2 Grundbegriffe

Bevor in diesem Kapitel die beiden Wortbestandteile des Buchtitels, »Software« und »Engineering«, gründlich betrachtet werden, geht es zunächst um den Begriff der Kosten. Denn die Kosten sind eines der wichtigsten, wenn nicht das wichtigste Merkmal jeder Software. Anschließend werden einige weitere Grundbegriffe eingeführt.

2.1 Kosten

Nach »Software« ist »Kosten« einer der Begriffe, die in diesem Buch am häufigsten vorkommen. Es ist darum notwendig, genau zu sagen, was wir unter Kosten verstehen; andernfalls wären Missverständnisse sehr wahrscheinlich. Wenn wir von Kosten sprechen, schließen wir alle Nachteile ein, die eine Lösung hat, seien sie in Geldwert auszudrücken oder nicht.

Analog ist beim *Nutzen* jede Art von Vorteil eingeschlossen. Wenn man sich auf eine Betrachtung der Kosten beschränken will, kann man den Nutzen als negativen Kostenbeitrag berücksichtigen. Dann kann das übergeordnete Ziel als *Minimierung der Gesamtkosten* formuliert werden. Das bedeutet also: Möglichst hohe Differenz zwischen Nutzen und Kosten.

Eine praktische Anwendung macht den Ansatz anschaulich: Wir betrachten die Situation, dass ein Mensch zu einem bestimmten Zeitpunkt an einem anderen Ort sein möchte oder sein muss. Wenn er die Entscheidung, auf welche Weise er dorthin kommt, rational treffen will, muss er die Kosten der Reise, die bei verschiedenen Varianten der Beförderung entstehen, vergleichen und sich für die Reise mit den geringsten Kosten entscheiden.

Dabei fallen zunächst die unmittelbaren Kosten ins Auge: Eine Bahnfahrkarte oder ein Flugticket haben einen Preis, der sich feststellen lässt. Auch zusätzliche Kosten für den Nahverkehr, für Versicherungen usw. lassen sich relativ gut erfassen.

Andere Ausgaben treten früher oder später auf und sind darum schwieriger zu kalkulieren. Wenn man die Reise mit dem eigenen Auto macht, ist sie auf den

ersten Blick kostenlos. Das Tanken lässt sich nicht sehr lange aufschieben, dagegen liegt der Ersatz der Reifen, des Motors und irgendwann des ganzen Autos für die meisten Menschen außerhalb des Blickfeldes. Aber natürlich sind auch das Kosten, die durch eine Fahrt anteilig erzeugt werden.

Kaum quantifizieren lassen sich all die Kosten, die durch Unbequemlichkeiten, Risiken, Langeweile oder Ängste entstehen. Wer sich vor dem Fliegen fürchtet, wer das Autofahren bei Dunkelheit anstrengend findet oder in der Bahn unter dem Geschwätz der anderen Reisenden leidet, bezieht entsprechende Kosten in seine Betrachtungen ein, natürlich in aller Regel unbewusst und irrational. Dass es letztlich Kosten sind, erkennt man daran, dass sich kaum ein Mensch solchen Einflüssen absolut unterwirft: Sind andere Vorteile groß genug, so werden die Einwände meist unterdrückt. Wer zufällig eine Gratisfahrt in Anspruch nehmen kann, setzt sich doch in die Bahn, und ein Australier, der den Nobelpreis in Empfang nehmen darf, überwindet wahrscheinlich die Flugangst. Wir wägen ab, d. h., wir quantifizieren, wenn auch nicht rational.

In vielen Fällen entsteht durch die eine oder andere Lösung auch zusätzlicher Nutzen. Wer mit dem Auto fährt, kann bei dieser Gelegenheit ein Möbelstück transportieren. Bei der Bahnfahrt ist das Umsteigen mühsam, es sei denn, man hatte schon lange vor, durch die Stadt, in der sich die beiden Züge treffen, zu bummeln oder dort ein Museum zu besuchen. Wir können solchen Nebennutzen einer Lösung wie negative Kosten betrachten.

Noch schwieriger wird die Betrachtung, wenn man auch die Kosten einbezieht, die anderen Bewohnern der Welt entstehen. Wer mit dem Auto fährt, verbrennt knappe Ressourcen, erzeugt Lärm und erhöht die Risiken, denen andere Menschen ausgesetzt sind. Wer billige Eier kauft, verurteilt damit Legehennen zur Tortur. Hier wird die Kostenbetrachtung offensichtlich zu einer Frage der Weltanschauung (im Wortsinn). Wollen wir solche Kosten berücksichtigen? Und wie sind sie gegeneinander abzuwägen? Dürfen wir, um das Leben eines Kranken zu verlängern, die Chancen der nächsten Generation beschädigen? Natürlich können und wollen wir solche Fragen nicht beantworten. Wir weisen nur darauf hin, dass wir, wenn wir von den Kosten einer Lösung sprechen, *sämtliche* direkten und indirekten Kosten meinen.

