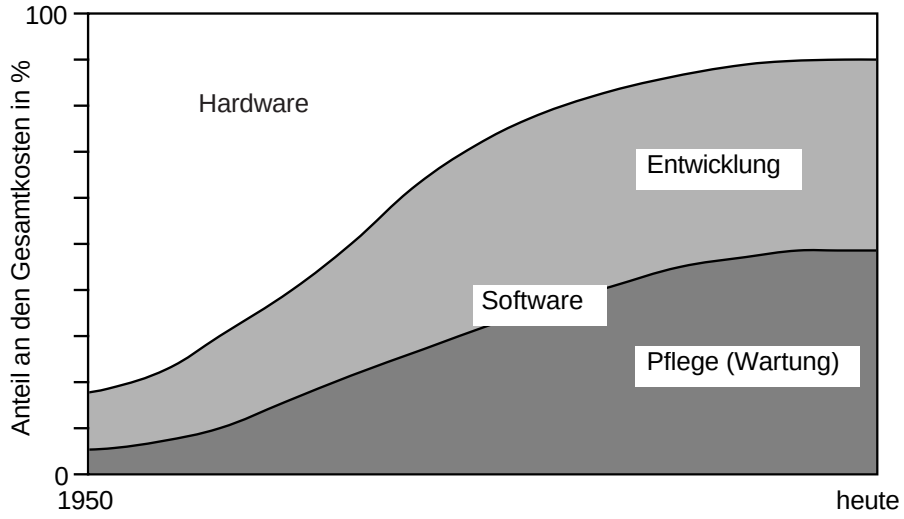
1. Einführung: Software-Entwicklung als Problem

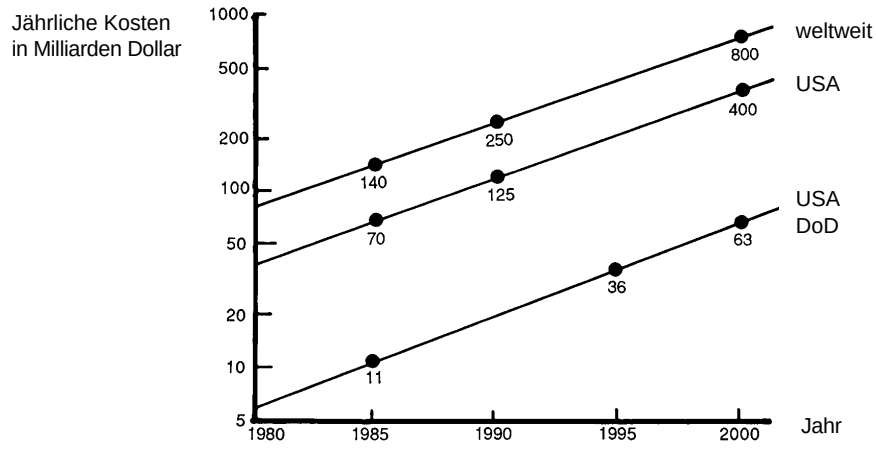
Software ist in den vergangenen Jahrzehnten zu einem derjenigen Faktoren geworden, ohne die in den hochentwickelten Ländern nichts mehr geht. Die Entwicklung von Software wird jedoch bis heute nur ungenügend beherrscht. Da die Softwarekosten den Hardwarekosten längst den Rang abgelaufen haben (Bild 1.1) und weltweit horrende Summen für Software ausgegeben werden (Bild 1.2)[1)], ist dies ein sehr unbefriedigender Zustand.

In diesem Kapitel wird geklärt, was eigentlich Software ist und warum Software Entwicklung schwierig ist. Auf diesem Hintergrund werden die Idee, die Ziele und die grundlegenden Mittel des Software Engineerings motiviert und eingeführt.



nach Boehm (1981)

BILD 1.1. Entwicklung des Kostenverhältnisses von Software und Hardware



nach Boehm (1987)

BILD 1.2. Tendenzen im Wachstum der Software-Kosten

---

1) Boehm hat 1987 eine exponenzielle Entwicklung der Software-Kosten vorausgesagt. Zuverlässige aktuelle Zahlen aus öffentlich zugänglichen Quellen gibt es leider nicht. Man kann aber davon ausgehen, dass weltweit allein im IT-Bereich über eine Billion (10^{12}) Euro pro Jahr für die Entwicklung, die Beschaffung und den Unterhalt von Software ausgegeben werden. Hinzu kommen die Kosten für Software in Produkten und Anlagen (siehe Anhang am Schluss dieses Kapitels).

1.1 Software

1.1.1 Die Rolle der Software

Rechner durchdringen alle Lebensbereiche – und mit ihnen die Software, welche diese Rechner steuert. Wirtschaft und Gesellschaft sind abhängig geworden von Software. Und die Abhängigkeit nimmt zu. Viele Selbstverständlichkeiten des Alltagslebens sind ohne Rechner und deren Software nicht mehr möglich.

In krassem Gegensatz zu dieser Abhängigkeit steht die Tatsache, dass weltweit die Erstellung von Software nur ungenügend beherrscht wird. Termin- und Kostenüberschreitungen bei Software-Projekten sind die Regel; Software, die Fehler enthält oder sich nicht entsprechend den Vorstellungen und Bedürfnissen der Benutzenden verhält, gehört zum Alltag. Immer wieder kommt es auch vor, dass ganze Projekte scheitern.

Stellvertretend seien zwei spektakuläre Beispiele aus jüngerer Zeit genannt: das Scheitern des CONFIRM-Projekts und der missglückte Erstflug der Ariane 5-Rakete.

Im CONFIRM-Projekt sollte im Auftrag großer amerikanischer Hotel- und Mietwagenunternehmen ein neues, umfassendes Reservierungssystem entwickelt werden. Nach dreieinhalb Jahren Entwicklungszeit und Investitionen von rund 125 Millionen Dollar wurde das Projekt im Juli 1992 eingestellt, als erkannt wurde, dass die gestellten Anforderungen mit dem gewählten Lösungsansatz nicht erreichbar waren (Oz, 1995).

