3 Software Engineering

Mit den in Kapitel 2 erläuterten Grundbegriffen können wir uns nun dem eigentlichen Gegenstand dieses Buches, dem Software Engineering, zuwenden. Wir gehen auf die geschichtliche Entwicklung ein und stellen die Ziele, die Chancen und auch die Schwierigkeiten des Software Engineerings dar. Am Ende des Kapitels stehen einige Hinweise auf Lehrbücher und andere Basisliteratur des Software Engineerings.

3.1 Fortschritte in Hardware und Software

Schon in den Sechzigerjahren bemerkte Gordon E. Moore, Chef der Firma Intel, dass die Kenngrößen der Hardware (Prozessorleistung, Integrationsdichte, Speicherkapazität pro Dollar) nicht nur zunehmen, sondern exponentiell wachsen. Etwa alle 18 Monate werden sie verdoppelt. Dieser Zusammenhang ist seitdem als *Moore's Law* bekannt.

Jedes Kind weiß, dass ein solches Wachstum nur eine begrenzte Zeit andauern kann; es ist zwar noch nicht völlig klar, wo die physikalischen Grenzen liegen, aber es ist sicher, dass es sie gibt. Vermutlich werden sie im zweiten Jahrzehnt des 21. Jahrhunderts erreicht.

Etwa zur selben Zeit, als Moore seine Entdeckung machte, entstand der Eindruck, dass die Hardware-Entwicklung der Software davonläuft. In den Fünfzigerjahren war der Computer erkennbar das schwächste Glied der Kette, und es wurde nicht an intellektuellem Aufwand gespart, um seine Unzulänglichkeiten zu kompensieren. Ein Beispiel waren Algorithmen, die auch dann am Ende zu einem korrekten Resultat führten, wenn der Rechner während der vielstündigen Berechnung ausfiel und repariert werden musste; bei aufwändigen Berechnungen gab es gar keine andere Wahl. Ein paar Jahre später waren diese Kinderkrankheiten überwunden, die Rechner wurden immer kleiner und billiger, dabei schneller und zuverlässiger.

Über die Software ließ sich das nicht sagen, sie wurde weder billiger noch zuverlässiger. Abbildung 3–1 zeigt die Fortschritte, die bei Hard- und Software von

1960 bis 1990 erzielt wurden. Die Grafik spiegelt die Tatsache, dass sich die Daten der Hardware nach Moore's Law entwickelten und entwickeln, während bei der Software-Entwicklung in etwa zwanzig Jahren eine Verdoppelung der Leistung beobachtet wurde (die sich etwa an Änderungen der Faktoren von COCOMO ablesen lassen, siehe Abschnitt 8.4.3). Dabei gab es – hier durch breite Balken anstelle von Linien angedeutet – beträchtliche »Phasenverschiebungen« zwischen verschiedenen Anwendungsbereichen.

Es ist bemerkenswert, dass sich die Kosten einer Zeile Code in Jahrzehnten kaum verändert haben. In höheren Sprachen leistet eine Zeile Code aber mehr als in primitiven Sprachen, die früher vorwiegend eingesetzt wurden.

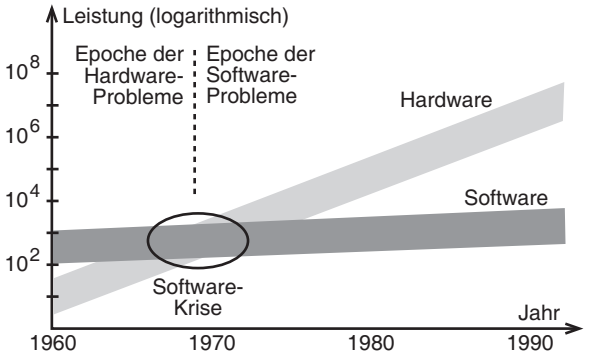

Abb. 3–1 *Entwicklung in der Hard- und Software*

3.1.1 Die Ausgaben für Hard- und Software

Wegen der unterschiedlichen Entwicklung in Hard- und Software wurde Ende der Sechzigerjahre vorausgesagt, dass zukünftig das meiste Geld nicht mehr für Hardware, sondern für Software ausgegeben werde (Abb. 3–2). Diese Aussage wurde in vielen Lehrbüchern übernommen.