Sowohl zum Nutzen als auch zu den Kosten gibt es viele Beiträge, auf die die Entscheidungen der Entwickler keinen Einfluss haben. In vielen Fällen haben wir beispielsweise keine Option, auf eine Lösung ganz zu verzichten. Ebenso wenig werden wir wegen einer Software-Entscheidung unser Gebäude erweitern oder verkaufen. Also brauchen wir weder den zentralen Nutzen noch die Gebäudekosten zu berücksichtigen. Das Gleiche gilt auch für viele andere Beiträge: Wenn eine Wahl aus n Varianten zu treffen ist, können die in allen n Varianten gleichen Summanden ignoriert werden.

Wer die Reduktion aller Wertungen auf eine Kosten-Skala als zynisch oder menschenverachtend einschätzt, macht sich nicht klar, dass es dazu keine Alter-

native gibt. Natürlich wird man den Wert von Leben und Gesundheit sehr hoch einstufen. Aber man wägt auch diesen Wert gegen andere Werte ab. Geradezu klassische Beispiele sind die politischen Entscheidungen über Geschwindigkeitsbeschränkungen, über die medizinische Versorgung und über die Prävention von Gewalttaten. Am Ende muss jeder Mensch Entscheidungen treffen, also beispielsweise zu einer bestimmten Geschwindigkeitsbeschränkung ja oder nein sagen. Damit sind wir bei einer binären Skala angelangt. Einfacher geht es nicht mehr. Natürlich kann man bindende Vorgaben (»K.-o.-Kriterien«) einführen, indem man jede Abweichung von der Vorgabe mit unendlich hohen Kosten verknüpft.

Dieser Abschnitt kann wie folgt zusammengefasst werden: Wir sprechen in diesem Buch von Kosten und decken damit alle Nachteile ab, durch das Konzept der negativen Kosten auch alle Vorteile. Wir erheben nicht den Anspruch, eine konkrete Abbildung aller Kosten auf eine Geld-Skala angeben zu können. Wir weisen aber darauf hin, dass wir am Ende vergleichbare Bewertungen brauchen, um irgendeine Entscheidung zu treffen, unabhängig davon, ob es dabei um Software oder um etwas anderes geht. Die Behauptung, dass manche Dinge, zwischen denen wir eine Wahl zu treffen haben, einfach nicht vergleichbar sind, ist ein Selbstbetrug.

2.2 Engineering und Ingenieur

Bei zusammengesetzten Wörtern, auch beim »Software Engineering«, bestimmt das letzte die Grundbedeutung. Wir beginnen darum mit dem *Engineering* und mit dem Ingenieur, der das Engineering beherrscht und repräsentiert.

2.2.1 Der Ingenieur als Leitbild

Aus etymologischer Sicht ist der Ingenieur als Leitbild kaum geeignet, eher als ein Beleg dafür, dass der Krieg tatsächlich der Vater zumindest vieler Dinge ist (Heraklit). Denn der Ingenieur war zunächst der Baufachmann für Festungen (in moderner Terminologie ein Bauingenieur), dann allgemein für technische Einrichtungen, die mit dem Krieg zu tun hatten, also für Geschütze usw.

Im 18. und 19. Jahrhundert erhielt das Wort die heute übliche Bedeutung, und in dieser Zeit entstand das Bild des Ingenieurs, der alle Probleme löst und kraft seines Verstandes Dinge vollbringt, die für simple Gemüter wie Zauberei erscheinen. Der schreibende Ingenieur Max Eyth (1836–1906), Verfasser des halb autobiografischen, halb fiktiven Buches »Hinter Pflug und Schraubstock«, hat das Bild in Deutschland wesentlich geprägt.

Im Vordergrund stehen die folgenden Merkmale:

- *Rationalität* und Anwendung aller wissenschaftlichen Erkenntnisse als Grundprinzip

  Der Ingenieur ist der natürliche Feind des Aberglaubens, der Verehrer der Zahlen und Formeln. Insofern ist er ein Kind der Aufklärung, der alles mechanistisch zu erklären sucht und ablehnt, was so nicht erklärt werden kann.

- *Problemlösen* als eigentliche Aufgabe
  Verbindet die Rationalität den Ingenieur mit jedem Naturwissenschaftler, so unterscheidet ihn dieses Merkmal deutlich: Der Ingenieur strebt letztlich nicht danach, die Welt zu erklären oder Probleme zu formulieren, zu vermeiden, ihre tieferen Ursachen zu erkennen und zu bekämpfen. Er will Probleme, echte oder vermeintliche, lösen, indem er die Welt auf technischem Wege verändert, also Distanzen überbrückt, Bauwerke errichtet, Energie verfügbar macht oder auch die größtmögliche Zerstörung anrichtet.

