HANSER

Leseprobe

zu

Software-Metriken

von Richard Seidl, Manfred Baumgartner und Harry M. Sneed

Print-ISBN: 978-3-446-47687-5
E-Book-ISBN: 978-3-446-47853-4
E-Pub-ISBN: 978-3-446-48058-2

Weitere Informationen und Bestellungen unter
https://www.hanser-kundencenter.de/fachbuch/artikel/9783446476875
sowie im Buchhandel

Inhalt

Vorwort

Dieses Buch „Software-Metriken“ ist das Ergebnis langjähriger Forschung und Entwicklung, die auf das ESPRIT-METKIT-Projekt im Jahre 1989 zurückgeht. Parallel zu dieser Forschungstätigkeit wurden über 30 Jahre lang Erfahrungen mit der Messung und Bewertung von Softwaresystemen in der industriellen Praxis gesammelt. Keiner hat sich in der Praxis so lange und so intensiv mit diesem Thema befasst wie der Autor Harry Sneed. Eine Erkenntnis, die er aus jener Erfahrung gezogen hat, ist die Bedeutung der Zahlen für die Softwarequalitätssicherung. Es ist nicht möglich, über Qualität zu reden, ohne auf Maßzahlen einzugehen. Es genügt nicht zu behaupten, System A sei viel schlechter als System B. Der Qualitätsgutachter muss erklären warum, denn Qualität ist relativ, und um die Qualität eines Softwareproduktes mit der Qualität eines anderen zu vergleichen, müssen beide Qualitäten in Zahlen ausgedrückt werden. Nur so kann man den Abstand zwischen den beiden Produktqualitäten erklären. Das Gleiche gilt für die Größe und die Komplexität eines Softwaresystems. Eine Aussage wie „Das System ist zu groß“ ist inhaltslos, ohne zu wissen, was „zu groß“ bedeutet. Auch Größe ist relativ zu den Vorstellungen des Menschen, die das System zu beurteilen haben. Sie müssen in der Lage sein, den Größenmaß mit einem Sollmaßstab für Softwaresysteme zu vergleichen. Voraussetzung dafür ist eine messbare und vergleichbare Zahl. Wer seine Aussagen nicht mit Zahlen belegen kann, wird nicht ernst genommen.

Es gibt zahlreiche Verwendungszwecke für die Zahlen, die wir aus der Software gewinnen:

- Wir können damit den Aufwand für ein Projekt kalkulieren.
- Wir können damit ein Projekt planen und steuern.
- Wir können damit Rückschlüsse auf die Qualität eines Produktes ziehen.
- Wir können damit die Produktivität unserer Mitarbeiter verfolgen.
- Wir können damit Ziele für die Produkt- und Prozessverbesserung setzen.
- Wir können damit Projekte und Produkte miteinander vergleichen.

Das sind auch längst nicht alle Zwecke. Zahlen sind eine unentbehrliche Voraussetzung für ein professionelles Projekt- und Produktmanagement. Dass wir bisher mit so wenig Zahlenmaterial ausgekommen sind, zeigt nur, wie unterentwickelt unsere Branche ist. Wenn wir weiterkommen wollen, müssen wir mehr mit Zahlen arbeiten.

An dieser Stelle möchten wir auf die Arbeit des Deutschen Zentrums für Softwaremetrik an der Universität Magdeburg unter der Leitung von Professor Dr. Reiner Dumke hinweisen. Diese Institution ist bemüht, in Zusammenarbeit mit der DASMA und der GI-Fachgruppe für Softwaremetrik Zahlen aus dem ganzen deutschsprachigen Raum zu sammeln und allen

interessierten Anwender bereitzustellen. Das Zentrum für Softwaremessung hat neben den vielen Tagungen und Workshops, die sie jährlich veranstaltet, und dem Rundbrief, den sie zwei Mal jährlich versendet, auch zahlreiche Veröffentlichungen zum Thema Softwaremessung herausgebracht, darunter:

- Dumke, R., Lehner, F.: *Software-Metriken*, Deutscher Universitätsverlag, Wiesbaden 2000
- Dumke, R., Abran, A.: *New Approaches in Software Measurement*, Springer-Verlag, Berlin Heidelberg, 2001
- Dumke, R., Rombach, D.: *Software-Messung und -Bewertung*, Deutscher Universitäts-Verlag, Wiesbaden 2002
- Dumke, R., Abran, A.: *Investigations in Software Measurement*, Shaker-Verlag, Aachen, 2003
- Abran, A., Dumke, R.: I*nnovations in Software Measurement*, Shaker-Verlag, Aachen, 2005
- Ebert, C., Dumke, R., Bundschuh, M., Schmietendorf, A.: *Best Practices in Software Measurement*, Springer-Verlag, Berlin Heidelberg, 2005
- Dumke, R., Büren, G., Abran, A., Cuadrado-Gallego, J.: *Software Process and Product Measurement*, Springer-Verlag, Berlin Heidelberg, 2008
- Büren, G., Dumke, R.: *Praxis der Software-Messung*, Shaker-Verlag, Aachen, 2009

Leser dieses Buches, die ihre Metrikkenntnisse vertiefen wollen, werden auf diese Veröffentlichungen hingewiesen. Wenn Sie auch noch bei der Weiterentwicklung der Softwaremetrik mitwirken wollen, möchten wir Sie ermutigen, der GI-Fachgruppe und/oder der DASMA beizutreten. Auf jeden Fall sollten Sie sich der deutschen Metrik Community anschließen, um auf diese Weise auf dem Laufenden zu bleiben. Dieses Buch wäre dann nur als Einstieg in die Welt der Softwarezahlen zu betrachten. Sie ist eine faszinierende Welt mit vielen Facetten.

Warum eine Neuauflage? Auch wenn die vorgestellten Konzepte und Metriken heute immer noch ihre Gültigkeit haben, hat sich die Welt des Software Engineerings weiterentwickelt. Und dieser Weiterentwicklung wollen wir Rechnung tragen. Gerade die agile Arbeitsweise erlaubt noch einmal einen neuen Blick auf Softwaremetriken, den wir gerne mit Ihnen hier teilen. Auch hat sich die Werkzeuglandschaft seit der ersten Auflage massiv verändert. Der Markt ist hier sehr dynamisch. Es entstehen ständig neue Tools, und ebenso verschwinden einige wieder oder werden nicht weiterentwickelt. Wir haben daher entschieden, konkrete Tools nur mehr punktuell zu nennen, wo sie dem Verständnis des dahinterliegenden Konzeptes dienlich sind.

Wien und Essen, im Januar 2024

Richard Seidl und Manfred Baumgartner

Geleitwort zur 1. Auflage

Zahlen sind aus unserem täglichen Leben nicht mehr wegzudenken. Wir planen Treffen zu bestimmten Zeitpunkten, kontrollieren die Gewichtsangaben von Produkten bezüglich möglicher Preisveränderungen, kalkulieren den Spritverbrauch für gefahrene Kilometer, klassifizieren Wohnungen nach ihren Quadratmetern, prüfen genau die Veränderungen des Kontostandes hinsichtlich der Buchungen, zählen die Häufigkeit auftretender Fehler bei der Nutzung von Haushaltsgeräten, mögen oder meiden die Zahl 13 für ein Hotelzimmer und vieles andere mehr. Wie sieht es aber bei Softwaresystemen aus? Kann man Software auch quantifizieren und Systemeigenschaften – insbesondere Qualität – genau bewerten oder gar exakt nachweisen? Was ist überhaupt Software?

Für die Beantwortung dieser und anderer Fragen hat sich eine Disziplin etabliert: das *Software Engineering*. Das bedeutet, dass Software etwas Reales ist, ein Artefakt als Softwaresystem, welches an eine (reale) Hardware gebunden ist und mit ingenieurtechnischen Methoden erstellt, gepflegt und somit auch analysiert und bewertet werden kann. Andererseits besteht Software nicht einfach nur aus (Computer-)Programmen, sondern umfasst alle dabei involvierten Entwicklungs-, Darstellungs- und Beschreibungsformen (also Dokumentationen). Für die Erstellung von Software wünscht man sich eigentlich

1. Beschreibungen von Methoden, die genau spezifizieren, was mit dieser Methode an Softwarequalität erreicht werden kann und was nicht,
2. Dokumentationen zu Entwicklungswerkzeugen, die zeigen, wie die Software mit all ihren Artefakten (entwicklungsbegleitend) an Komplexität, Performanz usw. zu- bzw. abnimmt,
3. Komponenten- bzw. Softwarebibliothekenbeschreibungen, die – analog zu einem elektronischen Handbuch – die genauen (Qualitäts-)Eigenschaften dieser Komponenten ausweisen,
4. schließlich: Softwaremaße, die einheitlich definiert und angewandt werden und damit eine generelle Vergleichbarkeit von Softwareeigenschaften gestatten.