Bei der Ariane 5 führten eine nicht behandelte Ausnahmebedingung in der Software der Trägheitsnavigationssysteme sowie die Fehlinterpretation der Meldung über den Ausfall dieser Systeme als Messdaten(!) durch die Software des Hauptrechners zu unsinnigen Befehlen an die Steuerdüsen und damit zur Zerstörung der Rakete (Lions, 1996).

Dabei erscheint Programmieren auf den ersten Blick gar nicht so schwierig. Was ist ein Programm denn eigentlich mehr als das Zusammensetzen einfacher Befehle zu einer geeigneten Folge von Anweisungen an einen Rechner? Was ist denn Software eigentlich und was hat sie für besondere Eigenschaften, die den Umgang mit ihr so schwierig machen?

1.1.2 Software

DEFINITION 1.1. *Software*. Die Programme, Verfahren, zugehörige Dokumentation und Daten, die mit dem Betrieb eines Computersystems zu tun haben (IEEE 610.12).

Diese Definition erlaubt uns, drei wichtige Feststellungen über Software zu machen, die wesentliche Hinweise darauf geben, warum die Entwicklung von Software schwierig ist.

FESTSTELLUNG[1)] **1.1.** Software umfasst erheblich mehr als nur Programme (Bild 1.3).

Der Aufwand für das Schreiben der Programme beträgt in der Regel nur 10-20 Prozent des Gesamtaufwands für die Entwicklung von Software. Bezogen auf die gesamten Lebensdauer einer Software sind es unter 10% der Kosten (Bild 1.4). Da die Programme jedoch der entscheidende Bestandteil von Software sind (ohne Programme geht nichts), wird bei der Schätzung des

---

1) Wir formulieren in diesem Text alle Gesetzmäßigkeiten über Software und Software Engineering als *Feststellungen* und *Regeln*. Die Wahl dieser Terminologie hat folgenden Hindergrund: In allen Lebensbereichen gilt, dass wir Gesetzmäßigkeiten vermuten, wenn wir gewisse Erfahrungen in gleicher Weise immer wieder machen. In den Wissenschaften wird dann versucht, solche Vermutungen durch Experimente und statistische Verfahren zu untermauern. Auf diese Weise gewinnen wir Gesetze, zum Beispiel das Gravitationsgesetz oder das Hebelgesetz in der Physik. Solche Gesetzmäßigkeiten kennen wir auch für Software. Im Gegensatz zu den Gesetzen der Naturwissenschaften sind die Software-Gesetzmäßigkeiten jedoch nur ansatzweise quantitativ gefasst und statistisch abgesichert. Wir sprechen daher im Folgenden von *Feststellungen*, wenn wir sichere oder empirisch erhärtete *Beobachtungen* beschreiben, und von *Regeln*, wenn wir entsprechende *Zusammenhänge* von Phänomenen beschreiben.
Endres und Rombach (2003) haben unlängst das empirisch erhärtete Wissen über Software Engineering systematisch zusammengetragen und publiziert.

Aufwands für die Entwicklung von Software allzuoft nur auf die Programme abgestellt. Dadurch wird der tatsächliche Aufwand um ein Vielfaches unterschätzt.

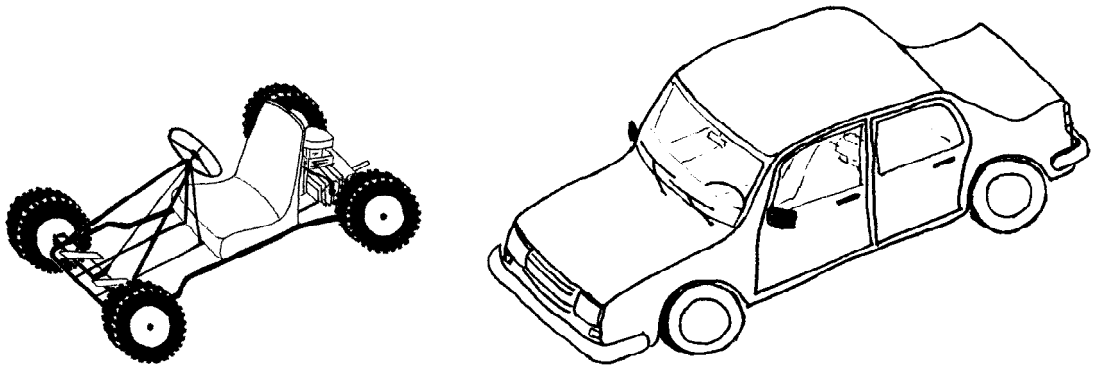
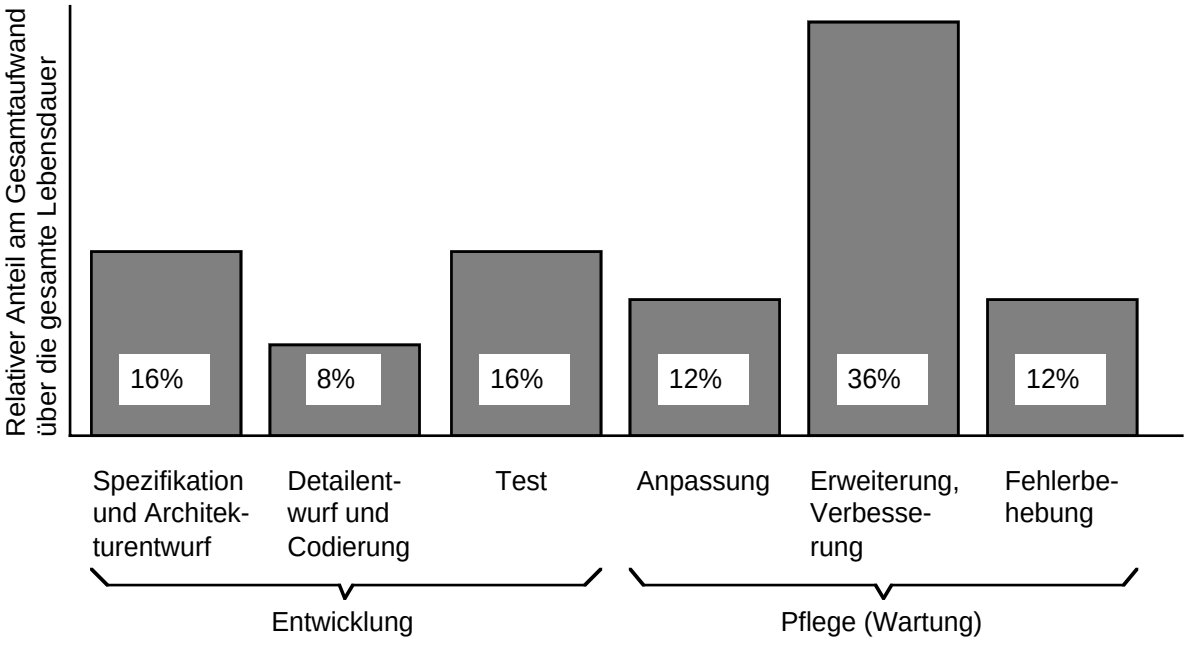