  Diesen Charakterzug verherrlicht Heinrich Seidel (1842–1906) in seinem Gedicht, dessen Anfang noch heute gern zitiert wird. Die erste der vier Strophen lautet:

  Dem Ingenieur ist nichts zu schwere –
  Er lacht und spricht: Wenn dieses nicht, so geht doch das!
  Er überbrückt die Flüsse und die Meere,
  Die Berge unverfroren zu durchbohren ist ihm Spaß.
  Er türmt die Bogen in die Luft,
  Er wühlt als Maulwurf in der Gruft,
  Kein Hindernis ist ihm zu groß –
  Er geht drauf los!

  Man beachte, dass hier die Frage nach dem Sinn des Tuns nicht gestellt wird, Problemlösen ist Selbstzweck.

  Auch Physiker arbeiten konstruktiv, sie bauen riesige Apparate, um ihre Versuche auszuführen. Aber sie bauen, um zu forschen. Die Ingenieure forschen, um zu bauen.

- *Kostensenkung* als wichtigste Randbedingung
  Eine Lösung ist gut, wenn sie das Problem zu möglichst niedrigen Kosten löst. Daher ist ein guter Ingenieur kein Perfektionist: Die perfekte Lösung hat nur selten ein günstiges Verhältnis zwischen Preis und Effekt.

- *Universeller Anspruch*, der nicht vor fachlichen Grenzen Halt macht
  Ein Ingenieur streicht nicht die Segel, weil das Problem die Grenzen seines Fachs überschreitet. Er ist in diesem Sinne Universalist.

Es ist leicht zu erkennen, dass es zwischen diesen Merkmalen zu Konflikten kommt. Denn in jeder konkreten Situation muss sich der Ingenieur fragen, ob er durch weitere Klärung Zeit verlieren oder mit dem Risiko eines Fehlers vorwärts-

gehen soll, ob die Kosten eine bessere Lösung erlauben usw. Auch im Software Engineering wird dieses Dilemma deutlich: Einerseits müssen wir dringend danach streben, ein wesentlich besseres Verständnis zu entwickeln, andererseits müssen wir auch mit dem ungenügenden heutigen Verständnis Beiträge zu einer Verbesserung der Situation leisten.

2.2.2 Merkmale der alten Ingenieurdisziplinen

Die Definition und Anwendung von *Methoden* zählt ebenso wie die Bereitstellung und Verwendung von *Werkzeugen* zu den Ansätzen, die auf allen Gebieten der menschlichen Kultur zu höheren Leistungen geführt haben. Den Ingenieuren können wir darüber hinaus folgende Prinzipien »abgucken«:

- *Kostendenken* als Grundlage von Bewertungen
  In der Technik sind Argumente der Art »Ich finde, dass ...« lächerlich. Wir müssen uns an objektiven Kriterien orientieren. Wo immer möglich, sollten das Kosten sein. Was insgesamt billiger ist, ist besser. Das verbietet eine nur vordergründige Sparsamkeit: An vielen Stellen sollte man zur Senkung der Gesamtkosten den Aufwand erhöhen.
- *Praktischer Erfolg* als einzige zulässige Beweisführung
  Der Ingenieur glaubt an eine Lösung erst, wenn er sie sieht. Jede Art von argumentativen Beweisen hat höchstens vorläufigen Nutzen; erst wenn sich die Konstruktion bewährt hat, ist sie anerkannt. Da die Differenzierung verschiedener Lösungen oft nur mit sehr genauen Beobachtungen möglich ist, schließt dieses Prinzip die Technik der Messungen ein. Nur was gezählt oder gemessen wurde, ist ein akzeptables Argument. Änderungen der Bearbeitungsprozesse sind nur sinnvoll, wenn sie zu nachweisbaren Verbesserungen führen. Damit sind Messungen die Voraussetzung.
- *Qualitätsbewusstsein* als »Grundfarbe« des Denkens
  In der Praxis ist es oft nicht klar, wie Nutzen und Kosten von einer Entscheidung beeinflusst werden. In diesen Fällen strebt der Ingenieur nach möglichst hoher Qualität. Diese Haltung kommt aus der handwerklichen Tradition, in der die Ingenieure stehen. Sie führt in aller Regel letztlich auch zu einer Kostensenkung.
- Die Einführung und Beachtung von *Normen*
  Normen sind das wichtigste Merkmal einer Ingenieurdisziplin. Sie schaffen auf allen Ebenen einheitliche Schnittstellen. Das beginnt mit dem Passstift, der zwei Maschinenteile verbindet (die erste Norm in Deutschland, 1918), und reicht bis hin zu den Begriffen, deren Normung die Kommunikation wesentlich erleichtert.

  Merke: Eine Norm erweist sich dadurch als Norm, dass sie in jedem Falle beachtet wird, selbst wenn es unpraktisch erscheint.

