwangsläufig Rückschlüsse auf die Komplexität des Programms erlauben, da die Zählweise der Zeilen nicht einheitlich ist und das Ergebnis außerdem von der Formatierung des Quelltextes, der Programmiersprache und anderen Faktoren abhängt. Je nach Zählweise sind insbesondere Rückschlüsse auf die investierte Arbeitszeit meistens sinnfrei.

Es gibt zwei gängige Varianten, LOC zu messen: physische Zeilen oder logische Zeilen. Bei der physischen Zählweise wird die Zahl der Zeilen im Quelltext, inklusive Kommentar- und Leerzeilen, gezählt. Bei der logischen Zählweise wird versucht, die Anzahl der Anweisungen zu zählen. Was wiederum unter einer Anweisung zu verstehen ist und wie die Komplexität derselben zu beurteilen ist, hängt von der verwendeten Programmiersprache ab. So kann etwa in einem mehrere Bildschirmseiten umfassenden Assembler-Quelltext letztlich das gleiche Programm formuliert sein, wie in einigen wenigen Zeilen einer höheren Programmiersprache.

Werden logische Zeilen gezählt, entfallen unter Umständen auch Unmengen an Dokumentation. So besteht beispielsweise die Haupt-Header-Datei der freien Graphikbibliothek Cairo zu knapp 60 % aus Doxygen-Kommentaren zur Dokumentation, die nicht minder wichtig sind.

Das folgende Beispiel zeigt den Unterschied der Bewertungsmethoden:

```
for (i=0; i<100; ++i) printf("hello"); /* Wieviele Zeilen Code sind das? */
```

Der Quelltext besteht aus:

- 1 physischen Codezeile (LOC),
- 2 logischen Codezeilen (LLOC, for-Anweisung und printf-Anweisung) sowie
- 1 Kommentarzeile.

Abhängig vom Programmierer und den verwendeten Formatierungsrichtlinien kann die obige Zeile in folgenden, programmtechnisch gesehen vollkommen gleichwertigen Quelltext umformuliert werden:

```
/* Wie viele Zeilen sind das? */
for (i = 0; i < 100; ++i)
{
    printf("hello");
}
```

Nun besteht der Quelltext aus:

- 5 physischen Codezeilen: Was kostet das Platzieren einer Klammer?
- 2 logischen Programmzeilen und
- 1 Kommentarzeile: Kommentare sind für den Betrachter des Quelltextes relevant; der Compiler überliest sie.

Selbst bei diesem simplen Beispiel kann man nun durchaus noch argumentieren, dass der Kopf der for-Schleife im Beispiel aus drei einzelnen Anweisungen besteht (Zuweisung, Vergleich und Inkrement). Dies zeigt, wie schwierig es selbst bei vorgegebener Programmiersprache ist, eine sinnvolle Definition für LLOC zu finden.

Beispiele

Hier sind einige Beispiele für die Anzahl der Zeilen von Programmcode in verschiedenen Betriebssystemen und Anwendungsprogrammen.

Jahr	Betriebssystem	SLOC (in Millionen)
1993	Windows NT 3.1	4-5[1]
1994	Windows NT 3.5	7-8[1]
1996	Windows NT 4.0	11-12[1]
2000	Windows 2000	mehr als 29[1]
2001	Windows XP	40[1]
2003	Windows Server 2003	50[1]

System	SLOC (in Millionen)
Debian 2.2	55-59[2] [3]
Debian 3.0	104[3]
Debian 3.1	215[3]
Debian 4.0	283[3]
OpenSolaris	9.7
FreeBSD	8.8
Mac OS X 10.4	86[4]
Linux Kernel 2.6.0	5.2
Linux Kernel 2.6.29	11.0
Linux Kernel 2.6.32	12.6[5]

SAP NetWeaver der SAP AG in 2007	238[6]

Literatur

- S. H. Kan. 2002. *Metrics and Models in Software Quality Engineering*. 2nd Edition. Addison-Wesley.

Weblinks

- CODECOUNT Toolset [7]
- Messung von LOC-Metriken mit Testwell CMT++/CMTJava [8]

Referenzen

[1] *How Many Lines of Code in Windows?* (http://www.knowing.net/PermaLink,guid,c4bdc793-bbcf-4fff-8167-3eb1f4f4ef99.aspx). Knowing.NET (April 2009 – Scholar search (http://scholar.google.co.uk/scholar?hl=en&lr=&q=intitle:How+Many+Lines+of+Code+in+Windows?&as_publication=&as_ylo=&as_yhi=&btnG=Search)). Abgerufen am 18. Oktober 2007.

[2] González-Barahona, Jesús M., Miguel A. Ortuño Pérez, Pedro de las Heras Quirós, José Centeno González, and Vicente Matellán Olivera. *Counting potatoes: the size of Debian 2.2* (http://people.debian.org/~jgb/debian-counting/counting-potatoes/). *debian.org*. Abgerufen am 12. August 2003.

[3] Robles, Gregorio. *Debian Counting* (http://libresoft.dat.escet.urjc.es/debian-counting/). Abgerufen am 16. Februar 2007.

[4] Jobs, Steve (August 2006). *Live from WWDC 2006: Steve Jobs Keynote* (http://www.engadget.com/2006/08/07/live-from-wwdc-2006-steve-jobs-keynote/). Abgerufen am 16. Februar 2007. „86 million lines of source code that was ported to run on an entirely new architecture with zero hiccups."

[5] http://www.h-online.com/open/features/What-s-new-in-Linux-2-6-32-872271.html?view=print What's new in Linux 2.6.32

[6] http://searchsap.techtarget.com/news/column/0,294698,sid21_gci1282035,00.html

[7] http://sunset.usc.edu/research/CODECOUNT/

[8] http://www.verifysoft.com/de_cmtx.html

Metriken

Softwaremetrik

Eine **Softwaremetrik**, oder kurz **Metrik**, ist eine (meist mathematische) Funktion, die eine Eigenschaft von Software in einen Zahlenwert, auch **Maßzahl** genannt, abbildet. Hierdurch werden formale Vergleichs- und Bewertungsmöglichkeiten geschaffen.

Hintergrund

Formell spricht man davon, die Metrik auf eine *Software-Einheit* anzuwenden. Das Ergebnis ist die Maßzahl. Mit Software-Einheit ist in der Mehrheit der Fälle der zugrundeliegende Quellcode gemeint. Da der Quellcode üblicherweise auf eine oder mehrere einzelne Dateien verteilt wird, kann die Metrik je nach Art auf den ganzen Quellcode oder Teile davon angewendet werden. Es gibt zudem Metriken, wie etwa die Function-Point-Analyse, die bereits auf der Spezifikation von Software angewendet werden können, um im Vorfeld den Aufwand zur Entwicklung der Software zu bestimmen.

In der Form des Zahlenwerts, der Maßzahl, dient die Metrik als Maß für eine Eigenschaft, ein Qualitätsmerkmal, von Software. Sie kann einen funktionalen Zusammenhang repräsentieren oder auch aus einer Checkliste abgeleitet werden. Einfache Metriken zeigen die Größe des Quellcode in Zeilen oder Zeichen auf, komplexere Metriken versuchen die Verständlichkeit des Quellcodes zu beurteilen. Mit einer geeigneten Zahl verschiedener Metriken kann beurteilt werden, wie aufwändig (sprich personal- und kostenintensiv) die Wartung, Weiterentwicklung und anschließende Tests der Software werden.

Von einem neu entwickelten Programm werden oft nicht nur bestimmte Funktionen gefordert, sondern auch Qualitätsmerkmale wie zum Beispiel Wartbarkeit, Erweiterbarkeit oder Verständlichkeit. Softwaremetriken können dabei keine korrekte Umsetzung der Funktionen bewerten, sie können allenfalls vorherbestimmen, welchen Aufwand die Erstellung der Software etwa bereiten wird und wie viele Fehler auftreten werden.

Werden während der langfristigen Weiterentwicklung einer Software regelmäßig Metriken angewendet, können negative Trends, also Abweichungen vom Qualitätsziel, frühzeitig entdeckt und korrigiert werden.

Die Interpretation der Daten einer Softwaremetrik ist Aufgabe der Disziplin der Softwaremetrie, dort stellen die Softwaremetriken einen Teil der Basisdaten für die Interpretation dar.

Definition nach IEEE Standard 1061

*Eine **Softwaremetrik** ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit* (IEEE Standard 1061, 1992)

Anmerkung: Mit *Software-Einheit* ist dabei in der Regel der zugrundeliegende Quellcode gemeint. Da der Quellcode üblicherweise auf eine odere mehrere einzelne Dateien verteilt wird, kann die Metrik (je nach Art) auf den ganzen Quellcode oder Teile davon angewendet werden. Es gibt zudem Metriken, wie etwa die Function-Point-Analyse, die bereits auf der Spezifikation der Software angewendet werden können.

Ordnung von Softwaremetriken

Metriken bedienen verschiedene Aspekte der entstehenden Software, des angewendeten Vorgehensmodells und der Bewertung der Erfüllung der Anforderungen.

Nutzung

Der Einsatz von Metriken erstreckt sich von der Beurteilung der Entwicklungsphasen über die Beurteilung der Phasenergebnisse bis hin zur Beurteilung der eingesetzten Technologien. Das Ziel der Anwendung einer Metrik in der Softwareentwicklung ist die Fehlerprognose und die Aufwandschätzung, wobei zwischen vorlaufendem, mitlaufendem und retrospektivem Einsatz unterschieden wird.

Beschränkung

Grundsätzlich sind Metriken, die überschaubar bleiben, eindimensional. Damit zwingen sie zur Vereinfachung. In der Regel wird das erreicht, indem jede Metrik auf eine Sicht eingeengt wird. Das bedeutet dann zwingend, dass andere Sichten nicht gleichzeitig in gleicher Qualität bedient werden.

1. Sicht des *Managements*
   - Kosten der Software-Entwicklung (Angebot, Kostenminimierung)
   - Produktivitätssteigerung (Prozesse, Erfahrungskurve)
   - Risiken (Marktposition, Time2Market)
   - Zertifizierung (Marketing)
2. Sicht des *Entwicklers*
   - Lesbarkeit (Wartung, Wiederverwendung)
   - Effizienz und Effektivität
   - Vertrauen (Restfehler, MTBF, Tests)
3. Sicht des *Kunden*
   - Abschätzungen (Budgettreue, Termintreue)
   - Qualität (Zuverlässigkeit, Korrektheit)
   - Return on Investment (Wartbarkeit, Erweiterbarkeit)

Klassifikation

Für die verschiedenen Aspekte der Bewertung gibt es *Entwurfsmetriken*, *wirtschaftliche Metriken*, *Kommunikationsmetriken* usw. Metriken können verschiedenen Klassen zugeordnet werden, die den Gegenstand der Messung oder Bewertung bezeichnen:

1. *Prozess-Metrik*
   - Ressourcenaufwand (Mitarbeiter, Zeit, Kosten)
   - Fehler
   - Kommunikationsaufwand
2. *Produkt-Metrik*
   - Umfang (Lines of Code, Wiederverwendung, Prozeduren, ...)
   - Komplexität
   - Lesbarkeit (Stil)
   - Entwurfsqualität (Modularität, Kohäsion, Kopplung, ...)
   - Produktqualität (Testergebnisse, Testabdeckung, ...)
3. *Aufwands-Metrik*
   - Aufwandsstabilität

- Aufwandsverteilung
- Produktivität
- Aufwand-Termin-Treue

4. *Projektlaufzeit-Metrik*
   - Entwicklungszeit
   - Durchschnittliche Entwicklungszeit
   - Meilenstein-Trend-Analyse
   - Termintreue
5. *Komplexitäts-Metrik*
   - Softwaregröße
   - Fertigstellungsgrad
6. *Anwendungs-Metrik*
   - Schulungsaufwand
   - Kundenzufriedenheit

Gütekriterien

Eine Metrik aus der Produktionsphase der Software allein ist noch kein Gütekriterium. In der Regel werden Gütemerkmale an der Erfüllung der Anforderungen des Kunden und seiner Anwendung gemessen. Dabei sind die Übertragbarkeit der Ergebnisse und die Repräsentanz der Messwerte für den Kundennutzen von Bedeutung:

- *Objektivität*: keine subjektiven Einflüsse des Messenden
- *Zuverlässigkeit*: bei Wiederholung gleiche Ergebnisse
- *Normierung*: Messergebnisskala und Vergleichbarkeitsskala
- *Vergleichbarkeit*: Maß mit anderen Maßen in Relation setzbar
- *Ökonomie*: minimale Kosten
- *Nützlichkeit*: messbare Erfüllung praktischer Bedürfnisse
- *Validität*: von messbaren Größen auf andere Kenngrößen zu schließen (schwierig)

Metriken

Einige der bekannteren Metriken sind:

- Anzahl der Codezeilen, engl. Lines Of Code, kurz LOC. Diese trivialste Metrik ist insb. von der Formatierung des Quellcodes und dem tatsächlichen enthaltenen Kommentarzeilen abhängig. LOC kann daher bei verschiedenen Quellcodes nicht vergleichbare Ergebnisse liefern. Dem kann aber mittels automatischer Formatierung weitgehend entgegen gewirkt werden.
- das Function-Point-Verfahren zur Aufwandsabschätzung in der Analysephase
- COCOMO
- die zyklomatische Komplexität (nach McCabe) zur Komplexitätsbestimmung eines Programmodules
- die Halstead-Metrik zur Implementierungsabschätzung in der Entwurfsphase
- Kontrollflussorientierte Metriken sind im Artikel Kontrollflussorientierte Testverfahren ausgeführt
  - Verhältnis von ausgeführten ELOC zu vorhandenen ELOC, wobei ELOC (executable line of code) eine ausführbare Quellcodezeile ist
  - C_0 Anweisungsüberdeckungskennzahl
  - C_1 Zweigüberdeckungskennzahl
  - C_2 Pfadüberdeckungskennzahl
  - C_3 Bedingungsüberdeckungskennzahl

Durch Kombination vorhandener Metriken werden immer wieder neue Metriken entwickelt, die zum Teil neue Entwicklungen im Software Engineering widerspiegeln. Ein Beispiel hierfür ist die 2007 vorgestellte C.R.A.P. (Change Risk Analysis and Predictions) Metrik zur Beurteilung der Wartbarkeit von Code.

Auswahl geeigneter Metriken

Zur Identifikation geeigneter Maße kann das Goal Question Metric (GQM) Verfahren eingesetzt werden.

Vorgehen

1. *Phasen- und Rollenmodell* festlegen
2. *Ziele* bestimmen
3. *Metrik-Maske* definieren
4. *Messplan* aufstellen
5. *Daten* sammeln
6. Daten validieren
7. Daten analysieren und interpretieren
8. Daten sichern und visualisieren

Literatur

- Christof Ebert und Reiner Dumke: *Software Measurement - Establish, Extract, Evaluate, Execute*. Springer-Verlag, 2007, ISBN 978-3-540-71648-8
- Georg E. Thaller: *Software-Metriken einsetzen - bewerten - messen*. Verlag Technik, 2000, ISBN 3-341-01260-5
- M. Rezagholi: *Prozess- und Technologie Management in der Softwareentwicklung*. Oldenbourg Verlag München Wien, 2004, ISBN 3-486-27549-6
- Ch. Bommer, M. Spindler, V. Barr: *Softwarewartung - Grundlagen, Management und Wartungstechniken*, dpunkt.verlag, Heidelberg 2008, ISBN 3-89864-482-0

Weblinks

- Gesellschaft für Informatik, Fachgruppe Software-Messung und -Bewertung (2.1.10) [1]
- Softwarequalitätsmanagement [2]
- Präsentation "Metrik basierte Technologie- und Prozessbewertung" [3] (PDF-Datei; 1,33 MB)
- Berechnung von McCabe- und Halstead-Metriken anhand eines Beispielprojekts [4] (PDF-Datei; 737 kB)
- Universität Linz: Seminar Softwaremetriken [5]

Referenzen

[1] http://ivs.cs.uni-magdeburg.de/sw-eng/us/giak/
[2] http://ivs.cs.uni-magdeburg.de/~dumke/ST2/qbegriffe.html
[3] http://www.m-boehmer.de/pdf/Metrik_basierte_Technologie-_und_Prozessbewertung.pdf
[4] http://www.verifysoft.com/de_cmtpp_mscoder.pdf
[5] http://www.ssw.uni-linz.ac.at/Teaching/Lectures/Sem/2002/reports/Holzmann/

Goal Question Metric

Goal Question Metric (**GQM**) ist eine systematische Vorgehensweise zur Erstellung spezifischer Qualitätsmodelle im Bereich der Softwareentwicklung. Diese lässt sich als Baumstruktur darstellen. Als Wurzel steht das Ziel (*Goal*), das über die Knoten (*Questions*) zu den Blättern (*Metric*) verfeinert wird. Auf diesem Weg werden Fragen und Softwaremaße abgeleitet. Auf dem Weg von den Blättern zur Wurzel werden die gemessenen Werte interpretiert.

Geeignete Metriken lassen sich in der Softwaremetrie identifizieren über die Beantwortung der folgenden Fragen:

- Welches Ziel soll durch die Messung erreicht werden? (*Goal*)
- Was soll gemessen werden bzw. welche Fragen soll die Messung beantworten? (*Question*)
- Welche Metrik(en) sind in der Lage, die notwendigen Eigenschaften zu beschreiben? (*Metric*)

Historie

„Wie kann man entscheiden, was man messen muss, um seine Ziele zu erreichen?“

Genau vor einem solchen Problem standen die Erfinder des Goal-Question-Metric-Modells, Prof. Victor R. Basili und Dr. David Weiss, als sie, in der Umgebung des Software Engineering Labors, des NASA Goddard Space Flight Center, an verschiedenen Flugdynamik-Projekten arbeiteten. Die Definition von Zielen (Goals) half ihnen, sich auf das Wesentliche und Wichtige zu konzentrieren. Das Aufschreiben von Fragen (Questions) erleichterte es, die Ziele genauer zu spezifizieren und hierdurch ließen sich gleichzeitig relevante Metriken (Metrics) ableiten. GQM war geboren und etablierte sich schnell als das Qualitätsmodell am Goddard Space Flight Center.

Goal-Question-Metric-Modell

Das GQM-Modell beschreibt die Verfahrensweise zur Erstellung eines Qualitätsmodells, wobei sich das Modell in sechs Einzelschritte unterteilen lässt:

1. Charakterisierung des Unternehmens- und Projektumfeldes, Erfassung des Mission Statement, Definition von Messzielen
2. Formulieren von Fragen zur genaueren Definition der Ziele
3. Messziele identifizieren und Metriken ableiten
4. Entwicklung von Mechanismen zur Datensammlung
5. Daten sammeln, analysieren und interpretieren
6. Erfahrungen zusammenfassen und anwenden

Kurz: *"Der GQM Prozess beginnt mit der Charakterisierung des Organisations- und Projektumfeldes. Unter Berücksichtigung des Umfeldes werden im zweiten Schritt Informationsbedürfnisse mittels Zielen und korrespondierenden Fragen erfasst. Anschließend werden im dritten Schritt Messungen dokumentiert, die der Quantifizierung dieser Informationsbedürfnisse dienen. Im vierten und fünften Schritt werden die Messungen durchgeführt und die resultierenden Daten interpretiert. Abschließend findet im sechsten Schritt eine Nachbereitung statt. Dabei werden beispielsweise die Qualitätsplanung und gewonnene Erkenntnisse gesichert."*

Die Literatur ist sich bei der Aufteilung der sechs Schritte allerdings nicht ganz einig. Daher stößt man bei der Recherche nach GQM auf verschiedenste Einteilungen und diverse unterschiedliche Bezeichnungen. Einig ist sich die Literatur nur darüber, dass es sechs Schritte sind und die Verfahren, die ihnen zu Grunde liegen, immer in der gleichen Reihenfolge angewendet werden. Die ersten drei Schritte des GQM-Modells werden häufig auch als Definitionsphase bezeichnet. In dieser Phase werden auch die Ziele (Goals), Fragen (Questions) und Metriken (Metrics) ermittelt, die dem GQM-Modell seinen Namen geben.