BILD 1.3. So wie ein Auto wesentlich mehr ist als nur ein fahrbarer Untersatz, so umfasst Software wesentlich mehr als nur Programme.



nach Boehm (1973)

BILD 1.4. Verteilung des Aufwands über die Lebensdauer eines Software-Produkts

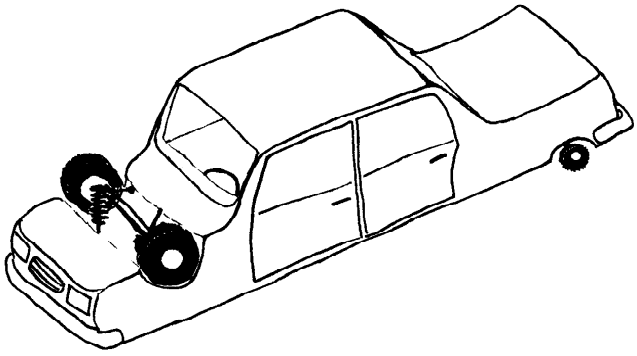
FESTSTELLUNG 1.2. Software ist ein immaterielles technisches Produkt. Man kann Software nicht anfassen.

Die Konsequenzen dieser trivial scheinenden Feststellung sind vielfältig:

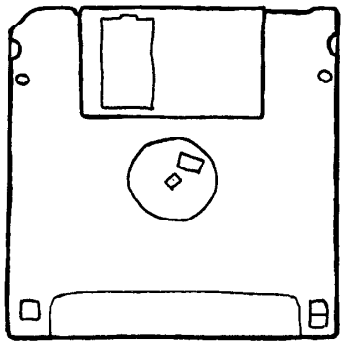
- Geistige Güter werden in unserer Gesellschaft in der Regel als *weniger wertvoll* eingestuft als mit dem gleichen Aufwand erstellte materielle Güter: Der Wert von Software wird *unterschätzt*.
- Im Gegensatz zu materiellen Produkten gibt es für Software keine natürlichen Grenzen in Form von Materialeigenschaften oder Naturgesetzen. Es gibt einzig die theoretische Grenze der Berechenbarkeit, aber diese wirkt sich in der Praxis kaum aus. Während ein Brückenbauingenieur bei seinen Konstruktionen Rücksicht nehmen muss auf die Eigenschaften der verwendeten Materialien und auf die Gesetze der Statik und Schwingungsdynamik, ist ein Software-Entwickler grundsätzlich in der Wahl seiner Konstruktionen frei. Software findet ihre praktischen Grenzen nur in der Begrenztheit des Könnens ihrer Entwickler. Da diese Grenzen sehr

weit gesteckt sind und Menschen außerdem dazu neigen, die Begrenztheit ihres Könnens zu verdrängen, werden bei Software häufiger und leichtsinniger (zu) schwierige Vorhaben angegangen als in anderen technischen Disziplinen.

- Bei der Entwicklung materieller Produkte sind viele *Fehler* leicht als solche erkennbar. Wenn beim Bauen das Dach statt des Kellers in die Baugrube gesetzt wird, so ist dies unschwer als Fehler zu erkennen. Ein Fehler ähnlicher Schwere in Software ist nur durch sorgfältige und aufwendige Prüfmaßnahmen zuverlässig erkennbar oder verhütbar (Bild 1.5).
- Gleiches gilt für die Beurteilung des *Entwicklungsstands*: Behauptet ein Bauunternehmen, der Bau sei praktisch fertig, obwohl erst der Rohbau steht, so ist leicht feststellbar, dass diese Behauptung nicht stimmt. Die Beurteilung des Fertigstellungsgrads von Software ist ungleich schwieriger und aufwendiger.
- Software ist scheinbar sehr flexibel und leicht zu ändern. Es gibt ja kein Material, das dabei umgeformt oder gar neu produziert werden muss. Kleinst-Software ist tatsächlich leicht änderbar. Mit zunehmender Größe der Software hat jede Änderung jedoch so viele Effekte und Konsequenzen, die alle bedacht sein müssen, dass die tatsächlichen Aufwendungen für Software-Änderungen oft höher sind als diejenigen für vergleichbare materielle Produkte.



Diese mechanische Konstruktion ist offensichtlich falsch.



Wie erkennen wir aber Fehler in der hier gespeicherten Software-Konstruktion?

BILD 1.5. Software kann man nicht anfassen.

FESTSTELLUNG 1.3. Software verhält sich (im mathematischen Sinn) unstetig.