- Denken in *Baugruppen*
  Durch Normen entsteht die Möglichkeit, Baugruppen zu verwenden, also die Lösung der Teilprobleme auszuklammern. Die gigantischen Leistungen der Technik wären ohne universelle Bausteine (z. B. Schrauben, Geräte mit Netzstecker, ICs usw.) undenkbar.

2.3 Software

Das Wort Hardware ist seit dem 16. Jahrhundert nachgewiesen; es bezeichnet wie das deutsche Wort »Eisenwaren« alle Gegenstände, die aus Metall gemacht sind, beispielsweise Nägel, Bleche, Werkzeuge, aber auch generell alle materiellen Bestandteile eines Systems. Als die Entwicklung der modernen Rechenmaschinen begann, wurden ihre materiellen Teile als Hardware bezeichnet. John W. Tukey[1] (1915–2000), der gut zehn Jahre zuvor bereits das »bit« als Einheit der Informationsmenge in die Diskussion gebracht hatte, kam 1957 auf die Idee, dem alten Wort »Hardware« das Kunstwort »Software« gegenüberzustellen.

2.3.1 Was ist Software?

Durch die Entstehungsgeschichte ist klar: Software ist, was nicht Hardware ist. Aber zur Definition reicht das nicht aus. Bis heute ist die Bedeutung umstritten. Wir halten uns an die Definition der IEEE Computer Society:

> **software** — Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.
>
> See also: application software; support software; system software
>
> Contrast with: hardware.
>
> IEEE Std 610.12 (1990)

Software umfasst danach also *Programme, Abläufe, Regeln, auch Dokumentation und Daten, die mit dem Betrieb eines Rechnersystems zu tun haben*.

Generell sollte Software als technisches Produkt betrachtet werden, das wie andere Produkte von Ingenieuren systematisch entwickelt werden kann und durch feststellbare Eigenschaften (Funktionalität, Qualität) gekennzeichnet ist. Wer die Besonderheit der Software betont, beansprucht ein unbegründetes Privileg. Software Engineering zielt darauf ab, die Sonderrolle der Software so weit wie möglich zu beseitigen!

1. Viele Informatiker kennen den Namen dieses vielseitigen Forschers durch den FFT-Algorithmus, den er 1965 zusammen mit Cooley publiziert hat.

2.3.2 Spezielle Eigenschaften der Software

Nicht alle Konzepte der Ingenieurwissenschaften lassen sich unverändert auf die Software anwenden, denn Software weist einige besondere und – wenigstens in dieser Kombination – einmalige Merkmale auf. (Dass einige der Aussagen auch auf integrierte Schaltungen zutreffen, überrascht nicht, denn diese werden heute weitestgehend wie Software entwickelt.)

Software ist immateriell.

Jede Software ist auf irgendwelchen Trägern repräsentiert, also auf Papier, auf Magnetplatten oder anderen Medien. In aller Regel interessiert uns dieser Träger aber nicht, sondern die Information. Unsere (individuelle wie stammesgeschichtliche) Erfahrung wurde aber in einer Welt akkumuliert, in der fast alle Effekte materieller Natur sind, von der Herdplatte, an der man sich die Finger verbrennen kann, bis zum aggressiven Hund, dessen Biss wir fürchten. Alles, was wir als natürlich empfinden, ist an materielle Eigenschaften geknüpft. Das hat eine Reihe von Konsequenzen:

- An Software ist *nichts natürlich*.
  Erfahrungen aus der natürlichen Welt sind nicht übertragbar, werden aber dennoch ständig übertragen, siehe z. B. unten die Verwendung des Wortes »Wartung«.
- Software wird nicht gefertigt, sondern *nur entwickelt*.
  Wir haben praktisch keine Fertigungskosten, wenn man als Fertigung die Reproduktion des ersten Musters versteht.
- Kopie und Original sind *völlig gleich*.
  Kopie und Original sind nicht zu unterscheiden. Darum müssen wir bei der Verwaltung von Software große Anstrengungen unternehmen, um nicht versehentlich Varianten zu erzeugen, die Konsistenzprobleme verursachen.
- Software *verschleißt nicht*.
  Die »Wartung« stellt nicht den alten Zustand wieder her, sondern einen neuen.
- *Software-Fehler* entstehen nicht durch Abnutzung.
  Sie sind von Beginn an (oder durch Änderungen) vorhanden. Darum kündigen sie sich nicht wie viele Defekte, die durch den Verschleiß materieller Mechanismen entstehen, langsam an.
- *Wiederverwendung* ist bei Software extrem lukrativ, wenn der Aufwand für Anpassungen relativ gering ist.
- Die Software-Leute *unterschätzen* meist das Problem – oder sie *überschätzen* sich selbst!

Natürliche Lokalität gibt es bei Software nicht.

Wenn man mit Distanz die Vorstellung verbindet, dass sich zwei Komponenten eines Systems umso weniger gefährden, je größer die Distanz ist, dann gibt es in einem Programm keine Distanz. Lokalität entsteht nur, wo sie gezielt, z. B. mittels spezieller Konstrukte der Programmiersprachen, herbeigeführt wird.