Genau diesem komplexen Thema widmet sich das vorliegende Buch von Sneed, Seidl und Baumgartner, welches den eigentlichen Kern des Software Engineering (die Softwaremessung und -bewertung) behandelt, die die grundlegenden Eigenschaften eines Softwareproduktes quantifiziert darstellt, alle Artefakte der Entwicklung, Anwendung und Wartung einbezieht und die jeweilige Systemausprägung berücksichtigt. Das ist heute leider noch keine Selbstverständlichkeit. Es gibt immer noch zahlreiche Bücher zur Software bzw. zum Software Engineering, die

- die Softwarequalität vornehmlich bzw. nur auf die Qualitätsbestimmung von Programmen einschränken,

- die Verifikation von Softwaremodellen für eine Qualitätssicherung als hinreichend postulieren,
- die Darstellung von Softwaremetriken ausschließlich auf die ersten Denkansätze von McCabe und Halstead reduzieren,
- die Definition und Anwendung von Metriken nicht im Kontext eines Messprozesses und damit von Softwareprozessen überhaupt verstehen.

Auch und vor allem in dieser Hinsicht stellt das vorliegende Buch eine besondere Bereicherung der Literatur zum Software Engineering dar. Die Softwaremessung wird stets in den Kontext einer *zielgerichteten Vorgehensweise* innerhalb *realer Softwareprojekte und -entwicklungen* gestellt. Als Kern der Bewertung wird die Softwarequalität unter Verwendung der Softwaremerkmale wie Umfang und Komplexität betrachtet. Auch wenn die oben genannten vier Punkte immer noch eine Wunschliste darstellen, zeigen die Autoren sehr anschaulich, wie in der jeweiligen konkreten Situation mit Anforderungsanalyse, Modellierung, Design, Codierung und Test einerseits und vor allem der weiteren Wartung der Softwaresysteme andererseits jeweils Messmethoden und Maße auszuwählen und anzuwenden sind, um die jeweiligen (Qualitäts-)Ziele zu erreichen.

Der besondere Wert des Buches besteht aber auch vor allem im immensen Erfahrungshintergrund der Autoren, der nicht nur in der Kenntnis verschiedenster Entwicklungsmethoden und Softwaresystemarten, sondern vor allem in den über Jahrzehnte hinweg miterlebten und mitgestalteten Methoden-, Technologie-, Paradigmen- und vor allem Anwendungsbereichswechseln besteht. Das versetzt die Autoren auch in die Lage, scheinbar spielerisch den komplexen Prozess der Softwareentwicklung mit Zahlen zu unterlegen, die genau die jeweils zu bewertenden Softwaremerkmale charakterisieren. Das abschließende Kapitel zur Softwaremessung in der Praxis zeigt noch einmal die noch offenen Fragen in diesem Bereich, denen sich auch vor allem die nationalen und internationalen Communities zu diesem Thema widmen, wie das Common Software Measurement International Consortium (COSMIC), das Metrics Association's International Network (MAIN), die Deutschsprachige Anwendergruppe für Software-Metrik und Aufwandschätzung (DASMA) und nicht zuletzt die Fachgruppe für Softwaremessung und -bewertung der Gesellschaft für Informatik (GI FG 2.1.20), in denen auch die Autoren dieses Buches aktiv mitarbeiten.

Das vorliegende Buch von Harry Sneed, Richard Seidl und Manfred Baumgartner ist sehr anschaulich geschrieben, sehr gut lesbar und kann von seiner Themenbreite als *Handbuch des Software Engineering* angesehen werden. Es ist vornehmlich für den im IT-Bereich praktisch Tätigen, aber vor allem auch als Ergänzungsliteratur für den Hochschulbereich hervorragend geeignet.

Reiner Dumke
Professor für Softwaretechnik, Otto-von-Guericke-Universität Magdeburg

Die Autoren

Harry M. Sneed

Harry M. Sneed ist seit 1969 Magister der Informationswissenschaften der University of Maryland. Seit 1977, als er für das Siemens ITS-Projekt die Rolle des Testmanagers übernommen hat, arbeitet er im Testbereich. Damals entwickelte er die erste europäische Komponententestumgebung namens PrüfStand und gründete gemeinsam mit Dr. Ed Miller das erste kommerzielle Testlabor in Budapest. Seit dieser Zeit hat Harry M. Sneed mehr als 20 verschiedene Testwerkzeuge für unterschiedliche Umgebungen entwickelt – von Embedded-Echtzeitsystemen über integrierte Informationssysteme auf Großrechnern bis hin zu Webapplikationen.

Am Beginn seiner Karriere hat er als Testprojektleiter gearbeitet; am Ende seiner langen Karriere war er für die ANECON GmbH in Wien in die Rolle eines Softwaretesters zurückgekehrt. Parallel zu seiner Projekttätigkeit hat Harry Sneed über 200 technische Artikel und 18 Bücher (davon vier über das Thema Test) verfasst. Er unterrichtete zudem Softwareentwicklung an der Universität von Regensburg, Softwarewartung an der technischen Hochschule in Linz sowie Softwaremessung, Reengineering und Test an den Universitäten von Koblenz und Szeged. 2005 wurde Sneed von der Deutschen Gesellschaft für Informatik zum „GI Fellow“ berufen und übte die Funktion des „general chair“ der Internationalen Konferenz für Softwarewartung in Budapest aus. 1996 wurde Sneed vom IEEE für seine Errungenschaften im Bereich des Software Reengineerings ausgezeichnet, und 2008 erhielt er den Stevens Award für seine Pionierarbeit in der Disziplin der Softwarewartung. 2011 wurde er für sein Lebenswerk mit dem renommierten Deutschen Preis für Softwarequalität (DPSQ) ausgezeichnet.

Richard Seidl

Richard Seidl ist Agile Quality Coach und Softwaretestexperte. In seiner abwechslungsreichen beruflichen Laufbahn hat er schon viel Software gesehen und getestet: gute und schlechte, große kleine, alte und neue. Seine Erfahrungen bündelt er nun zu einem ganzheitlichen Ansatz, denn Entwicklungs- und Testprozesse können nur dann erfolgreich sein, wenn die unterschiedlichsten Kräfte sowie Stärken und Schwächen ausbalanciert sind. So wie ein Ökosystem nur mit allen Aspekten in seiner ganzen Qualität harmonisch existieren kann, müssen die Prozesse im Testumfeld als ein Netzwerk verschiedener Akteure betrachtet werden. Agilität und Qualität wird dann zu einer Haltung, die wir wirklich leben können, anstatt sie nur abzuarbeiten. Als Autor und Co-Autor hat er verschiedene Fachbücher und Artikel veröffentlicht, darunter „Der Systemtest – Von den Anforderungen zum Qualitätsnachweis“ (2006, 2008, 2011), „Der Integrationstest – Von Entwurf und Architektur zur Komponenten- und Systemintegration“ (2012) und „Basiswissen Testautomatisierung“ (2012, 2015, 2021). Seit April 2023 betreibt er zudem den Podcast „Software-Testing“.

Manfred Baumgartner

Manfred Baumgartner verfügt über mehr als 30 Jahre Erfahrung in der Softwareentwicklung, insbesondere in der Softwarequalitätssicherung und im Softwaretest. Nach dem Studium der Informatik an der Technischen Universität Wien war er als Softwareentwickler bei einem großen Softwareunternehmen im Bankensektor und später als Quality Director eines CRM-Lösungsanbieters tätig. Seit 2001 hat er die QS-Beratungs- und Schulungsangebote der ANECON, später Nagarro GmbH, eines der führenden Dienstleistungsunternehmen im Bereich Softwaretest, auf- und ausgebaut. Er ist Vorstandsmitglied im Arbeitskreis für Softwarequalität und Fortbildung (ASQF) und Mitglied des Austrian Testing Board (ATB). Seine umfangreichen Erfahrungen sowohl in der klassischen als auch in der agilen Softwareentwicklung bringt er als beliebter Referent auf international renommierten Konferenzen und als Autor und Co-Autor einschlägiger Fachbücher ein: „Der Systemtest – Von den Anforderungen zum Qualitätsnachweis“ (2006, 2008, 2011), „Software in Zahlen“ (2010), „Basiswissen Testautomatisierung“ (2012, 2015, 2021), „Agile Testing – Der agile Weg zur Qualität“ (2013, 2018, 2023).

1 Softwaremessung

1.1 Das Wesen von Software

Software ist Sprache. Sie dient der Kommunikation zwischen den Menschen und Rechnern ebenso wie zwischen Rechnern und Rechnern und zwischen Menschen und Menschen (siehe Bild 1.1). Programmcode ist jene Sprache, in der der Mensch der Maschine Anweisungen erteilt. Der Mensch schreibt den Code, der Rechner liest ihn. Er muss sowohl von den Menschen als auch vom Rechner, in diesem Fall dem Compiler, verstanden werden [DeLi99].