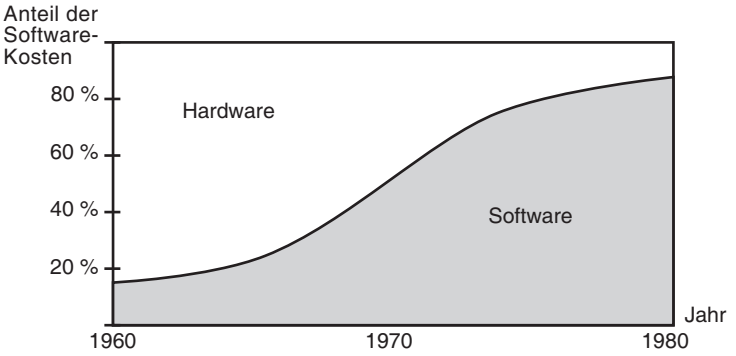

Abb. 3–2 *Falsche Prognose für die Kostenverteilung nach Boehm (1973)*

Obwohl die genannten Ursachen zweifellos bestehen, ist dieser Effekt nach Untersuchungen von Gurbaxani und Mendelson (1987) in Wahrheit nicht eingetreten, weil Software durch Hardware substituiert wird. Beispielsweise haben wir heute schlechte Betriebssysteme auf gigantischen Rechnern, statt guter, wesentlich teurer Betriebssysteme auf kleinen, billigen Rechnern. Zudem sind einige wenige Basiskomponenten der Software heute über die ganze Welt verbreitet, sodass die Kosten pro Exemplar sehr gering sind.

Das Verhältnis zwischen Hardware- und Software-Ausgaben bleibt nach dem zitierten Artikel etwa konstant bei 6:4. Dafür kann es viele Gründe geben. Unterstellt man, dass damit tatsächlich das Optimum des Nutzens erzielt wird, so kann man ein Modell suchen, das den Effekt erklärt.

Wenn die Hardware oder die Software völlig nutzlos ist, wird auch das ganze System nutzlos sein. Daraus folgt als einfachstes Modell, den Gesamtnutzen als Produkt aus Hardware-Nutzen und Software-Nutzen zu betrachten. Nehmen wir nun an, dass der Hardware-Nutzen N_{HW} proportional dem eingesetzten Geld G_{HW} ist, der Software-Nutzen N_{SW} entsprechend proportional G_{SW}, wobei die Faktoren unterschiedlich sein können:

$$N_{HW} = k_{HW} \cdot G_{HW}$$

$$N_{SW} = k_{SW} \cdot G_{SW}$$

$$N = N_{HW} \cdot N_{SW} = k_{HW} \cdot G_{HW} \cdot k_{SW} \cdot G_{SW}$$

Steht insgesamt das Geld G zur Verfügung, so gilt außerdem $G_{HW} + G_{SW} = G$. Wir können also schreiben

$$N = k_{HW} \cdot k_{SW} \cdot G_{HW} \cdot (G - G_{HW})$$

Offensichtlich erreicht N sein Maximum unabhängig von den Faktoren k_{HW} und k_{SW} für $G - 2G_{HW} = 0$; es ist also zweckmäßig, jeweils genau die Hälfte des Budgets für Hardware und für Software aufzuwenden.

Die beobachtete Abweichung kann mit den beiden folgenden Hypothesen erklärt werden:

- Manager kaufen lieber Hardware als Software, weil Hardware in vieler Hinsicht kaufmännisch einfacher zu handhaben ist: Die Bilanzierung ist klar, Wartungskosten und Abschreibung sind bekannt.
- Der Nutzen steigt nicht linear mit dem Aufwand, sondern bei der Hardware progressiv (überproportional) und/oder bei der Software degressiv (unterproportional). In diesem Falle verschiebt sich das Maximum in Richtung höherer Ausgabenanteile für die Hardware. Diese Erklärung ist nicht abwegig, weil sich auf der Software-Seite mit höherem Aufwand kaum höhere Leistung

erzielen lässt, während man bei der Hardware z. B. durch eine Speichererweiterung Wachstum zu linearem Preis kaufen kann.

3.1.2 Die Entstehung des Software Engineerings

In den späten Sechzigerjahren zeigte sich erstmals, dass sich manche Software-Projekte selbst mit gigantischem Aufwand nicht zu einem befriedigenden Ende bringen ließen. In vielen anderen Fällen wurde der Abschluss nur mit unbefriedigenden Resultaten, viel zu spät oder mit extremen Kostenüberschreitungen erreicht. Kurz: Es war nicht alles machbar, was die Entwickler für machbar *hielten*. Dabei nahm die Bedeutung der Rechner laufend zu, sodass der Wunsch nach Methoden und Werkzeugen zur schnellen und billigen Entwicklung fehlerarmer Software immer drängender wurde.