Ein Programm realisiert keine stetige Funktion.

Diese Eigenschaft »erbt« die Software von der Hardware, dem Digitalrechner, auf dem sie ausgeführt wird. Der Rechner kann keine stetigen Funktionen realisieren, sondern nur endlich viele verschiedene Werte auf endlich viele Werte abbilden. Damit ist es grundsätzlich – und, wie die Erfahrung zeigt, leider auch praktisch – möglich, dass einzelne Werte völlig »aus der Reihe tanzen«, also gänzlich andere Resultate liefern als die benachbarten Werte. Die Änderung eines einzigen Bits, sei es im Programmcode oder in den Daten, mit denen das Programm arbeitet, kann beliebige Folgen haben, also die Ausgabe des Programms völlig verändern; es gibt keine uns natürlich erscheinende Gesetzmäßigkeit, dass kleine Ursachen auch nur kleine Wirkungen haben.

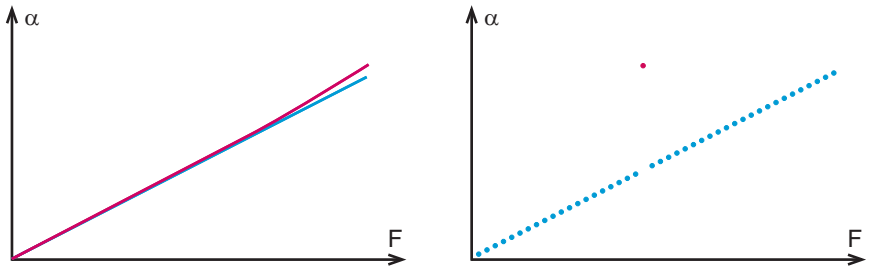

Abb. 2–1 *Das Hook'sche Gesetz, gemessen und digital modelliert*

Ein mechanisches Teil kann diese Eigenschaft nicht haben. Darum kann es durch einige Versuche geprüft werden. Software dagegen müsste für *alle* möglichen Werte geprüft werden, was wegen der großen Zahl unmöglich ist, zumal es nicht ein einzelner Wert sein muss, der das Verhalten bestimmt, es kann auch eine bestimmte Kombination verschiedener Werte sein. Darum ist die Funktionalität eines Programms durch Test nicht prüfbar, ein Test zeigt eventuell die Anwesenheit eines Fehlers, aber niemals die Abwesenheit von Fehlern (Dijkstra, 1970).

Abbildung 2–1 zeigt das Verhalten einer Stahlfeder nach dem Hook'schen Gesetz. Links ist das Verhalten einer realen Feder zu sehen, das vom idealen (linearen) Verhalten abweicht. Rechts sieht man eine fehlerhafte digitale Modellierung.

Software-Systeme sind sehr komplex.

Komplexität ist ein schwieriger Begriff (vgl. Abschnitt 14.4.3); intuitiv bezeichnen die meisten Menschen damit, wie viel Information einer Sache unvermeidbar zugeordnet ist. Was nur kompliziert dargestellt ist, kann auch einfacher dargestellt werden, was komplex ist, verweigert sich einer Vereinfachung.

Schon mittelgroße Software-Systeme bestehen aus verschiedenen Dokumenten, darunter der Code, der einige tausend Zeilen lang ist; bei großen Systemen sind es Millionen. Betrachtet man diese Informationen als die präzise Beschreibung des Systems, dann erkennt man, dass die Beschreibung umfangreicher ist als bei den meisten anderen Systemen, die offensichtlich sehr komplex sind: Gesetzeswerke, Verwaltungen, komplizierte Maschinen, Schiffe und Flugzeuge.

Nicht selten ist die Software-Lösung komplizierter, als der Sache angemessen ist; dann sollte sie vereinfacht werden. Trotzdem wird es nicht gelingen, den Umfang drastisch zu vermindern.

Software gehört also zu den komplexesten Artefakten, die Menschen bislang geschaffen haben. Der Aufwand, um Software zu entwickeln, ist darum auch unvermeidlich sehr hoch.

Software-Systeme müssen autonom funktionieren.

Andere komplexe Artefakte, z. B. die im vorigen Punkt genannten, funktionieren nur, wenn und solange Menschen immer wieder eingreifen und Mängel, Defekte und Widersprüche beseitigen oder kompensieren. Gesetze werden laufend ausgelegt und korrigiert, Maschinen werden überwacht und gewartet. Nur relativ einfache Mechanismen (z. B. die Mechanik der Plattenspeicher in modernen Rechnern) funktionieren ohne menschliche Eingriffe über längere Zeit fehlerlos.