Ziele (goals) identifizieren, was wir erreichen möchten; Fragen (questions), sofern beantwortet, sagen uns, ob wir die Ziele erreichen, oder helfen uns, sie zu verstehen und zu interpretieren; und die Metriken (metrics) identifizieren die

Messungen, die nötig sind, um die Fragen (questions) zu beantworten, und quantifizieren die Ziele (goals). Die Ziele, Fragen und Metriken mit deren dazugehörigen Maßen, Schaubildern und sonstigen Ausweitungen, werden als GQM-Plan zusammengefasst. Wie man in obigem Schaubild erkennen kann, ist es auch möglich, mehrere Ziele zu definieren, wobei sich mehrere Ziele eine Frage und mehrere Fragen eine Metrik teilen können. Anders gesagt: Die Beziehung zwischen „Zielen und Fragen“ und „Fragen und Metriken“ ist beide Male 1 zu n. Das gesamte GQM-Modell wird „Top-Down“ definiert, d.h. es ist ein zielorientierter Ansatz, der von den Zielen ausgehend nach unten Fragen und Metriken definiert. Die Analyse und Interpretation geschieht dann wiederum „Bottom-Up“, d.h. von unten nach oben.

Schritt 1: Charakterisierung des Unternehmens- und Projektumfeldes, Erfassung des Mission Statement, Definition von Messzielen

In diesem ersten Schritt geht es darum, GQM-Ziele (goals) zu definieren. Es gibt zwei Arten von GQM-Zielen:

- Geschäftsziele (business goals)
- Messziele (measurement goals)

Geschäftsziele, gerne auch als Mission Statement bezeichnet, sind das übergeordnete Leitbild eines Unternehmens und definieren dessen langfristige Ziele. Ein Unternehmen definiert immer Geschäftsziele. Diese müssen identifiziert und als Fokus zur Erstellung von Messzielen verwendet werden. Ohne Geschäftsziele hätte die gesamte Messung keine Ausrichtung. Geschäftsziele treiben infolgedessen die Identifizierung von Messzielen an, wobei sich Geschäftsziele und Messziele nicht zwangsläufig voneinander unterscheiden müssen, daher wird im Folgenden nur noch von GQM-Zielen gesprochen.

Jedes GQM-Ziel sollte durch fünf Aspekte ausgedrückt werden:

- Objekt der Messung
- Zweck
- Qualitätsfokus
- Blickwinkel
- Kontext

Festlegen des Messobjekts

Objekt der Messung festlegen bedeutet herauszufinden, über was man mehr erfahren möchte: „Was möchte man messen?“ GQM erlaubt es, verschiedenste Objekte zu untersuchen. Dies könnten verallgemeinert gesagt Prozesse, Produkte, Ressourcen, Projekte, etc. sein, um nur einige Beispiele zu nennen.

Festlegen des Zwecks

Zweck festlegen bestimmt, was erreicht werden soll. Beispiele hierfür wären Charakterisierung, Verbesserung, Überwachung, Auswertung, Vorhersage oder Optimierung.

Festlegen des Qualitätsfokus

Qualitätsfokus festlegen: Der Qualitätsfokus ist ein Element des Messziels. Die folgenden Beispiele zeigen eine große Auswahl an Qualitätsattributen:

- Zuverlässigkeit/Verfügbarkeit
- Benutzbarkeit
- Sicherheit
- Funktionale Sicherheit
- Skalierbarkeit
- Wartbarkeit
- Anpassbarkeit

- Time to Market
- Performance
- Effizienz
- Portabilität
- Schnittstellenkompatibilität

Festlegen des Blickwinkels

Blickwinkel festlegen: Dies ist – wie der Name schon sagt – die Perspektive, von der das Messobjekt betrachtet wird, wobei dies in der Regel der Kunde, der Entwickler, der Tester, der Projektleiter oder der Vertragsnehmer ist.

Festlegen des Kontexts

Kontext festlegen: Hier wird bestimmt, in welchem Zusammenhang das Messobjekt steht. Der Kontext sollte möglichst genau erläutert werden, wobei es sich anbietet, Angaben über Firma, Abteilung, Projekt und Zeitraum zu machen.

Merksatz

Die Definition der GQM-Ziele kann mit folgendem Hilfssatz dargestellt werden:

```
Analysiere den Entwicklungsprozess
zum Zwecke der Änderung
in Bezug auf Korrektheit
vom Blickwinkel des Entwicklers
im Kontext des Projekts X.
```

GQM-Plan

Das Sammeln der Ziele kann auf verschiedene Arten und Weisen passieren. Die meist verbreitetsten Methoden sind Workshops und Interviews. Wichtig ist hierbei, dass Ziele von den verschiedensten Prozessbeteiligten erhoben werden, um ein möglichst breites Spektrum zu erhalten. Die Ziele selbst werden in einem sogenannten Abstraction Sheet dokumentiert. Häufig werden mehrere Abstraction Sheets in einem GQM-Plan festgehalten, die dann vor der weiteren Bearbeitung mit Prioritäten versehen werden. Der GQM-Plan ist das Dokumentationswerkzeug der Goal-Question-Metric. Der Aufbau ist einfach. Er beginnt mit einer thematischen Einleitung passend zum definierten Ziel. Anschließend werden Fragen, mit den jeweils dazugehörigen Metriken niedergeschrieben. Die nachfolgende Vorlage kann bei der Fertigung eines GQM-Plans helfen.

```
Q.1 Frage1
 M.1.1 Metrik1
 M.1.2 Metrik2
Q.2 Frage2
 M.2.1 Metrik1
 M.2.2 ...
Q.3 ...
```

Abstraction Sheets

Abstraction Sheets werden genutzt, um einem GQM-Ziel wichtige Informationen zuzuordnen, bzw. sie mit dem GQM-Ziel zu gruppieren. Es können parallel mehrere Abstraction Sheets existieren. Das Sheet ist in fünf Teile aufgeteilt:

- Die fünf Aspekte (Objekt der Messung, Zweck, Qualitätsfokus, Blickwinkel, Kontext) drücken aus, zu welchem GQM-Ziel das Sheet gehört.
- Qualitätsfaktoren beziehen sich unmittelbar auf das GQM-Ziel. Um sie herauszufinden, sollte man sich die Frage stellen: „Welche Faktoren müssen bei diesem GQM-Ziel betrachtet werden und welche beeinflussen es?“
- Die Hypothese ist die geschätzte, bzw. erwartete Antwort auf die definierten Qualitätsfaktoren.
- Einflussfaktoren beziehen sich wiederum direkt auf die Qualitätsfaktoren: „Was beeinflusst die Qualitätsfaktoren?“
- Einflüsse auf die Hypothesen werden im letzten Abschnitt des Sheets festgehalten: „Wie beeinträchtigen die Einflussfaktoren die Hypothesen?“

Schritt 2: Formulieren von Fragen zur genaueren Definition der Ziele

In dieser Phase des Modells werden Fragen (questions) gestellt, die sich aus den Abstraction Sheets ableiten. Für jede festgehaltene Komponente eines Abstraction Sheets lässt sich in der Regel auch eine Frage formulieren. Das Beantworten dieser Fragen hilft somit, dem Ziel (goal) näher zu kommen. Um Fragen zu definieren, sollte man zunächst ermitteln, was man über die Eigenschaften im Qualitätsfokus des Ziels lernen möchte. Hierbei sollten folgende Punkte beachtet werden:

- Übereinstimmung mit dem Ziel
  Beim Thema bleiben! Gerade den Zweck und den Blickwinkel sollen die Fragen widerspiegeln.
- Quantifizierbarkeit
  Die Fragen müssen so formuliert werden, dass sie über Messwerte beantwortet werden können. Also nicht „Welche Testfälle wären in diesem Kontext gut?“, sondern „Welche Testfälle haben die meisten Fehler gefunden?“. Kompliziertere Fragen sollten in mehrere einfache Fragen unterteilt werden.
- Bedeutung der Antwort
  Man sollte sich die Fragen stellen: „Was machen wir mit der Antwort auf diese Frage?“, „Was sind mögliche Maßnahmen, um dieses Ziel zu erreichen?“. Wenn die definierte Frage zu keinen Verbesserungen/Handlungen führt, sollte die Notwendigkeit der Frage noch mal überdacht werden.
- Verständliche Diagramme
  Es muss klar sein, was auf den Achsen des Diagramms dargestellt wird, welche Analyse der Darstellung vorausgeht und welche Daten ausgeschlossen wurden.
- Benutzung von bekannten Diagrammtypen
  Man sollte bei der Auswahl der Diagrammarten nicht zu kreativ sein, sondern, wann immer möglich, auf bekannte (zum Fragentyp passende) Diagramme zurückgreifen.
- Kosten/Nutzen
  Wiegt der Nutzen, den man sich aus der Beantwortung der Frage verspricht, die Kosten der Datensammlung auf?
- Klarheit der Begriffe
  Es sollte sichergestellt sein, dass Begriffe wie Qualität, Effektivität, Effizienz oder Zuverlässigkeit präzise definiert sind.
- Vernünftige Anzahl von Fragen
  Erfahrungen aus erfolgreich durchgeführten Messprogrammen zeigen, dass drei bis sieben Fragen pro Ziel definiert werden sollten.
- Gutes Verhältnis zwischen Qualitäts- und Einflussfaktoren
  Es sollte durch die Frage „Was beeinflusst diesen Qualitätsaspekt“ überprüft werden, ob es wichtige Einflüsse

gibt. Jedoch sollten nicht mehr als etwa dreimal so viele Einflussfaktoren wie Qualitätsfaktoren definiert sein, da man letztendlich an Letzteren interessiert ist.

Schritt 3: Messziele identifizieren und Metriken ableiten

Dieser Schritt baut auf den Vorherigen auf und versucht, mit Hilfe der zuvor ausgearbeiteten GQM-Ziele, Metriken (metrics) abzuleiten. Die Essenz dieser Phase ist es, die passende Metrik zu der gestellten Frage zu finden, was sich oftmals als einfacher herausstellt, als zuvor gedacht. Als besonders geeignet stellen sich folgende Maße, bzw. Metriken und die sich daraus ergebenden Maße, heraus:

- Depth of Inheritance Tree (DIT)
- Number of Children of a Class (NOC)
- Response for a Class (RFC)
- Weighted Methods per Class (WMC)
- Coupling between Object Classes (CBO)
- Lack of Cohesion of Methods (LCOM)
- Constructive Cost Model I+II (COCOMO I+II)
- Halstead-Metrik
- McCabe-Metrik (zyklomatische Komplexität)
- Lines of Code (LOC)
- Function-Point-Verfahren
- Fehlerdichte
- Schwierigkeitsgrad

Schritt 4: Entwicklung von Mechanismen zur Datensammlung

In diesem Arbeitsschritt wird beschrieben, wie die Daten und Maße zu erfassen sind, die für den GQM-Plan notwendig sind. Man spricht auch von der Erstellung eines Messplans. Folgende Punkte sollten mit Hilfe des Messplans abgedeckt werden:

- Formale Definition der Messungen
- Schriftliche Abfassung der Messungen
- Alle Output-Werte der Messungen
- Die Person oder Rolle, die die Messungen durchführt
- Der Zeitpunkt, wann die Messungen durchgeführt werden
- Das Medium (Werkzeug oder Fragebogen), das benutzt wird, um die Messungen durchzuführen

Für die gesamte Datenerhebung und -messung gilt, dass Personen, die Messungen durchführen, bzw. Daten sammeln, speziell geschult werden müssen. Dies ist notwendig, damit die Datensammlung gültig, einheitlich und vergleichbar ist. Nach diesem Schritt liegen Rohdaten, oder auch Primärdaten genannt, vor. Sie erhalten diesen Namen, da sie noch in keiner Weise aufbereitet wurden und in dieser Form noch nicht repräsentativ sind.

Messplan

Der Messplan beinhaltet, wie Rohdaten am Effizientesten erhoben werden und an wen sie zur Bearbeitung weitergegeben werden. Er wird definiert durch die Metriken, die zuvor ausgewählt wurden. Er handelt die Vorgehensweise während der Messung ab, welche Rohdaten wie erhoben werden müssen.

Schritt 5: Daten sammeln, analysieren und interpretieren

In diesem Schritt werden die zuvor im Messplan festgelegten Rohdaten gesammelt. Dieser Schritt beschreibt nun, auf welche Art die Rohdaten erfasst werden. Die Hauptarbeit ist dennoch nicht die Erhebung der Rohdaten, sondern die Analyse und Interpretation der Rohdaten, so dass man am Ende dieses Schrittes aufbereitete Daten, sogenannte Sekundärdaten, erhält. Man kann Rohdaten mit Hilfe verschiedener Verfahren erfassen:

- Fragebögen
- Datenblätter
- Interviews
- Beobachtung
- Automatisierte Tools

Fragebögen

Fragebögen können halbautomatisch über ein Webfrontend im Inter- oder Intranet abgefragt werden oder klassisch über Papierbögen. Mit Fragebögen lassen sich idealerweise Erfahrungen, Wahrnehmungen und Gefühle erfassen.

Datenblätter

Datenblätter können ebenfalls halbautomatisch oder klassisch abgefragt werden. Der Unterschied zu Fragebögen liegt darin, dass sie statischer und technischer sind. Sie können also viel exakter technische Maße aufnehmen.

Interviews

Interviews sind Meetings mit einer oder mehreren Personen. Interviews sind dynamischer als vorige Methoden, da neben den vorgegebenen Fragen auch noch zusätzlicher Input gesammelt werden kann. Der Aufwand ist zwar größer, dies kann sich aber lohnen.

Beobachtung

Beobachtung bedeutet, dass eine speziell geschulte Person (der Beobachter) am Entwicklungsprozess im Unternehmen teilnimmt. Der Beobachter observiert einen definierten Prozess oder eine Aktion eines Prozesses wie beispielsweise die Erstellung von Berichten oder die Zusammenarbeit im Team. Die Beobachtung kann, je nach Prozessgröße, einige Wochen bis hin zu Jahren dauern. Die Ergebnisse dieser Methode werden in einem Logbuch festgehalten.

Automatisierte Tools

Automatisierte Tools können nicht immer verwendet werden, machen aber die Arbeit leichter und schneller. Besonders wird die automatische Datenerhebung bei allen Prozessen verwendet, deren Output (z.B. Quellcode) bereits in elektronischer Form vorliegt.

Validierung

Unabhängig vom verwendeten Verfahren müssen die erfassten Rohdaten validiert und aufbereitet werden, bevor Sie zur Analyse verwendet werden können. Validieren bedeutet in diesem Zusammenhang nicht nur das reine Überprüfen der Rohdaten auf Korrektheit, sondern auch das Kontrollieren auf Vollständigkeit und Konsistenz . Es kommt immer wieder vor, dass in den Rohdaten Ausreißer vorhanden sind. Ausreißer sind Werte, die von den restlich erhobenen Werten stark abweichen. Diese müssen aufgespürt werden, i.d.R. automatisiert, und entfernt werden, da sie sonst das Messergebnis verfälschen würden.

Analyse

Sobald valide Rohdaten vorliegen beginnt der Abschnitt der Analyse. Die Datenanalyse ist nötig, um Messergebnisse zu erhalten. Die validierten und analysierten Messergebnisse helfen dann wiederum, die Messziele (oder GQM-Ziele) anzutreiben und Verbesserungen einzulei-ten. Hier erkennt man auch sehr gut die Bottom-Up Interpretation des GQM-Modells.

Schritt 6: Erfahrungen zusammenfassen und anwenden

In diesem letzten Arbeitsschritt des GQM-Modells werden alle zuvor gewonnenen Ergebnisse zusammen mit einer Beschreibung gebündelt. Es gibt hier wiederum zwei Arten der Ergebnisbündelung:

- Messergebnisse
- Lessons Learned

Messergebnisse

Messergebnisse in Form von validierten und analysierten Daten liegen mit dem Abschluss des dritten Schritts vor. Diese werden hier nochmals zusammengefasst und stehen zur weiteren Bearbeitung zur Verfügung.

Lessons Learned

Lessons Learned ist eine komprimierte Sammlung von Erfahrungen, Entwicklungen, Hinweisen, Fehlern, Risiken etc., die während eines Prozesses gemacht wurden. Man bündelt sie in Lessons Learned zusammen, um sie zu einem späteren Zeitpunkt in einer ähnlichen Situation, bzw. für eine gleiche Problemstellung nochmals verwenden zu können. Im Gegensatz zu den Messergebnissen sind Lessons Learned eher eine qualitative Auswertung.

Zusammenfassung

Das Goal-Question-Metric-Modell ist ein Qualitätsmodell zur Sicherung und Verbesserung der Qualität von Prozessen, welches von Dr. Victor R. Basili entwickelt wurde. GQM ist nicht auf einen bestimmten Prozess spezialisiert, sondern vielseitig einsetzbar. Es hilft technische, wie auch personenbezogene Prozesse zu analysieren. Weiterhin handelt es sich um ein zielorientiertes Modell, das in sechs Schritten aufgebaut ist. Die ersten drei Schritte nennen sich auch „Definitionsphase“. In der Definitionsphase werden zunächst die Ziele (goals) definiert, die erreicht werden sollen. Fragen (questions) helfen die Ziele besser zu spezifizieren und Metriken (metrics) werden definiert, um den Prozess messbar zu machen. Die letzten drei Schritte werden auch „Interpretationsphase“ bezeichnet. In dieser Phase wird ein Plan erstellt, der beschreibt, wie die nötigen Messdaten, für die in der Definitionsphase festgelegten Metriken, eingeholt werden. Mit Hilfe des Plans werden die Rohdaten erfasst, anschließend validiert und analysiert. Zum Schluss werden die gemachten Ergebnisse festgehalten, um sie nochmals anwenden zu können und Verbesserungsmaßnahmen einzuleiten. Ein Zusammenschrift der Ergebnisse in Form von „Lessons Learned“ ist ebenfalls eine gängige Maßnahme.

Literatur

- V. R. Basili, H. D. Rombach: *The TAME project. Towards improvement-oriented software environments.* In: *IEEE Transactions on Software Engineering.* Bd. 14, Nr. 6, 1988, S. 758–773, doi:10.1109/32.6156 [1].
- Reiner Dumke: *Software Engineering. Eine Einführung für Informatiker und Ingenieure: Systeme, Erfahrungen, Methoden, Tools.* Bd. 4, 2003, S. 234.
- Helmut Balzert: *Lehrbuch der Softwaretechnik: Softwaremanagement*. Spektrum Akademischer Verlag, 2008, ISBN 978-3-8274-1161-7
- Rini van Solingen und Egon Berghout: *Goal/Question/Metric Method*. McGraw-Hill Publishing Company, 1999, ISBN 978-0-07-709553-6

Weblinks

- V. Basili, G. Caldiera, H. D. Rombach: *The Goal Question Metric Approach* [2]. In: *Encyclopedia of Software Engineering.* John Wiley & Sons, 1994, S. 528–532.
- Software Acquisition Gold Practice Goal-Question-Metric (GQM) Approach [3]
- Handbuch: Messen und Bewerten von Webapplikationen mit der Goal/Question/Metric Methode [4] (PDF-Datei)

Referenzen

[1] http://dx.doi.org/10.1109%2F32.6156
[2] http://www.cs.umd.edu/~basili/publications/technical/T87.pdf
[3] http://www.goldpractices.com/practices/gqm/index.php
[4] http://publica.fraunhofer.de/eprints/urn:nbn:de:0011-n-176425.pdf

Halstead-Metrik

Die **Halstead-Metrik** ist eine 1977 von Maurice Howard Halstead vorgestellte Softwaremetrik. Sie gehört zu den statischen, analysierenden Verfahren der Komplexitätsmessung von Software.

Hierbei wird die Systemkomponente nicht aktiv ausgeführt (wie bei den dynamischen Verfahren), sondern gezielt Informationen über den Prüfling mit analytischen Mitteln gesammelt.

Die Halstead-Metrik bedient sich hierbei der Annahme, dass ausführbare Programmteile aus Operatoren und Operanden aufgebaut sind. Die Definition, was die zu betrachtenden Operatoren und Operanden sind, ist dabei eine der Aufgaben vor dem Einsatz einer Halstead-Metrik. Typischerweise werden z. B. Variablen und Konstanten als Operanden betrachtet; Schlüsselwörter, logische und Vergleichsoperatoren usw. als Operatoren.

Es werden dann für jedes Programm folgende Basismaße gebildet:

- Anzahl der verwendeten unterschiedlichen Operatoren (n_1) und Operanden (n_2), zusammen die Vokabulargröße n .
- Anzahl der insgesamt verwendeten Operatoren (N_1) und Operanden (N_2), zusammen die Implementierungslänge N .

Hieraus werden dann die Größen Halstead-Länge (HL) und Halstead-Volumen (HV) errechnet:

- $HL = n_1 \cdot \log_2 n_1 + n_2 \cdot \log_2 n_2$
- $HV = N \cdot \log_2 n$

Aus den Basisgrößen kann man verschiedene Kennzahlen berechnen, z. B.:

- $D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$

(D = Schwierigkeit ein Programm zu schreiben bzw. zu verstehen, z. B. bei einem Code-Review)

Die Halstead-Metrik ist leicht zu ermitteln und zu berechnen, automatisierbar, für alle Programmiersprachen einsetzbar und überraschenderweise meist ein wirklich gutes Maß für die Komplexität. Der Nachteil ist, dass sie nur lexikalische/textuelle Komplexität misst.