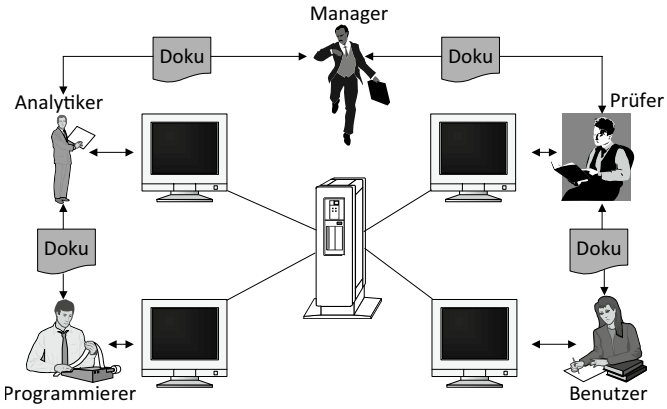

Bild 1.1 Software als Kommunikation zwischen Mensch und Maschine

Anforderungsspezifikationen und Entwurfsdiagramme sind ebenfalls Software, also auch Sprachen. Sie dienen der Kommunikation zwischen Menschen. Der eine Mensch schreibt sie, z. B. der Analytiker, der andere Mensch – der Programmierer – liest sie. Wenn sie nicht für beide Seiten verständlich sind, haben sie ihren Zweck verfehlt. Eine Spezifikation, die von einem Rechner interpretiert werden kann, z. B. eine domänenspezifische Sprache, ist zugleich eine Kommunikation zwischen Mensch und Rechner, ähnlich dem Programmcode.

Kommunikationsprotokolle wie XML-Dateien und Web-Service SOAP-Nachrichten sind desgleichen Software. Sie dienen der Kommunikation zwischen Rechnern. Der eine Rechner

schreibt sie, der andere liest sie. Sie muss daher von beiden Rechnern verstanden werden. Ein Protokoll ist eine Vereinbarung zwischen zwei Rechnerarten, wie sie sich verständigen wollen, ebenso wie eine Sprache eine Vereinbarung zwischen Menschen ist, die sich verständigen wollen. Natursprachen sind aus dem Zusammenleben der Menschen heraus erwachsen. Programmier-, Spezifikations- und Testsprachen sind wiederum aus dem Zusammenleben der Menschen mit Computern hervorgegangen [Rose67].

Wenn es nun um die Messung und Erforschung von Software geht, geht es also um die Analyse und Bewertung von Sprachen und den in diesen Sprachen geschriebenen Werken.

Eine Rechnersprache besteht genauso wie eine Sprache der Menschen aus Begriffen und Regeln für die Zusammensetzung jener Begriffe. Der Umfang einer Sprache wird an der Anzahl ihrer Begriffe bzw. Wörter gemessen. Oft legen Schüler Wörterbücher zweier unterschiedlicher Sprachen nebeneinander, um zu sehen, welches dicker ist. Dies ist in der Tat eine sehr grobe Messung des Sprachumfangs und setzt voraus, dass die Seitenaufteilung und die Schriftgröße gleich sind, aber sie ist nichtsdestotrotz eine Messung. Genauer wäre es, die Worteinträge zu zählen und zu vergleichen, aber auch hier ist die Messung ungenau, denn wer weiß, ob in den Wörterbüchern alle möglichen Wörter in beiden Sprachen berücksichtigt sind? Die Zählung der Wörter ist auf jeden Fall genauer als der Vergleich der beiden Wörterbücher. Das Gleiche gilt für Softwaresprachen. Ihr Umfang in vereinbarten Begriffen bzw. Symbolen lässt sich grob und fein vergleichen [LiGu88].

Aber nicht nur die Sprachen selbst können gemessen und miteinander verglichen werden. Auch die Ergebnisse von Sprache wie zum Beispiel Theaterstücke, Bücher, Essays etc. können nach unterschiedlichen Kriterien und zu unterschiedlichen Zwecken gemessen werden. Ist die Schularbeit lang genug? Durch Zählung der Wörter erhält man die Antwort. Warum ist das Buch „Die Buddenbrooks“ von Thomas Mann schwerer zu lesen als Astrid Lindgrens „Pippi Langstrumpf“ und kann man den Unterschied messen? Der Umfang alleine scheint dafür nicht der Grund zu sein und die Zählung der Seiten oder Worte wohl eine zu einfache Erklärung. Sind die Sätze durchschnittlich länger? Haben die beiden Werke einen unterschiedlichen Wortschatz? Wenn jedes Wort, welches mehrfach vorkommt, nur einmal gezählt wird, hätten wir das Vokabular des Schriftstücks. Ähnlich verfuhr M. Halstead, als er begann, Programmcode zu messen [Hals77]. Er zählte alle Wörter, also Operatoren und Operanden, um die Programmlänge zu ermitteln, und zählte jedes verwendete Wort, um das Programmvokabular zu bestimmen. Daraus berechnete er einen Wert für die Schwierigkeit, ein Programm zu verstehen.

Wäre eine Sprache nur eine beliebige Aneinanderreihung von Begriffen, könnte man sich mit der Messung der Größe zufriedengeben. Aber eine Sprache hat auch eine Grammatik. Darin befinden sich die Regeln für die Zusammensetzung der Wörter. Den Wörtern werden Rollen zugewiesen. Es gibt Hauptwörter, Eigenschaftswörter, Zeitwörter usw. Ähnliche Regeln gibt es auch in der Software. Für jede Sprache – Spezifikationssprache, Entwurfssprache, Programmiersprache und Testsprache – gibt es Regeln, wie die Wörter und Symbole verwendet werden können. Man spricht hier von der Syntax der Sprache. Mit der Syntax kommt die Komplexität. Je nachdem, wie umfangreich die Regeln sind, ergeben sich mehr oder weniger mögliche Wortkonstrukte. Je mehr Wortkonstrukte möglich sind, desto komplexer ist die Sprache.

Durch den Vergleich der Grammatik bzw. der Sprachregel ist es möglich, die Komplexität der Sprachen zu vergleichen. Dies trifft für Deutsch, Englisch und Latein ebenso zu wie für COBOL, Java, UML und VDM. Erschwert wird dies allerdings durch die informale Definition

der Regeln und den vielen erlaubten Ausnahmen für die Sprache. In der Softwarewelt wird der Vergleich durch die vielen herstellerspezifischen Abweichungen erschwert. Es gibt kaum eine bekannte Softwaresprache, von der es nicht eine Reihe von Derivaten, sprich Dialekte gibt, die sich mehr oder weniger stark unterscheiden [Jone01].

In natürlichen Sprachen gibt es das Kunstwerk Satz: Das ist eine Zusammensetzung von Wörtern nach einem geregelten Muster. Ein Satz hat ein Subjekt, ein Objekt und ein Prädikat. Subjekt und Objekt sind Operanden bzw. Hauptwörter. Sie können durch Eigenschaftswörter ergänzt werden. Die Prädikate, sprich Zeitwörter, können gleichfalls Eigenschaftswörter haben, welche die Handlung ergänzen. Diese Wortarten müssen in einem gewissen Satzmuster vorkommen, um einen sinnvollen Satz zu bilden. Je mehr Muster zugelassen sind, desto komplexer die Satzbildung.

In Softwaresprachen entspricht der Satz einer Anweisung. Auch hier gibt es Syntaxregeln für die Satzbildung. Es gibt Operanden (= Objekte) und Operatoren (= Prädikate). Das Subjekt fehlt. Es wird impliziert als die ausführende Maschine. Der Rechner oder das System liest eine Datei, errechnet Datenwerte, vergleicht zwei Werte oder sendet Nachrichten. Je nachdem, wie viele Anweisungsarten eine Sprache hat, ist sie mehr oder weniger komplex. Die Zahl der einzelnen Anweisungen ist wie die Zahl der Sätze im Prosatext ein Größenmaß. Die Zahl der verschiedenen Anweisungsarten ist wiederum ein Komplexitätsmaß. Sie deutet auf die Komplexität der Sprache bzw. der jeweiligen Sprachanwendung hin.

Sprachen lassen sich in Form von Syntaxbäumen oder Netzdiagrammen darstellen. Peter Chen hat bewiesen, dass sich jeder Sprachtext, auch in einer natürlichen Sprache, mit einem „Entity/Relationship-Diagramm“ abbilden lässt [Chen76]. Die Begriffe sind die Entitäten, die Zusammensetzung der Begriffe ergeben die Beziehungen. Ursprünglich war das E/R Model für die Datenmodellierung gedacht, wobei die Entitäten die Datenobjekte sind. Es lässt sich jedoch genauso gut für die Funktionsmodellierung verwenden, wobei hier die Entitäten die Funktionen sind. Die Zahl der Entitäten bestimmt die Größe einer Beschreibung. Die Zahl der Beziehungen bestimmt deren Komplexität. Je mehr Beziehungen es zwischen Entitäten relativ zur Anzahl der Entitäten gibt, desto komplexer ist die Beschreibung.

Sprachen sind Beschreibungsmittel. Ihr Umfang hängt von der Zahl ihrer Begriffe, sprich den Entitäten ab. Ihre Komplexität hängt wiederum von der Zahl ihrer erlaubten Konstrukte bzw. möglichen Beziehungen zwischen ihren Begriffen ab. Eine Sprachanwendung ist eine ganz bestimmte Beschreibung. Softwaresysteme sind letztendlich nur Beschreibungen. Die Anforderungsspezifikation ist die Beschreibung einer fachlichen Lösung zu einem Zielproblem. Der Systementwurf, z. B. ein UML-Modell, ist die Beschreibung einer rechnerischen Lösung zum Zielproblem, die an die fachliche Beschreibung angelehnt werden sollte [ErPe00]. Der Programmcode ist ebenfalls nur eine Beschreibung, allerdings eine sehr detaillierte Beschreibung der technischen Lösung eines fachlichen Problems, das mehr oder weniger der Entwurfsbeschreibung und der Anforderungsbeschreibung entspricht. Schließlich ist die Testspezifikation nochmals eine Beschreibung dessen, wie sich die Software verhalten sollte.