Für diese Probleme kam das Schlagwort »Software Crisis« auf. Die Software-Krise war Anlass für viele Diskussionen und Tagungen. Bei der Vorbereitung der NATO-Konferenz in Garmisch (Naur, Randell, 1969) prägte F.L. Bauer, der schon zehn Jahre zuvor u.a. durch die Erfindung des Kellerspeichers bedeutende Beiträge zur Informatik geleistet hatte, das Wort Software Engineering als Provokation (siehe dazu Bauer, 1993):

> *The whole trouble comes from the fact that there is so much tinkering with software. It is not made in a clean fabrication process, which it should be. What we need, is software engineering.*

Die Geschichte des Software Engineerings beginnt also nach diesem Ereignis, sie entwickelt in den Siebzigerjahren eine Eigendynamik. Wesentliche Beiträge in der Frühzeit des Software Engineerings waren

- der sogenannte Software Life Cycle (Royce, 1970; siehe Abschnitt 9.1.3) und
- die Sammlung empirischer Daten (beginnend mit Boehm, 1973).

Damit wurden vor allem die Kostentreiber und die Verteilung der Kosten besser verstanden.

3.1.3 Definitionen für »Software Engineering«

»Software Engineering« wurde oft definiert. Drei wichtige Definitionen (hier dem Sinn nach wiedergegeben) sind:

Software Engineering ist die Entdeckung und Anwendung solider Ingenieurprinzipien mit dem Ziel, auf wirtschaftliche Art Software zu bekommen, die zuverlässig ist und auf realen Rechnern läuft.

F.L. Bauer

Software Engineering ist die Herstellung und Anwendung einer Software, wobei mehrere Personen beteiligt sind oder mehrere Versionen entstehen.

D.L. Parnas

software engineering — (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
(2) The study of approaches as in (1).

IEEE Std 610.12 (1990)

Auf Deutsch sagt der Standard also: Software Engineering ist das systematische, disziplinierte und quantifizierbare Vorgehen zur Entwicklung, zum Betrieb und zur Wartung von Software, kurz: die Arbeit an Software nach Ingenieurprinzipien.

Diese Definition ist nicht gut gelungen. Sie gibt uns die Auswahl zwischen zwei Kriterien: Entweder werden an das Software Engineering Erwartungen geknüpft, die unrealistisch sind und in der Praxis kaum erfüllt werden. Was hier definiert wird, ist eher »optimales Software Engineering«. Oder das Ingenieurwesen wird als Fundament verwendet, das leider selbst nicht definiert ist. Vor allem Leute, die keine Ingenieure sind, scheinen eine idealisierte Vorstellung von der Arbeit der Ingenieure zu haben. Tatsächlich verlassen sich diese viel mehr auf Erfahrung und Intuition, als es ihren Bewunderern recht sein kann.

Wir verwenden folgende sehr pragmatische, »anspruchslose« Definition, die keine Wertung impliziert (Ludewig, 2001):

Software Engineering ist jede Aktivität, bei der es um die Erstellung oder Veränderung von Software geht, soweit mit der Software Ziele verfolgt werden, die über die Software selbst hinausgehen.

Das heißt: Überall, wo Software entwickelt oder bearbeitet wird, um ihre Funktionalität nutzbar zu machen, sehen wir Software Engineering oder, in einer unbefriedigenden Halbübersetzung, *Softwaretechnik*. Ausgenommen sind vor allem Programme, die nur zum Spaß oder zur Übung (in der Ausbildung) geschrieben werden.

3.2 Grundideen des Software Engineerings

Die ungewöhnliche Entstehungsgeschichte dieses Gebietes hatte zur Folge, dass über die »richtige« Auslegung des Begriffs bis heute gestritten wird. Die oben zitierten Definitionen deuten schon darauf hin. Aber in einigen fundamentalen Punkten besteht doch Einigkeit. Diese Punkte werden nachfolgend aufgezählt.

3.2.1 Software Engineering versus Kunst der Programmierung

Mit der Einführung des Engineering-Begriffs in die Software-Welt war nicht nur die positive Aussage in Richtung der Ingenieure verbunden, sondern auch die Abwendung vom Künstlertum. Sie erscheint aus der Perspektive des 21. Jahrhunderts unnötig. Daher ist daran zu erinnern, dass es eine Zeit gab (in den Sechzigerjahren), in der IBM gern Menschen mit künstlerischer Ausbildung einstellte,

da die Software-Entwicklung als ein Metier galt, in dem die (den Künstlern unterstellte) Kreativität besonders wichtig ist. Tabelle 3–1 stellt die Merkmale der Werkstatt den Merkmalen des Künstler-Ateliers gegenüber.