Siehe auch

- McCabe-Metrik

Literatur

- Maurice Howard Halstead: *Elements of software science.* Elsevier, New York u.a. 1977, ISBN 0-444-00205-7 (Operating and programming systems series; 2).

Weblinks

- Messung von Halstead-Metriken [1]
- Berechnung von McCabe- und Halstead-Metriken anhand eines Beispielprojekts [4] (PDF-Datei; 737 kB)

http://www.fit-for-bit.de/Software-mit-Halstead-Metriken-bewerten.php

Referenzen

[1] http://www.verifysoft.com/de_halstead_metrics.html

McCabe-Metrik

Die **McCabe-Metrik** (auch **zyklomatische Komplexität** - *cyclomatic complexity*) ist eine Software-Metrik, mit der die Komplexität eines Software-Moduls gemessen werden kann. Die zyklomatische Komplexität wurde bereits 1976 durch Thomas J. McCabe eingeführt.

Hinter der Software-Metrik von McCabe steckt der Gedanke, dass ab einer bestimmten Komplexität das Modul für den Menschen nicht mehr begreifbar ist. Er definiert seine *cyclomatic complexity* auf dem Kontrollflussgraphen eines Moduls. Einfach ausgedrückt ist das Komplexitätsmaß nach McCabe gleich der Anzahl der binären Verzweigungen plus 1.

Die so ermittelte *McCabe-Zahl* ist ein Maß für die Komplexität des Kontrollflusses eines Programms (Funktion, Prozedur oder Stück Code). Betrachtet man den Kontrollflussgraphen, wobei die Anweisungen als Knoten und der Kontrollfluss zwischen den Anweisungen als Kanten dargestellt sind, dann ist die McCabe-Zahl M definiert als $M = e - n + 2p$

e: Anzahl Kanten im Graphen

n: Anzahl Knoten im Graphen

p: Anzahl der einzelnen Kontrollflussgraphen (ein Graph pro Funktion/Prozedur)

Bei Betrachtung eines einzelnen Kontrollflussgraphen (also p=1) gilt $M = b + 1$ mit

b: Anzahl Binärverzweigungen, also bedingte Anweisungen mit genau zwei Zweigen, z. B. IF-Anweisungen.

M ist eine untere Schranke für die Anzahl der möglichen Wege durch ein Programm.

M ist außerdem eine obere Schranke für die Anzahl der Testfälle, die nötig sind, um eine vollständige Kantenabdeckung des Kontrollflussgraphen zu erreichen.

Laut McCabe sollte die zyklomatische Zahl eines in sich abgeschlossenen Teilprogramms nicht höher als 10 sein, da sonst das Programm zu komplex und zu schwer zu testen ist. Diese Regel ist allerdings umstritten, da sich die

zyklomatische Zahl nur dann erhöht, wenn verzweigende Anweisungen wie IF eingefügt werden, aber nicht beim Einfügen sonstiger Anweisungen (zum Beispiel einer Bildschirmausgabe). Es kann also lediglich eine Aussage über den Testaufwand (Anzahl der zu testenden unabhängigen Programmpfade) getroffen werden.

Kritik

Komplexitätsmaße sind für Menschen mitunter nicht intuitiv zu erfassen, so kann im folgenden Beispiel von Unübersichtlichkeit für Menschen keine Rede sein, jedoch wird als Komplexitätsmaß ein recht hoher Wert angegeben, zum Beispiel von dem Werkzeug CodeAnalyzer.

```
const String wochentagsName(const int nummer) {
  switch(nummer)
  {
    case 1: return "Montag";
    case 2: return "Dienstag";
    case 3: return "Mittwoch";
    case 4: return "Donnerstag";
    case 5: return "Freitag";
    case 6: return "Samstag";
    case 7: return "Sonntag";
  }
  return "(unbekannter Wochentag)"
}
```

In der Praxis wird die switch-Konstruktion häufig für Nachschlageaufgaben dieser Art eingesetzt. Die Methode umfasst acht Kontrollflusspfade, hat entsprechend eine hohe Komplexitätszahl von 9 und ist trotzdem vom Menschen auf einen Blick zu überblicken. Hier geht der Wert des Komplexitätsmaßes in Bezug auf den Menschen also verloren. Für die Einschätzung des Aufwandes für eine vollständige Testabdeckung spielt die subjektive Bewertung im Vergleich zum Komplexitätsmaß jedoch keine Rolle.

Literatur

- T. J. McCabe: *A Complexity Measure.* in: *IEEE Transactions on Software Engineering*, Band SE-2, 308-320. 1976.
- Helmut Balzert: *Lehrbuch der Software-Technik ; Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*, Seiten 481-482.

Weblinks

- A Complexity Measure [1] - Thomas J. McCabe, 1976 (PDF-Datei; 1,68 MB)
- Berechnung der McCabe-Metrik (Beispielprojekt) [4] - (PDF-Datei; 0,74 MB)
- Messung von McCabe-Metriken [2]

Referenzen

[1] http://www.literateprogramming.com/mccabe.pdf
[2] http://www.verifysoft.com/de_mccabe_metrics.html

Randthemen

Softwarequalität

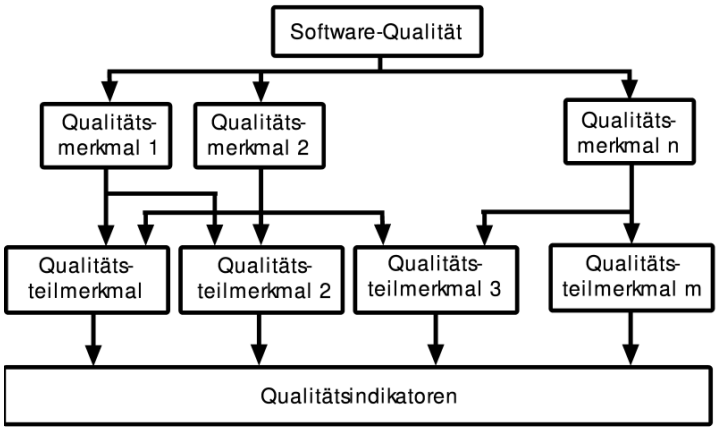
"Unter **Softwarequalität** versteht man die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen" (Ist/Soll) [1] . Diese Definition bezieht sich damit ausschließlich auf die Produktqualität und nicht auf die Prozessqualität.

Qualitätsmodelle

Konzept

Der Begriff der Softwarequalität selbst ist nicht operabel und in der Praxis direkt anwendbar. [1] Deshalb existieren Qualitätsmodelle, die durch eine weitere Detaillierung und Konkretisierung das Konzept der Softwarequalität operationalisieren. Dies leisten die Qualitätsmodelle durch Ableiten von Unterbegriffen. Dadurch entsteht ein Baum (oder ein Netz) von Begriffen und Unterbegriffen.

Die Qualitätsmerkmale tragen im Englischen die Bezeichnung *factor*, ein Qualitäts-Teilmerkmal heißt *criterion* und die Qualitäts-Indikatoren *metrics*. Deswegen erscheinen derartige Qualitätsmodelle in der Literatur auch als FCM-Modelle (z.B. FURPS, Boehm et al. 1978, DGQ-Modell 1986, McCall et al. 1977[2]). Bei den Blattknoten im Baum des Qualitätmodells, den Qualitätsindikatoren, sollte es sich um beobachtbare oder messbare Sachverhalte handeln. Hier können beispielsweise Softwaremetriken zum Einsatz kommen.



Qualitätsmodell

Modelle

Es existieren bereits ausgearbeitete Qualitätsmodelle, wie zum Beispiel die ISO/IEC 9126 bzw. DIN 66272.

Es existieren allerdings auch Vorgehensmodelle, wie der GQM-Ansatz, die zu individuellen Qualitätsmodellen führen.

Sicherstellung der Qualität

Für die Sicherstellung, dass die Software bezüglich der verschiedenen Qualitätsmerkmale den Anforderungen entspricht, existieren verschiedene Vorgehensmodelle und -methoden.

Einige Modelle:

- Organisatorische Qualitätsmodelle, zum Beispiel das Capability Maturity Model (CMM)
- Prozessmodelle, wie zum Beispiel
  - Capability Maturity Model Integration
  - der Rational Unified Process (RUP)
  - Agile Methoden
  - das V-Modell

Diese Modelle lassen sich eher dem Konzept der Prozessqualität zuordnen. Dieses geht davon aus, dass ein qualitativ hochwertiger Prozess der Produkterstellung die Entstehung von qualitativ hochwertigen Produkten begünstigt. Deshalb stellen die obigen Modelle Qualitätsanforderungen an den Prozess, in dem die Software entwickelt wird.

Einige Methoden:

- iterative Software-Entwicklung
- das zur Methode gewordene Spiralmodell
- Softwaretests
  - statische Analyse
  - Code Reviews
  - Keyword-Driven Testing
- Refaktorisierung
- Paarprogrammierung
- Testgetriebene Entwicklung

Softwaretests, Refaktorisierung und Code Reviews gehen direkt auf die Produktqualität ein. Das konkrete Produkt wird untersucht und bearbeitet, damit es die gestellten Qualitätsanforderungen möglichst gut erfüllt.

Die Modelle lassen sich teils, die Methoden größtenteils miteinander kombinieren. Interessant sind die Modelle der agilen Prozesse wie das Extreme Programming insbesondere deshalb, weil sie Synergieeffekte des gleichzeitigen Einsatzes verschiedener Methoden nutzen.

Weblinks

- Global Association for Software Quality [3]
- Arbeitskreis Software-Qualität und Fortbildung [4]
- International Software Quality Institute [5]
- Motor Industry Software Reliability Association (MISRA) [6]
- Software-Engineering und Software-Qualität in Open-Source Projekten [7] (Creative Commons Lizenz)
- iqnite - Die Konferenz für Software-Qualitätsmanagement und -Testen [8]

Referenzen

[1] Helmut Balzert', 'Lehrbuch der Softwaretechnik, Teil 2: Softwaremanagement, Software-Qualitaetssicherung, Unternehmensmodellierung, *1998, S. 257, ISBN 3-8274-0065-1*
[2] McCall, J.A., Richards, P.K. and Walters, G.F. (1977) Factors in software quality, Vols I-III, Rome Air Development Centre, Italy
[3] http://www.gasq.org/
[4] http://www.asqf.de/
[5] http://www.isqi.org
[6] http://www.misra.org.uk
[7] http://www.ebusiness-akademie.de/expertenwissen/7_software-engineering-und-software-qualitaet-in-open-source-projekten.htm
[8] http://www.iqnite-conferences.com/iqnite-de/

Wirtschaftlichkeit

Wirtschaftlichkeit ist ein allgemeines Maß für die Effizienz, bzw. für den rationalen Umgang mit knappen Ressourcen. Sie wird allgemein als das Verhältnis zwischen erreichtem Erfolg und dafür benötigten Mitteleinsatz definiert. Das Ziel ist, mit einem möglichst geringen Aufwand einen gegebenen Ertrag zu erreichen oder mit einem gegebenen Aufwand einen möglichst großen Ertrag zu erreichen.

Dies lässt sich mit folgender Formel darstellen:

$$\text{Wirtschaftlichkeit} = \frac{\text{Ertrag}}{\text{Aufwand}}$$

Hierbei ist:

- *Ertrag* = der in Geld gemessene Wertezuwachs zum Zeitpunkt der Betrachtung
- *Aufwand* = der in Geld gemessene Wert aller verbrauchten Güter und/oder Leistungen

Wirtschaftlichkeit ist gegeben, wenn der Quotient aus Ertrag und Aufwand gleich oder größer 1 ist.

- Wenn das Ergebnis größer als 1 ist, so ist eine Wirtschaftlichkeit gegeben – Wertezuwachs
- Wenn das Ergebnis gleich 1 ist, so ist die Wirtschaftlichkeit gegeben – kostendeckend
- Wenn das Ergebnis kleiner als 1 ist, so ist *keine* Wirtschaftlichkeit gegeben – Verlust

Unterschied zwischen Wirtschaftlichkeit und Rentabilität

Die Rentabilität ist das Verhältnis zwischen erzieltem Erfolg (z. B. Gewinn) und eingesetztem Kapital (Gesamt- oder Eigenkapital). Hierbei wird das Kapital, daher der in Geld gemessene Wert, in Beziehung gesetzt. Die Rentabilität ist eine Kennzahl für den Erfolg und wird als Prozentsatz angegeben.

Bei der Wirtschaftlichkeit kann der Ertrag als Wertezuwachs, nur als Wert von verkauften Gütern oder auch nur als erbrachte Leistung in Geldwert eingesetzt werden. Der Aufwand kann auch in Arbeitsstunden, Materialbedarf oder anderen Leistungen, umgerechnet in Geldwert, eingesetzt werden. Die Wirtschaftlichkeit ist ein Maß für Sparsamkeit oder Effizienz; sie ist dimensionslos.

Wirtschaftlichkeit und Rentabilität sind Synonyme für Ökonomie.

Wirtschaftlichkeit in der Produktionsplanung

Die Produktions- und Kostentheorie formen die theoretischen Grundlagen der Produktionsplanung.

Die Kostentheorie hat das Ziel, die kostengünstigsten Verfahren für eine vorgegebene Produktmenge zu bestimmen. Die Kennzahl der Kostentheorie ist die Wirtschaftlichkeit.

$$\text{Wirtschaftlichkeit} = \frac{\text{Erlös}}{\text{Kosten}}$$

Hierbei ist:

- *Erlös* = der in Geld gemessene Wert der durch den Verkauf von Waren (Erzeugnissen) oder Dienstleistungen sowie aus Vermietung oder Verpachtung erzielt wurde.
- *Kosten* = der in Geld gemessene Wert aller verbrauchten Güter und Leistungen zur Produktion einer vorgegebenen Erzeugnismenge.

Auch hier gilt, dass die Wirtschaftlichkeit gegeben ist, wenn der Quotient aus Erlös und Kosten gleich oder größer 1 ist. Ist das Ergebnis gleich 1, wurde nur kostendeckend produziert.

Beurteilung der Wirtschaftlichkeit eines Unternehmens

Eine Beurteilung und Kontrolle der Wirtschaftlichkeit eines Unternehmens kann durch Umschlagskennzahlen ermöglicht werden. Dabei werden die Kosten den Leistungen gegenübergestellt.

Umschlagskennzahlen sind:

- Lagerumschlagshäufigkeit
- durchschnittliche Lagerdauer
- Umsatzhäufigkeit
- durchschnittliche Kreditdauer

Verfahren der Wirtschaftlichkeitsrechnung in der Investitionsrechnung

Die Wirtschaftlichkeitsrechnung untersucht ein oder mehrere Investitionskandidaten auf deren Vorteile bei Investitionen unter bestimmten Voraussetzungen.

In der Investitionsrechnung werden statische oder dynamische Verfahren der Wirtschaftlichkeitsrechnung angewendet.

Statische Verfahren der Wirtschaftlichkeitsrechnung sind:

- Kostenvergleichsrechnung
- Gewinnvergleichsrechnung
- Rentabilitätsrechnung
- Amortisationsrechnung

Nachteile der statischen Verfahren sind:

- kurzfristige Betrachtungsweise
- Nichtberücksichtigung des zeitlichen Anfalls von Ein- und Auszahlungen

Dynamische Verfahren der Wirtschaftlichkeitsrechnung sind:

- Kapitalwertmethode
- Annuitätenmethode
- Interner Zinsfuß
- Endwertmethode
- Marktzinsmethode.

Die dynamischen Verfahren berücksichtigen

- die Bewertung von Ein- und Auszahlungen entsprechend ihrem zeitlichen Anfall
- die genaue Erfassung von Ein- und Auszahlungen während der Nutzungsdauer

In der Privatwirtschaft wird in der Regel keine möglichst große Wirtschaftlichkeit, sondern ein möglichst großer Gewinn angestrebt. Dies hat je nach Marktform Konsequenzen für die Wirtschaftlichkeit der Privatwirtschaft.

„Auf dem Mengenanpassermarkt führen das Streben nach Gewinn und das Streben nach Wirtschaftlichkeit zu identischen Ergebnissen. Anders ist es z. B. beim Monopolisten, der die angebotene Menge einschränkt und gleichwohl – zu unwirtschaftlichen Stückkosten anbietend – seinen Gewinn maximiert“.

Siehe auch

- Kostenwirtschaftlichkeit
- Ökonomisches Prinzip
- Rechnungshof
- Effektivität

Literatur

- Erich Gutenberg: Kategorische Umklammerung des Prinzips der Wirtschaftlichkeit.
- Günter Wöhe und Ulrich Döring *Einführung in die Allgemeine Betriebswirtschaftslehre*, 22. Aufl., München 2005, 1200 S., ISBN 3800632543
- Volker Oppitz und Volker Nollau: *Taschenbuch Wirtschaftlichkeitsrechnung*, Carl Hanser Verlag 2003, 400 S., ISBN 3446224637
- Volker Oppitz: *Gabler Lexikon Wirtschaftlichkeitsberechnung*, Gabler-Verlag 1995, 629 S., ISBN 3409199519
- Hans Jung „Allgemeine Betriebswirtschaftslehre“. R. Oldenbourg Verlag, München 1994.

Fehlerquotient

Als **Fehlerquotient**, **Fehlerdichte** oder **Fehlerrate** bezeichnet man den relativen Anteil von fehlerhaften Elementen im Verhältnis zur Gesamtheit.

Der Fehlerquotient bei der Speicherung und Übertragung binärer Daten ist die Bitfehlerhäufigkeit.

Anwendungsbeispiel

Der Fehlerquotient könnte beispielsweise in der Schule zur Verwendung kommen:

> Beim Verfassen eines Textes von 400 Wörtern hat ein Schüler elf Rechtschreibfehler gemacht. Um den Fehlerquotienten zu erhalten teilt man die Anzahl der Fehler durch die Gesamtzahl der Wörter. Er lautet also:
>
> $$\frac{11}{400} = 0{,}0275 = 2{,}75\%$$
>
> Allgemein lässt sich schreiben:
>
> $$\frac{\text{Fehler}}{\text{Wörter}} = \frac{\text{Fehler} \cdot 100}{\text{Wörter}}\%$$
>
> Anhand des Fehlerquotienten, in diesem Beispiel wären es 2,75 %, kann der Lehrer dann eine Note vergeben; falls es sich um einen Aufsatz o.Ä. handelt, muss der Inhalt natürlich noch separat bewertet werden.

Fehlerdichte in der Informatik

In der Informatik bezeichnet die Fehlerdichte die Anzahl an Fehlern pro 1.000 Zeilen Code (Lines of Code). Fehlerfreie Software ist aus betriebswirtschaftlichen und technischen Gründen in der Praxis unmöglich zu erstellen; anzustreben ist daher eine möglichst geringe, zu den Anforderungen der Software passende Fehlerrate - die angestrebte Fehlerrate ist somit während der Analysephase zu definieren. Bei Software, deren Ausfall Menschenleben kosten könnte (wie beispielsweise Militätsysteme oder Krankenhaussysteme), wird üblicherweise eine Fehlerdichte von < 0,5 Fehlern pro 1.000 Zeilen Code angestrebt.

Anzustreben ist üblicherweise eine Fehlerdichte von < 2, normal ist 2 - 6 und akzeptable aber nur im Webbereich ist 6 - 10. Über 10 gilt als Malpractice und kann vor einem Gericht zur Kompensationszahlung führen.[1]

Die Fehlerdichte kann auch zur Klassifizierung der Produktreife von Software herangezogen werden:

Fehlerdichte	Klassifizierung der Programme
< 0,5	stabile Programme
0,5 .. 3	reifende Programme
3 .. 6	labile Programme
6 .. 10	fehleranfällige Programme
> 10	unbrauchbare Programme

[2]

Referenzen

[1] Linda Laird & Carol Brennan: "Software Measurement and Estimation: A Practical Approach.", Wiley & Sons, 2006

[2] Carper Jones: "Programming Productivity", McGraw-Hill, New York, 1986

Paretoprinzip

Das **Paretoprinzip**, auch **Pareto-Effekt**, **80-zu-20-Regel**, besagt, dass 80 % der Ergebnisse in 20 % der Gesamtzeit eines Projekts erreicht werden. Die verbleibenden 20 % der Ergebnisse verursachen die meiste Arbeit.

Ableitung

Die Pareto-Verteilung beschreibt das statistische Phänomen, wenn eine *kleine* Anzahl von *hohen* Werten einer Wertemenge mehr zu deren Gesamtwert beiträgt, als die *hohe* Anzahl der *kleinen* Werte dieser Menge.