Auch bei Software treten Mängel auf, beispielsweise in Form sogenannter Systemabstürze. In der Regel liegen aber Milliarden von korrekt ausgeführten Operationen zwischen solchen Ereignissen. Die Software funktioniert also trotz ihrer hohen Komplexität über weite Strecken autonom.

Die wenigen »Werkstoffe« der Software, die Sprachen, implizieren keine sinnvolle Strukturierung.

Software ist immateriell und besteht darum auch aus immateriellen »Materialien«, nämlich aus den Sprachen und Notationen, in denen sie formuliert ist (vgl. Abschnitt 2.5.2). Mindestens zwei Sprachen werden regelmäßig eingesetzt, eine natürliche Sprache wie Deutsch oder Englisch und eine Programmiersprache. Zusätzlich können Notationen wie UML (z. B. Klassen- und Interaktionsdiagramme) verwendet werden.

Fast alle größeren Artefakte bestehen aus vielen verschiedenen Materialien; beispielsweise besteht ein Fahrrad aus Stahl, Aluminium, Kupfer, Gummi, Kunststoffen, Leder, Glas, Lacken und vielen anderen Werkstoffen. Diese werden zunächst separat hergestellt und bearbeitet, dann kombiniert. Aus Kupferdraht und Kunststoff wird eine Leitung, aus Stahlblech, Kupferleitern und Glas eine Lampe. Der Rahmen wird geschweißt oder gelötet, dann mit Lack überzogen. Die Werkstoffe geben also die Schritte vor, in denen das Produkt aufgebaut wird, und sie bestimmen in der Regel auch die Abgrenzung der Komponenten, die später repariert oder ausgetauscht werden können: Reifen, Schlauch und Felge sind voneinander unabhängig, sie können, wenn ein Teil defekt ist, einzeln erneuert werden. Werkstoffvielfalt impliziert Modularität.

Glocken aus Bronzeguss und billige Gartenstühle aus Kunststoff unterscheiden sich von den meisten anderen Produkten dadurch, dass sie beide nicht modular aufgebaut sind, sondern nur aus einem Material und nur aus einem Stück bestehen, die Glocke, weil sie sonst nicht klingt, der Stuhl, weil es so am billigsten ist. Darum sind weder Glocken noch Plastiksessel reparierbar, eine Glocke mit einem Riss ist verloren, ein zerbrochener Plastiksessel ist Müll.

Die zwei oder drei Werkstoffe der Software schaffen offensichtlich Modularität, aber leider keine sinnvolle. Die Trennung zwischen dem Code auf der einen und den übrigen Dokumenten auf der anderen Seite ist schuld an den praktischen Schwierigkeiten, die gesamte Dokumentation aktuell zu halten, während der Code den Änderungen der realen Welt nachgeführt wird. Dagegen erlauben es die Programmiersprachen, beliebig komplexe Programme zu schreiben, unstrukturiert und darum unverständlich für jeden, der sie bearbeiten soll.

Leider sorgen also die Werkstoffe nicht für sinnvolle Strukturen, das müssen die Software-Entwickler tun.

Software spiegelt (in vielen Fällen) die Realität.

Die Software verbindet Hardware und Anwendungsbereich. Gab es früher noch viele spezielle Schaltungen und Mechanismen, die direkt und ohne Software Aufgaben der Steuerung und Regelung übernahmen, werden heute praktisch überall Prozessoren mit Software eingesetzt, denn Software ist flexibel und erlaubt die (relativ) einfache Anpassung des Systems an fast beliebige Anforderungen.

Damit ist eine bestimmte Aufgabenverteilung zwischen Hard- und Software zur Regel geworden: Die Hardware garantiert mit ihrer phantastischen Leistung Quantität, also Tempo und Speicherkapazität, die Software sorgt für Qualität, also für die Abbildung der Anforderungen, und seien sie noch so bizarr, auf die strukturell primitive Hardware. Das bedeutet: Die Hardware ist von der Realität der Anwendungen weitgehend entkoppelt, ihre Entwicklung ist von der Anwendung höchstens durch Kostenargumente und Leistungsforderungen beeinflusst. Die Software ist dagegen eng an die Anwendung gebunden, sie muss ihr auch fol-

gen, wenn sie sich ändert. Tatsächlich kompensiert heute die Software auch Mängel der Hardware; wenn sich Fehler der Prozessoren zeigen, werden sie, wenn irgend möglich, durch Erweiterungen der Software »geheilt«.

> *... This complexity [of the software] is compounded by the necessity to conform to an external environment that is arbitrary, unadaptable, and ever-changing.*
>
> F.P. Brooks (1987)

2.4 Arbeiten, die an Software ausgeführt werden

Dieser Abschnitt enthält eine kurze Übersicht der Arbeiten, die in der Regel auszuführen sind, um Software zu entwickeln oder zu verändern. Die einzelnen Tätigkeiten werden in den weiteren Kapiteln des Buches näher beschrieben; dabei sind allerdings Vorwärtsreferenzen, also Verweise auf spätere Kapitel, unvermeidlich. Aus diesem Grund erscheint es sinnvoll, zunächst alle Arbeiten kurz zu charakterisieren. Die Frage, ob die Arbeiten in derselben Reihenfolge ausgeführt werden können oder gar müssen, wie sie hier beschrieben sind, wird bei den Vorgehensmodellen (siehe Kap. 9) behandelt.