	Werkstatt (technisches Produkt)	Atelier (Kunstwerk)
Die geistige Voraussetzung	ist das vorhandene und verfügbare technische Know-how	ist u. a. die Inspiration des Künstlers
Die Termine	sind in der Regel mit genügender Genauigkeit planbar	sind wegen der Abhängigkeit von der Inspiration nicht planbar
Der Preis	ist an den Kosten orientiert und darum kalkulierbar	ist nur durch den Marktwert, nicht durch die Kosten bestimmt
Normen und Standards	existieren, sind bekannt und werden in aller Regel respektiert	sind rar und werden, wenn sie bekannt sind, nicht respektiert
Eine Bewertung, ein Vergleich	kann nach objektiven, quantifizierten Kriterien durchgeführt werden	ist nur subjektiv möglich, das Ergebnis ist umstritten
Der Urheber	bleibt meist anonym, hat keine emotionale Bindung zum Produkt	betrachtet das Kunstwerk als Teil seiner selbst
Gewährleistung und Haftung	sind klar geregelt, können nicht ausgeschlossen werden	sind undefiniert und praktisch kaum durchsetzbar

Tab. 3–1 *Werkstatt und Atelier*

In der Praxis hat sich das Leitbild des Ingenieurs in der Software-Welt auch zu Beginn des 21. Jahrhunderts noch nicht durchgesetzt, und man muss nicht lange suchen, um einen Blick in die Software-Steinzeit zu werfen:

- Termine und Kosten werden ohne rationale Grundlage geschätzt und im Verlauf des Projekts vielfach weit überschritten.
- Regeln sind nicht existent oder nicht bekannt oder werden ignoriert, das Gleiche gilt verstärkt für Normen.
- Programmtests liefern nur eine sehr schwache Aussage über die Funktionalität der Programme, andere Eigenschaften (z. B. die Portabilität oder die Robustheit) können nur subjektiv bewertet werden. Für den Vergleich von Software-Produkten fehlen anerkannte Kriterien.
- Der Programmierer baut seine Persönlichkeit in die Software ein, sodass sie für andere Menschen unverständlich ist. Kritik an seinem Werk kann er kaum ertragen.
- Trotzdem übernehmen weder die Entwickler noch ihre Firma die Verantwortung für die Qualität des Produkts.

Der Anspruch des Software Engineerings ist also noch nicht eingelöst. Doch die oben genannten Merkmale technischer Produkte spiegeln sich bereits in den Aktivitäten dieses Gebietes: Es gibt seit Mitte der Siebzigerjahre eine wachsende Zahl veröffentlichter Arbeiten über Software-Management, -Kostenschätzung, -Spezifi-

kation, -Entwurf, -Metriken, -Validierung und formale Verifikation. Seit Beginn der Achtzigerjahre sind empirische Daten (Einsatzerfahrungen und vergleichende Statistiken) hinzugekommen. Auch der Begriff des »egoless programming« (Weinberg, 1971 und 1999) passt zu diesem Ansatz: Er bezeichnet einen Programmierstil, bei dem das Produkt nicht mehr die Handschrift seines Verfassers trägt, sondern vorgegebenen Normen entspricht und damit auch anderen Programmierern leicht verständlich ist. Zu Beginn des 21. Jahrhunderts haben sich auch die gesetzlichen Regelungen verändert, sodass der Haftungsausschluss immer schwieriger wird. Alle professionellen Software-Entwickler sollten diese Tendenzen begrüßen, denn sie werden dazu führen, dass die Software besser wird – und dass ihre unqualifizierten Konkurrenten mehr und mehr Probleme haben werden, mit minderwertiger Software Geld zu verdienen.

Wenn wir im Software Engineering nach dem Vorbild anderer Ingenieurdisziplinen arbeiten wollen, dann müssen auch wir uns an den Kosten orientieren. Unser Ziel ist die globale Optimierung der Bearbeitungsprozesse, d. h., ein Fortschritt im Sinne des Software Engineerings ist eine Veränderung, die zu einer *Senkung der Gesamtkosten* (siehe Abschnitt 2.1) führt.

3.2.2 Quantität als Qualität

Software Engineering ist nicht zu verstehen, wenn man sich auf kleine Einheiten konzentriert, z. B. auf einzelne Algorithmen. Denn bei größerem Umfang schlägt Quantität in Qualität um, d. h., das Große ist nicht das Vielfache von etwas Kleinem, sondern etwas ganz anderes.