Alle diese Beschreibungen ähneln den Schatten in Platons Höhlengleichnis [Plat06]. Sie sind nur abstrakte Darstellungen eines Objekts, das wir in Wahrheit gar nicht wahrnehmen können. Zum einen handelt es sich um abstrakte Darstellungen konkreter Vorstellungen und Anforderungen seitens eines Kunden an ein Softwaresystem, zum anderen um Beschreibungen von Rechenvorgängen auf unterschiedlichsten Abstraktionsebenen. Da wir das eigentliche Objekt selbst nicht messen können, messen wir die Beschreibungen des

Objekts und damit die Sprachen, in denen die Beschreibungen formuliert sind. Was wir bekommen, sind nur die Größe und die Komplexität einer Beschreibung. So gesehen ist jedes Softwaremaß ein Maß für eine Darstellung und kann nur so zuverlässig sein wie die Darstellung selbst.

Eine Beschreibung bzw. eine Darstellung hat nicht nur eine Quantität und eine Komplexität, sie hat außerdem noch eine Qualität, und diese soll auch messbar sein. Die Frage stellt sich, was die Qualität einer Beschreibung ist. Man könnte genauso gut nach der Qualität der Schatten in Platons Höhle fragen. Wir würden gerne antworten, die Qualität eines Schattens sei der Grad an Übereinstimmung mit dem Objekt, das den Schatten wirft. Demnach müsste die Qualität des Programmcodes am höchsten sein, weil diese Beschreibung am nächsten an den eigentlichen Rechenvorgang herankommt. Dies entspricht der Behauptung von DeMillo und Perles, die besagt, „die einzige zuverlässige Beschreibung eines Programms ist der Code selbst“ [DePL79]. Lieber würde der Mensch sich mit den Entwurfsbildern befassen, aber diese sind verzerrte Darstellungen der Wirklichkeit. Je leichter verständlich eine Darstellung ist, desto weiter ist es von der Wirklichkeit entfernt.

Aber was ist die Wirklichkeit? Was ist, wenn das real existierende System nicht dem entspricht, was der Auftraggeber haben wollte? Wie sollen wir wissen, ob die verwirklichte Funktionalität mit der gewünschten Funktionalität samt allen Eigenschaften übereinstimmt? Auch Platon unterscheidet zwischen den sichtbaren Schatten, die wir sehen können, und den projektierten Schatten, die wir sehen wollen. Ein Abgleich kann nur stattfinden, wenn wir zwei Beschreibungen vergleichen: die Beschreibung, die dem wahren Rechenvorgang am nächsten kommt, mit der Beschreibung, die den Vorstellungen des Auftraggebers am ehesten entspricht. In der Welt der Softwarekonstruktion wäre dies die Anforderungsspezifikation. Um diese Beschreibung mit der Beschreibung Programmcode zu vergleichen, müssen die beiden Beschreibungen einander begrifflich und syntaktisch zuordenbar sein. Das heißt, sie müssen sich in etwa auf der gleichen semantischen Ebene befinden. Eine grobe Anforderungsbeschreibung ist jedoch mit einer feinen Codebeschreibung nicht vergleichbar. Die Anforderungsbeschreibung müsste fast so fein sein wie die des Codes. Da dies mit Ausnahme der formalen Spezifikationssprachen wie Z, VDM und SET selten der Fall ist, wird die Anforderungsbeschreibung stellvertretend über die Testfälle mit dem echten Systemverhalten verglichen. Dabei darf nicht übersehen werden, dass die Testfälle zur Bestätigung der Erfüllung der Anforderungen auch in einer Sprache verfasst sind und als solche allen Unzulänglichkeiten jener Sprache ausgesetzt sind [Fetz88].

Der statische Zustand von Softwareprodukten, also Struktur und Inhalt ihrer Beschreibungen, kann entsprechend einer Vielzahl von Qualitätseigenschaften bewertet werden. So sollte z. B. der Programmcode als Beschreibung modular aufgebaut, flexibel, portabel, wiederverwendbar, testbar und vor allem verständlich sein. Dieses sind alles Kriterien, die sich unmittelbar auf die Beschreibung beziehen. Um sie messen zu können, werden Richtlinien und Konvention benötigt. Diese können in Form einer Checkliste, eines Musterbeispiels oder einer Soll-Metrik vorliegen. Auch hier handelt es sich um einen Soll-Ist-Vergleich. Die eigentliche Softwarebeschreibung wird gegen die Soll-Beschreibung abgeglichen. Jede Abweichung vom Soll wird als Mangel oder als Regelverletzung betrachtet. Die statische Qualität der Software wird anhand der Anzahl gewichteter Mängel relativ zur Größe gemessen. Je mehr Mängel eine Softwarebeschreibung hat und je schwerer diese Mängel wiegen, desto niedriger ist die statische Qualität [ZWNS06].

Softwareprodukte haben aber nicht nur einen statischen Zustand, sondern auch ein dynamisches Verhalten. Das alles erschwert die Messung der Systemqualität. Der Grad der dynamischen Qualität ist der Grad, zu dem das tatsächliche Systemverhalten mit dem erwarteten Systemverhalten übereinstimmt. Jede Abweichung zwischen Soll und Ist wäre als Abweichung zu betrachten, egal ob es sich um die Nichterfüllung einer funktionalen Anforderung, um die falsche Erfüllung einer solchen Anforderung oder um die Nichterfüllung einer nichtfunktionalen Anforderung handelt. Mit jedem zusätzlich festgestellten Fehler sinkt die Qualität. Die konventionelle Art, Softwarequalität zu messen, ist anhand der Anzahl der Fehler gewichtet durch die Fehlerschwere relativ zur Softwaregröße.

Bild 1.2 Messskalen nach Zuse

Es ist jedoch zu betonen, dass in beiden Fällen – der statischen Qualitätsmessung wie auch der dynamischen Qualitätsmessung – der Begriff Qualität relativ zu einer Beschreibung, nämlich der Beschreibung der erwarteten Qualität ist. Ohne eine derartige Beschreibung lässt sich Qualität nicht messen. Die Messung von Qualität impliziert den Vergleich einzelner Ist-Eigenschaften mit entsprechenden Soll-Eigenschaften. Es gibt keinen Weltstandard für Fehlerhaftigkeit – ebenso wenig wie es einen Weltstandard für Wartbarkeit oder Testbarkeit gibt. Hinter jedem Qualitätsmaß steckt eine heuristische Regel, die zu einer lokalen Norm erhoben wurde. Wie wir später sehen werden, kann jede Qualitätsnorm quantifiziert und auf eine Werteskala gebracht werden. Hinter jeder solchen Werteskala steckt jedoch eine heuristisch begründete oder willkürliche Vereinbarung, was gut und was schlecht ist (siehe Bild 1.2).

1.2 Sinn und Zweck der Softwaremessung

Ein wesentlicher Zweck der Softwaremessung ist, die Software besser zu verstehen. Dazu dienen uns die Zahlen. Zahlen helfen uns, die Zusammensetzung eines komplexen Gebildes wie ein Softwaresystem zu begreifen: „Comprehension through Numbers“ [Sned95]. Durch sie erfahren wir, wie viele verschiedene Bauelemente es gibt und wie viele Ausprägungen jedes hat, wir erhalten Informationen über deren komplexe Beziehungen und Maßzahlen über die Qualität der Softwaresysteme.

Ein weiterer Zweck ist die Vergleichbarkeit. Zahlen geben uns die Möglichkeit, Softwareprodukte mit anderen Softwareprodukten zu vergleichen bzw. verschiedene Versionen ein und desselben Produktes zu vergleichen. Nicht nur Produkte, auch Projekte und Prozesse lassen sich vergleichen – allerdings nur, wenn sie in Zahlen abbildbar sind.

Ein dritter Zweck ist die Vorhersage. Um planen zu können, müssen wir die Zukunft vorhersagen, z. B. schätzen können, was ein Projekt kosten wird. Dazu brauchen wir Zahlen aus der Vergangenheit, die wir in die Zukunft projizieren können.

Ein vierter Zweck ist, Zahleninformationen für die Steuerung von Projekten und Produktentwicklungen zu erhalten: Wenn z. B. wöchentlich hundert neue Fehler im Fehlermanagementtool erfasst werden, aber gleichzeitig nur dreißig geschlossen werden, sind entsprechende Steuerungsmaßnahmen zu ergreifen (z. B. Behebung der Fehler vor Implementierung neuer Funktionalität).