- *Analyse*
  Software-Entwicklung ist kein Selbstzweck. Software wird entwickelt, um ein Problem zu lösen. Ziel der Analyse ist, das bestehende Problem zu durchdringen und zu verstehen. Dabei wird auch geklärt, inwieweit Software eingesetzt werden kann, um das Problem zu lösen, und welche Aufgaben von der zu entwickelnden Software übernommen werden sollen.
- *Spezifikation der Anforderungen*
  Die in der Analyse festgestellten Anforderungen müssen geordnet, dokumentiert, geprüft, ergänzt und korrigiert werden. Da sich viele andere Arbeiten auf die Spezifikation stützen, ist dieser Schritt besonders wichtig.
- *Architekturentwurf, Spezifikation der Module*
  Software-Systeme werden nicht monolithisch gebaut, sondern bestehen aus Modulen oder Komponenten, die miteinander die Gesamtfunktionalität des Systems bieten. Die Gesamtstruktur wird durch die Software-Architektur beschrieben. Die Schnittstellen der Komponenten werden so präzise wie möglich festgelegt, damit die Komponenten parallel entwickelt und am Ende problemlos integriert werden können.
- *Codierung und Modultest*
  Die einzelnen Module werden in der gewählten Programmiersprache codiert. Sind sie fertiggestellt, dann können sie u. a. auf der Basis ihrer Spezifikation getestet werden. Die dabei entdeckten Fehler werden anschließend korrigiert. Die Module stehen dann für die Integration zum Gesamtsystem zur Verfügung.

- *Integration, Test, Abnahme*
  Das System wird aus den fertiggestellten Modulen zusammengebaut (integriert). Dann wird es getestet. Ein spezieller Test ist die Abnahme durch den Kunden.
- *Betrieb und Wartung*
  Das Software-System wird beim Auftraggeber installiert und in Betrieb genommen. Während der Nutzung des Systems fallen Fehler auf, und oft werden auch neue Anforderungen an das System gestellt. Fehler müssen korrigiert werden; neue Anforderungen werden in weiteren Entwicklungsprojekten (sog. Wartungsprojekten) umgesetzt.
- *Auslauf und Ersetzung*
  Jedes Software-System wird irgendwann aus dem Betrieb genommen und in der Regel durch ein Nachfolgesystem ersetzt. Für die Ablösung kann es viele verschiedene Gründe geben; oft ist das alte System nicht mehr wartbar oder kann nicht an neue oder veränderte Anforderungen angepasst werden. Wie alle anderen Schritte muss auch dieser präzise geplant sein.

Eine Reihe von Tätigkeiten findet – wenn auch mit unterschiedlicher Intensität – während der gesamten Entwicklungszeit statt. Dazu gehören die Leitungsfunktionen, also Planung und Management, ebenso die Qualitätssicherung, die dafür sorgt, dass die Qualität geprüft und durch verschiedene Maßnahmen verbessert wird. Die Dokumentation erscheint nicht in der Liste, weil sie Bestandteil aller Tätigkeiten ist.

2.5 Weitere Grundbegriffe

Der präzise Umgang mit den Begriffen ist im Software Engineering besonders wichtig. Denn da Software immateriell ist (Abschnitt 2.3.2), können wir nicht wie der Konstrukteur einer Maschine darauf ausweichen, den Gegenstand selbst zu zeigen oder zu betrachten. Hier ist der Begriff der Gegenstand.

In vielen Fällen arbeiten wir darum mit Metaphern, also mit Begriffen, die sich aus der Erfahrungswelt in die Software übertragen lassen. So ist der *Bug* (»Käfer«) eine Metapher für einen Defekt (nachdem Grace Hopper 1946 tatsächlich einen Defekt entdeckt hatte, an dem ein Falter schuld gewesen war); der (Fehler-)*Ausgang* eines Programmteils ist der Punkt, wo seine Ausführung beendet wird; ein *Werkzeug* (siehe Abschnitt 2.5.2) dient uns zur Bearbeitung des Werkstücks, nämlich der Software. In anderen Fällen war der Begriff schon vor Übernahme in die Informatik abstrakt, aber noch anschaulich, etwa beim Wort »Life Cycle«.

Gerade wenn wir durch die Verwendung vertrauter Begriffe Klarheit zu erreichen suchen, ist die Gefahr groß, dass wir unbemerkt auch Vorstellungen übernehmen, die unangemessen, irreführend oder gefährlich sind. Das zeigt sich z. B.

beim Begriff der Transparenz (der ohne erkennbaren Sinn wahlweise für Unsichtbarkeit oder volle Sichtbarkeit verwendet wird). Hier ist besondere Vorsicht angebracht!