Das wird im Vergleich mit einem anderen System deutlich: Ein Rinnsal kann man mit einem simplen Brett überbrücken. Um einen Bach zu queren, braucht man einen Steg, der mit Pfosten verankert ist. Ein Fluss von einigen Metern Breite erfordert eine Brücke, und für eine Meerenge wie das Golden Gate bei San Francisco brauchte man eine Hängebrücke in einer völlig neuen Konstruktion. Die beiden Extreme, das Brett und die Golden Gate Bridge, unterscheiden sich nicht nur in den Dimensionen (Maße, Gewichte, Kosten), sondern sie haben so gut wie nichts gemeinsam, was die Voraussetzungen und die notwendigen Arbeiten betrifft. Der Aufwand für die Planung, für die statische Berechnung und für die Baugenehmigung ist beim Brett nicht gering, sondern exakt null. Arbeiter werden nicht eingestellt, Zufahrtstraßen werden nicht gebaut, neue Verfahren, um Pfeiler im Meeresboden zu verankern, brauchen nicht entwickelt zu werden. Umgekehrt bedeutet das: Wer den Brückenbau erlernen will, darf sich nicht an Bauwerken orientieren, die höchstens einen halben Meter überspannen. Denn die Lösungen für kleine Probleme lassen sich nicht für große Probleme *skalieren*.

Die gleiche Betrachtung lässt sich anstellen, wenn man n Personen organisieren muss oder wenn man ein Programm mit n Zeilen schreiben oder verstehen will (mit n von 3 bis 3 000 000). In allen diesen Beispielen ist das Große nicht

einfach durch Extrapolation vom Kleinen her zu verstehen bzw. zu realisieren; beim Großen kommt vor allem Aufwand für die Organisation hinzu, kurz: *Größe verursacht Aufwand*.

Die Probleme, die durch die Größe entstehen, werden durch einen statistischen Effekt verschärft: Während es bei einfachen Systemen aus wenigen Komponenten einfach ist, ihre Korrektheit zu überprüfen und, wenn nötig, herzustellen, sodass mit hoher Wahrscheinlichkeit ein funktionierendes Gesamtsystem entsteht, wird das bei Systemen aus vielen Komponenten *sehr* viel schwieriger. Das zeigt sich, wenn man die Möglichkeit analysiert, ein korrektes Programm zu schreiben. Dazu soll das folgende sehr einfache, aber nicht abwegige Modell (nach Dijkstra, 1974) dienen:

Ein Software-System sei aus n Modulen zusammengesetzt. Jedes einzelne Modul sei korrekt mit der Wahrscheinlichkeit p; n Module sind dann korrekt mit der Wahrscheinlichkeit p^n. Dabei ist vereinfachend vorausgesetzt, dass es keine Abhängigkeiten zwischen den Wahrscheinlichkeiten für die Korrektheit der einzelnen Module gibt. Für n = 100 können wir beispielsweise die folgenden Werte berechnen.

Wenn p	=	0,9	0,95	0,99	0,999
dann ist p^{100}	=	0,000 027	0,006	0,37	0,90

Bei einem kleinen Programm (nur ein Modul) kommen wir also auch mit p = 0,9 gut aus, nur in einem Zehntel der Fälle funktioniert es nicht. Um die gleiche Funktionssicherheit bei einem System aus hundert Modulen zu erreichen, muss jedes einzelne Modul zu 99,9 % korrekt sein! Wir müssen also extrem fehlerarm arbeiten.

Hinzu kommt, dass unsere Fähigkeit, etwas Komplexes zu verstehen, keineswegs langsam abnimmt, wenn die Komplexität steigt. Vielmehr kommen wir mit kleinen Schwierigkeiten gut zurecht, sind aber schon bei etwas höherer Komplexität völlig überfordert und bringen nichts Korrektes mehr zustande. Dabei wirkt es sich besonders tückisch aus, dass wir wesentlich leistungsfähiger zu sein scheinen, wenn wir konstruktiv arbeiten, als wenn wir analytisch an ein Problem herangehen. Wir erzeugen also ohne große Mühe Software, die wir anschließend nicht mehr verstehen können (Abb. 3–3).

Sehr kurz gesagt: Der Mensch kann nur Probleme von geringer Komplexität handhaben und macht auch dabei noch Fehler. Die banale, aber verlässliche Regel KISS (keep it small and simple) oder in verstärkter Form KISSS (keep it small, stupid and simple) weist uns den richtigen Weg.