Der letzte Zweck ist eher abstrakt. Es geht darum, die Kommunikation zwischen Menschen zu verbessern. Wir kennen alle die Unzulänglichkeiten der natürlichen Sprachen. Es gibt viele uneindeutige Begriffe und solche, die nichtssagend sind. Die zwischenmenschliche Kommunikation leidet an Missverständnissen und Fehlinterpretationen. Die natürliche Sprache stößt schnell an ihre Grenzen, wenn es darum geht, komplexe technische Gebilde exakt zu beschreiben. Zahlen sind eine eindeutige Sprache. Urvölker kannten keine Zahlen. Sie konnten sagen, dass es einen Löwen gibt, wenige Löwen oder viele Löwen. Heute wissen wir, dass es drei Löwen gibt oder dass der Weltumfang etwa 40 000 Kilometer beträgt. Das ist eine andere Aussage als die, dass die Welt groß ist. So gesehen tragen Zahlen dazu bei, die zwischenmenschliche Kommunikationsfähigkeit zu steigern. Wie Lord Kelvin es so trefflich formuliert hat: „Erst wenn wir etwas in Zahlen ausdrücken können, haben wir es wirklich verstanden. Bis dahin ist unser Verständnis oberflächlich und unzulänglich“ [Kelv67]. Das heißt, erst wenn wir Software quantifizieren können, haben wir sie wirklich im Griff. Der englische Professor Norman Fenton behauptet, dass es ohne Metrik kein Software Engineering geben kann. Messung ist die Voraussetzung für jegliche Engineering-Disziplin [Fent94].

Zusammenfassend ist der Zweck der Softwaremessung fünferlei:

- Sie dient dem Softwareverständnis.
- Sie dient der Vergleichbarkeit.
- Sie dient der Vorhersage.
- Sie dient der Steuerung.
- Sie dient der zwischenmenschlichen Verständigung.

1.2.1 Zum Verständnis (Comprehension) der Software

Wenn wir Software verstehen wollen, müssen wir wissen, wie sie zusammengesetzt ist, d. h. aus welchen Bausteintypen sie besteht und welche Beziehungen zwischen jenen Bausteintypen existieren. Die Eigenschaften der Bausteintypen helfen, diese Typen zu klassifizieren. Am besten lassen sich diese Eigenschaften in Zahlen ausdrücken wie z. B. die Größe in Zeilen oder Wörtern oder Symbole. Die Zahl der Beziehungen zwischen den Bausteinen hilft uns, den Zusammenhang der Softwareelemente zu verstehen. Zahlen sind neben Sprache und Grafik ein weiteres Verständigungsmittel. Sie sind genauer als die anderen beiden Mittel.

1.2.2 Zum Vergleich der Software

Gesetzt den Fall, ein IT-Anwender muss zwischen zwei Softwareprodukten entscheiden, welche die gleiche Funktionalität haben. Wie soll er sie vergleichen? Ohne Zahlen wird der Vergleich schwer möglich oder sehr subjektiv sein. Mit Zahlen lassen sich Größe und Komplexität, ja sogar Qualität vergleichen. Er kann z. B. feststellen, dass das eine Produkt mit der Hälfte des Codes dasselbe leistet oder dass das eine Produkt um 20 % komplexer ist als das andere. Durch einen Performanztest kann er die Laufzeiten und die Antwortzeiten vergleichen. Das Gleiche gilt für den Vergleich von Versionen desselben Systems. Durch die Messung der Unterschiede wird erkennbar, ob ein System sich verbessert oder verschlechtert hat. Für den Vergleich sind Zahlen Grundvoraussetzung.

1.2.3 Zur Vorhersage

Solange Softwareentwicklung und -wartung Geld und Zeit kosten, wird der Käufer der Software wissen wollen, was diese kostet und wie lange ein Vorhaben dauern wird. Außerdem will der Käufer wissen, was er für sein Geld bekommt, also welche Funktionalität zu welcher Qualität. Damit wir diese verständlichen Wünsche erfüllen können, brauchen wir Zahlen. Die Dauer eines Projekts in Tagen oder Monaten ist eine Zahl, die jeder Auftraggeber wissen will, ebenso die Anzahl der Personentage, die er bezahlen muss. Falls es zu lange dauert oder zu viel kostet, wird er bereit sein, auf das Projekt zu verzichten, oder er wählt eine andere Lösung. Wenn er sieht, dass die Funktionalität zu wenig und die Qualität zu gering sein wird, wird er sich nach Alternativen umsehen. Der Kunde braucht Informationen für seinen Entscheidungsprozess. Durch die Softwaremessung erhält er nicht nur Zahlen zur Projektabwicklung, sondern auch detaillierte und objektive Informationen über das Softwaresystem und dessen Entwicklung selbst. Zahlen über Zahlen sind die beste Information, die er bekommen kann. Nur mit Zahlen ist eine fundierte Aussage möglich, alles andere ist reine Spekulation.

1.2.4 Zur Projektsteuerung

Ist ein Projekt einmal genehmigt und gestartet, sind Zahlen erforderlich, um den Stand des Projektes festzustellen. Die Projektleitung soll wissen, welcher Anteil der Software bereits fertig ist und was noch zu entwickeln ist. Sie soll auch wissen, wie es um die Qualität des fertigen Anteils bestellt ist. Entspricht diese der vereinbarten Qualität und wenn nicht, wie weit ist sie davon entfernt? Hierzu braucht man Zahlen: über den Umfang der gefertigten Software sowie Zahlen über den Qualitätszustand. Ohne Zahlen hat die Projektleitung kaum eine Chance, die Entwicklung oder Wartung von Software zu verfolgen und nach Bedarf einzugreifen. Wie Tom DeMarco es formulierte: „You cannot control what you cannot measure“ [DeMa82]. Messung ist die Vorbedingung für Steuerung; und zur Messung gehört eine Metrik. Das Wort „Metrik“ kommt aus dem Altgriechischen und bezeichnet im Allgemeinen ein System von Kennzahlen oder ein Verfahren zur Messung einer quantifizierbaren Größe [Wik07].

1.2.5 Zur zwischenmenschlichen Verständigung

Die Menschen haben genug Schwierigkeiten, sich über Alltagsprobleme wie den Kauf eines neuen Autos oder den Anbau einer neuen Garage zu verständigen. Zahlen wie die der Pferdestärke, Höchstgeschwindigkeit und Hubraum erleichtern die Verständigung. Software ist eine unsichtbare Substanz – desto schwerer ist es deshalb, sich darüber zu verständigen. Niemand kann wissen, was der andere meint, wenn er sagt, die Software ist „groß“ oder die Aufgabe ist „komplex“. Man fragt sich sofort: Relativ zu was? Was bedeutet groß oder komplex? Man sucht nach einer Messskala für Größe oder Komplexität. Das Gleiche gilt für Qualität: Wenn einer sagt, das System wäre fehlerhaft, was meint er damit? Kommt ein Fehler bei jeder Nutzung oder bei jeder zehnten Nutzung vor? Damit sind wir bei Zahlen angelangt. Die Nutzung von Zahlen ist ein Indikator für die Genauigkeit der zwischenmenschlichen Kommunikation.

Für die Beschreibung von Software gilt dies umso mehr. Statt zu sagen, die Software sei groß, ist es genauer, wenn man sagt, die Software habe 15 557 Anweisungen. Wir setzen damit jedoch voraus, dass der Kommunikationspartner dies einordnen kann. Wer noch nie einen Source-Code und seine Anweisungen gesehen hat, für den hat auch die Zahl 15 557 keine Bedeutung.

1.3 Dimensionen der Substanz Software

Software ist eine multidimensionale Substanz. Sie hat bestimmt mehr Dimensionen, drei davon sind allerdings messbar. Die eine Dimension ist die Größe bzw. die Quantität der Software. Die zweite Dimension ist die Zusammensetzung bzw. die Komplexität der Software. Die dritte Dimension ist die Güte bzw. die Qualität der Software. Wenn also von Messung bei Software die Rede ist, dann von einer dieser drei Metrikarten:

- Quantitätsmetrik
- Komplexitätsmetrik
- Qualitätsmetrik (siehe Bild 1.3)

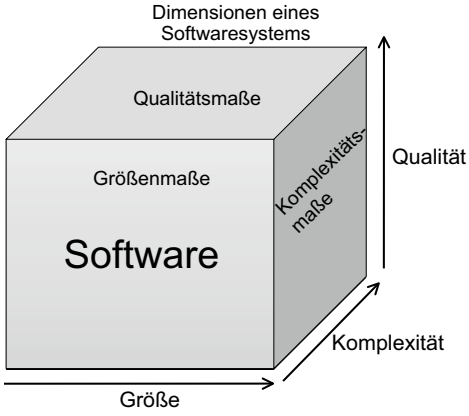

Bild 1.3
Drei Dimensionen von Software

1.3.1 Quantitätsmetrik von Software

Mit der Quantitätsmessung sind Mengenzahlen gemeint, z. B. die Menge aller Wörter in einem Dokument, die Menge der Anforderungen, die Menge der Modelltypen in einem Entwurfsmodell und die Menge aller Anweisungen in einer Source-Bibliothek. Mengenzählungen sind Aussagen über den Umfang von Software. Sie werden benutzt, um den Aufwand für die Entwicklung einer vergleichbaren Menge zu kalkulieren. Aus der Menge der Datenelemente wird die Größe der Datenbank projiziert, aus der Menge der Anforderungen wird die Menge der Entwurfsentitäten und aus dieser die Menge der Codeanweisungen abgeleitet. Aus der Menge der Anforderungen und Anwendungsfälle wird auch die Menge der Testfälle projiziert. In einem Softwaresystem gibt es etliche Mengen, die wir zählen könnten. Manche sind relevant, andere nicht. Unsere Aufgabe als Software-Ingenieure besteht darin, die relevanten Mengen zu erkennen. Eine weitere Herausforderung besteht darin, diese Mengen richtig zu zählen. Dafür brauchen wir Zählregeln. In diesem Buch werden mehrere davon behandelt.