Vilfredo Pareto untersuchte die Verteilung des Volksvermögens in Italien und fand heraus, dass ca. 20 % der Familien ca. 80 % des Vermögens besitzen. Banken sollten sich also vornehmlich um diese 20 % der Menschen kümmern und ein Großteil ihrer Auftragslage wäre gesichert.

Daraus leitet sich das Pareto-Prinzip ab. Es besagt, dass sich viele Aufgaben mit einem Mitteleinsatz von ca. 20 % so erledigen lassen, dass 80 % aller Probleme gelöst werden. Es wird häufig kritiklos für eine Vielzahl von Problemen eingesetzt, ohne dass die Anwendbarkeit im Einzelfall belegt wird. Allerdings ist das „Prinzip“ eine gute Merkhilfe für den Wertebereich eines für zwei Quantile berechneten Theil-Indexes (s. u.): Dieses Ungleichverteilungsmaß hat bei einer 50-50-Verteilung den Wert „0“. Knapp über einer 80-20-Verteilung ist der Wert "1". (Bei einem weiteren Anstieg in Richtung einer 100-0-Verteilung steigt der Theil-Index theoretisch ins Unendliche.)

Die hier vorgenommene Aufteilung einer Gesellschaft in zwei Teile ist eine Aufteilung in zwei „a-Fraktile“ (*siehe Hauptartikel Quantil*).

Pareto weist darauf hin, dass diese Regel nur gilt, wenn die Elemente des Systems unabhängig voneinander sind. Durch Interdependenz der Elemente (wie etwa in einer Organisation und allen soziotechnischen Systemen) wird die Situation verändert. In der Praxis ist folglich die Zahl der relevanten Elemente sehr gering; sehr wenige Elemente bestimmen fast den gesamten Effekt.

Beispiele

Das Pareto-Prinzip kann bei vielen - auch alltäglichen - Fragestellungen beobachtet werden. 20 % der eingesetzten Zeit bringt 80 % der Ergebnisse (*siehe auch: Zeitmanagement*). In einem durchschnittlichen Haushalt verursachen 20 % der Kostenpositionen 80 % der Kosten. In einer Wohnung weisen 20 % des Teppichs 80 % der Gesamtabnutzung auf. In einem Unternehmen werden 80 % des Umsatzes mit 20 % der Kunden erzielt. 80 % eines Textes werden mit 20 % der Wörter bestritten (z. B. der, die, das usw.).

Viele Verteilungen in der Natur folgen einem Skalengesetz, sehr oft einem Potenzgesetz, also einer Pareto-Verteilung.

- Wohlstandsverteilung auf Individuen: siehe oben
- Größe von menschlichen Siedlungen: Viele kleine Dörfer mit wenig Einwohnern, die Masse der Menschen wohnt aber in wenigen großen Städten.
- Werte im Lager eines Industrieunternehmens: Viele Schrauben etc., die nicht viel kosten, aber wenige sehr teure Zukaufsartikel.
- Aufwände bei Vorhaben: 20 % Aufwand bringen 80 % Ergebnis, die restlichen 20 % des Ergebnisses brauchen aber 80 % des gesamten Aufwandes.
- 75 % des Welthandels finden unter 25 % der Menschen statt.
- Ankunftszeiten und Paketgrößen in Netzwerken, insbesondere WWW, da hier Nutzerbedenkzeiten zu beachten sind.[1]
- In der Informetrie: Lotkas Gesetz

- Die Pareto-Verteilung wird in der Versicherungs- und Finanzmathematik zur Modellierung von extremen Ereignissen (z. B. Großschäden, starke Kursveränderungen von Aktien) eingesetzt.
- 80 % aller Supportanfragen im Internet beziehen sich immer wieder auf die gleichen 20 % (oder weniger) der Problemstellungen.

Literatur

- Koch, Richard: *Das 80/20-Prinzip. Mehr Erfolg mit weniger Aufwand*. Frankfurt/M.; New York, 1998. OT: The 80/20 principle. The secret of achieving more with less, 1997. (Zusammenfassung aus 'Campus Management') [2]

Referenzen

[1] http://www.cs.bu.edu/faculty/crovella/paper-archive/web-tails.ps
[2] http://www.ephorie.de/das_80-20_prinzip.htm

Mean Time Between Failures

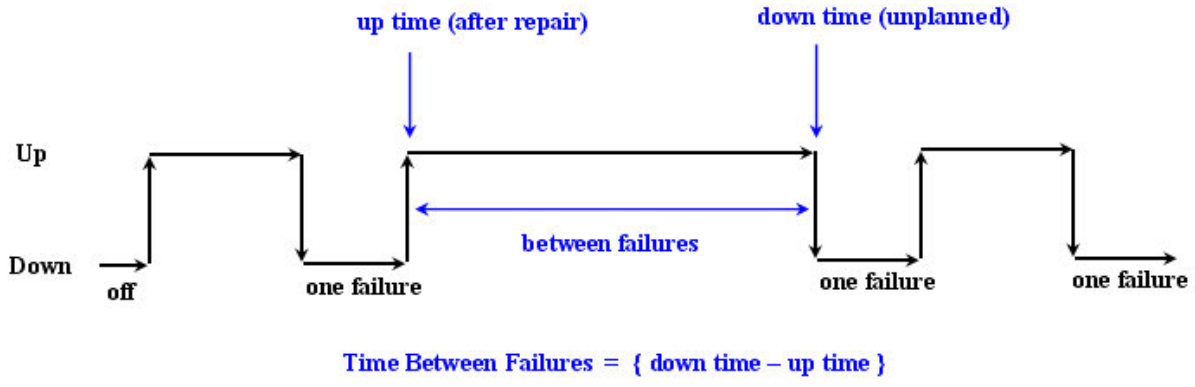
MTBF ist die Abkürzung für das englische *Mean Time Between Failures*, zu deutsch die **mittlere Betriebsdauer zwischen Ausfällen**. Sie gilt für Einheiten, die instandgesetzt werden; Betriebsdauer meint die Betriebszeit zwischen zwei aufeinanderfolgenden Ausfällen einer instandzusetzenden Einheit.

Die Definition nach IEC 60050 (191) lautet: *Der Erwartungswert der Betriebsdauer zwischen zwei aufeinanderfolgenden Ausfällen.*

Für Einheiten die *nicht* instandgesetzt werden, ist der Erwartungswert (Mittelwert) der Verteilung von Lebensdauern die mittlere Lebensdauer MTTF (engl. *mean time to failure*). Umgangssprachlich werden die Begriffe oft synonym verwendet (in diesem Fall hat sich das Backronym „mean time *before* failure“ eingebürgert).

Grundlegendes

MTBF ist *ein* Maß für die Zuverlässigkeit von Einheiten (Baugruppen, Geräte oder Anlagen), die nach einem Ausfall (*down*) instandgesetzt (*up*) werden. Dieses Verhalten lässt sich anhand folgender Grafik verdeutlichen:



Der Betrieb der Einheit liegt zwischen dem Ereignis der Inbetriebnahme (*up-time*) und dem Ereignis des Ausfalls (*down-time*). Formal lässt sich die MTBF über einen langen Zeitraum, in welchen mit vielen Ausfällen und Inbetriebnahmen zu rechnen ist, ausdrücken als:

$$\mathrm{MTBF} = \frac{\Sigma(\text{down-time} - \text{up-time})}{n}.$$

wobei der Wert *n* die Anzahl der Ausfälle über den betrachteten, langen Zeitabschnitt angibt. Für den Fall, dass die Betriebsdauern exponentialverteilt sind, erhält man während der Brauchbarkeitsdauer einer Einheit die MTBF aus dem Kehrwert der dann konstanten Ausfallrate λ:

$$\mathrm{MTBF} = \frac{1}{\lambda}.$$

Je höher der MTBF-Wert, desto „zuverlässiger“ ist das Gerät. Ein Gerät mit einer MTBF von 100 Stunden wird im Mittel öfter ausfallen als ein gleichartiges Gerät mit einer MTBF von 1.000 Stunden.

Werden MTBF-Angaben gemacht, so sollten zusätzlich die umgebungs- und funktionsbedingten Beanspruchungen, die Ausfallkriterien und die Geltungsdauer mit angegeben werden (z. B. Umgebungstemperatur, Anzahl der Start-/Stop Zyklen pro Tag, Einhaltung von Wartungsvorschriften, etc.). Unter ungünstigen Betriebsbedingungen können wesentlich geringere MTBF-Werte (höhere Ausfallraten) auftreten. Andererseits kann durch ein Derating die MTBF größer (Ausfallrate kleiner) sein.

Aussagen über MTBF-Werte sind nur während der geplanten Geltungsdauer (z. B. Brauchbarkeitsdauer) gültig. Danach kann die Ausfallrate aufgrund von Abnutzungserscheinungen deutlich ansteigen.

Die MTBF muss weiters unterschieden werden von der Brauchbarkeitsdauer (engl. *useful life*) eines Geräts: Die Brauchbarkeitsdauer gibt die Zeitdauer an, auf die ein Gerät bei der Entwicklung ausgelegt wurde. Sie ist u.a. auch durch die Dimensionierung von Verschleißteilen bestimmt.

Ermittlung

Die MTBF kann zur Abschätzung von Ausfällen in Zeitintervallen verwendet werden. Beispielsweise sind bei Festplatten MTBF-Werte von 1'200'000 Stunden (WD RE3 500G) üblich, dies entspricht 137 Jahren. Aus dieser Zahl kann die Wahrscheinlichkeit berechnet werden, dass es während der Nutzungsdauer (oft 5 Jahre bei Festplatten) zu einem Ausfall kommt. Sie beträgt etwa:

$$p(T) = 1 - e^{-\frac{T}{MTBF}}$$

$$p(5a) = 1 - e^{-\frac{5a}{137a}} = 3,6\%$$

Schätzwerte für die MTBF können durch Lebensdauerversuche - fallweise auch mit erhöhten Beanspruchungen - ermittelt werden, in denen das Gerät z.B. Strahlungen, Feuchtigkeit, Erschütterungen, Hitze und Ähnlichem ausgesetzt wird. Konkret heißt das, dass zum Beispiel eine Computer-Maus auf einem Laufband 24 Stunden ununterbrochen betrieben wird, um zu schauen, wie viele Kilometer diese ohne Fehler zurücklegen kann. Tests sind nicht standardisiert, deshalb sind alle angegebenen Werte in der Regel nur innerhalb der Produktreihen eines Herstellers vergleichbar.

Eine andere Möglichkeit der Ermittlung der MTBF, die oft in den frühen Entwicklungsphasen angewendet wird, ist die Zuverlässigkeitsprognose. Damit lässt sich abschätzen, ob gesetzte Zuverlässigkeitsziele erreicht werden können. Dazu sind genaue Kenntnisse des Aufbaus des Gerätes und der verwendeten Bauelemente notwendig. Für viele Bauelemente existieren in Handbüchern Ausfallraten (Werte oft in **FIT** angegeben (1 FIT=10^(-9) 1/h) , Failure in Time bezeichnet). Die MTBF ist der Kehrwert der berechneten Ausfallrate der Baugruppe/Einheit, die sich aus der Summe der in Abhängigkeit von der Beanspruchung gewichteten Bauelementeausfallraten ergibt.

Die MTBF wird auch für die Berechnung der „stationären“ Verfügbarkeit (engl. *Availability*) eingesetzt. Die Verfügbarkeit gibt an, wie groß die Wahrscheinlichkeit ist, dass ein System bei Anforderung den spezifizierten Dienst anbietet:

$$A = 100\% \cdot \frac{MTBF}{MTBF + MTTR}$$

Mathematisch betrachtet beträgt die Ausfallwahrscheinlichkeit bei MTTF 63,2%, exakt *(1-1/e)*, oder anders ausgedrückt bei der Zeit MTTF sind etwa 2/3 der Geräte ausgefallen.

Def. Verfügbarkeit: der Anteil der Zeit, zu der man über ein System verfügen kann

Ähnliche Begriffe

- MTBM Mean Time Between Maintenance
- MTTF Mean Time To Failure
- MTTR Mean Time To Recover
- MTBO Mean Time Between Overhaul
- Hazardrate
- MCBF Mean Cycles Between Failure

Maßeinheit

Angaben über die Zuverlässigkeit technischer Produkte werden in Power-On Hours (POH) gemacht, zum Beispiel: "MTBF: 60.000 POH"

Normen

Für die Berechnung existieren Normen, beispielsweise

- DIN EN/IEC61709
- SN 29500 der Siemens AG (wird laufend aktualisiert, neben elektronischen auch vermehrt elektromechanische Komponenten wie Relais, Schütze, Leistungsschalter, Überlastrelais, Zeitrelais usw. Befehls- und Meldegeräte, Drucktaster, Leuchtmelder, Positionsschalter sind in Arbeit)
- SR-332 von Telecordia Technologies
- MIL-HDBK-217 des Militärs (wird nicht mehr gepflegt)

Literatur

- Patrick Gehlen: *Funktionale Sicherheit von Maschinen und Anlagen. Umsetzung der Europäischen Maschinenrichtlinie in der Praxis.* 1. Auflage. Publicis Corporate Publishing, ISBN 978-3-89578-281-7.
- Martin L. Shooman: *Reliability of Computer Systems And Networks.* 1. Auflage. Wiley Interscience, 2002, ISBN 0-471-29342-3.

Weblinks

- Genauere Erklärung zur MTBF [1] (englisch, pcguide.com)
- noch eine Erklärung zu MTBF [2] (englisch, storagereview.com)
- Ausführliche Darstellung zur MTBF am Beispiel von Festplatten (englisch) [3]
- Tutorial MTBF-Berechnung [4]
- Edinn M2 [5]

Referenzen

[1] http://www.pcguide.com/ref/hdd/perf/qual/specMTBF-c.html
[2] http://faq.storagereview.com/tiki-index.php?page=MTBF
[3] http://www.digit-life.com/articles/storagereliability/
[4] http://mtbf.polimore.com/index.php?Filename=help.1.1.en.php&LANGUAGE=de
[5] http://edinn.com/de/mttr.html

Testabdeckung

Als **Testabdeckung** bezeichnet man das Verhältnis an tatsächlich getroffenen Aussagen eines Tests gegenüber den theoretisch möglich treffbaren Aussagen bzw. der Menge der gewünschten treffbaren Aussagen. Die Testabdeckung spielt als Metrik zur Qualitätssicherung und zur Steigerung der Qualität insbesondere im Maschinenbau und der Softwaretechnik eine große Rolle.

In der Praxis wird die Testabdeckung durch verschiedene Kriterien beeinflusst. Die Testabdeckung lässt sich durch eine Erhöhung der Zahl an Messungen, Stichproben und Testfällen verbessern. Begrenzt wird die Testabdeckung in der Praxis jedoch durch die Kosten, die mit jedem Test verbunden sind.

Testabdeckung im Maschinenbau

Je nach Art, Aufwand und Nutzen der Tests werden einige Tests stichprobenartig, andere Tests vollständig durchgeführt. Ein einfach und automatisch durchzuführender Test wird mit jedem Produkt durchgeführt, da seine Kosten die Produktionskosten nur geringfügig erhöhen. Ein Crash-Test mit einem Fahrzeug wird jedoch natürlich nur mit Stichproben durchgeführt, da das getestete Produkt anschließend unbrauchbar wird.

Für 1000 produzierte Fahrzeuge könnte dies z.B. bedeuten, dass besonders aufwendige Tests und Crashtests nur mit einem einzigen Fahrzeug durchgeführt werden, während weniger aufwendige Tests mit einer größeren Zahl oder gar allen Fahrzeugen durchgeführt werden.

Notwendige aber aufwendige Tests werden in ihrer Häufigkeit und damit der Testabdeckung variiert. Liefert ein Test überwiegend oder ausschließlich positive Ergebnisse, wird seine Zahl verringert. Liefert ein Test negative Ergebnisse, wird er häufiger eingesetzt, bis die Veränderungen an der Produktion zu einer deutlichen Steigerung positiver Ergebnisse und damit wieder einer höheren Produktqualität geführt hat.

Die Kosten-Nutzen-Rechnung solcher Tests wird mit Hilfe der Stochastik durchgeführt. Wird z.B. nur mit 5 von 1000 Fahrzeugen ein Test durchgeführt, ob die elektrischen Fensterscheibenheber einwandfrei funktionieren, lässt sich mit Hilfe der Stochastik die statistische Relevanz und die Wahrscheinlichkeit einer Fehleinschätzung aufgrund des Testergebnisses berechnen.

Testabdeckung in der Softwaretechnik

Für die Testabdeckung in der Softwaretechnik spielt die Stochastik praktisch keine Rolle, da es sich bei Computerprogrammen nicht um seriengefertigte Einzelprodukte handelt, bei denen Tests mit Stichproben durchgeführt werden. Stattdessen werden Tests anhand der Spezifikation (Eigenschaften der Schnittstelle) oder der inneren Struktur einer zu testenden Software-Einheit definiert.

In der Softwaretechnik wird eine hohe Testabdeckung durch die Definition einer hohen Anzahl an Testfällen erreicht.

Nur für einige Testmethoden ist beim Softwaretest die Angabe eines Maßes für die Testabdeckung (Softwaremetrik) möglich, da die Bestimmung der Anzahl der möglichen Testfälle für reale Probleme oft nicht möglich ist.

Eine vollständige Testabdeckung stellt eine Ausnahme dar, weil die Anzahl möglicher Testfälle sehr schnell ungeheuer groß wird (durch kombinatorische Explosion). Ein vollständiger Funktionstest für eine einfache Funktion, die zwei 16-Bit-Werte als Argument erhält, würde schon 2^(16+16), also ca. 4 Milliarden Testfälle bedeuten, um die Spezifikation vollständig zu testen.

Stattdessen beschränkt man sich auf eine Auswahl sinnvoll erscheinender Tests für Grenzfälle. Beispiel: Eine Wurzelfunktion für rationale Zahlen könnte z.B. mit sämtlichen Elementen der Menge {-10; -1; -0,0000001; 0; 0,0000001; 1; 2; 3; 4; 5,25; 9; 10000} getestet werden. Als sinnvolle Auswahl von Testfällen für eine angemessene Testabdeckung gelten in der Regel verschiedenartige gültige Argumente, für Komponenten mit Robustheitsanforderung zusätzlich Grenzelemente (gerade noch gültige Argumente und gerade ungültige Argumente). Es hat sich zudem als erfolgreich erwiesen, im Fehlerfall das den Fehler auslösende Argument in die Menge der Testelemente aufzunehmen.

Zur Bestimmung der Testfälle wird zwischen verschiedenen Testmethoden unterschieden. Eine Klassifikation ist die in White-Box-Test und Black-Box-Test. Ein Blackbox-Test arbeitet auf der Spezifikation der Software-Einheit, ein White-Box-Test arbeitet mit Kenntnissen über die innere Struktur der zu testenden Software-Einheit.

Tools zur Messung der Software-Testabdeckung

- Bullseye Coverage - C++
- Clover - Java, .Net
- Cobertura - Java
- CodeCover - Java, Cobol
- coverage.py - Python
- Devel::Cover - Perl
- gcov - C
- EMMA - Java
- LDRA Testbed - C++
- NCover - .Net
- PartCover - .Net
- rcov - Ruby
- shcov - shell / bash Scripts
- Simulink Verification and Validation - Simulink-Modelle
- Sonar - Java
- Tessy - C/C++
- Testwell ctc++ - C, C++, Java, C#
- VBWatch - Visual Basic
- XDebug - PHP

Siehe auch

- Überdeckungstest

Personenstunde

Eine **Personenstunde** (auch veraltet **Mannstunde**, auf Englisch person hour genannt) ist die Menge an Arbeit, die eine Person durchschnittlich in einer Stunde schafft. Man verwendet diesen Begriff, um Schätzungen für die Gesamtmenge an Arbeit für die Erledigung einer Aufgabe zu errechnen. Zum Beispiel benötigt man vielleicht zwanzig Personenstunden, um einen Aufsatz zu schreiben, oder zehn Personenstunden, um ein großes Familienessen vorzubereiten.

Personenstunden enthalten keine Zeit für Pausen, sie stehen für die reine Arbeitszeit. Will man die Gesamtzeit für eine Aufgabe errechnen, müssen zusätzliche Pausen berücksichtigt werden. Der oben genannte Aufsatz wird nicht in zwanzig aufeinander folgenden Stunden fertig, sondern wird durch andere Aufgaben, Mahlzeiten, Schlaf und andere Ablenkungen unterbrochen.