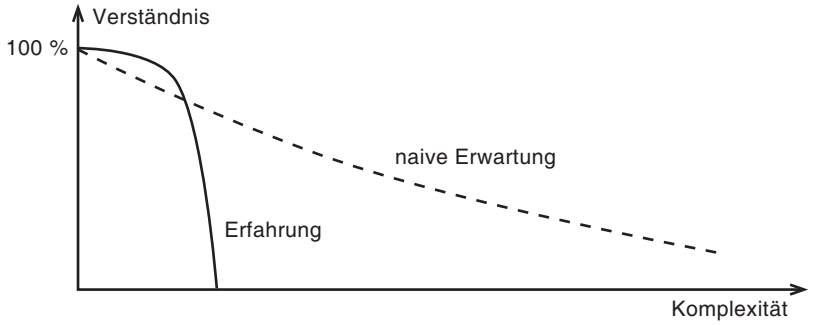

Abb. 3–3 *Das Verständnis in Abhängigkeit von der Komplexität*

3.3 Probleme und Chancen des Software Engineerings

Wer sich aufmacht, die Welt der Software-Bearbeitung zu verbessern, sollte die Schwierigkeiten kennen, auf die er sehr wahrscheinlich stoßen wird.

3.3.1 Software Engineering als globale Optimierung

Wir wollen, wie in Abschnitt 2.1 festgestellt wurde, die Gesamtkosten senken. Ein naheliegender Ansatz wäre, die einzelnen Aktivitäten des Software Engineerings zu untersuchen und auf Möglichkeiten zur Einsparung abzuklopfen. Dieser naive Ansatz führt aber nicht zum Ziel, denn die Aktivitäten sind wie die einzelnen Budgets im Staatshaushalt miteinander verknüpft. Wenn man sparen will, kann man die Ausgaben für Bildung und Familien senken, sabotiert damit aber die zukünftige Entwicklung. Wenn man in der Software-Bearbeitung sparen will, kann man die Spezifikation und den Architekturentwurf weglassen, wird dafür aber später, schon beim Test und bei der Integration, mehr noch in der Wartung, hart bestraft.

Es ist also notwendig, das *globale Optimum* zu suchen, nicht das lokale Optimum der einzelnen Tätigkeiten. Dafür gibt es keine simple Formel. Wir müssen ein differenziertes Bild der Kosten und ihrer Beziehungen entwickeln, um uns dem Optimum zu nähern (siehe Kap. 4). Und wir müssen dort, wo uns auch das differenzierte Bild der Kosten nicht weiterhilft, weil es zu viele Details bietet, nach dem Vorbild der Ingenieure das Prinzip der hohen Qualität anwenden (siehe Kap. 5).

3.3.2 Software Engineering als defensive Disziplin

Software Engineering spielt in der Informatik eine ähnliche Rolle wie die Hygiene in der Medizin: Sie nützt nichts, sondern verhindert vielmehr Schäden und sollte generell beachtet werden. Ist dieses Ideal erreicht, so ist sie als eigenständiges

Fach überflüssig. Software Engineering zielt also darauf ab, selbstverständlich und damit überflüssig zu werden. Darum kann das Fach Software Engineering keine atemberaubenden Visionen bieten; wenn man einem Patienten sagt, dass man hart daran arbeite, ihn während seines Aufenthaltes im Krankenhaus vor einer Infektion zu schützen, wird er wahrscheinlich sagen oder denken, dass er eigentlich gekommen sei, um gesund zu werden, nicht, um noch eine neue Krankheit aufzulesen. Software Engineering ist – wie die Hygiene – langweilig und frustrierend für Leute, die die Abwehr von Fehlschlägen und Katastrophen nicht als positive Leistung betrachten.

3.3.3 Software Engineering als ein Gestrüpp von Problemen

Eine besondere Schwierigkeit sowohl in der praktischen Arbeit als auch in der Lehre ist die wechselseitige Verknüpfung zwischen den Maßnahmen, die zum Software Engineering gerechnet werden. Beispielsweise ist eine Qualitätssicherung kaum möglich ohne eine Konfigurationsverwaltung, also eine zuverlässige Verwaltung der Teilprodukte. Die Konfigurationsverwaltung setzt natürlich die Definition und Stabilität der Teilprodukte voraus, was nur durch Qualitätssicherung möglich ist. Abbildung 3–4 zeigt diesen Zusammenhang in Form der Gewölbe-Metapher; die Steine stützen sich gegenseitig. Wenn auch nur ein einziger Stein fehlt, trägt das Gewölbe nicht. Unbedingte Voraussetzungen sind als Fundamente dargestellt: Ohne qualifizierte Software-Leute und ohne die aktive Unterstützung durch das Management sind Verbesserungen im Software Engineering chancenlos.