1.3.2 Komplexitätsmetrik von Software

Mit der Komplexitätsmetrik sind Verhältniszahlen für die Beziehungen zwischen den Mengen und deren Elementen gemeint. Ein Element wie das Modul XY hat Beziehungen zu anderen Elementen wie zu weiteren Modulen oder zu weiteren Datenelementen. Die Zahl der Beziehungen ist eine Aussage über Komplexität. Die Menge aller Module hat Beziehungen zu der Menge aller Daten. Sie werden benutzt, erzeugt und geändert. Sie haben auch Beziehungen zur Menge aller Testfälle, die das Modul testen. Je mehr Beziehungen eine Menge hat, desto höher ist ihre Komplexität. Komplexität steigt und fällt mit der Zahl der Bezie-

hungen. Also gilt es hier, Beziehungen zu zählen und miteinander zu vergleichen. Das Problem ist hier dasselbe wie bei der Quantität, nämlich zu erkennen, welche Beziehungen relevant sind. Es ist nur sinnvoll, relevante Komplexitäten zu messen. Dafür müssen wir aber zwischen relevanten und irrelevanten Beziehungen unterscheiden können. Komplexität ist somit wie Quantität eine Frage der Definition.

1.3.3 Qualitätsmetrik von Software

Mit der Qualitätsmetrik wollen wir die Güte einer Software beurteilen. Wenn schon die Größe und Komplexität von Software unklar sind, dann ist deren Qualität um ein Vielfaches mehr verschwommen. Was gut und was schlecht ist, hängt von den Sichten des Betrachters ab. Die Klassifizierung von Software in gut und schlecht kann erst in Bezug zu einer definierten Norm stattfinden. Ohne Gebote und Gesetze ist ein Qualitätsurteil weder für menschliches Verhalten noch für Software möglich. Gut ist das, was den Geboten entspricht, und schlecht ist das, was zu ihnen im Widerspruch steht. Aufgrund von Erfahrungen lassen sich einige Schlüsse ziehen wie etwa der, dass unstrukturierter und undokumentierter Code ohne sprechende Namen schwer lesbar und somit auch schwer weiterzuentwickeln ist. Übergroße Source-Module sind bekanntlich schwer handzuhaben. Nicht abgesicherte Klassen sind leicht zu knacken. Mehrfache Verbindungen zwischen Code-Bausteinen erschweren deren Wiederverwendbarkeit. Tief verschachtelte Entscheidungslogik ist fehleranfällig. Diese und viele andere als schädlich empfundene Codierpraktiken können durch Regeln verboten werden.

Verstöße gegen die Regel gelten als qualitätsmindernd. Demnach ist die Qualität des Codes mit der Einhaltung von Regeln eng verknüpft. Ohne ein derartiges Regelwerk kann Qualität nur post factum nachgewiesen werden. Eine Software, in der viele Fehler auftreten oder die unverhältnismäßig langsam ist, gilt als qualitätsarm. Hierfür ist aber der Benutzer auch in der Pflicht zu definieren, was im speziellen Fall zu viele Fehler sind oder was zu langsam ist. Schlechthin kann es ohne Qualitätsnorm keine Qualitätsmessung geben. Qualität ist der Grad, zu dem eine vereinbare Norm eingehalten wird. Sie ist die Distanz zwischen dem Soll- und dem Ist-Zustand. Liegt die Ist-Qualität unter der Soll-Qualität, ist die Qualität zu gering. Liegt sie darüber, ist sie eventuell zu hoch. Zu wenig Qualität verursacht Kosten für den Betrieb und die Erhaltung eines Systems. Zu viel Qualität verursacht Mehrkosten bei der Entwicklung des Systems. In beiden Fällen sind dies Kosten, die der Auftraggeber nicht tragen möchte. Bei Qualität wie bei Quantität kommt es darauf an, genau das zu liefern, was der Kunde bestellt hat, nicht mehr und nicht weniger [DGQ86a].

1.4 Sichten auf die Substanz Software

Ein Softwaresystem besteht aus vielen verschiedenen Typen von Elementen, nicht nur Code, sondern auch Texte, Diagramme, Tabellen und Daten jeglicher Art. Wenn es darum geht, ein solches System zu messen, müssen die Elementtypen genau definiert werden. Die Definition der Messobjekte ist der erste Schritt in einem Messprozess. Es muss für alle Beteiligten klar sein, was gemessen wird [Jone91].

Eine mögliche Kategorisierung der Messobjekte ist nach deren Darstellungsform bzw. Elementtyp wie z. B. Softwarecode, Textdokumente, Diagramme oder Tabellen.

Ein anderes Gliederungsschema ist nach dem Zweck der Elemente. Manche Elemente dienen dazu, die Anforderungen an ein System zu beschreiben. Mit anderen Elementen werden die Konstruktion bzw. Architektur des Systems beschrieben. Eine dritte Kategorie von Elementen sind dann die Codebausteine, die von einer Maschine ausgeführt werden. Eine vierte bilden die Elemente, die dazu dienen, das System zu testen. Eine letzte Kategorie umfasst alle Elemente, die dazu dienen, die Bedienung des Systems zu beschreiben. Diese fünf Kategorien entsprechen den fünf Schichten eines Softwareprodukts:

- Anforderungsdokumentation
- Entwurfsdokumentation
- Code
- Testware
- Nutzungsanleitung (siehe Bild 1.4)

Eine weitere Gliederungsmöglichkeit ist nach dem Gesichtspunkt der Beteiligten. Auf der einen Seite stehen die Benutzer der Software. Aus ihrer Sicht besteht ein System aus Bildschirmoberflächen, Telekommunikationsnachrichten, Papierausdrucken, gespeicherten Daten und Bedienungsanleitungen. Auf der anderen Seite stehen die Entwickler von Software. Aus ihrer Sicht besteht ein System aus Codebausteinen, Dokumenten, Dateien, Datenbanken und Steuerungsprozeduren. Diese beiden Sichten – die fachliche und die technische – sind oft unverträglich, da sie verschiedene Ontologien verwenden. Der Benutzer verwendet die Begriffe aus der Fachwelt, die von der Software abgebildet wird. Der Entwickler verwendet die Begriffe aus der Welt der Maschinen, in welcher die Software implementiert ist.

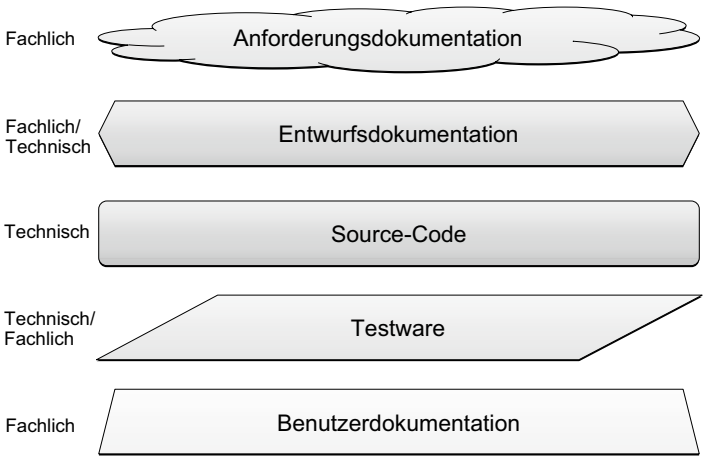

Bild 1.4 Fünf Schichten eines Softwareproduktes

Deshalb gibt es noch eine dritte Sichtweise auf die Software – die Sicht des Integrators, der versucht, die beiden anderen Sichten miteinander zu vereinen. In der IT-Projektpraxis nimmt der Tester die Rolle des Integrators ein und vertritt diese dritte, übergreifende Sicht. Demnach gibt es

- fachliche Beschreibungselemente,
- technische Beschreibungselemente,
- integrative Beschreibungselemente.

Schließlich wird unterschieden zwischen statischen und dynamischen Sichten auf ein Softwaresystem. Eine statische Sicht nimmt die Elemente wahr, die zu einem bestimmten Zeitpunkt existieren, z. B. die Struktur einer Datenbank oder die Zusammenstellung einer Komponente. Diese Elemente können sich zwar verändern, aber zu einem gegebenen Zeitpunkt sind sie statisch invariant. Die statischen Elemente eines Systems bieten sich am besten als Messobjekte an.

Die dynamische Sicht auf die Software nimmt Bewegungen bzw. Zustandsveränderungen wahr. Hier werden Abfolgen von Aktionen und Veränderungsfolgen von Daten beobachtet. Auch diese Bewegungen bzw. Zustandsveränderungen der Systemelemente lassen sich messen, aber dies ist viel schwieriger und verlangt besondere Messinstrumente.

1.5 Objekte der Softwaremessung

Aus den Sichten auf die Software ergeben sich die Objekte der Softwaremessung.