Anwendungen

Ein Einsatzbereich der Personenstundenberechnung ist die Schätzung von Teamgröße und Gesamtdauer eines Projekts. Es darf aber nicht einfach die Anzahl von Personenstunden durch die Anzahl der Teammitglieder dividiert werden, genauso wenig wie neun Frauen ein Kind in einem Monat wachsen lassen können. Es gibt gerade für Softwaresysteme viele Formeln, die eine gewisse Mindestdauer von Projekten und die unterschiedlichen Schwierigkeitsgrade berücksichtigen. (siehe Barry Boehm, "Software Metrics")

Nur bei Fließbandarbeit ist es möglich, durch die Einstellung weiterer Mitarbeiter mehr Arbeit zu schaffen oder vorhandene Arbeit schneller zu schaffen.

Ein Fehler in der reinen Berechnung von Personenstunden entsteht zum Beispiel dadurch, dass Organisation, Ausbildung und Koordination zusätzlicher Arbeitskräfte mehr Zeit in Anspruch nehmen können, als die zusätzlichen Kräfte einsparen. Dies zeigte Frederick P. Brooks in seinem Software-Engineering-Buch *The Mythical Man-Month*, deutsch *Vom Mythos des Mann-Monats*.

Personenjahre

Ein ähnliches Konzept, Personenjahre (*PJ*, auch „Bearbeiterjahr“, *BJ* oder „Mannjahr“), wird für sehr große Projekte verwendet. Es ist die Arbeitsmenge, die eine Person durchschnittlich während eines Jahres arbeitet. Dieses hängt natürlich von der üblichen Arbeitswochendauer ab, sowie von der Anzahl der Urlaubstage, die von Land zu Land stark variieren. Als Wert wird aber oft 2000 Personenstunden eines Personenjahres verwendet.

Abhängig vom Stundenlohn entspricht ein Personenjahr einem Wertäqivalent von weniger als 1000 USD (Entwicklungsland) bis zu über 100.000 Dollar (in den G8-Staaten).

Andere Einheiten

Weitere gebräuchliche Einheiten sind der Personentag, die Personenwoche sowie der Personenmonat.

- Der *Personentag* (kurz *PT*, auch „Bearbeitertag“, *BT*, „Manntag“, *MT*) wird oft als 8 Personenstunden berechnet.
- Die *Personenwoche* (kurz *PW*, „Mannwoche“, *MW*) wird oft als fünf Personentage berechnet.
- Der *Personenmonat* (kurz *PM*, „Mannmonat“, *MM*) wird oft als 20 Personentage berechnet.

Nur implizit personenbezogene Aufwandsbezeichnungen sind Zeit-Werke, d.h. Tagewerk (TW), Wochenwerk (WW), usw.

Verwandte Themen

- Personenstunden sind eine Kenngröße der wissenschaftlichen Betriebsführung, des so genannten Taylorismus, der von Frederick Winslow Taylor, Henry Gantt, Frank Bunker Gilbreth und anderen begründet wurde.

Literatur

- Barry Boehm, Software Engineering Economics, ISBN 0-13-822122-7 ,1981
- Barry Boehm, Chris Abts und A. Winsor Brown, Software Cost Estimation with Cocomo II, ISBN 0-13-026692-2, 2000
- *The Principles of Scientific Management* [1] F.W.Taylor, 1911, online
- *Shop Management* [2], F.W.Taylor, 1911, online
- A Selection from Frederick Taylor's Essays [3], online

Weblinks

Personenjahr

- Definition als 2000 Personenstunden: http://www.onr.navy.mil/02/matoc/05_09/solicitations/docs/05-0002-02.pdf (Solicitation Number 05-0002-02: Support Services for the Office of Naval Research for the Legislative Affairs Office (United States Navy Office of Naval Research: Arlington, Virginia, USA, 2004)
- Definition als 2087 Personenstunden: Report 5, International Federation Of Professional And Technical Engineers Local 32: San Diego, California, 2000 [4](counting 311 "Non-available/Nonproductive" man-hours)

Referenzen

[1] http://www.eldritchpress.org/fwt/taylor.html
[2] http://onlinebooks.library.upenn.edu/webbin/gutbook/lookup?num=6464
[3] http://www.fordham.edu/halsall/mod/1911taylor.html
[4] http://www.ifptelocal32.com/CA/Report%205.htm

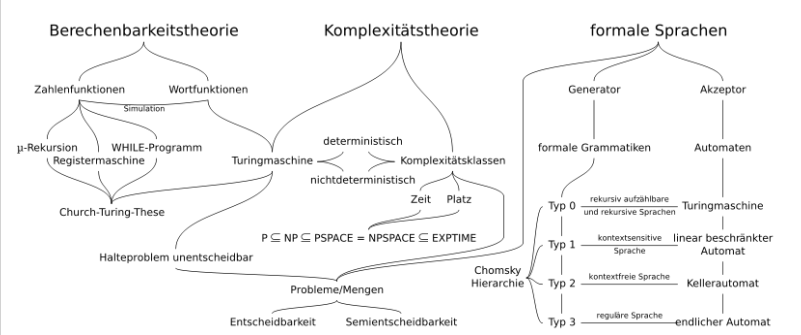
Komplexitätstheorie

Die **Komplexitätstheorie** als Teilgebiet der Theoretischen Informatik befasst sich mit der Komplexität von algorithmisch behandelbaren Problemen auf verschiedenen mathematisch definierten formalen Rechnermodellen. Die Komplexität von Algorithmen wird in deren Ressourcenverbrauch gemessen, meist Rechenzeit oder Speicherplatzbedarf. Es werden jedoch auch speziellere Komplexitätsmaße wie die Größe eines Schaltkreises oder die Anzahl benötigter Prozessoren bei parallelen Algorithmen untersucht.

Die Komplexitätstheorie unterscheidet sich von der Berechenbarkeitstheorie, die sich mit der Frage beschäftigt, welche Probleme prinzipiell algorithmisch gelöst werden können. Demgegenüber besteht das wichtigste Forschungsziel der Komplexitätstheorie darin, die Menge aller lösbaren Probleme zu klassifizieren. Insbesondere versucht man, die Menge der effizient lösbaren Probleme von der Menge der inhärent schwierigen Probleme abzugrenzen.

Einordnung in die Theoretische Informatik

Die Komplexitätstheorie gilt, neben der Berechenbarkeitstheorie und der Theorie der Formalen Sprachen, als einer der drei Hauptbereiche der Theoretischen Informatik. Zu ihren wesentlichen Forschungszielen gehört die Klassifizierung von Problemen im Hinblick auf den zu ihrer Lösung notwendigen Aufwand. Eine besondere Rolle spielt dabei die Abgrenzung der praktisch effizient lösbaren Probleme. Die Komplexitätstheorie grenzt daher diejenigen Probleme ein, in denen andere Disziplinen der Informatik überhaupt sinnvollerweise nach effizienten Lösungen suchen sollten und motiviert so die Entwicklung von praxistauglichen Approximationsalgorithmen.

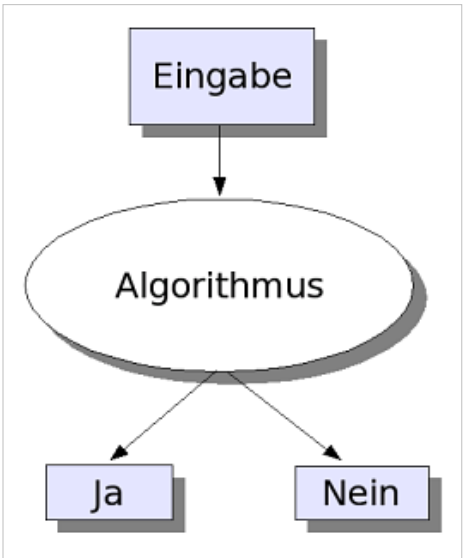


Die Komplexitätstheorie in der Theoretischen Informatik

Neben dem reinen Erkenntnisgewinn bereichert auch das Methodenarsenal der komplexitätstheoretischen Forschung zahlreiche angrenzende Gebiete. So führt etwa ihre enge Verzahnung mit der Automatentheorie zu neuen Maschinenmodellen und einem umfassenderen Verständnis der Arbeitsweise von Automaten. Die häufig konstruktive Beweisführung findet auch im Rahmen des Entwurfs und der Analyse von Algorithmen und Datenstrukturen Anwendung.

Probleme aus Sicht der Komplexitätstheorie

Entscheidungsprobleme als formale Sprachen



Den zentralen Gegenstand der Komplexitätstheorie bilden Probleme. Ein Problem wird dabei oft als formale Sprache verstanden. Die Aufgabe besteht nun darin, für ein gegebenes Wort zu entscheiden, ob es zu der Sprache gehört oder nicht. Daher spricht man hier auch von Entscheidungsproblemen. Man sagt dazu: Die Berechnung akzeptiert oder verwirft das Wort.

Ein mögliches Eingabewort könnte beispielsweise ein beliebiger Graph sein. Das Problem könnte darin bestehen zu entscheiden, ob der gegebene Graph zusammenhängend ist oder nicht. Die zugehörige entschiedene Sprache wäre demnach die Menge aller zusammenhängenden Graphen.

Man könnte annehmen, dass die Einschränkung auf solche Entscheidungsfragen viele wichtige Probleme ausschließt. Das ist jedoch nicht so. So lassen sich alle im Sinne der Komplexitätstheorie relevanten Probleme auch als Entscheidungsproblem formulieren. Betrachtet man zum Beispiel das Problem der Multiplikation zweier Zahlen, so besteht die dazugehörige Sprache des Entscheidungsproblems aus allen Zahlen-Tripeln (a, b, c) für die der Zusammenhang $a \cdot b = c$ gilt. Die Entscheidung, ob ein gegebenes Tripel zu dieser Sprache gehört, entspricht dem Lösen des Problems der Multiplikation zweier Zahlen.

Berechnungsprobleme als Abbildungen

Darüber hinaus ist man in der Komplexitätstheorie manchmal daran interessiert, Problemlösungen auch wirklich zu generieren. Hier begnügt man sich nicht mit der einfachen Ja/Nein-Antwort eines Wortproblemes für formale Sprachen. Stattdessen versteht man das Problem als eine Abbildung aus einem Definitionsbereich in seinen Lösungsraum. Die Lösung des Problemes der Multiplikation zweier Zahlen in etwa würde man als das Ergebnis der Abbildung f mit $f(a, b) = a \cdot b$ verstehen. Für die Definition der meisten Komplexitätsklassen wird jedoch die Formulierung durch Entscheidungsprobleme bevorzugt.

Eine wichtige Unterkategorie der Berechnungsprobleme stellen die Optimierungsprobleme dar. Bei Optimierungsproblemen besteht der funktionale Zusammenhang aus der Forderung, dass ein Maximum bzw. Minimum zu einer gegebenen Kostenfunktion zu der Eingabe ausgegeben werden soll. Man könnte so zum Beispiel beim Problem des Handlungsreisenden nach der optimalen Route durch die Landeshauptstädte Deutschlands fragen, welche die geringste Gesamtlänge besitzt. Viele Optimierungsprobleme sind von großer praktischer Bedeutung.

Probleminstanzen

Eine Probleminstanz ist nicht mit dem Problem selbst zu verwechseln. Ein Problem stellt in der Komplexitätstheorie die abstrakte Fragestellung dar. Dem entgegen bezeichnet die Instanz eines Problems eine ganz konkrete Ausprägung, die als Eingabe für eine solche Entscheidungsfrage dienen könnte.

Eine Instanz des Problems des Handlungsreisenden könnte zum Beispiel die Frage sein, ob eine Route durch die Landeshauptstädte Deutschlands mit einer maximalen Länge von 2000 km existiert. Die Entscheidung über diese Route hat jedoch nur begrenzten Wert für andere Probleminstanzen, wie etwa eine Rundreise durch die Sehenswürdigkeiten Mailands mit gegebener Längenschranke. In der Komplexitätstheorie interessiert man sich daher für Aussagen, die unabhängig von konkreten Instanzen sind.

Problemrepräsentationen

Als formale Sprachen werden Probleme und deren Instanzen über abstrakten Alphabeten definiert. Häufig wird das binäre Alphabet mit den Symbolen 0 und 1 gewählt, da dies der Verwendung von Bits bei modernen Rechnern am nächsten kommt. Eingaben werden dann durch Alphabetsymbole kodiert. An Stelle von mathematischen Objekten wie Graphen verwendet man möglicherweise eine Bitfolge die der Adjazenzmatrix des Graphen entspricht, an Stelle von natürlichen Zahlen zum Beispiel deren Binärdarstellung.

Auch wenn sich Beweise komplexitätstheoretischer Aussagen in der Regel konkreter Repräsentationen der Eingabe bedienen, versucht man Aussagen und Betrachtung unabhängig von Repräsentationen zu halten. Dies kann etwa erreicht werden, indem man sicherstellt, dass die gewählte Repräsentation bei Bedarf ohne allzu großen Aufwand in eine andere Repräsentation transformiert werden kann, ohne dass sich hierdurch die Berechnungsaufwände insgesamt signifikant verändern. Um dies zu ermöglichen, ist unter anderem die Auswahl eines geeigneten universellen Maschinenmodells von Bedeutung.

Problemgröße

Hat man ein Problem formal definiert (zum Beispiel das Problem des Handlungsreisenden in Form eines Graphen mit Kantengewichten), so möchte man Aussagen darüber treffen, wie sich ein Algorithmus bei der Berechnung der Problemlösung in Abhängigkeit von der Schwierigkeit des Problems verhält. Im Allgemeinen sind bei der Beurteilung der Schwierigkeit des Problems viele verschiedene Aspekte zu betrachten. Dennoch gelingt es häufig, wenige skalare Größen zu finden, die das Verhalten des Algorithmus im Hinblick auf den Ressourcenverbrauch maßgeblich beeinflussen. Diese Größen bezeichnet man als die Problemgröße. In aller Regel entspricht diese der Eingabelänge (bei einer konkret gewählten Repräsentation der Eingabe).

Man untersucht nun das Verhalten des Algorithmus in Abhängigkeit von der Problemgröße. Die Komplexitätstheorie interessiert sich für die Frage: *Wie viel* Mehrarbeit ist für wachsende Problemgrößen notwendig? Steigt der Aufwand (in Relation zur Problemgröße) zum Beispiel linear, polynomial, exponentiell oder gar überexponentiell?

So kann man beim Problem des Handlungsreisenden die Problemgröße als Anzahl der vorgegebenen Orte definieren (wobei man vernachlässigt, dass auch die vorgegebenen Streckenlängen verschieden große Eingabegrößen aufweisen können). Dann ist dieses Problem für die Problemgröße 2 trivial, da es hier überhaupt nur eine mögliche Lösung gibt und diese folglich auch optimal sein muss. Mit zunehmender Problemgröße wird ein Algorithmus jedoch mehr Arbeit leisten müssen.

Bester, schlechtester und durchschnittlicher Fall für Problemgrößen

Auch innerhalb einer Problemgröße lassen sich verschiedene Verhaltensweisen von Algorithmen beobachten. So hat das Problem des Handlungsreisenden für die 16 deutschen Landeshauptstädte dieselbe Problemgröße $n = 16$wie das Finden einer Route durch 16 europäische Hauptstädte. Es ist keineswegs zu erwarten, dass ein Algorithmus unter den unterschiedlichen Bedingungen (selbst bei gleicher Problemgröße) jeweils gleich gut arbeitet. Da es jedoch in der Regel unendlich viele Instanzen gleicher Größe für ein Problem gibt, gruppiert man diese zumeist grob in drei Gruppen: bester, schlechtester und durchschnittlicher Fall. Diese stehen für die Fragen:

- Bester Fall: Wie arbeitet der Algorithmus (in Bezug auf die in Frage stehende Ressource) im günstigsten Fall?
- Schlechtester Fall: Wie arbeitet der Algorithmus im schlimmsten Fall?
- Durchschnittlicher Fall: Wie arbeitet der Algorithmus durchschnittlich (wobei die zugrundegelegte Verteilung für die Berechnung eines Durchschnitts nicht immer offensichtlich ist)?

Untere und obere Schranken für Probleme

Die Betrachtung bester, schlechtester und durchschnittlicher Fälle bezieht sich stets auf eine feste Eingabelänge. Auch wenn die Betrachtung konkreter Eingabelängen in der Praxis von großem Interesse sein kann, ist diese Sichtweise für die Komplexitätstheorie meist nicht abstrakt genug. Welche Eingabelänge als groß oder praktisch relevant gilt, kann sich aufgrund technischer Entwicklungen sehr schnell ändern. Es ist daher gerechtfertigt, das Verhalten von Algorithmen in Bezug auf ein Problem gänzlich unabhängig von konkreten Eingabelängen zu untersuchen. Man betrachtet hierzu das Verhalten der Algorithmen für immer größer werdende, potentiell unendlich große Eingabelängen. Man spricht vom asymptotischen Verhalten des jeweiligen Algorithmus.

Bei dieser Untersuchung des asymptotischen Ressourcenverbrauchs spielen untere und obere Schranken eine zentrale Rolle. Man möchte also wissen, welche Ressourcen für die Entscheidung eines Problems mindestens und höchstens benötigt werden. Für die Komplexitätstheorie sind die unteren Schranken von besonderem Interesse: Man möchte zeigen, dass ein bestimmtes Problem *mindestens* einen bestimmten Ressourcenverbrauch beansprucht und es folglich keinen Algorithmus geben kann, der das Problem mit geringeren Ressourcen entscheidet. Solche Ergebnisse helfen, Probleme nachhaltig bezüglich ihrer Schwierigkeit zu separieren. Jedoch sind bisher nur vergleichsweise wenige aussagekräftige untere Schranken bekannt. Der Grund hierfür liegt in der Problematik, dass sich Untersuchungen unterer Schranken stets auf alle denkbaren Algorithmen für ein Problem beziehen müssen; also auch auf Algorithmen, die heute noch gar nicht bekannt sind.

Im Gegensatz dazu gelingt der Nachweis von oberen Schranken in der Regel durch die Analyse konkreter Algorithmen. Durch den Beweis der Existenz auch nur eines Algorithmus, der die obere Schranke einhält, ist der Nachweis bereits erbracht.

Maschinenmodelle in der Komplexitätstheorie

Kostenfunktionen

Zur Analyse des Ressourcenverbrauchs von Algorithmen sind geeignete Kostenfunktionen zu definieren, welche eine Zuordnung der Arbeitsschritte des Algorithmus zu den verbrauchten Ressourcen ermöglichen. Um dies tun zu können, muss zunächst festgelegt werden, welche Art von Arbeitsschritt einem Algorithmus überhaupt erlaubt ist. Diese Festlegung erfolgt in der Komplexitätstheorie über abstrakte Maschinenmodelle - würde man auf reale Rechnermodelle zurückgreifen, so wären die gewonnenen Erkenntnisse bereits in wenigen Jahren überholt. Der Arbeitsschritt eines Algorithmus erfolgt in Form einer Befehlsausführung auf einer bestimmten Maschine. Die Befehle, die eine Maschine ausführen kann, sind dabei durch das jeweilige Modell streng limitiert. Darüber hinaus unterscheiden sich verschiedene Modelle etwa in der Handhabung des Speichers und in ihren Fähigkeiten zur parallelen Verarbeitung, d. h. der gleichzeitigen Ausführung mehrerer Befehle. Die Definition der Kostenfunktion erfolgt nun durch eine Zuordnung von Kostenwerten zu den jeweils erlaubten Befehlen.

Kostenmaße

Häufig wird von unterschiedlichen Kosten für unterschiedliche Befehle abstrahiert und als Kostenwert für eine Befehlsausführung immer 1 gesetzt. Sind auf einer Maschine beispielsweise Addition und Multiplikation die erlaubten Operationen, so zählt man für jede Addition und jede Multiplikation, die im Laufe des Algorithmus berechnet werden müssen, den Kostenwert von 1 hinzu. Man spricht dann auch von einem *uniformen Kostenmaß*. Ein solches Vorgehen ist dann gerechtfertigt, wenn sich die erlaubten Operationen nicht gravierend unterscheiden und wenn der Wertebereich, auf dem die Operationen arbeiten, nur eingeschränkt groß ist. Dies wird schon für eine einfache Operation wie die Multiplikation klar: Das Produkt zweier einstelliger Dezimalzahlen dürfte sich ungleich schneller errechnen lassen als das Produkt zweier hundertstelliger Dezimalzahlen. Bei einem uniformen Kostenmaß würden beide Operationen dennoch mit einem Kostenwert von 1 veranschlagt. Sollten sich die möglichen Operanden im Laufe eines Algorithmus tatsächlich so gravierend unterscheiden, so muss ein realistischeres Kostenmaß gewählt

werden. Häufig wählt man dann das *logarithmische Kostenmaß*. Der Bezug auf den Logarithmus ergibt sich daraus, dass sich eine Dezimalzahl n im Wesentlichen durch $\log_2(n)$ Binärziffern darstellen lässt. Man wählt zur Repräsentation der Operanden Binärziffern aus und definiert die erlaubten booleschen Operationen. Sollte das jeweilige Maschinenmodell Adressen verwenden, so werden auch diese binär codiert. Auf diese Weise werden die Kosten über die Länge der Binärdarstellung logarithmisch gewichtet. Andere Kostenmaße sind möglich, werden jedoch nur selten eingesetzt.