Software ist Bestandteil eines digital arbeitenden Rechnersystems. Solche Systeme haben die unangenehme Eigenschaft, dass kleinste Veränderungen in Programmen oder Daten (Im Extremfall genügt schon die Setzung eines Punkts anstelle eines Kommas) massive Veränderungen im Verhalten des Systems bewirken können. Es ist daher ungleich schwieriger und aufwendiger, das wunschgemäße Funktionieren von Software mit einer gegebenen Wahrscheinlichkeit zu gewährleisten, als dies bei Produkten der klassischen technischen Disziplinen der Fall ist.

1.1.3 Wozu dient Software?

Software dient dazu, ein Problem zu lösen oder zu dessen Lösung beizutragen, indem menschliche oder technische Arbeitsvorgänge automatisiert oder unterstützt werden. Software steht daher in einer ständigen Wechselwirkung mit Arbeits- und Produktionsprozessen und mit den daran beteiligten Menschen. Daraus folgen drei weitere wesentliche Eigenschaften von Software.

FESTSTELLUNG 1.4. Wenn ein Problem von seiner Natur her komplex und schwierig zu lösen ist, so ist die Software zur Lösung dieses Problems in der Regel nicht weniger komplex und schwierig.

FESTSTELLUNG 1.5. Bei der Lösung von Problemen mit Software sind immer zwei Schwierigkeiten gleichzeitig zu bewältigen: (1) Das Problem ist im Kontext seines Sachgebiets zu verstehen und befriedigend zu lösen. (2) Die Problemlösung muss auf adäquate Software-Strukturen abgebildet werden.

FESTSTELLUNG 1.6. Problemlösungen schaffen neue Realitäten und wecken neue Bedürfnisse.

Software ist daher nicht einfach ein Abbild der Realität und bisheriger manueller Problemlösungen. Sie *konstruiert und verändert* die Realität. Beispielsweise ermöglicht Logistik-Software die Fertigung von Gütern nach dem "Just in time"-Prinzip (Zulieferteile werden genau dann angeliefert, wenn sie in der Produktion benötigt werden). Dies schafft streng termingebundenen Lastwagenverkehr als neue Realität und softwaregestützte Optimierung solchen Verkehrs als neues Bedürfnis.

1.2 Software-Entwicklung

1.2.1 Definition

Software-Entwicklung umfasst alle Tätigkeiten und Ressourcen, die zur Herstellung von Software notwendig sind.

DEFINITION 1.2. *Software-Entwicklung*. Die Umsetzung der Bedürfnisse von Benutzern in Software. Umfasst Spezifikation der Anforderungen, Konzept der Lösung, Entwurf und Programmierung der Komponenten[1)] , Zusammensetzung der Komponenten und ihre Einbindung in vorhandene Software, Inbetriebnahme der Software sowie Überprüfung des Entwickelten nach jedem Schritt.

1.2.2 Einige grundlegende Gesetzmäßigkeiten

In diesem Abschnitt zeigen wir einige grundlegende Gesetzmäßigkeiten für die Entwicklung von Software, die wesentlich dazu beitragen, dass Software-Entwicklung etwas Schwieriges ist.

FESTSTELLUNG 1.7. Die Entwicklung von Klein-Software unterscheidet sich fundamental von der Entwicklung größerer Software.

Letzteres stellt dabei den Normalfall dar. Tabelle 1.1 charakterisiert die Unterschiede. Viele Probleme existieren für Klein-Software gar nicht. Die Erfahrung, dass die Entwicklung kleiner Programme recht einfach ist, ist der Hauptgrund für den Trugschluss vieler Leute, dass Software-Entwicklung generell etwas Einfaches sein müsse.

REGEL 1.1. Der Aufwand für die Erstellung von Software steigt mit wachsender Produktgröße überproportional an. Nach Boehm (1981) wird der Zusammenhang beschrieben durch eine Gleichung der Art $A = bP^m$

Dabei ist A der Aufwand, P die Größe der Software, b ein konstanter Faktor und m ein Exponent im Bereich zwischen 1,05 und 1,2.

Die Regel 1.1 ist eine der wenigen, welche empirisch erhärtet sind (vgl. Boehm (1981) und Kapitel 5).

---

1) Dabei wird unter «Programmieren» sowohl das Schreiben neuer Programme wie auch das Erweitern oder Modifizieren vorhandener Programme verstanden.

Ursachen für dieses überproportionale Wachstum sind unter anderem, dass der Aufwand zur Beherrschung der wachsenden Komplexität und der Kommunikationsaufwand überproportional wachsen. Das Wachstum einzelner Faktoren ist dabei nicht kontinuierlich, sondern weist Quantensprünge auf (Bild 1.6).

TABELLE 1.1. Klein-Groß-Gegensätze in der Software-Entwicklung

klein	**groß**
Programme von 1 bis ca. 300 Zeilen Länge	Längere Programme
Für den *Eigengebrauch*	Für den Gebrauch durch *Dritte*
Vage Zielsetzung genügt, das Produkt ist seine eigene Spezifikation	*Genaue Zielbestimmung*, d.h. die Spezifikation von Anforderungen, erforderlich
Ein Schritt vom Problem zur Lösung genügt: die Lösung wird direkt programmiert	*Mehrere Schritte* vom Problem zur Lösung erforderlich: Spezifikation der Anforderungen, Konzept der Lösung, Entwurf der Teile, Programmieren der Teile, Zusammensetzen der Teile, Inbetriebnahme
Validierung (Feststellen, ob die Software sich den Erwartungen entsprechend verhält) und nötige Korrekturen finden am *Endprodukt* statt	*Auf jeden Entwicklungsschritt muss ein Prüfschritt* folgen, sonst wird das Risiko, dass das Endergebnis unbrauchbar ist, viel zu groß
Eine Person entwickelt: Keine Koordination mehrerer beteiligter Personen erforderlich, keine Kommunikationsbedürfnisse	*Mehrere Personen* entwickeln gemeinsam: Koordination und Kommunikation notwendig
Komplexität des Problems in der Regel *klein*, Strukturieren der Software und Behalten der Übersicht nicht schwierig	*Komplexität* des Problems *größer bis sehr groß*, explizite Maßnahmen zur Strukturierung und Modularisierung erforderlich
Software besteht aus *wenigen Komponenten*	Software besteht aus *vielen Komponenten*, die spezielle Maßnahmen zur Komponentenverwaltung erfordern
In der Regel wird *keine Dokumentation* erstellt	*Dokumentation* dringend *erforderlich*, damit Software wirtschaftlichen betrieben und gepflegt werden kann
Keine Planung und *Projektorganisation* erforderlich	*Planung* und *Projektorganisation zwingend* erforderlich für eine zielgerichtete, wirtschaftliche Entwicklung