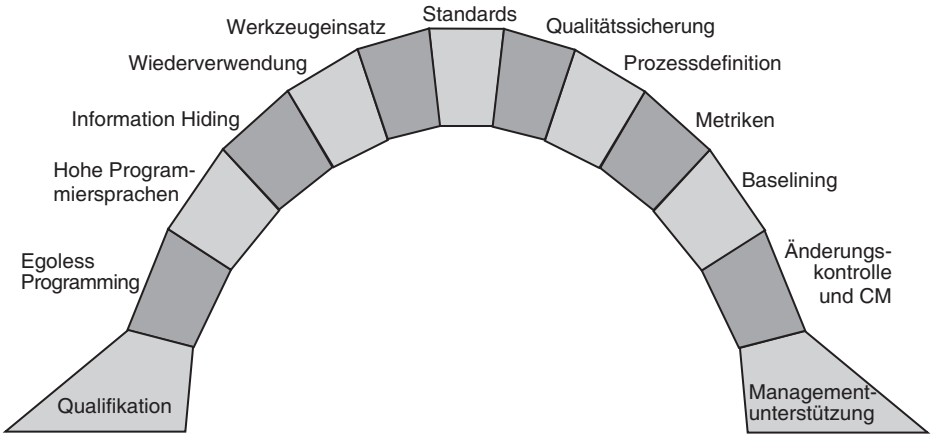

Abb. 3–4 *Das Software-Engineering-Gewölbe*

3.3.4 Software Engineering als Technologie für die Köpfe

Wissenschaftliche Erkenntnis und neue Technologie lassen sich gut in die Anwendung transferieren, wenn sie sich in Produkten niederschlagen, z. B. in neuen Autos, neuen Telefonen, neuen Werkstoffen. Die Zahl der Leute, die die Erkenntnis, die Technologie, verstehen müssen (z. B. die Technik eines Funktelefons), ist sehr gering, die Anwender der Produkte brauchen kein Verständnis. Das gilt auch in der Software: Einen Compiler oder ein Betriebssystem kann man benutzen, ohne die Technik zu verstehen.

Leider lassen sich die Fortschritte im Software Engineering nur zu einem kleinen Teil in Werkzeuge gießen. Ganz überwiegend entstehen im Software Engineering Einsichten und Regeln, die von den Software-Entwicklern umgesetzt werden müssen. Das sind sehr viele Menschen, die zudem zu einem großen Teil völlig unzureichend ausgebildet sind. Software Engineering zielt darauf ab, das Verhalten dieser Leute zu verändern. Damit ist eine rasche Veränderung der Praxis völlig ausgeschlossen.

Natürlich lässt sich gutes Software Engineering nicht durch Vorschriften durchsetzen; wer nicht systematisch und diszipliniert vorgehen will, wird es auch dann nicht tun, wenn es eine Richtlinie oder das Projekthandbuch von ihm fordern. Wir brauchen darum eine handwerkliche Tradition, wie sie bei den Mechanikern, Technikern und Ingenieuren über Jahrhunderte gewachsen ist. Nur, wenn den Entwicklern gute Arbeit, Beachtung der Normen, penible Prüfung usw. *selbstverständlich* sind, können wir erwarten, solide Produkte zu erhalten.

3.3.5 Software Engineering als Goldader

Den Schwierigkeiten stehen allerdings auch ungewöhnliche Chancen gegenüber: Auf kaum einem anderen Gebiet ist der Bedarf an Know-how, und damit auch an qualifizierten Leuten, so hoch wie im Software Engineering, auf kaum einem anderen Gebiet lassen sich ähnliche Erfolge erzielen (oder genauer: ähnliche Desaster verhindern) – einfach, weil der Abstand zwischen der Praxis und dem Stand der Wissenschaft so groß ist. In vielen Universitäten der Welt, nicht zuletzt auch in Stuttgart, wurden spezielle Studiengänge eingerichtet. Sie repräsentieren die Bedeutung des Fachs und das gestiegene Selbstbewusstsein der Software-Ingenieure. Dass die Absolventen dieses Studiengangs auf dem Arbeitsmarkt besonders gesucht sind, bestätigt das Konzept einer speziellen Ausbildung der Software-Ingenieure (siehe Kap. 25).

> *Even though no technological breakthrough promises to give the sort of magical results with which we are so familiar in the hardware area, there is both an abundance of good work going on now, and the promise of steady, if unspectacular progress.*
>
> F.P. Brooks (1987)

3.4 Lehrbücher und andere Basisliteratur

Bereits 1979 war unter dem holprigen Titel »Einführung in Software Engineering« ein deutschsprachiges Lehrbuch verfügbar (Kimm et al., 1979). Blättert man knapp dreißig Jahre später darin, dann stellt man fest, dass sich damals das »Programming in the Large« (DeRemer, Kron, 1976) noch nicht gegen das geläufigere Programmieren im Kleinen behaupten konnte. Es ging aber bereits um dieselben Fragen und Probleme, um die es auch heute geht, und die Lösungsansätze waren nicht fundamental anders.