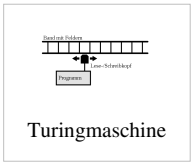
Maschinenmodelle und Probleme

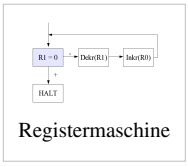
Man unterscheidet verschiedene Berechnungsparadigmen: der pragmatischste Typ ist sicher der der deterministischen Maschinen; weiterhin gibt es den in der Theorie besonders relevanten Typ der nichtdeterministischen Maschinen; weiterhin gibt es noch probabilistische Maschinen, alternierende und andere. In der Regel kann man jedes Maschinenmodell mit jedem der obigen Paradigmen definieren. Einige Paradigmen, so zum Beispiel der Nichtdeterminismus, modellieren dabei einen Typ, der der Theorie vorbehalten bleiben muss, da man den Nichtdeterminismus in der dort definierten Form physikalisch nicht bauen kann, (sie „erraten“ einen richtigen Pfad in einem Berechnungsbaum), lassen sich jedoch häufig leicht zu einem gegebenen Problem konstruieren. Da eine Transformation von nichtdeterministischen in deterministische Maschinen immer relativ einfach möglich ist, konstruiert man daher zunächst eine nichtdeterministische Maschinenversion und transformiert diese später in eine deterministische.

Daraus geht eine wichtige Beweistechnik der Komplexitätstheorie hervor: Lässt sich zu einem gegebenen Problem ein bestimmter Maschinentyp konstruieren, auf dem das Problem mit bestimmten Kosten entschieden werden kann, so kann damit bereits die Komplexität des Problems eingeschätzt werden. Tatsächlich werden sogar die unterschiedlichen Maschinenmodelle bei der Definition von Komplexitätsklassen zugrundegelegt. Dies entspricht einer Abstraktion von einem konkreten Algorithmus: Wenn ein Problem auf Maschine M entscheidbar ist (wobei ein entsprechender Algorithmus evtl. noch gar nicht bekannt ist), so lässt es sich unmittelbar einer bestimmten Komplexitätsklasse zuordnen, nämlich derjenigen, die von M definiert wird. Dieses Verhältnis zwischen Problemen und Maschinenmodellen ermöglicht Beweisführungen ohne die umständliche Analyse von Algorithmen.

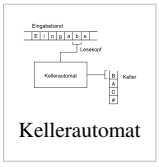
Häufig eingesetzte Maschinenmodelle

Besonders häufig eingesetzte Modelle sind:

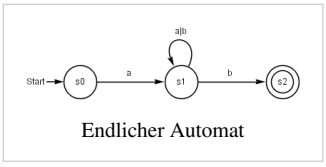

Turingmaschine



Registermaschine



Kellerautomat



Endlicher Automat

Zur Untersuchung parallelisierbarer Probleme können darüber hinaus auch parallelisierte Versionen dieser Maschinen zum Einsatz kommen, insbesondere die parallele Registermaschine.

Die erweiterte Church-Turing-These

Für die Verwendung von Maschinenmodellen in der Komplexitätstheorie ist eine Erweiterung der Church-Turing-These von Bedeutung, die auch als *erweiterte Church-Turing-These* bezeichnet wird. Sie besagt, dass alle universellen Maschinenmodelle in Bezug auf die Rechenzeit bis auf polynomielle Faktoren gleich mächtig sind. Dies ermöglicht dem Komplexitätstheoretiker eine relativ freie Wahl des Maschinenmodells im Hinblick auf das jeweilige Untersuchungsziel. Auch diese These ist nicht beweisbar; im Gegensatz zur gewöhnlichen Church-Turing-These wäre es aber möglich, sie durch ein Gegenbeispiel zu widerlegen.

Modellmodifikationen für Speicherplatzanalysen

Zur Untersuchung des Mindestspeicherbedarfs, der für die Lösung von Problemen benötigt wird, nimmt man häufig die folgenden Modifikationen des verwendeten Maschinenmodells (in der Regel eine Turingmaschine) vor:

- Der Eingabespeicher darf *nur gelesen* werden.
- Auf die Ausgabe darf *nur geschrieben* werden. Der Schreibkopf wird nur nach Schreibvorgängen und immer in dieselbe Richtung bewegt (falls das Maschinenmodell eine solche Bewegung vorsieht).

Für die Untersuchung des Speicherbedarfs dürfen dann Ein- und Ausgabe der Maschine unberücksichtigt bleiben. Die Motivation für diese Änderungen ist die folgende: Würde zum Beispiel der Eingabespeicher in die Speicherplatzanalyse einbezogen, so könnte kein Problem in weniger als $\mathcal{O}(n)$ Platzbedarf gelöst werden, denn das Eingabewort hat ja immer genau die Länge und damit den Speicherbedarf n. Indem man die Eingabe nur lesbar macht, verhindert man, dass sie für Zwischenrechnungen verwendet werden kann. Man kann dann die Eingabe bei der Berechnung des Platzbedarfs vernachlässigen. Eine ähnliche Argumentation führt zu der Einschränkung der Ausgabe. Durch die zusätzliche Einschränkung einer möglichen Kopfbewegung wird verhindert, dass die Kopfposition verwendet wird, um sich Information zu „merken“. Insgesamt stellen all diese Einschränkungen sicher, dass Ein- und Ausgabe bei der Speicherplatzanalyse nicht berücksichtigt werden müssen.
Die vorgenommenen Modifikationen beeinflussen das Zeitverhalten der Maschine übrigens nur um einen konstanten Faktor und sind damit vernachlässigbar.

Verwendung und Rechtfertigung der O-Notation

Bei der Untersuchung von Größenordnungen für Aufwände wird in der Komplexitätstheorie ausgiebig von der O-Notation Gebrauch gemacht. Dabei werden lineare Faktoren und Konstanten aus der Betrachtung ausgeblendet. Diese Vorgehensweise mag zunächst überraschen, wäre doch für den Praktiker häufig bereits eine Halbierung der Aufwände von hoher Bedeutung.

Der Standpunkt der Komplexitätstheorie lässt sich theoretisch mit einer Technik rechtfertigen, die man als *lineares Beschleunigen* oder auch Speedup-Theorem bezeichnet. (Wir beschränken uns hier auf das Zeitverhalten. Analoge Beweise kann man auch für den Speicherbedarf oder andere Ressourcen führen.) Das Speedup-Theorem besagt vereinfachend, dass sich zu jeder Turingmaschine, die ein Problem in Zeit T entscheidet, eine neue Turingmaschine konstruieren lässt, die das Problem in Zeit weniger als $\varepsilon \cdot T$ entscheidet. Dabei kann $\varepsilon > 0$ beliebig klein gewählt sein. Das bedeutet nichts anderes, als dass sich jede Turingmaschine, die ein bestimmtes Problem löst, um einen beliebigen konstanten Faktor beschleunigen lässt. Der Preis für diese Beschleunigung besteht in einer stark vergrößerten Arbeitsalphabetgröße und Zustandsmenge der verwendeten Turingmaschine (letztlich also „Hardware“).

Diese Beschleunigung wird unabhängig von der Problemgröße erreicht. Es ergibt daher keinen Sinn, konstante Faktoren bei der Betrachtung von Problemen zu berücksichtigen – solche Faktoren ließen sich durch Anwendung der Beschleunigungstechnik wieder beseitigen. Die Vernachlässigung konstanter Faktoren, die sich in der O-Notation ausdrückt, hat daher nicht nur praktische Gründe, sie vermeidet auch Verfälschungen im Rahmen komplexitätstheoretischer Betrachtungen.

Bildung von Komplexitätsklassen

Eine wesentliche Aufgabe der Komplexitätstheorie besteht darin, sinnvolle Komplexitätsklassen festzulegen, in diese die vorliegenden Probleme einzusortieren und Aussagen über die wechselseitigen Beziehungen zwischen den Klassen zu finden.

Einflussgrößen bei der Bildung von Komplexitätsklassen

Eine Reihe von Faktoren nehmen Einfluss auf die Bildung von Komplexitätsklassen. Die wichtigsten sind die folgenden:

- Das zugrunde liegende *Berechnungsmodell* (Turingmaschine, Registermaschine usw.).
- Der verwendete *Berechnungsmodus* (deterministisch, nichtdeterministisch, probabilistisch usw.).
- Die betrachtete *Berechnungsressource* (Zeit, Platz, Prozessoren usw.).
- Das angenommene *Kostenmaß* (uniform, logarithmisch).
- Die eingesetzte *Schrankenfunktion*.

Anforderungen an Schrankenfunktionen

Zur Angabe oder Definition von Komplexitätsklassen verwendet man Schrankenfunktionen. Schreibt man beispielsweise DTIME(f), so meint man damit die Klasse aller Probleme, die auf einer deterministischen Turingmaschine in der Zeit $\mathcal{O}(f)$ entschieden werden können. f ist dabei eine Schrankenfunktion. Um als Schrankenfunktion für komplexitätstheoretische Analysen eingesetzt werden zu können, sollte die Funktion mindestens die folgenden Anforderungen erfüllen:

- $f : \mathbb{N} \to \mathbb{N}$ (Schrittzahl, Speicher usw. werden als natürliche Zahlen berechnet).
- $f(n+1) \geq f(n)$ (monotones Wachstum).
- Die Funktion f muss selbst in Zeit $\mathcal{O}(f)$ und mit Raum $\mathcal{O}(f)$ berechenbar sein.

Eine Funktion, die diesen Anforderungen genügt, bezeichnet man auch als *echte Komplexitätsfunktion*. Der Sinn der Anforderungen ist zum Teil technischer Natur. Die Schrankenfunktion kann selbst auf konstruktive Art (zum Beispiel als Turingmaschine) in Beweise einfließen und sollte sich daher für diese Zwecke „gutartig“ verhalten. An dieser Stelle soll nur darauf hingewiesen werden, dass bei der Wahl der Schrankenfunktion eine gewisse Vorsicht walten muss, weil sonst bestimmte algorithmische Techniken nicht angewandt werden können.

Die meisten „natürlichen“ Funktionen entsprechen den oben genannten Einschränkungen, so etwa die konstante Funktion, die Logarithmusfunktion, die Wurzelfunktion, Polynome, die Exponentialfunktion sowie alle Kombinationen dieser Funktionen. Es folgt eine Übersicht der wichtigsten Schrankenfunktionen mit der jeweils üblichen Sprechweise. Die Angabe erfolgt wie üblich in O-Notation.

Die wichtigsten Schrankenfunktionen

konstant	$\mathcal{O}(1)$
logarithmisch	$\mathcal{O}(\log n)$
polylogarithmisch	$\mathcal{O}(\log^k n)$ **für** $k \geq 1$
linear	$\mathcal{O}(n)$
n-log-n	$\mathcal{O}(n \log n)$
quadratisch	$\mathcal{O}(n^2)$
polynomial	$\mathcal{O}(n^k)$ **für** $k \geq 1$
exponentiell	$\mathcal{O}(d^n)$ **für** $d > 1$

Hierarchiesätze

Für die gebildeten Klassen möchte man möglichst beweisen, dass durch zusätzlich bereitgestellte Ressourcen tatsächlich *mehr* Probleme gelöst werden können. Diese Beweise bezeichnet man als *Hierarchiesätze*, da sie auf den Klassen der jeweiligen Ressource eine Hierarchie induzieren. Es gibt also Klassen, die in eine echte Teilmengenbeziehung gesetzt werden können. Wenn man solche echten Teilmengenbeziehungen ermittelt hat, lassen sich auch Aussagen darüber treffen, wie groß die Erhöhung einer Ressource sein muss, um damit eine größere Zahl von Problemen berechnen zu können. Von besonderem Interesse sind dabei wiederum die Ressourcen Zeit und Raum. Die induzierten Hierarchien bezeichnet man auch als *Zeithierarchie* und *Raumhierarchie*.

Die Hierarchiesätze bilden letztlich das Fundament für die *Separierung* von Komplexitätsklassen. Sie waren daher auch eines der frühesten Ergebnisse der Komplexitätstheorie. Ohne die Hierarchiesätze müsste für die Beziehungen der unterschiedlichen Klassen statt einer $\subset$-Hierarchie eine $\subseteq$-Hierarchie angenommen werden. Es bliebe dann zu befürchten, dass die gesamte Hierarchie zu einer einzigen Klasse kollabiert. Tatsächlich muss ergänzt werden, dass alle Hierarchiesätze auf diversen Voraussetzungen beruhen. Eine dieser Voraussetzungen ist etwa, dass die oben genannten Anforderungen an *echte Komplexitätsfunktionen* erfüllt werden. Ohne die Einhaltung dieser Anforderungen bricht tatsächlich die gesamte Klassenhierarchie in sich zusammen.[1]

Zeithierarchiesatz

Der Zeithierarchiesatz besagt:

$$\mathrm{DTIME}\big(f(n)\big) \subsetneq \mathrm{DTIME}\big(f(n) \cdot \log^2(f(n))\big)$$

Dies bedeutet also, dass man die Zeitressource um einen Faktor $\log^2(f(n))$ erhöhen muss, um auf einer deterministischen Turingmaschine mehr Probleme lösen zu können. Eine ähnliche Beziehung lässt sich für nichtdeterministische Turingmaschinen finden.

Raumhierarchiesatz

Der Raumhierarchiesatz besagt:

$$\mathrm{DSPACE}\big(f(n)\big) \subsetneq \mathrm{DSPACE}\big(f(n) \cdot \log(f(n))\big)$$

Die Aussage ist analog zum Zeithierarchiesatz. Man erkennt jedoch, dass im Vergleich zur Zeit bereits eine geringere Steigerung des Raumes ausreicht (Faktor $\log(f(n))$ im Vergleich zu $\log^2(f(n))$), um eine größere Klasse zu bilden. Dies entspricht auch einer intuitiven Erwartung, verhält sich doch der Raum insgesamt aufgrund seiner Wiederverwendbarkeit (alte Zwischenergebnisse können überschrieben werden) gutmütiger.

Wofür die Hierarchiesätze nicht gelten

Die Hierarchiesätze beziehen sich ausschließlich auf den jeweils gleichen Berechnungsmodus und eine einzelne Ressource, also zum Beispiel auf die Ressource Zeit auf einem deterministischen Maschinenmodell. Es wird jedoch keine Aussage darüber getroffen, wie sich etwa Raum- und Zeitklassen zueinander verhalten oder in welchem Verhältnis deterministische oder nichtdeterministische Klassen zueinander stehen. Dennoch gibt es derartige Zusammenhänge. Sie werden in den Abschnitten *Beziehungen zwischen Raum- und Zeitklassen* und *Beziehungen zwischen deterministischen und nichtdeterministischen Klassen* behandelt.

Wichtige Zeitklassen

- DTIME(f): Allgemeine Schreibweise für deterministische Zeitklassen.
- P: Deterministisch in Polynomialzeit entscheidbare Sprachen.
- EXPTIME: Deterministisch in Exponentialzeit entscheidbare Sprachen.
- NTIME(f): Allgemeine Schreibweise für nichtdeterministische Zeitklassen.
- NP: Nichtdeterministisch in Polynomialzeit entscheidbare Sprachen.
- NEXPTIME: Nichtdeterministisch in Exponentialzeit entscheidbare Sprachen.
- NC: Parallel in polylogarithmischer Zeit berechenbare Funktionen.

Wichtige Raumklassen

- DSPACE(f): Allgemeine Schreibweise für deterministische Raumklassen.
- L: Deterministisch mit logarithmisch beschränktem Raum entscheidbare Sprachen.
- PSPACE: Deterministisch mit polynomial beschränktem Raum entscheidbare Sprachen.
- NSPACE(f): Allgemeine Schreibweise für nichtdeterministische Raumklassen.
- NL: Nichtdeterministisch mit logarithmisch beschränktem Raum entscheidbare Sprachen.
- CSL: Kontextsensitive Sprachen sind die nichtdeterministisch mit linear beschränktem Raum entscheidbaren Sprachen.

Siehe auch: Liste von Komplexitätsklassen

Komplementbildungen

Für jede Komplexitätsklasse lässt sich ihre Komplementklasse bilden: Die Komplementklasse enthält genau die Komplemente der ursprünglichen Klasse. Dagegen kann man auch das Komplement einer Klasse betrachten, das sind alle Mengen, die in dieser Klasse nicht sind; diese Mengen sind in der Regel viel schwerer als die der ursprünglichen Komplexitätsklasse. Die Komplementklasse hingegen besitzt mit der ursprünglichen Klasse in der Regel einen nichtleeren Durchschnitt. Der neu gebildeten Komplementklasse stellt man das Präfix *Co* vor. Heißt also die Ausgangsklasse K, so heißt ihr Komplement CoK. Für deterministische Maschinen gilt in der Regel K = CoK, da in der Übergangsfunktion einfach nur die Übergänge zu akzeptierenden Zuständen durch Übergänge zu verwerfenden Zuständen ausgetauscht werden müssen und umgekehrt. Für andere Berechnungsmodi gilt dies jedoch nicht, da hier die Akzeptanz anders definiert ist.

Beispielsweise ist bislang unbekannt, ob NP = CoNP gilt. P = CoP ist wahr, da das zugrunde liegende Modell deterministisch ist und hier die akzeptierenden und ablehnenden Zustände in den Berechnungen einfach ausgetauscht werden können, wie im vorherigen Absatz angesprochen. So sehen wir sofort, dass P im Durchschnitt von NP und CoNP enthalten ist. Ob dieser Durchschnitt genau P ist, ist nicht bekannt.

Das P-NP-Problem und seine Bedeutung

Eines der wichtigsten und nach wie vor ungelösten Probleme der Komplexitätstheorie ist das P-NP-Problem. *Ist die Klasse P gleich der Klasse NP?* Diese Frage kann als eine zentrale Forschungsmotivation der Komplexitätstheorie angesehen werden, und eine Vielzahl der komplexitätstheoretischen Ergebnisse wurde erzielt, um der Lösung des P-NP-Problems näher zu kommen.

Die Klasse P: Praktisch lösbare Probleme

Die Tragweite des P-NP-Problems resultiert aus der Erfahrung, dass die Probleme der Klasse P in der Regel praktisch lösbar sind: Es existieren Algorithmen, um Lösungen für diese Probleme effizient oder doch mit vertretbarem zeitlichen Aufwand zu berechnen. Der zeitliche Aufwand zur Problemlösung wächst für die Probleme der Klasse P maximal polynomial. Meist lassen sich sogar Algorithmen finden, deren Zeitfunktionen Polynome

niedrigen Grades sind. Da das gewählte Maschinenmodell dieser Zeitklasse deterministisch und damit tatsächlich realisierbar ist, bilden die Probleme der Klasse P gerade die Grenze des algorithmisch sinnvollerweise Machbaren.

Die Klasse NP: Praktisch (vermutlich) nicht lösbare Probleme

Im Gegensatz zu den Problemen in P basieren die Algorithmen zur Lösung der Probleme in NP auf einem nichtdeterministischen Maschinenmodell. Für solche Maschinen wird eine unbeschränkte Parallelisierbarkeit der sich verzweigenden Berechnungspfade angenommen, die technisch nicht realisiert werden kann. Zwar arbeiten auch die Algorithmen zur Lösung der Probleme in NP in polynomialer Zeit, aber eben auf der Basis eines unrealistischen Maschinenmodells. Die Probleme in NP gelten daher für praktische Zwecke als nicht lösbar: Der Aufwand zu ihrer Berechnung steigt auf einer deterministischen Maschine *mutmaßlich* mit wachsender Problemgröße mehr als polynomial an. Dies wäre nicht weiter tragisch, wenn nicht so viele Probleme von größter Bedeutung zu NP gehören würden. Tatsächlich finden sich jedoch in NP Probleme aus fast allen Bereichen der Informatik, deren effiziente Lösung enorm wichtig wäre.

Der Fall P = NP

Würde das P-NP-Problem im Sinne von P = NP gelöst, so wüssten wir, dass es selbst für NP-vollständige Probleme Algorithmen geben muss, die mit polynomiellem Zeitaufwand arbeiten.