Aus der Sicht der Elementtypen gibt es Folgendes zu messen:

- Natürlichsprachliche Texte
- Diagramme
- Tabellen
- Codestrukturen (siehe Bild 1.5)

Aus der Sicht des Zwecks der Elemente kann Folgendes gemessen werden:

- Anforderungselemente
- Entwurfselemente
- Codeelemente
- Testelemente
- Beschreibungselemente

Aus der Sicht des Systembenutzers lässt sich Folgendes messen:

- Die System/Benutzer-Interaktionen
- Die Systemkommunikation
- Die Systemausgabe
- Die Benutzerdokumentation

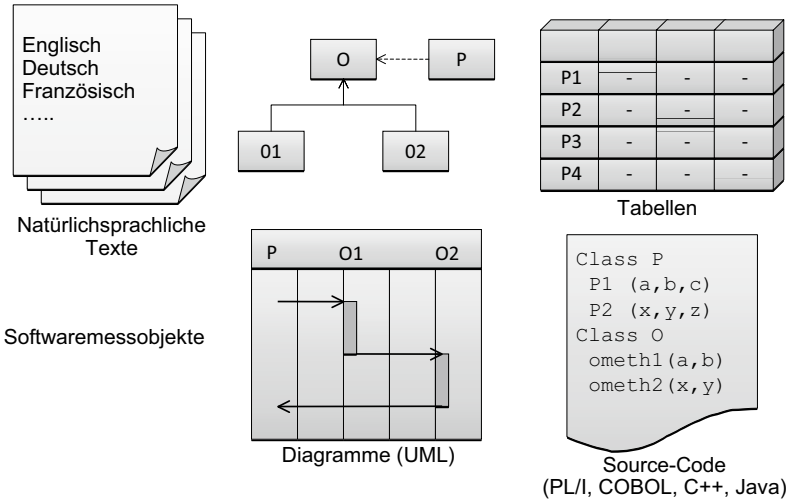

Bild 1.5 Objekte der Softwaremessung

Aus Sicht des Systemintegrators kann Folgendes gemessen werden:

- Die Programme
- Die Daten
- Die Schnittstellen
- Die Systemdokumentation
- Die Fehlermeldungen

Aus statischer Sicht sind alle Elementtypen zu messen, die als Dateien in einem Verzeichnis abgelegt sind. Dazu gehören Testdaten, Tabellen, Grafiken und Diagramme, Source-Code-Texte, Listen und Dateien im Zeichenformat. Aus dynamischer Sicht lässt sich die Ausführung des Codes, die Anzahl an Fehlern, die Veränderung der Daten, die Nutzung der Maschinenressourcen und die Dauer der Computeroperationen messen. Auch Zeiteinheiten wie Ausfallzeiten, Reparaturzeiten und Reaktionszeiten gelten als dynamische Messobjekte. Im Prinzip lässt sich fast alles an einem Softwaresystem messen. Die Frage ist nur immer, ob es sich lohnt, etwas zu messen. Denn Messwerte sind lediglich ein Mittel zum Zweck. Zuerst muss das Ziel der Messung definiert sein. Was will man damit erreichen? Die Kosten schätzen, Qualitätsaussagen treffen oder Mitarbeiter bewerten? Erst wenn diese Ziele klar sind, können aus der großen Anzahl potenzieller Messobjekte die richtigen ausgewählt werden. Es macht wenig Sinn, sämtliche Objekte zu messen, bloß weil sie da sind. Auf diese Weise entstehen die berühmt-berüchtigten Zahlenfriedhöfe. Wer Software messen will, muss eine definierte Messstrategie haben und dieses Konzept verfolgen. Die Messstrategie bestimmt, welche Messobjekte letztendlich herangezogen werden und welche Metriken zur Anwendung kommen.

1.6 Ziele einer Softwaremessung

Im Hinblick auf die Ziele einer Softwaremessung ist es wichtig, zwischen einer einmaligen und einer kontinuierlichen Messung zu unterscheiden. Optimalerweise misst ein Software-Entwicklungsbetrieb bzw. ein Anwenderbetrieb seine Projekte und Produkte ständig, so wie es z. B. im CMMI-Modell vorgesehen ist [ChKS03]. Dazu braucht er eine zuständige Stelle, die dem Qualitätsmanagement untersteht. Diese Stelle vereinbart die Ziele der Softwaremessung mit der IT-Leitung und führt die erforderlichen Messinstrumente ein. Es gibt aber leider nur wenig Anwender im deutschsprachigen Raum, die sich eine solche permanente Messung leisten wollen oder können.

Dies liegt zum einen daran, dass sie den Nutzen nicht erkennen können, andererseits daran, dass ihnen die Kosten zu hoch erscheinen, oder drittens daran, dass selbst wenn sie den Nutzen erkennen und die Kosten tragen können, sie kein qualifiziertes Personal finden. Nur wenig Informatiker haben sich mit Metriken befasst, und die meisten von ihnen sind irgendwo an der Hochschule oder einem Forschungsinstitut. Die Zahl der verfügbaren Metrikspezialisten ist viel zu klein, um den Bedarf zu decken. Demzufolge werden Messungen nur sporadisch durchgeführt.

Die Gründe für einmalige Messungen sind unter anderem:

- Der Anwender steht vor einem betrieblichen Merger und muss entscheiden, welche der doppelten Anwendungssysteme beibehalten werden.
- Der Anwender übernimmt ein Softwaresystem zur Wartung und möchte wissen, worauf er sich einlässt.
- Der Anwender hat vor, seine bestehenden Anwendungen zu migrieren, und möchte wissen, um welchen Umfang es sich handelt.
- Der Anwender hat vor, seine Anwendungen auszulagern, und möchte wissen, was ihre Erhaltung und Weiterentwicklung kosten soll.
- Der Anwender steht vor einer Neuentwicklung und möchte wissen, wie groß und wie komplex die alte Anwendung war.
- Der Anwender hat massive Probleme mit der bestehenden Software und möchte diese genaueren Analysen unterziehen.

Die Ziele einer laufenden Messung unterscheiden sich von denen einer einmaligen Messung. Bei der einmaligen Messung ist das Ziel, den aktuellen Stand eines Systems zu bewerten und daraus Informationen für Entscheidungen zu gewinnen:

- Kosten und Nutzen alternativer Strategien
- Vergleiche verschiedener Systeme
- Vergleiche mit den Industriestandards (Benchmarking)
- Informationen über den Gesundheitsstand eines Softwaresystems

Bei der fortlaufenden Messung geht es darum, Veränderungen in der Produktivität und Termintreue der Projekte sowie in der Größe, der Komplexität und der Qualität der Produkte zu verfolgen.

- Veränderungen der Quantität
- Reduzierung der Komplexität

- Erhöhung der Qualität
- Verbesserung der Schätzgenauigkeit

Da die Ziele so vielfältig sind, müssen sie vor jeder Messung neu definiert werden. Diese Erkenntnis hat Victor Basili und Hans-Dieter Rombach dazu bewogen, die Methode Goal-Question-Metric (GQM) ins Leben zu rufen [BaRo94]. Diese Methode gilt seitdem als Grundlage für jede Softwaremessung (siehe Bild 1.6).

Nach der GQM-Methode werden zunächst die Ziele gesetzt. Zu diesen Zielen werden Fragen gestellt, um sich darüber klar zu werden, wann die Ziele erreicht sind bzw. wie diese zu erreichen sind. Auf die Fragen folgen Maße und Metriken, die uns wissen lassen, wo wir im Verhältnis zu unseren Zielen stehen bzw. wie weit wir noch von ihnen entfernt sind. Das Ziel ist also der Gipfel, den wir besteigen wollen. Die Frage ist, auf welchem Weg man ihn besteigt, und die Metrik ist die Entfernung vom Ausgangspunkt bzw. zum Zielpunkt.

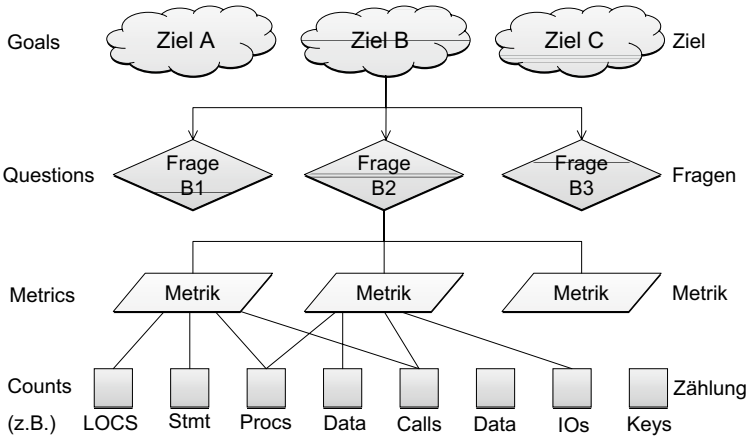

Bild 1.6 Zielorientierte Softwaremessung mit der GQM-Methode

Eigentlich müsste die GQM-Methode um eine weitere Stufe ergänzt werden, und zwar um die der Kennzahlen. Denn eine Metrik ist eine Gleichung mit Kennzahlen als Parameter, die ein bestimmtes, numerisches Ergebnis liefert [Kütz07]. In der gängigen Literatur werden alle Zahlen (auch Summen einzelner Eigenschaften) als Metrik bezeichnet. Dies ist aus Sicht der Metrik eine Verfälschung. Eine Metrik benutzt Zählungen in einer Gleichung, um damit ein Ergebnis zu errechnen. Die Function-Point-Metrik etwa vereint die Zahl der gewichteten Ein- und Ausgaben mit der Zahl der gewichteten Datenbestände und der Zahl der externen Schnittstellen, um daraus Function-Points zu errechnen. Dies ist eine Metrik für die Systemgröße. Die Zahl der Ein- und Ausgaben ist eine Kennzahl bzw. im Englischen ein „count“. Die Anzahl Codezeilen und die Anzahl Anweisungen sind ebenfalls „counts“. In diesem Buch wird zwischen Metriken und Kennzahlen unterschieden. Metriken basieren auf Kennzahlen. Demzufolge wird die GQM-Methode um eine Stufe erweitert:

G = Goal = Ziel

Q = Question = Frage

M = Metric = Metrik

C = Counts = Kennzahl

Als Beispiel dient das Ziel „Die Software soll möglichst fehlerfrei sein“. Die erste Frage, die sich dazu stellt, ist: Was bedeutet möglichst fehlerfrei? Die zweite Frage wäre: Wie fehlerfrei ist die Software jetzt? Das Messziel für die erste Frage könnte eine Restfehlerwahrscheinlichkeit von 0,015 sein. Als Metrik für die zweite Frage könnte die Berechnung der Anzahl der noch nicht entdeckten Fehler auf Basis der bisherigen Fehlerrate in Bezug zur Testüberdeckung dienen.

$$\text{Restfehler} = \text{bisherige Fehler} * \left(\frac{1}{\text{Testüberdeckung}} - 1 \right)$$

wobei Testüberdeckung auf verschiedenen Stufen betrachtet werden kann. Auf der Codestufe könnte sie getestete Logikzweige/alle Logikzweige, auf der Entwurfsstufe getestete Modelelemente/alle Modelelemente und auf der Anforderungsebene getestete Anforderungen/alle Anforderungen sein.

Dies wäre die Metrik. Die Kennzahlen sind:

- Anzahl bisheriger Fehler
- Anzahl getesteter Elemente
- Anzahl aller Elemente

Die GQM-Methode wurde ursprünglich im Jahre 1984 von V. Basili und D. Weis im Rahmen einer Softwaremessung beim NASA Goddard Space Flight Center entwickelt [BaWe84]. Sie wurde in Europa erst Anfang der 90er Jahre bei der Schlumberger Petroleum AG in den Niederlanden eingesetzt, um die dortige Prozessverbesserung zu messen. Im Jahre 1999 brachte R. van Solingen und E. Berghout ein Buch mit dem Titel „The Goal/Question/Metric Method“ heraus [SoBe99]. In diesem Buch beschreiben die Autoren ihre Erfahrungen mit der Methode in mehreren europäischen Unternehmen. Trotz der üblichen Probleme mit Ziel- und Begriffsdefinitionen konnten damit einige Prozesse und Produkte gemessen und bewertet werden. Welche Maßnahmen auf die Messungen folgten, bleibt unbeschrieben. Jedenfalls konnten die Anwender erkennen, wo sie sich im Verhältnis zu ihren Zielen befanden. Auch der Autor Sneed hat mit der Methode gute Erfahrungen gemacht, vor allem in Bezug auf die Optimierung der Wartungsprozesse im Anwendungsbetrieb. Ausschlaggebend für den Erfolg der Methode ist die Definition messbarer Ziele wie z. B. die Reduktion der Kundenfehlermeldungen um 25 %. Auf welchem Weg das Ziel zu erreichen ist, ist eine andere Frage, die wiederum von anderen Messungen abhängt.

Die Wahl des Weges zum Ziel wird von der Korrelation diverser Metriken bestimmt wie etwa der Korrelation zwischen Codequalität oder Architekturqualität und Fehlerrate. Ein Großteil der Metrikforschung ist darauf ausgerichtet, solche Korrelationen zwischen Ziel und Mittel herauszustellen. Erst wenn wir wissen, was einen Zustand verursacht, können wir daran gehen, die Ursachen des Zustands zu verändern, sei es die Codequalität, die Prozessreife, die Werkzeugausstattung oder die Qualifikation der Mitarbeiter.

Ein Ziel der Metrik ist, derartige Zusammenhänge aufzudecken, damit wir die betroffenen Zustände ändern können. Ein weiteres Ziel ist, die Zustände zu verfolgen, wo sie im Verhältnis zum Soll stehen. Ein drittes Ziel ist es zu kalkulieren, welche Mittel man braucht, um die Zustände zu verändern. Hier ist ein Projekt als Zustandsänderung bzw. als Zustandsübergang zu betrachten.

1.7 Zur Gliederung dieses Buches

In Anlehnung an die Dimensionen und Schichten eines Softwareproduktes sowie an die Ziele eines Softwareprozesses ist dieses Buch in drei Teile mit elf Kapiteln gegliedert (siehe Bild 1.7)

	Entwicklung				Wartung/Evolution	
Messobjekte	**Anforderungs-spezifikation**	**Systementwurf**	**Source-Code**	**Testware**	**Entwicklungs-maße**	**Wartungs-maße**
Quantität	Geschäftsprozesse Geschäftsobjekte Geschäftsregeln Anwendungsfälle	Klassen/Module Methoden/Procs Schnittstellen Daten	Codezeilen Anweisungen Bedingungen Referenzen	Testobjekte Testfälle Testläufe Fehlermeldungen	Func-Points Obj-Points UC-Points	LOCS Anweisungen Module
Komplexität	Strukturiert Textuell Fachlich	Entitäten Beziehungen Interaktionen	Ablaufe Zugriffe Datennutzung	Zustandsdichte Pfadanzahl Schnittstellen-breite	Objekte Relationen Ereignisse	Koordinaten Zweige Pfade
Qualität	Konsistenz Vollständigkeit Exaktheit	Kohäsion Kopplung Ausbaufähigkeit	Modularität Konvertierbarkeit Konformität	Fehlerdichte Testüberdeckung Fehlerfindung	Vollständig Konsistent Plausibel	Koordinaten Zweige Pfade
Produktivität	Testzeilen Zeilen pro PT	Diagramme pro PT	Codezeilen/ Anweisungen pro PT	Testfälle pro PT	FPs OPs pro PT TCs	LOCs Stmts pro PT TCs

Bild 1.7 Dreifache Gliederung des Buches

Der erste Teil befasst sich mit den Dimensionen der Software bzw. mit deren Größe, Komplexität und Qualität. Das zweite Kapitel beschreibt die Maße für die Größe eines Softwareprodukts, Maße wie Anforderungen, Dokumentationsseiten, Modeltypen, Codezeilen, Anweisungen, Object-Points, Function-Points und Testfälle. Das dritte Kapitel geht auf die Komplexitätsmessung ein und behandelt solche Komplexitätsmetriken wie Graphenkomplexität, Verschachtelungstiefe, Kopplungsgrad und Datennutzungsdichte. Das vierte Kapitel setzt sich mit dem Thema Qualitätsmessung auseinander. Dabei geht es um Maßstäbe für Qualitätsmerkmale wie Zuverlässigkeit, Korrektheit, Sicherheit und Wiederverwendbarkeit. Hier kommt die GQM-Methode zur Geltung.

Der zweite Teil befasst sich mit den einzelnen Softwareschichten und wie sie zu messen sind. Die hier behandelten Softwareschichten sind:

- Die Anforderungsdokumentation
- Der Systementwurf
- Der Code
- Die Testware

Kapitel 5 behandelt die Messung natursprachlicher Anforderungsdokumente. Kapitel 6 schlägt eine Metrik für den Systementwurf im Allgemeinen und im Speziellen für UML vor. Das Kapitel 7 beschäftigt sich mit der Messung und Bewertung sowohl von prozeduralem als auch objektorientiertem Code. Kapitel 8 ist dem Thema Testmessung gewidmet. Darin werden diverse Testmetriken vorgestellt, die nicht nur das dynamische Verhalten des Systems, sondern auch den statischen Zustand der Testware messen. Für alle vier Schichten werden die drei Dimensionen Quantität, Komplexität und Qualität behandelt.

Im dritten und letzten Teil des Buches geht es um die Messung der Softwareprozesse. Kapitel 9 geht auf die Messung der Produktivität in Entwicklungsprojekten ein. Hier werden diverse Ansätze zur Ermittlung der Produktivität zwecks Planung und Steuerung von Entwicklungsprojekten vorgestellt. Kapitel 10 befasst sich mit dem schwierigen Thema „Wartungsmessung“. Es geht dabei sowohl um die Wartbarkeit der Softwareprodukte als auch um die Messung der Wartungsproduktivität. Kapitel 11 schildert den Messprozess, den die Autoren bereits in zahlreichen Messprojekten verwendet haben, und die Werkzeuge, die sie eingesetzt haben, um die Messergebnisse zu erzeugen. Hier wird Softwaremessung als ein – im Sinne des CMMI – definierter und wiederholbarer Prozess dargestellt.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z