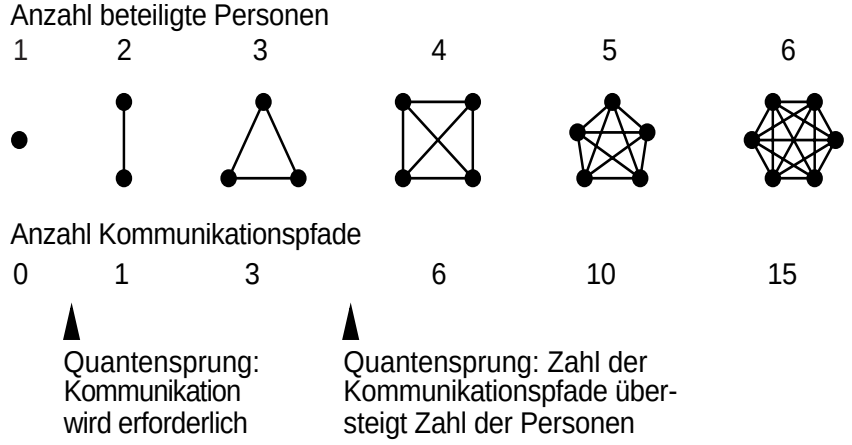


BILD 1.6. Wachstum und Quantensprünge beim Kommunikationsbedarf

FESTSTELLUNG 1.8. Software ist einer Evolution unterworfen (vgl. Kapitel 3.1).

Die Umwelt und die Bedürfnisse der Benutzenden verändern sich ständig. Software, die sich in Gebrauch befindet, bleibt daher ohne ständige Anpassungen und Erweiterungen nicht gebrauchstauglich. Ferner werden im Betrieb auch immer wieder Fehler entdeckt, die behoben werden müssen. Solche Entwicklungsarbeiten an in Betrieb befindlicher Software werden *Pflege* oder *Wartung* genannt.

Das Tempo der Software-Evolution ist so hoch, dass häufig schon während der Entwicklung die Entwicklungsziele an veränderte Bedürfnisse angepasst werden müssen.

Die erforderlichen Anpassungen und Erweiterungen bei der Pflege bringen es mit sich, dass Umfang und innere Unordnung von in Gebrauch befindlicher Software ständig zunehmen. Die Pflege einer Software wird daher mit zunehmendem Alter immer schwieriger und teurer. Dies ist ein Effekt, den man auch in anderen Disziplinen beobachten kann. Wird zum Beispiel ein Haus über Jahrzehnte hinweg ständig erweitert und umgebaut, so wird es immer größer und unübersichtlicher. Je planloser und je häufiger an- oder umgebaut wird, desto schwieriger ist es, den Bau so auszuführen, dass nicht Teile des Gebäudes dabei einstürzen oder danach ihren Verwendungszweck nicht mehr erfüllen.

FESTSTELLUNG 1.9. Software wird von Menschen gemacht.

Die Fähigkeiten dieser Menschen ebenso wie ihre emotionalen Einstellungen und alle ihre Unzulänglichkeiten haben einen direkten und erheblichen Einfluss auf die von ihnen entwickelte Software (vgl. DeMarco und Lister 1991, Glinz 1988 und Weinberg 1971).

1.2.3 Warum ist Software-Entwicklung schwierig?

Wenn wir obenstehende Feststellungen und Regeln und die ihnen zugrundeliegenden Phänomene analysieren, so stellen wir fest, dass es Wesentlichen vier Faktoren sind, welche die Entwicklung von Software schwierig machen. Es sind dies

- Die Größe der zu lösenden Probleme. Software ist nicht einfacher, als die Probleme, die sie löst. Je größer und schwieriger die Software, desto aufwendiger und schwieriger ist ihre Entwicklung. Fortschritte bei der Beherrschung des Software-Entwicklungsprozesses werden durch das Angehen größerer und schwierigerer Probleme immer wieder kompensiert.
- Die Tatsache, dass Software ein immaterielles Produkt ist. Die Immaterialität macht das Arbeiten mit Software schwieriger als dasjenige mit materiellen technischen Produkten vergleichbarer Komplexität. Die Risiken sind schwieriger zu erkennen; ad-hoc-Vorgehensweisen führen schneller ins Desaster.
- Sich permanent verändernde Ziele aufgrund der Evolution. Schon das Bestimmen und Erreichen fixierter Ziele bei der Entwicklung eines Produkts ist keine leichte Aufgabe. Sich verändernde Ziele machen das Ganze noch um eine Größenordnung schwieriger.
- Fehler infolge von emotionalen Fehleinschätzungen (Immaterielles hat keinen Wert und ist total flexibel, was im Kleinen geht, geht genauso im Großen, etc.). Software-Entwicklung wird daher unbewusst-emotional meist als viel einfacher eingeschätzt, als sie tatsächlich ist. Dies führt zu unrealistischen Erwartungen und zu von Beginn weg zu tiefen Kosten- und Terminschätzungen. Eine besonders häufige Fehlerursache ist das Denken und Handeln in der Welt von Klein-Software, während tatsächlich große Software zu entwickeln ist (vgl. Tabelle 1.1). Besonders verheerend wirkt sich die lineare Extrapolation von Erfahrungen mit Klein-Software auf große Software aus (Bild 1.7).