Da umgekehrt die Definition der NP-Vollständigkeit Algorithmen voraussetzt, mit denen es gelingt, beliebige Probleme aus NP in polynomieller Zeit auf NP-vollständige Probleme zu reduzieren, wären mit der polynomialen Lösbarkeit auch nur eines einzigen NP-vollständigen Problems sofort sämtliche Probleme der Klasse NP in polynomieller Zeit lösbar. Dies hätte eine Problemlösekraft in der gesamten Informatik zur Folge, wie sie auch durch noch so große Fortschritte in der Hardware-Entwicklung nicht erreicht werden kann.

Andererseits ist für bestimmte Anwendungsfälle eine Lösung des P-NP-Problems im Sinne von P = NP eher unerwünscht. Beispielsweise würden asymmetrische Verschlüsselungsverfahren erheblich an Sicherheit verlieren, da diese dann in Polynomialzeit gebrochen werden könnten.

Der Fall P ≠ NP

Würde das P-NP-Problem im Sinne von P ≠ NP gelöst, so wäre klar, dass weitere Bemühungen, polynomielle Lösungen für NP-vollständige Probleme zu finden, sinnlos wären. Man kann sich leicht vorstellen, dass aufgrund der hohen Bedeutung der Probleme in NP die Bemühungen um eine effiziente Lösung erst dann eingestellt werden, wenn diese nachgewiesenermaßen unmöglich ist. Bis zu diesem Zeitpunkt wird noch viel private und öffentliche Forschungsenergie aufgewandt werden.

In vielen Theoremen wird heute jedoch angenommen, dass P ≠ NP gilt, denn nur so kann ohne einen Beweis der Gleichheit trotzdem effektive Forschungsarbeit geleistet werden. Man sucht nach Auswegen durch Approximationen und Heuristiken, nach Problemeinschränkungen, die die Praxis nicht vernachlässigen.

Konsequenzen für die Komplexitätstheorie

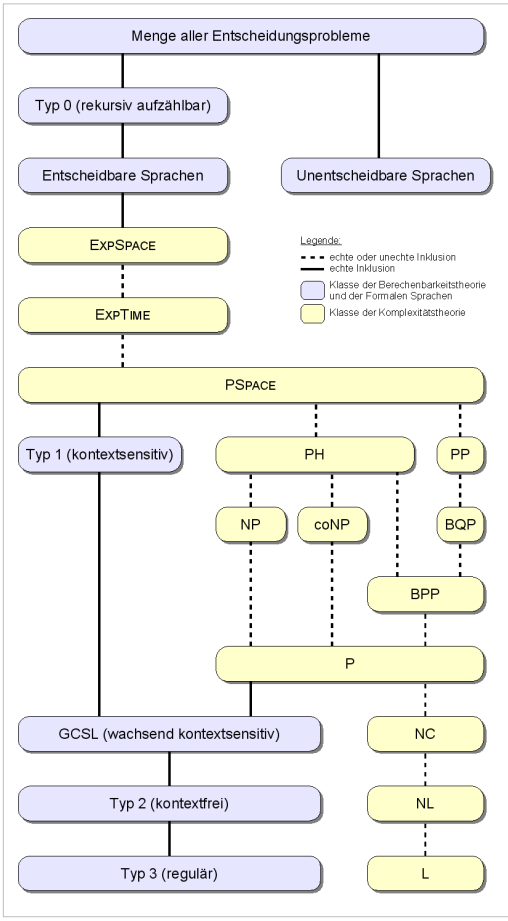
Zu den wichtigsten Forschungszielen der Komplexitätstheorie gehört die Abgrenzung des praktisch Machbaren und damit des Betätigungsfeldes der Informatik schlechthin. Die vorherigen Abschnitte haben die Wichtigkeit dieser Grenzziehung verdeutlicht. Im Zuge der Versuche, das P-NP-Problem zu lösen, hat die Komplexitätstheorie zahlreiche Ergebnisse zu Tage gefördert und ihre Analysemethoden Zug um Zug verfeinert. Diese Ergebnisse werden insbesondere beim Entwurf und der Analyse praktisch wichtiger Algorithmen angewandt und wirken so auch unmittelbar auf die Praktische Informatik.

Die seit über dreißig Jahren andauernden Bemühungen, das P-NP-Problem zu lösen, gewähren darüber hinaus dem praktischen Informatiker ein großes Maß an Sicherheit, dass isolierte Bemühungen zur effizienten Lösung von

Problemen aus NP mehr oder weniger sinnlos sind. Die praktische Informatik konzentriert sich daher bei der Lösung für Probleme aus NP auf Näherungslösungen oder die Abwandlung der ursprünglichen Probleme. So kann sich beispielsweise die Problemkomplexität von Optimierungs-Algorithmen enorm verringern, wenn man keine optimale Lösung anstrebt, sondern mit einer fast optimalen Lösung zufrieden ist. Die Komplexitätstheorie liefert für diese Vorgehensweise die theoretische Rückendeckung.

Sprachen und Komplexitätsklassen

Das folgende Inklusionsdiagramm gibt einen – recht groben – Überblick über die Zusammenhänge zwischen Klassen der Berechenbarkeitstheorie, der Chomsky-Hierarchie und den bedeutendsten Komplexitätsklassen.



Geschichte der Komplexitätstheorie

Nachdem in den vorhergehenden Abschnitten zahlreiche Grundbegriffe und wichtige Ergebnisse der Komplexitätstheorie erläutert wurden, wird in den folgenden Abschnitten ein geschichtlicher Abriss gegeben, der die zeitliche Abfolge dieser Ergebnisse einordnen helfen soll.

Grundlagen

Vor dem eigentlichen Beginn der explizit auf die Komplexität von Algorithmen bezogenen Forschung wurden zahlreiche Grundlagen erarbeitet. Als wichtigste kann dabei die Konstruktion der Turingmaschine durch Alan Turing im Jahr 1936 angesehen werden, die sich für spätere Algorithmen-Analysen als ausgesprochen flexibles Modell erwies.

Als erste informelle komplexitätstheoretische Untersuchungen werden Ergebnisse von John Myhill (1960), Raymond Smullyan (1961) und Hisao Yamada (1962) angesehen, die sich mit speziellen raum- und zeitbeschränkten

Problemklassen beschäftigt haben, jedoch in ihren Arbeiten noch keinen Ansatz zu einer allgemeinen Theorie entwickelten.

Beginn der komplexitätstheoretischen Forschung

Einen ersten großen Schritt in Richtung einer solchen Theorie unternehmen Juris Hartmanis und Richard Stearns in ihrer 1965 erschienenen Arbeit „*On the computational complexity of algorithms*“. Sie geben bereits eine quantitative Definition von Zeit- und Platzkomplexität und wählen damit bereits die beiden Ressourcen aus, die bis heute als die wichtigsten angesehen werden. Dabei wählen sie die Mehrband-Turingmaschine als Grundlage und treffen damit eine sehr robuste Entscheidung, die in vielen komplexitätstheoretischen Feldern Bestand hat. Sie erarbeiten auch bereits erste Hierarchiesätze.

In den folgenden Jahren kommt es zu einer Reihe fundamentaler Ergebnisse. 1967 veröffentlichte Manuel Blum das Speedup-Theorem. 1969 folgt das Union-Theorem von Edward M. McCreight und Albert R. Meyer. Und 1972 veröffentlicht Allan Borodin das Gap-Theorem. Diese Ergebnisse lassen sich nicht nur als grundlegend für die Komplexitätstheorie ansehen, sie stellen auch ein Abtasten des noch neuen Forschungsgebietes dar, das sich zugleich noch durch möglichst „spektakuläre“ Ergebnisse rechtfertigen muss. So treffen diese Theoreme z.T. zwar überraschende Aussagen, sind aber mitunter auf Annahmen gebaut, die man heute einschränken würde. Beispielsweise werden keine *echten Komplexitätsfunktionen* (siehe oben) vorausgesetzt.

In derselben Zeit, die etwa die ersten zehn Jahre komplexitätstheoretischer Forschung umfasst, kommt es zur Formulierung der Klasse P als der Klasse der „praktisch lösbaren“ Probleme. Alan Cobham zeigt, dass die Polynomialzeit robust unter der Wahl des Maschinenmodells ist (was man heute unter der erweiterten Church-Turing These zusammenfasst). Darüber hinaus erweisen sich viele mathematische Funktionen als in Polynomialzeit berechenbar.

Erforschung der Klasse NP

Die Klasse NP tritt zuerst bei Jack Edmonds auf den Plan, der jedoch zunächst nur eine informelle Definition gibt. Die Tatsache, dass zahlreiche wichtige Probleme in NP zu liegen scheinen, lässt diese Klasse jedoch bald als attraktives Forschungsfeld erscheinen. Der Begriff der Reduzierbarkeit und die darauf basierende NP-Vollständigkeit wird entwickelt und findet zuerst im Satz von Cook (1971) prägnanten Ausdruck: Das Erfüllbarkeitsproblem (SAT) ist NP-vollständig und damit ein *schwerstes* Problem in NP. Nebenbei bemerkt bezog sich die ursprüngliche Arbeit von Stephen Cook auf Tautologien (aussagenlogische Formeln, die durch *jede* Belegung erfüllt werden), während der Begriff der Erfüllbarkeit nicht erwähnt wird. Da die Ergebnisse bezüglich der Tautologien jedoch relativ einfach auf die Erfüllbarkeit übertragen werden können, rechnet man sie Stephen Cook zu. Einen Teil dieser Übertragung leistet Richard Karp (1972), indem er die Technik der Reduktion ausarbeitet. Völlig unabhängig von diesen Arbeiten entwickelte Leonid Levin (1973) in der damaligen Sowjetunion eine Theorie der NP-Vollständigkeit, die im Westen für lange Zeit unbeachtet blieb.

1979 veröffentlichen Michael R. Garey und David S. Johnson ein Buch, welches 300 NP-vollständige Probleme beschreibt (*Computers and intractability*). Dieses Buch wurde für künftige Forscher zu einer wichtigen Referenz.

Randomisierte Komplexitätsklassen

1982 stellt Andrew Yao das Konzept der Falltürfunktionen (trapdoor functions) vor, die eine spezielle Art von Einwegfunktionen (one way functions) darstellen, und zeigt deren grundlegende Wichtigkeit in der Kryptographie auf. Jedoch genügt für die Schwierigkeit, einen Code zu knacken, die Worst-Case-Betrachtungsweise der Komplexitätsklassen wie NP nicht. Es dürfen vielmehr auch keine Algorithmen existieren, die diese Probleme in einem signifikanten Anteil aller Fälle effizient lösen. Dies korrespondiert zum Modell der probabilistischen Turingmaschine und motiviert die Einführung randomisierter Komplexitätsklassen wie ZPP, RP oder BPP (alle eingeführt von John T. Gill, 1977).

Mit dieser Übersicht wurden die wesentlichen Grundsteine der Geschichte der Komplexitätstheorie gelegt. Wie in anderen Forschungsgebieten auch, fächern sich die neueren Ergebnisse in viele, teils sehr spezielle Teilbereiche auf.

Weblinks

- Complexity Zoo [2] -- „Zoo“ der Komplexitätsklassen

Literatur

- Ingo Wegener: *Komplexitätstheorie. Grenzen der Effizienz von Algorithmen*. 1. Auflage. Springer, Berlin 2003, ISBN 3-540-00161-1.
- K. Rüdiger Reischuk: *Komplexitätstheorie - Band I: Grundlagen: Maschinenmodelle, Zeit- und Platzkomplexität, Nichtdeterminismus*. 2. Auflage. Teubner, Stuttgart/Leipzig 1999, ISBN 3-519-12275-8.
- Christos H. Papadimitriou: *Computational Complexity*. Addison-Wesley, Reading/Mass. 1995, ISBN 0-201-53082-1.
- Lance Fortnow, Steve Homer: *A Short History of Computational Complexity*. Online-Manuskript [3] (PDF, 225 kB)
- Jan van Leeuwen (Hrsg.): *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. The MIT Press/Elsevier, Amsterdam 1994, ISBN 0-262-72020-5.
- Juris Hartmanis, Richard Edwin Stearns: *On the computational complexity of algorithms*. In: *Trans. American Mathematical Society*. 117/1965, S. 285-306, ISSN 0002-9947 [4].
- Ding-Zhu Du, Ker-I Ko: *Theory of Computational Complexity.* John Wiley & Sons, New York 2000, ISBN 0-471-34506-7.
- Michael R. Garey, David S. Johnson: *Computers and Intractability: A guide to the theory of NP-completeness.* Freeman, New York 2003, ISBN 0-7167-1045-5.
- Michael Sipser: *Introduction to the Theory of Computation*. 2. Auflage. Thomson, Boston 2006, ISBN 0-534-95097-3. (International Edition)

Referenzen

[1] Derartige Probleme gibt es beispielsweise im Bereich der probabilistischen Komplexitätsklassen. Schränkt man hier - wie für praktisch verwendbare probabilistische Algorithmen erforderlich - die Fehlerwahrscheinlichkeit ein, so resultiert daraus unter anderem, dass die Komplexitätsklassen nicht mehr aufzählbar sind. Dies ist aber für alle Separationsverfahren eine Voraussetzung. Als Ergebnis lassen sich Polynomzeitalgorithmen plötzlich durch Linearzeitalgorithmen ersetzen. Das Beispiel zeigt, wie sensibel das Geflecht aus Voraussetzungen und den abgeleiteten Sätzen insgesamt ist.

[2] http://qwiki.stanford.edu/wiki/Complexity_Zoo

[3] http://people.cs.uchicago.edu/~fortnow/papers/history.pdf

[4] http://dispatch.opac.d-nb.de/DB=1.1/CMD?ACT=SRCHA&IKT=8&TRM=0002-9947

Komplexität (Informatik)

Komplexität bezeichnet in der Informatik die „Kompliziertheit“ von Problemen, Algorithmen oder Daten. Die Komplexitätstheorie befasst sich dabei mit dem Ressourcenverbrauch von Algorithmen, die Informationstheorie dagegen verwendet den Begriff für den Informationsgehalt von Daten (siehe unten).

Komplexität von Algorithmen

Unter der Komplexität (auch *Aufwand* oder *Kosten*) eines Algorithmus (nicht zu verwechseln mit der *Algorithmischen Komplexität*, siehe unten) versteht man in der Komplexitätstheorie seinen maximalen Ressourcenbedarf. Dieser wird oft in Abhängigkeit von der Länge n der Eingabe angegeben und für große n asymptotisch unter Verwendung eines Landau-Symbols abgeschätzt. Analog wird die Komplexität eines Problems definiert durch den Ressourcenverbrauch eines optimalen Algorithmus zur Lösung dieses Problems. Die Schwierigkeit liegt darin, dass man somit alle Algorithmen für ein Problem betrachten müsste, um die Komplexität desselben zu bestimmen.

Die betrachteten Ressourcen sind fast immer die Anzahl der benötigten Rechenschritte (Zeitkomplexität) oder der Speicherbedarf (Platzkomplexität). Die Komplexität kann aber auch bezüglich einer anderen Ressource bestimmt werden. Dabei interessiert nicht der Aufwand eines konkreten Programms auf einem bestimmten Computer, sondern viel mehr, *wie* der Ressourcenbedarf wächst, wenn mehr Daten zu verarbeiten sind, also z. B. ob sich der Aufwand für die doppelte Datenmenge verdoppelt oder quadriert (Skalierbarkeit).

Oft ist es sehr aufwändig oder ganz unmöglich, eine Funktion anzugeben, die allgemein zu jeder beliebigen Eingabe für ein Problem den zugehörigen Aufwand an Ressourcen angibt. Daher begnügt man sich in der Regel damit, statt jede Eingabe einzeln zu erfassen, sich lediglich mit der *Eingabelänge* $n = |w|$ zu befassen. Es ist aber meist sogar zu schwierig, eine Funktion $f_L : n \to f_L(n), n = |w|$ anzugeben. Daher beschränkt man sich häufig darauf, eine obere und untere Schranke für das asymptotische Verhalten anzugeben. Hierfür wurden die Landau-Symbole entwickelt.
Algorithmen und Probleme werden in der Komplexitätstheorie gemäß ihrer so bestimmten Komplexität in so genannte Komplexitätsklassen eingeteilt. Diese sind ein wichtiges Werkzeug, um bestimmen zu können, welche Probleme „gleich schwierig“, beziehungsweise welche Algorithmen „gleich mächtig“ sind. Dabei ist die Frage, ob zwei Komplexitätsklassen gleichwertig sind, oft nicht einfach zu entscheiden (zum Beispiel P-NP-Problem).

Die Komplexität eines Problems ist zum Beispiel entscheidend für die Kryptographie und insbesondere für die asymmetrische Verschlüsselung: So verlässt sich zum Beispiel das RSA-Verfahren auf die Vermutung, dass die Primfaktorzerlegung von großen Zahlen nur mit sehr viel Aufwand zu berechnen ist – anderenfalls ließe sich aus dem öffentlichen Schlüssel leicht der private Schlüssel errechnen.

Komplexität von Daten

In der Informationstheorie versteht man unter der Komplexität von Daten bzw. einer Nachricht ihren Informationsgehalt. Neben der klassischen Definition dieser Größe nach Claude Shannon gibt es verschiedene andere Ansätze, zu bestimmen, wie viel Information in einer Datenmenge enthalten ist:

Zum einen gibt es die so genannte *Kolmogorow-Komplexität* (auch *Algorithmische Komplexität* oder *Beschreibungskomplexität*), die den Informationsgehalt als die Größe des kleinsten Programms definiert, das in der Lage ist, die betrachteten Daten zu erzeugen. Sie beschreibt eine absolut optimale Komprimierung der Daten. Eine Präzisierung des Ansatzes Andrei Kolmogorows bezüglich des Maschinenmodells bietet die *Algorithmische Informationstheorie* von Gregory Chaitin.

Dagegen betrachtet *Algorithmische Tiefe* (auch *Logische Tiefe*) die Zeitkomplexität eines optimalen Algorithmus zur Erzeugung der Daten als Maß für den Informationsgehalt.

Komplexitätsmetriken

Es gibt eine Reihe von Softwaremetriken, welche die Komplexität von Daten und Algorithmen in der Informatik auf unterschiedliche Art und Weise messen. Diese sind:

- Chapins-Data-Metrik – misst den Anteil der Bedingungs- und Ergebnisdaten von allen verwendeten Daten.
- Elshofs-Data-Flow-Metrik – misst die Anzahl der Datenverwendungen relativ zur Anzahl der Daten. Sie ist verwandt mit hoher Kohesion. Hohe Kohesion entspricht einer hohen Verwendung von möglichst wenig Variablen.
- Cards-Data-Access-Metrik – misst das Verhältnis der Anzahl Zugriffe auf externe Dateien und Datenbanken relativ zur Anzahl derselben.
- Henrys-Interface-Metrik – misst die Anzahl der Zugriffe von fremden Funktionen/Methoden in ein Modul (englisch *fan-in*) beziehungsweise Anzahl der Aufrufe fremder Funktionen/Methoden aus einem Modul (englisch *fan-out*) relativ zu allen Funktionen/Methoden des Moduls.
- McCabe-Metrik bzw. Eulers Maß bzw. Zyklomatische Komplexität – misst die Komplexität des Ablaufgraphen als Verhältnis Kanten zu Knoten.
- McClures-Decision-Metrik – misst den Anteil Entscheidungen von allen Anweisungen.
- Sneeds-Branching-Metrik – misst das Verhältnis der Anzahl Verzweigungen jeglicher Art zur Summe aller Anweisungen.
- Halstead-Metrik – misst die Anzahl der verschiedenen Wörter (hier Anweisungstypen) relativ zur Anzahl verwendeter Wörter (hier Anweisungen). Sie behauptet, je weniger verschiedene Anweisungstypen man verwendet, desto einfacher ist der Code, was sehr umstritten ist.

Siehe auch

- Effizienz

Testbarkeit

Testbarkeit ist der Grad, zu dem ein Software-Artefakt (ein Software-System, ein Software-Modul, ein Anforderungs- oder Entwicklungsdokument) den Test in einem gegebenen Testkontext unterstützt.

Testbarkeit ist keine intrinsische Eigenschaft von Software-Artefakten und kann nicht direkt (wie z. B. der Software-Umfang) gemessen werden. Stattdessen ist Testbarkeit eine extrinsische Eigenschaft, die sich aus der Wechselwirkung der Software mit den Testzielen, Test-Ressourcen und eingesetzten Testverfahren (d. h. dem Testkontext) ergibt.

Je geringer die Testbarkeit ist, desto höher ist der Testaufwand. Im Extremfall ist bei schlechter Testbarkeit der Test von Teilen der Software-Anforderungen gar nicht möglich.