Zur Mitte der Achtzigerjahre erschienen zwei englischsprachige Lehrbücher (Fairley, 1985, und die erste Auflage des »Sommerville«), die das Gebiet nachhaltig geprägt haben. Inzwischen sind einige weitere Lehrbücher hinzugekommen, zum Teil bereits mehrmals revidiert und erweitert (was nicht notwendig eine Verbesserung bedeutet). Die Rangliste der Auflagen beginnt mit:

- Sommerville (2010) 9. Aufl. (1. Aufl. 1982)
- Pressman (2010) 7. Aufl. (1. Aufl. 1992)
- Jalote (2005) 3. Aufl. (1. Aufl. 1991)
- van Vliet (2008) 3. Aufl. (1. Aufl. 1993)
- Ghezzi, Jazayeri, Mandrioli (2003) 2. Aufl. (1. Aufl. 1991)

Im schmalen Angebot auf Deutsch ist vor allem das mittlerweile dreibändig Werk »Lehrbuch der Softwaretechnik« von Helmut Balzert zu nennen. Es besteht aus den Bänden »Basiskonzepte und Requirements Engineering«, »Entwurf, Implementierung, Installation und Betrieb« und »Softwaremanagement«.

- Balzert (2009), Band 1 3. Aufl. (1. Aufl. 1996)
- Balzert (2011), Band 2 3. Aufl. (1. Aufl. 1996)
- Balzert (2008), Band 3 2. Aufl. (1. Aufl. 1998)

Da das Software Engineering traditionell und aus guten Gründen eng mit dem Gebiet der Programmiersprachen verbunden ist, gehört im weiteren Sinne auch der Klassiker von Sebesta (2012), bereits die zehnte Auflage, in diese Liste.

Die IEEE Computer Society, die mit den »Transactions on Software Engineering« und »Software« zwei der wichtigsten Zeitschriften auf unserem Gebiet herausgibt, hat in der Vergangenheit immer wieder dafür gesorgt, dass wichtige Artikel zu einem Thema, die in verschiedenen Tagungsbänden und Zeitschriften erschienen sind, in großformatigen Paperbacks zusammengefasst wurden. Stellvertretend für die große Auswahl an solchen Sammlungen seien hier die beiden aktuellsten zum Software Engineering genannt: Thayer und Christensen (2005), Thayer und Dorfman (2005) sowie die Sammlung »klassischer« Arbeiten (Thayer, Dorfman, 1996).

Glücklich, wer um die Jahresmitte 2001 Gelegenheit hatte, die großen Persönlichkeiten des Software Engineerings fast vollzählig im alten Bundestag in

Bonn zu sehen und zu hören. Alle anderen können wenigstens die Beiträge in der Dokumentation von Broy und Denert (2002) nachlesen und mittels der vier beiliegenden DVD-Scheiben noch an den Vorträgen teilnehmen.

David L. Parnas, einer der bedeutendsten Vor- und Nachdenker des Software Engineerings, ist nicht durch Bücher, sondern durch viele brillante Artikel berühmt. Schön, dass diese Artikel in Form eines Parnas-Lesebuches vorliegen (Parnas, Hoffman, Weiss, 2001). Richard Selby (2007) hat ein ähnliches Buch mit den wichtigsten Artikeln von Barry W. Boehm, einem anderen Großen des Software Engineerings, zusammengestellt.

Das kleinste, aber sicher nicht das schlechteste Lehrbuch des Software Engineerings hat Alan Davis (1995) verfasst. Seine 201 Regeln sind jeweils in wenigen Worten begründet und erläutert. Sein leider nur noch antiquarisch erhältliches Buch schürft nicht tief, aber an den richtigen Stellen.

Im weiteren Sinne gehört in diesen Überblick auch die Normensammlung des IEEE (siehe Abschnitt 26.3.2).

Wenn man irgendetwas sucht und in der einschlägigen Literatur nicht findet, dann ist sehr oft die »freie Enzyklopädie WIKIPEDIA« (Wikipedia, o.J.) die letzte Rettung. Auch wenn die Artikel von sehr unterschiedlicher Qualität und manchmal offensichtlich nicht neutral sind, ist diese Quelle doch oft außerordentlich hilfreich.