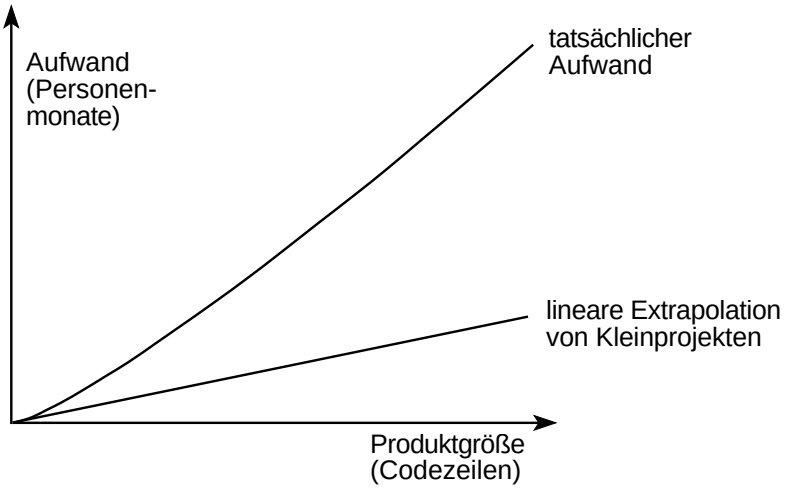


BILD 1.7. Fehleinschätzungen bei linearer Extrapolation

1.3 Software Engineering

In der Anfangszeit der Informatik gab es mit der Entwicklung von Software kaum Probleme. Dies ist nicht weiter verwunderlich, denn die Software bestand aus einzelnen, weitgehend voneinander unabhängigen Programmen, die jedes für sich eine beherrschbare Größe hatten. Die Schwierigkeiten der Software-Entwicklung manifestierten sich erst in den 60er Jahren, als immer umfangreichere Software entwickelt wurde und die bis anhin verwendeten ad-hoc-Vorgehensweisen zunehmend versagten. Zur Überwindung dieser *Software-Krise* erhob Friedrich Ludwig Bauer 1968 die Forderung nach *Software Engineering*, d.h. einem systematischen Vorgehen bei der Entwicklung, wie es in klassischen Ingenieurdisziplinen seit langem üblich ist.

Dabei muss man wissen, dass zur damaligen Zeit allgemein die Auffassung herrschte, das Schreiben von Software sei eine Kunst und erfordere daher individuelle künstlerische Freiheit für die Programmierer. Auf diesem Hintergrund gesehen war Bauers Forderung revolutionär.

In der Zwischenzeit hat sich die Erkenntnis weitestgehend durchgesetzt, dass bei nichttrivialen Aufgaben eine wirtschaftliche und termintreue Entwicklung qualitativ guter Software ohne Software Engineering nicht möglich ist.

Trotz aller seither erzielter Fortschritte ist uns die Software-«Krise» treu geblieben. Das hat verschiedene Gründe. Einerseits zeigt sich, dass gegen gesicherte Erkenntnisse des Software Engineerings immer wieder verstoßen wird. Die Ursachen dafür liegen (neben Unkenntnis) meistens in emotional bedingten Fehleinschätzungen. Andererseits werden Fortschritte dazu benutzt, immer größere (und schwierigere) Software-Systeme zu entwickeln (vgl. Abschnitt 1.2.3).

Rückblickend wird klar, dass es sich um keine Krise gehandelt hat, sondern um die erste Manifestation der grundsätzlichen Schwierigkeit der Entwicklung großer Software.

Die von Fairley (1985) gegebene Definition gibt das heutige Verständnis von Software Engineering sehr gut wieder:

DEFINITION 1.3. *Software Engineering* ist das technische und planerische Vorgehen zur systematischen Herstellung und Pflege von Software, die zeitgerecht und unter Einhaltung der geschätzten Kosten entwickelt bzw. modifiziert wird.

Das IEEE Standard Glossary of Software Engineering Terminology (IEEE 610.12, 1990) nimmt das quantitative, auf gemessenen Größen basierene Arbeiten als weiteres Kriterium hinzu.

DEFINITION 1.4. *Software Engineering*. Die Anwendung eines systematischen, disziplinierten und quantifizierbaren Ansatzes auf die Entwicklung, den Betrieb und die Wartung von Software, das heißt, die Anwendung der Prinzipien des Ingenieurwesens auf Software (IEEE 610.12).

Im deutschen Sprachraum ist anstelle von Software Engineering auch der Terminus *Softwaretechnik* gebräuchlich.

1.4 Ziele und Mittel des Software Engineerings

Mit Software Engineering werden drei grundsätzliche Ziele verfolgt.

Die *Produktivität* bei der Herstellung von Software soll gesteigert werden. Angesichts der horrenden Summen, die jährlich weltweit für Software ausgegeben werden (Bild 1.2), sind auch kleine Produktivitätssteigerungen von großem wirtschaftlichem Interesse.

Die *Qualität* der erstellten Software soll verbessert werden. Dies bringt neben Wettbewerbsvorteilen durch zufriedene Kunden auch Kostensenkungen bei der Pflege und Weiterentwicklung von in Betrieb befindlicher Software.

Die *Führbarkeit* von Software-Entwicklungsprojekten soll erleichtert werden. Dies bringt Erleichterungen durch bessere Termin- und Kostentreue und erleichtert die Kontrolle der Projektrisiken. Letztlich bedeutet dies Senkung der Kosten bei verbesserter Qualität.

Die Mittel, die zur Erreichung dieser Ziele zur Verfügung stehen, sind in Bild 1.8 skizziert. Alle folgenden Kapitel sind der näheren Beschreibung dieser Mittel gewidmet.