Hintergrund

Der Aufwand und die Effektivität eines Software-Tests hängen unter anderem von den folgenden Faktoren ab:

- Software-Anforderungen
- Eigenschaften der Software wie Umfang, Komplexität und Testbarkeit
- Eingesetzte Testmethoden
- Entwicklungs- und Testprozess
- Qualifikation und Motivation der am Test beteiligten Personen

Testbarkeit von Software

Die Testbarkeit von Software wird u.a. durch folgende Faktoren bestimmt:

- **Kontrollierbarkeit**: Das Testobjekt kann in den für den Test erforderlichen Zustand gebracht werden.
- **Beobachtbarkeit**: Das Testergebnis kann beobachtet werden.
- **Isolierbarkeit**: Das Testobjekt kann isoliert getestet werden.
- **Trennung der Verantwortlichkeit**: Das Testobjekt hat **eine** wohldefinierte Verantwortlichkeit.
- **Verständlichkeit**: Das Testobjekt ist selbsterklärend bzw. gut dokumentiert.
- **Automatisierbarkeit**: Die Tests lassen sich automatisieren.
- **Heterogenität**: Unterschiedliche Technologien erfordern den gleichzeitigen Einsatz von unterschiedlichen Testverfahren und -Werkzeugen.

Die Testbarkeit der Software wird verbessert durch:

- Testgetriebene Entwicklung
- Entwurf für Testbarkeit

Testbarkeit von Anforderungen

Anforderungen sind testbar, wenn sie folgende Kriterien erfüllen:

- **konsistent**
- **vollständig**
- **eindeutig**: Die Anforderung kann nicht unterschiedlich interpretiert werden.
- **quantitativ formuliert**: Eine Anforderung wie "schnelle Antwortzeit" kann nicht verifiziert werden.
- **praktisch verifizierbar**: Der Test ist nicht nur theoretisch möglich, sondern auch mit begrenztem Aufwand in der betrieblichen Praxis durchführbar.

Weblinks

- www.testbarkeit.de [1] Website mit Hinweisen und Literatur zum Thema Software-Testbarkeit

Literatur

- Stefan Jungmayr: Improving testability of object-oriented systems [2], ISBN 3-89825-781-9
- Johannes Link: Softwaretests mit JUnit - Techniken der testgetriebenen Entwicklung [3], ISBN 3-89864-325-5
- Frank Westphal: *Testgetriebene Entwicklung mit JUnit und FIT*, ISBN 3-89864-220-8

Referenzen

[1] http://www.testbarkeit.de

[2] http://www.dissertation.de/index.php3?active_document=/FDP/sj929.pdf

[3] http://stmj.developertests.de/

Anhang

Quelle(n) und Bearbeiter des/der Artikel(s)

Aufwandsschätzung (Softwaretechnik) *Quelle*: http://de.wikipedia.org/w/index.php?oldid=79217189 *Bearbeiter*: Aka, Amtiss, Avron, Bernburgerin, Cactus26, Ephraim33, Erkan Yilmaz, Eryakaas, Estimatax, Exhrenda, Gardini, HaSee, Jaellee, KressnerD, MBq, Ma-Lik, Mixia, Norro, OttoK, Revvar, Richardigel, Schlurcher, SchmidtNils, Sebastian.Dietrich, Sozi, Superbass, TableSitter, Thomas P. Hartmann, Video2005, Wnme, Xqt, 24 anonyme Bearbeitungen

Schätzmethode *Quelle*: http://de.wikipedia.org/w/index.php?oldid=63398811 *Bearbeiter*: BratzelWatz, Cramer, Nobart, Pgergen, Zaungast, 4 anonyme Bearbeitungen

Delphi-Methode *Quelle*: http://de.wikipedia.org/w/index.php?oldid=79162392 *Bearbeiter*: ALE!, Aka, Bahnemann, Beek100, BurninLeo, Chrisqwq, Christian Stroppel, ChristophDemmer, Conversion script, Ennie, Felixjansen, Flominator, Ghw, Gibsonmann, Heinz Wittenbrink, Jivee Blau, LKD, Macadamia, MarvinMonroe, MichaelHaeckel, Mitch77, Nerd, Ngomes, Ot, Patchworker, Pgergen, Philphilphilphil, Pilawa, Qwqchris, RKoenig, Rainer Wasserfuhr, Robert "BuRnEr" Schadek, Rohrbeck, Saibo, Salvina, Sinn, Sommerkom, StefanRybo, Till.niermann, To old, Wissons, pop-mu-5-1-dialup-100.freesurf.ch, 46 anonyme Bearbeitungen

Zwei-Zeiten-Methode *Quelle*: http://de.wikipedia.org/w/index.php?oldid=23070459 *Bearbeiter*: AHZ, Cramer, Etagenklo, Jergen, Silberchen, WortUmBruch, 1 anonyme Bearbeitungen

Drei-Zeiten-Methode *Quelle*: http://de.wikipedia.org/w/index.php?oldid=61592608 *Bearbeiter*: Andreas aus Hamburg in Berlin, Antrios, Batke, Cramer, Erkan Yilmaz, Etagenklo, Snorky, 3 anonyme Bearbeitungen

Best Practice *Quelle*: http://de.wikipedia.org/w/index.php?oldid=73072874 *Bearbeiter*: A Ruprecht, Abdull, Abubiju, Aka, Aloiswuest, Catrin, Daveboy123, Ephraim33, Erkan Yilmaz, Jjumper, LoisLane, Manja, Rauchfarbenes strahlenloses Licht, Rudolfox, Spacebirdy, Stern, T34, W!B:, WissensDürster, 31 anonyme Bearbeitungen

COCOMO *Quelle*: http://de.wikipedia.org/w/index.php?oldid=79345769 *Bearbeiter*: Abubiju, Balumir, ChristophDemmer, Flea, HaSee, Hydro, Jens m0, JuTa, Kaneiderdaniel, Krawi, MFM, Ma-Lik, Moros, Mstolt, Philipp3286, Rax, Rufus46, SRyll, Schlurcher, Schubbay, Slychief, Sparti, Speck-Made, Swoon, Tambora, Turdus, Weissbier, WiseWoman, Zubi, 71 anonyme Bearbeitungen

Function-Point-Verfahren *Quelle*: http://de.wikipedia.org/w/index.php?oldid=77100234 *Bearbeiter*: 1000, Amtiss, Bernd2150, Christian Storm, DasFliewatüüt, EdBever, Erkan Yilmaz, Eryakaas, Gerbil, Huerten, JeeAge, Jpp, Kaneiderdaniel, Krawi, Markus Großmann, Mstolt, Reddy, Roland Kaufmann, Saibo, Septembermorgen, Sparti, Stse, Tjark, Wahldresdner, Wolfgang H., 60 anonyme Bearbeitungen

Function Point Analyse *Quelle*: http://de.wikipedia.org/w/index.php?oldid=56472664 *Bearbeiter*: Crazy1880, DasBee, Fp modeler, HAL Neuntausend, Kein Einstein, Pittimann, Tröte, Wahldresdner, 5 anonyme Bearbeitungen

Lines of Code *Quelle*: http://de.wikipedia.org/w/index.php?oldid=79499781 *Bearbeiter*: Ben-k86, Bitsandbytes, Erkan Yilmaz, Hystrix, Johannesbauer, King gabson, Lagota, Lkwg82, Mike Krüger, Mstolt, PaterMcFly, PeterGerstbach, Rabenkind, Saehrimnir, Schmitty, Srbauer, Sven Pauli, Tyset, 20 anonyme Bearbeitungen

Softwaremetrik *Quelle*: http://de.wikipedia.org/w/index.php?oldid=77408732 *Bearbeiter*: Avron, BenediktM, Berlinschneid, Boemmels, Capaci34, ChristianHujer, Dathomas, Elwood j blues, Erkan Yilmaz, Frank Jacobsen, Froggy, Gaius L., Hans-Jörg Günther, JeeAge, Jpp, Levin, Mboehmer, Mstolt, Niemeyerstein, Olei, Patmuk, PeterVitt, QualiStattQuanti, Rene Mas, RobertAUlm, Sebastian.Dietrich, Solid State, Sparti, Surrounder, Tamaraz, Test-tools, Uncopy, Uwe Hermann, W!B:, Wiegels, 48 anonyme Bearbeitungen

Goal Question Metric *Quelle*: http://de.wikipedia.org/w/index.php?oldid=74956982 *Bearbeiter*: AHZ, BenediktM, Checkdaflo83, Erkan Yilmaz, ExilWestfale, Heinte, Hydro, JensKohl, Kaneiderdaniel, Louis Bafrance, Pentachlorphenol, S.K., STBR, Sebastian.Dietrich, 17 anonyme Bearbeitungen

Halstead-Metrik *Quelle*: http://de.wikipedia.org/w/index.php?oldid=79216722 *Bearbeiter*: ChristophDemmer, Ew-h2002, Froggy, MovGP0, Mstolt, S.K., Tziemer, WaLn, 12 anonyme Bearbeitungen

McCabe-Metrik *Quelle*: http://de.wikipedia.org/w/index.php?oldid=79216646 *Bearbeiter*: BB-Froggy, Chiccodoro, Exil, FelixReimann, Froggy, Harrga, Infohoschie, Jpp, Kerbel, Ktnagel, Mdetting, Mstolt, Peter200, Quarim, Rabus, Schmiddtchen, Sparti, Srbauer, Trublu, Uncopy, Zaungast, 20 anonyme Bearbeitungen

Softwarequalität *Quelle*: http://de.wikipedia.org/w/index.php?oldid=78631836 *Bearbeiter*: Avron, BlackMamba, Christian Meyer, ChristianHujer, ChristophDemmer, Erkan Yilmaz, Fleshgrinder, GRRD, Ghw, Gustavf, Hubertl, Ich hab hunga, Jens611, Joslankes, Jwilkes, Kku, MarZilein, Mkleine, Mnh, Paddy, STBR, Softq, Sparti, Staro1, Sterling, ToSter, Torsten.Stefan, Urizen, Uwe Hermann, W!B:, Zahnradzacken, 57 anonyme Bearbeitungen

Wirtschaftlichkeit *Quelle*: http://de.wikipedia.org/w/index.php?oldid=78188518 *Bearbeiter*: Ahoerstemeier, Babyauditor, Benatrevqre, Boehm, Breßler, Carab, Cybazyrfa, Docmo, Drahreg01, Ephraim33, Gamma, Geisslr, Gratisaktie, HeRaider, Ing. Schröder Walter, Inkowik, JaPPe, Jack (tR), Jan eissfeldt, Johnny Controletti, Kerbel, Krems, Liberatus, Licht-Ausmach-Män, Livani, Lucutus84, Matt1971, Millbart, Norbach, Oratio, Ordnung, P. Birken, Priwo, Ralf Z., Reni Tenz, RobbyBer, Sicherlich, Sinn, Spielblau, Stauba, Strelok, TOCO, Tolanor, TomK32, Wolfgang1018, Yotwen, Zaibatsu, Zenit, 56 anonyme Bearbeitungen

Fehlerquotient *Quelle*: http://de.wikipedia.org/w/index.php?oldid=66794871 *Bearbeiter*: Aka, Baumfreund-FFM, CollectiveStupidity, D, Gerbil, Gunther, LocoVincent, Nb, Sebastian.Dietrich, Serpens, Siehe-auch-Löscher, 8 anonyme Bearbeitungen

Paretoprinzip *Quelle*: http://de.wikipedia.org/w/index.php?oldid=77776824 *Bearbeiter*: AN, BeeDotGee, Conny, Grindinger, Laufe42, Megaman7de, Nicolas G., Superplus, Tjmoel, Trustable, Tubas, YourEyesOnly, 15 anonyme Bearbeitungen

Mean Time Between Failures *Quelle*: http://de.wikipedia.org/w/index.php?oldid=77396938 *Bearbeiter*: 9xl, Adema, Aka, Andi8086, Avron, Baumbart, Björn König, Brownout, Carstenrun, CyRoXX, DimaS, ERuede, Ephraim33, Fgb, Fuenfundachtzig, Gapeev, Gustavf, HaSee, Itu, JCS, Jergen, Jowi24, Jw188, JøMa, Kdwnv, Kku, Matthäus Wander, Mik81, Ordnung, Peter200, PhilippWeissenbacher, Polluks, Pragette, Priwo, Rellek, SDB, Saluk, Sarge Baldy, Smial, Sparti, Srbauer, Th-Schf, Trickser, Vossi75, Wdwd, WikipediaMaster, WissensDürster, Wkw1959, 97 anonyme Bearbeitungen

Testabdeckung *Quelle*: http://de.wikipedia.org/w/index.php?oldid=74694821 *Bearbeiter*: Avron, Boemmels, ChristianHujer, Hob Gadling, Jschlosser, Leider, Mstolt, Orci, Sebastian.Dietrich, Sparti, Test-tools, 9 anonyme Bearbeitungen

Personenstunde *Quelle*: http://de.wikipedia.org/w/index.php?oldid=78389564 *Bearbeiter*: Abubiju, Aleks-ger, Avron, Eriom, Gratisaktie, Jan eissfeldt, Jpp, Mps, Ocrho, Onee, Revolus, RokerHRO, Sabine0111, Siehe-auch-Löscher, WikipediaMaster, WiseWoman, Yarin Kaul, 12 anonyme Bearbeitungen

Komplexitätstheorie *Quelle*: http://de.wikipedia.org/w/index.php?oldid=78955561 *Bearbeiter*: AF666, Accountalive, Ahoerstemeier, Aka, Akribix, AlfonsGeser, Anarchitect, Anhi, Archiv, Artanh, Baumfreund-FFM, Beyer, Bitbert, Bsmuc64, Bunt, Carbidfischer, Chaddy, Christian Gawron, ChristophDemmer, Complex, Crux, Divisor, Duesentrieb, Esperantisto, Expent, FRR, FordPrefect42, Head, Herbert Lehner, Hongkongpfui, Horrorist, Hschaefer, Hydro, Igrimm12, J.e, JFKCom, JakobVoss, Jpp, JuTa, KaPe, Karl-Henner, Kku, Koethnig, Korelstar, Kubieziel, Kölsche Jung, LimoWreck, MFM, MRA, Manoridius, Marc van Woerkom, Mkleine, Mr-duke, N-regen, Nachtagent, NeoUrfahraner, Nina, Ollio, Paeng, Paul Ebermann, Pinguin.tk, Pinoccio, Rdb, RokerHRO, Romanticor, S1, SRyll, Scherben, Schlurcher, Sdo, Shiut, Shurakai, Snotty, Spid, Stefan, Stern, Störfix, Th., The Lake, Thire, Tobias1983, VictorAnyakin, Wasseralm, Wiegels, Wilfried Elmenreich, YMS, Youandme, Zap, Zeno Gantner, Zooloo, 80 anonyme Bearbeitungen

Komplexität (Informatik) *Quelle*: http://de.wikipedia.org/w/index.php?oldid=76167431 *Bearbeiter*: Accountalive, Complex, Conny, Conversion script, Duesentrieb, Esperantisto, Fomafix, FordPrefect42, Harald Tribune, Head, Homer Landskirty, Hystrix, Kku, Kubieziel, Lustiger seth, Mathiasl26, Ost38, PeterVitt, Piefke, Pinguin.tk, Planegger, Rade Kutil, Regnaron, RobM, Sebastian.Dietrich, Sinn, Spid, Spuk968, Stern, Trustable, Vernanimalcula, Wst, Zefram, 32 anonyme Bearbeitungen

Testbarkeit *Quelle*: http://de.wikipedia.org/w/index.php?oldid=78856800 *Bearbeiter*: Aka, Ammit, Erkan Yilmaz, JFKCom, JMetzler, Jergen, ManWing2, UlrichAAB, 3 anonyme Bearbeitungen

Quelle(n), Lizenz(en) und Autor(en) des Bildes

Bild:Fcm-tree.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Fcm-tree.png *Lizenz*: GNU General Public License *Bearbeiter*: Chaddy, Crux, Joslankes

Datei:Time between failures.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Time_between_failures.jpg *Lizenz*: Public Domain *Bearbeiter*: Maurizio.Cattaneo

Datei:Theoretische-informatik.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Theoretische-informatik.svg *Lizenz*: GNU Free Documentation License *Bearbeiter*: Benutzer:Paeng

Datei:Entscheidungsproblem.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Entscheidungsproblem.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Original uploader was Mkleine at de.wikipedia

Datei:Turingmaschine.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Turingmaschine.svg *Lizenz*: GNU Free Documentation License *Bearbeiter*: User:TripleWhy

Datei:Registermaschine-beispiel.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Registermaschine-beispiel.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Mkleine, N-regen

Datei:Kellerautomat.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Kellerautomat.svg *Lizenz*: Public Domain *Bearbeiter*: User:N-regen

Datei:Nea02.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Nea02.png *Lizenz*: unbekannt *Bearbeiter*: Benutzer:APPER

Datei:Inklusionsdiagramm_Komplexitätsklassen.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Inklusionsdiagramm_Komplexitätsklassen.png *Lizenz*: unbekannt *Bearbeiter*: Benutzer:MRA

Lizenz

Wichtiger Hinweis zu den Lizenzen

Die nachfolgenden Lizenzen bezieht sich auf den Artikeltext. Im Artikel gezeigte Bilder und Grafiken können unter einer anderen Lizenz stehen sowie von Autoren erstellt worden sein, die nicht in der Autorenliste erscheinen. Durch eine noch vorhandene technische Einschränkung werden die Lizenzinformationen für Bilder und Grafiken daher nicht angezeigt. An der Behebung dieser Einschränkung wird gearbeitet.
Das PDF ist daher nur für den privaten Gebrauch bestimmt. Eine Weiterverbreitung kann eine Urheberrechtsverletzung bedeuten.

Creative Commons Attribution-ShareAlike 3.0 Unported - Deed

Diese "Commons Deed" ist lediglich eine vereinfachte Zusammenfassung des rechtsverbindlichen Lizenzvertrages (http://de.wikipedia.org/wiki/Wikipedia:Lizenzbestimmungen_Commons_Attribution-ShareAlike_3.0_Unported) in allgemeinverständlicher Sprache.
Sie dürfen:

- das Werk bzw. den Inhalt **vervielfältigen, verbreiten und öffentlich zugänglich machen**
- **Abwandlungen und Bearbeitungen** des Werkes bzw. Inhaltes anfertigen

Zu den folgenden Bedingungen:

- **Namensnennung —** Sie müssen den Namen des Autors/Rechteinhabers in der von ihm festgelegten Weise nennen.
- **Weitergabe unter gleichen Bedingungen —** Wenn Sie das lizenzierte Werk bzw. den lizenzierten Inhalt bearbeiten, abwandeln oder in anderer Weise erkennbar als Grundlage für eigenes Schaffen verwenden, dürfen Sie die daraufhin neu entstandenen Werke bzw. Inhalte nur unter Verwendung von Lizenzbedingungen weitergeben, die mit denen dieses Lizenzvertrages identisch, vergleichbar oder kompatibel sind.

Wobei gilt:

- **Verzichtserklärung —** Jede der vorgenannten Bedingungen kann aufgehoben werden, sofern Sie die ausdrückliche Einwilligung des Rechteinhabers dazu erhalten.
- **Sonstige Rechte —** Die Lizenz hat keinerlei Einfluss auf die folgenden Rechte:
  - Die gesetzlichen Schranken des Urheberrechts und sonstigen Befugnisse zur privaten Nutzung;
  - Das Urheberpersönlichkeitsrecht des Rechteinhabers;
  - Rechte anderer Personen, entweder am Lizenzgegenstand selber oder bezüglich seiner Verwendung, zum Beispiel Persönlichkeitsrechte abgebildeter Personen.
- **Hinweis —** Im Falle einer Verbreitung müssen Sie anderen alle Lizenzbedingungen mitteilen, die für dieses Werk gelten. Am einfachsten ist es, an entsprechender Stelle einen Link auf http://creativecommons.org/licenses/by-sa/3.0/deed.de einzubinden.

Haftungsbeschränkung
Die „Commons Deed" ist kein Lizenzvertrag. Sie ist lediglich ein Referenztext, der den zugrundeliegenden Lizenzvertrag übersichtlich und in allgemeinverständlicher Sprache, aber auch stark vereinfacht wiedergibt. Die Deed selbst entfaltet keine juristische Wirkung und erscheint im eigentlichen Lizenzvertrag nicht.

GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies

of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.
This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.
We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.
A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.
A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.
The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.
The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.
A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".
Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.
The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.
A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.
The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.
You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.
If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.
If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.
It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.
- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.
You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.
You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.
The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.
The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.
You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.
If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.
If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.
Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
>
> Permission is granted to copy, distribute and/or modify this document
>
> under the terms of the GNU Free Documentation License, Version 1.2
>
> or any later version published by the Free Software Foundation;
>
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
>
> A copy of the license is included in the section entitled
>
> "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the
>
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.
If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.