2.5.1 Taxonomie

taxonomy — A scheme that partitions a body of knowledge and defines the relationships among the pieces. It is used for classifying and understanding the body of knowledge.

IEEE Std 610.12 (1990)

Taxonomien sind allgegenwärtig. Beispielsweise werden Lehrveranstaltungen klassifiziert in Vorlesungen, Übungen, Praktika usw.; Vorlesungen sind weiter klassifiziert in Grundvorlesungen und Spezialvorlesungen usw.

Weil im Software Engineering die Begriffe oft unklar sind, werden wir an vielen Stellen gezwungen sein, zunächst Ordnung in die Wörter zu bringen und ihre Beziehungen zu klären. Wo möglich werden dabei Normen wie der oben zitierte Begriffsstandard des IEEE zu Grunde gelegt. Wo diese Quellen unergiebig sind, ist es notwendig, neue Definitionen vorzuschlagen.

Sehr vorteilhaft ist eine baumartige Begriffszerlegung, d. h. die rekursive Zerlegung der durch einen Begriff bezeichneten Menge in disjunkte Klassen (Unterbegriffe).

2.5.2 Methoden, Sprachen, Werkzeuge

Auch das Werkzeug ist im Software Engineering ursprünglich nur eine Metapher, deren Sinn nicht immer erkennbar ist; vor allem die Abgrenzung zu den Sprachen und Methoden ist oft unklar. In diesem Buch wird der Begriff in folgender Bedeutung verwendet:

Ein **Werkzeug** dient zur Ausführung einer Arbeit, die – wenigstens prinzipiell – auch ohne das Werkzeug geleistet werden könnte. Im Produkt ist das Werkzeug nicht mehr enthalten, es sei denn als Ausrüstung für die Wartung.

Im Software Engineering dienen Werkzeuge zur automatischen oder vom Benutzer gesteuerten Transformation, evtl. auch Speicherung, von Information. Typische Werkzeuge sind Editoren, Übersetzer, CASE-Tools und Datenbanksysteme.

Werkstoffe sind von den Werkzeugen deutlich unterschieden, weil aus ihnen das Produkt geformt wird; die Werkstoffe bleiben im Produkt. Wir benutzen als Werkstoffe unserer Produkte Sprachen (siehe Abschnitt 2.3.2), sowohl »natürliche« als auch formale. Eine Sprache oder – hier grundsätzlich synonym – Notation legt die möglichen Aussagen fest (Syntax) und deren Bedeutungen (Semantik).

Methoden sind Handlungsanweisungen, also Regeln, die den Menschen bei der Wahl seiner Aktionen führen.

Methode, Sprache und Werkzeug bilden ein System, wenn sie durch gemeinsame Konzepte verbunden sind (Abb. 2–2). Beispielsweise verbindet das Konzept der Iteration die Methode, gleichartige Daten durch die Wiederholung bestimmter Operationen zu bearbeiten, mit den Sprachkonstrukten für die Konstruktion von Programmschleifen und den Werkzeugen, die uns die Rechner (Index-Register) und Compiler dafür bieten.

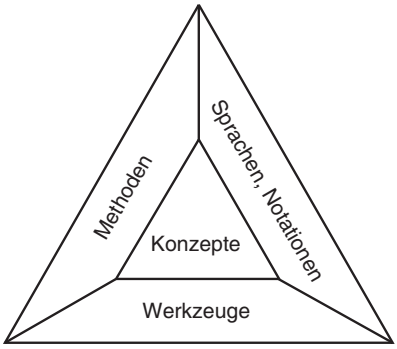

Abb. 2–2 *Das Systemdreieck*

2.5.3 Effektiv und effizient

Im Abschnitt 5.2 werden einige Qualitätsmerkmale definiert und in eine Taxonomie geordnet; dem soll hier nicht vorgegriffen werden. Einzig die Bedeutung der beiden Begriffe »effektiv« und »effizient« soll klargestellt werden, damit keine Missverständnisse entstehen:

- Eine Lösung, die die geforderte Funktionalität aufweist, also im üblichen Sinne korrekt ist, ist *effektiv*. Wir verwenden das Wort allgemeiner auch bei Werkzeugen und Verfahren; ein effektives Werkzeug erfüllt seinen Zweck, ein effektives Verfahren führt zum Ziel.
- Eine Lösung, die von den Betriebsmitteln sparsamen Gebrauch macht, also mit wenig Rechenzeit und wenig Speicher auskommt, ist *effizient*.

Darum ist Effektivität eine Eigenschaft, die gegeben ist oder nicht, man kann sie nicht steigern. Effizienz ist dagegen in unterschiedlichem Maße vorhanden: Ein Programm kann effizienter sein als ein anderes.