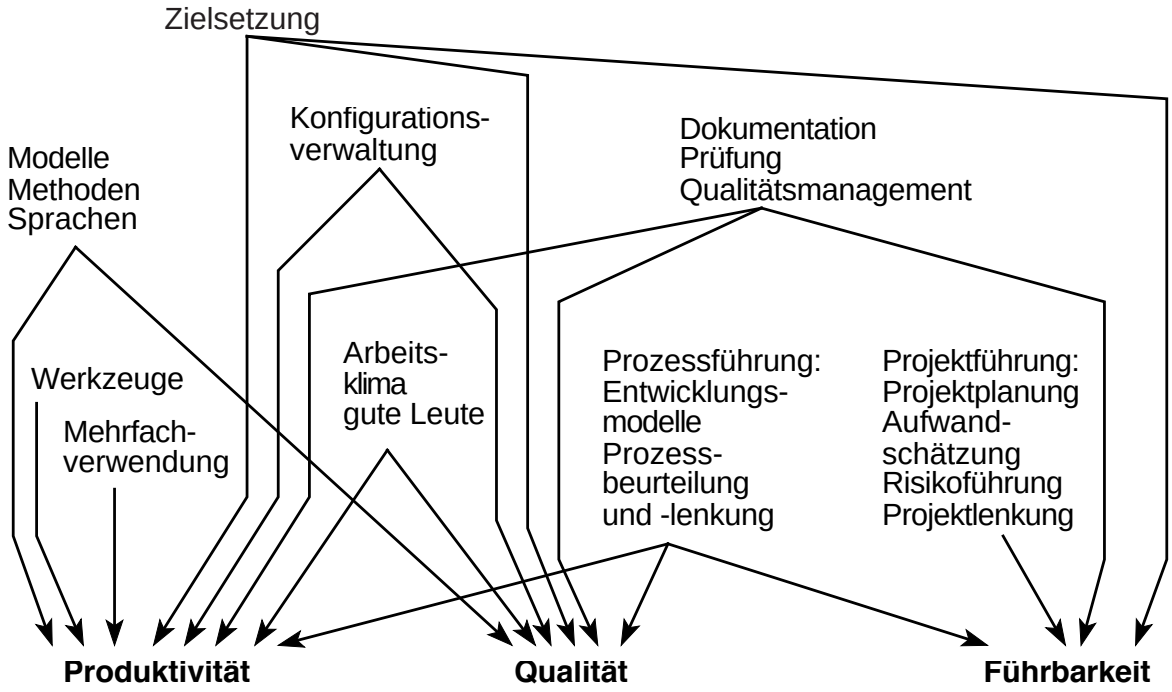


BILD 1.8. Mittel des Software Engineerings und ihre Wirkung

Aufgaben

1.1 Begründen Sie die vier Hauptschwierigkeiten bei der Entwicklung von Software aus den Feststellungen 1.1 bis 1.9 und der Regel 1.1.

1.2 «Ein Mann braucht zum Bau einer 2 m langen Brücke 0,5 Tage. Wie lange brauchen 100 Leute für den Bau einer 2 km langen Brücke? Rechne.»
Begründen Sie, warum das eine Milchmädchenrechnung ist. Ziehen Sie Parallelen zur Entwicklung von Software.

1.3 Was sind die drei grundlegenden Ziele des Software Engineerings?

1.4 Eine Kundenbetreuerin im Firmenkundengeschäft einer Bank hat auf der Grundlage eines Tabellenkalkulationsprogramms eine kleine persönliche Anwendung geschrieben, die sie bei der Überprüfung der Kredite der von ihr betreuten Firmen unterstützt. Die notwendigen Daten gibt sie jeweils von Hand ein. Der Abteilungsleiter sieht diese Anwendung zufällig, ist davon angetan und beschließt, sie allen Kundenbetreuerinnen und -betreuer zur Verfügung zu stellen. Die notwendigen Daten sollen neu automatisch als den Datenbanken der Bank übernommen werden.
Die Kundenbetreuerin gibt an, für die Entwicklung ihrer Anwendung insgesamt etwa vier Arbeitstage aufgewendet zu haben. Der Abteilungsleiter veranschlagt daher für die Übernahme und die gewünschten Änderungen einen Aufwand von einer Arbeitswoche. Als die geänderte Anwendung endlich zur Zufriedenheit aller Beteiligten läuft, sind jedoch rund acht Arbeitswochen Aufwand investiert.
Der Abteilungsleiter erzählt die Geschichte einem befreundeten Berater als Beispiel, dass Informatik-Projekte nie ihre Termine einhalten. Darauf meint der Berater trocken, der investierte Aufwand sei völlig realistisch und normal. Begründen Sie, warum.

1.5 Ein Unternehmen stellt Messgeräte her, in denen Messungen und Gerätebedienung mit Hilfe von Software erfolgen. Bei einem in Entwicklung befindlichen neuen Gerät findet die für die Hardware verantwortliche Abteilung heraus, dass pro Gerät 20 Franken eingespart werden können, wenn der geplante Prozessor durch einen primitiveren ersetzt wird. Der Verkauf rechnet damit, während 5 Jahren je etwa 400 Geräte abzusetzen; es kann also eine Einsparung von ca. 40'000 Franken erwartet werden. Die Änderung wird daher beschlossen, kleine Anpassungen in der Software sind ja kein Problem. Als die Softwareentwickler zwei Monate später mit ersten Tests auf dem Zielsystem beginnen wollen, stellt sich heraus, dass ihr bisher immer verwendetes Echtzeit-Betriebssystem auf dem neuen Prozessor nicht läuft und dass der stattdessen angebotene Echtzeit-Kern für ihre Bedürfnisse nicht ausreicht. Durch die Verzögerung und die notwendigen Software-Erweiterungen des Echtzeit-Kerns entstehen Mehrkosten von ca. 100'000 Franken.
Geben Sie Gründe an, wie es zu einer solchen Fehlentscheidung kommt. Wie hätte der Fehler vermieden werden können?

Anhang: Schätzung der weltweit für Software ausgegebenen Summen

Die Schätzung der jährlich für Software ausgegebenen Beträge ist schwierig. Marktforschungsfirmen weisen als Software-Kosten in der Regel nur die Kosten für den Kauf von Software-Lizenzen aus. Diese machen jedoch nur einen Bruchteil der gesamten Software-Kosten aus. Aufschlussreicher sind die Zahlen zu den IT- (information technology) Kosten. Diese enthalten allerdings auch die Kosten für die Hardware (ca. 10 Prozent) und die Kommunikationsinfrastruktur (ca. 12 Prozent). Andererseits sind die gesamten Kosten für Software in Produkten (Geräte, Maschinen, Verkehrsmittel, etc.) und Anlagen (Industrieanlagen, Kraftwerke, Verkehrsinfrastruktur, etc.) in den IT-Kosten *nicht* enthalten.

Zudem variieren die Schätzungen erheblich. Für das Jahr 2003 beispielsweise liegen die Angaben von vier verschiedenen Marktforschungsunternehmen zu den IT-Kosten in den USA zwischen 407 und 864 Milliarden Dollar. Geht man davon aus, dass die Software-Kosten (Lizenzen sowie Entwicklung und Pflege von Software, aber ohne Betriebskosten) mindestens zwei Drittel der gesamten IT-Kosten ausmachen und die USA einen Anteil von ca. 40 Prozent an den weltweiten Kosten haben, so werden weltweit allein im IT-Bereich über eine Billion (10^{12}) Euro pro Jahr für Beschaffung, Entwicklung und Pflege von Software ausgegeben.

Ferner ist zu beachten, dass die Kosten für Software in Produkten und Anlagen einen erheblichen Anteil an den Gesamtkosten für Software haben. Dem Verfasser liegen hierzu leider keine zuverlässigen Angaben vor. Die folgenden Zahlen mögen aber als Anhaltspunkt dienen:

Im Jahr 1997 betrug das IT-Budget des US Verteidigungsministeriums (DoD) rund 10 Milliarden Dollar, während das DoD im gleichen Zeitraum zwischen 20 und 30 Milliarden Dollar für Software in Waffensystemen ausgab.

Ergänzende und vertiefende Literatur

Brooks (1995) liefert in einer Sammlung von Essays eine unterhaltsame und lesenswerte Beschreibung von Fehlern im Software Engineering und ihren Ursachen. Das Buch von 1995 ist eine Neuausgabe des Originals von 1975 mit vier neuen Kapiteln. Dieses Buch ist ein zeitloser Klassiker. Wegen der zusätzlichen Essays wird empfohlen, die Neuausgabe zu lesen.

IEEE 610.12 (1990) ist die Standardreferenz für die englischsprachige Terminologie im Software Engineering. Viele Definitionen in diesem Text sind Übersetzungen aus IEEE 610.12 oder lehnen sich an diese an.

Lehman und Belady (1985) verdanken wir die Erkenntnisse über die Software-Evolution.

McDermid (1991) behandelt alle Gebiete der Informatik mit Bezug zum Software Engineering einschließlich der Grundlagen- und Randgebiete in knappen Übersichtsartikeln in der Art einer Enzyklopädie, aber nach Sachgebieten, nicht nach Stichworten geordnet.

Der Begriff «Software Engineering» wurde von Bauer auf einer NATO-Konferenz in Garmisch-Partenkirchen geprägt. Der Tagungsband dieser Konferenz (Naur, Randell und Buxton 1976) gibt einen Eindruck der damaligen Situation.

Weinberg (1971), DeMarco und Lister (1991) und Glinz (1988) behandeln Probleme der Software-Psychologie, der Menschenführung in der Software-Entwicklung und den Gegensätzen zwischen rationalen und emotionalen Einstellungen im Software Engineering.

Endres und Rombach (2003) haben sich die Mühe gemacht, das empirisch erhärtete Wissen über Software Engineering systematisch in einem Buch zusammenzutragen.

Zitierte Literatur

Boehm, B, (1973). Software and Its Impact: A Quantitative Assessment. *Datamation* May 1973. 48-59.

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

Boehm, B. (1987). Improving Software Productivity. *IEEE Computer* **20**, 9 (Sept. 1987). 43-57.

Brooks, F.P. (1995). *The Mythical Man Month. Essays on Software Engineering*. Anniversary Edition Reading, Mass., etc.: Addison-Wesley.

DeMarco, T., T. Lister (1991). *Wien wartet auf Dich! Der Faktor Mensch im DV-Management*. München-Wien: Hanser.

Endres, A., D. Rombach (2003). *A Handbook of Software and Systems Engineering*. Harlow, England, etc.: Pearson Addison-Wesley.

Fairley, R. (1985). *Software Engineering Concepts*. New York, etc.: McGrawHill.

Glinz, M. (1988). Emotionales und Rationales im industriellen Software Engineering. *Technische Rundschau* 20/88, 78-81.

IEEE (1990). *Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12-1990. IEEE Computer Society Press.

Lehman, M.M., L.A. Belady (Hrsg.) (1985). *Program Evolution: Processes of Software Change*. London, etc.: Academic Press.

Naur, P., B. Randell, J.N. Buxton (Hrsg.) (1976). Software engineering: concepts and techniques: proceedings of the NATO conferences. (Neuausgabe der Proceedings der NATO-Konferenz von 1968 und der Nachfolgekonferenz von 1969). New York: Petrocelli.

McDermid, J. (1991). *Software Engineer's Reference Book*. Oxford: Butterworth-Heinemann. ca. 1200 p.

Oz, E. (1994). When Professional Standards are Lax. The CONFIRM Failure and its Lessons. *Communications of the ACM* **37**, 10 (Oct 1994). 29-36.

Weinberg, G.M. (1971). *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold.

Lions, J.L. (1996). *ARIANE 5 Flight 501 Failure*. Report by the Inquiry Board. Paris: ESA. http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf