Things are more complex than they seem to be.
Things take longer than expected.
Things cost more than expected.
If something can go wrong, it will.

Murphyls Law

1. Programmierung digitaler Rechner

1.1. Einige Begriffe

Durch die sehr standardisierte gewählte Aufbaustruktur digitaler Rechner wird deren Einsatz in den unterschiedlichsten Anwendungsgebieten ermöglicht. Diese sehr allgemein gehaltene Aufbaustruktur bewirkt jedoch auch, dass aus der reinen physikalischen Existenz eines digitalen Rechners nicht dessen sofortige Einsatzmöglichkeit folgt. Vielmehr ist es vorher unumgänglich, sich eine Arbeitsvorschrift auszudenken, die auf den geplanten speziellen Einsatz zugeschnitten ist und diese dem Rechner mitzuteilen. Diese Vorgehensweise hat zwei Vorteile.

- Der physikalische Aufbau des digitalen Rechners ist für sehr viele Anwendungsgebiete gleich. Damit ist eine billige Massenproduktion möglich.

- Die nachträgliche Erweiterung oder Änderung des speziellen Einsatzgebietes erfordert in den meisten Fällen hauptsächlich die Änderung der Arbeitsvorschrift. Der physikalische Aufbau des Rechners bleibt unverändert.

Es gibt in der englischen Fachsprache zwei unübersetzbare Fachworte, die diesen Sachverhalt beschreiben:

- Die **Hardware** eines digitalen Rechners umfasst all seine physikalisch existierenden Teile, so also z. B. die Zentraleinheit, die Ein-/Ausgabegeräte, die Prozessankopplung und auch die Datenträger wie CD-ROMs, Disketten usw.
- Die **Software** umfasst die Gesamtheit aller Arbeitsvorschriften für einen digitalen Rechner, stellt also reines Gedankengut ihres Entwicklers dar. Sie gibt an, auf welche Art und Weise die Hardwarekomponenten mit einem Bediener oder einem

technischem Prozess zusammenarbeiten sollen und ergänzt diese so zu einem funktionsfähigen Rechnersystem.

Die Gesamtheit aller für einen digitalen Rechner existierenden Software besteht aus einer Vielzahl von Programmen. Ein Programm besteht wiederum aus einer sequentiellen Folge logisch aufeinander abgestimmter Anweisungen, die in ihrem Zusammenwirken eine Lösungsvorschrift für ein spezielles Problem darstellen. Eine Anweisung ist somit eine Arbeitsvorschrift, die einen nicht weiter aufspaltbaren Teil der Aufgabe in einer vereinbarten Sprache, einer Programmiersprache, formuliert.

1.2 Die Softwarekrise

Der heutige Stand der Entwicklung im Bereich Datenverarbeitung ist gekennzeichnet durch zwei gegensinnig laufende Entwicklungen. Auf der einen Seite zeigt es sich zunehmend, dass durch Massenproduktion und Einsatz immer besserer Technologien die Hardwarekosten rapide fallen. Gleichzeitig steigen Integrationsdichte, Leistungsumfang und Schnelligkeit der einzelnen Chips. So existiert heute für jede Standardanwendung eines digitalen Rechners auf der Hardwareseite eine Schubladenlösunq.

Diese Tatsache gilt auf der anderen Seite jedoch nicht für die Software. Die unterschiedliche Art von Problemstellungen in den unterschiedlichen Anwendungen bringt es mit sich, dass oft die gesamte Software für eine neue Anwendung neu zu erstellen ist.

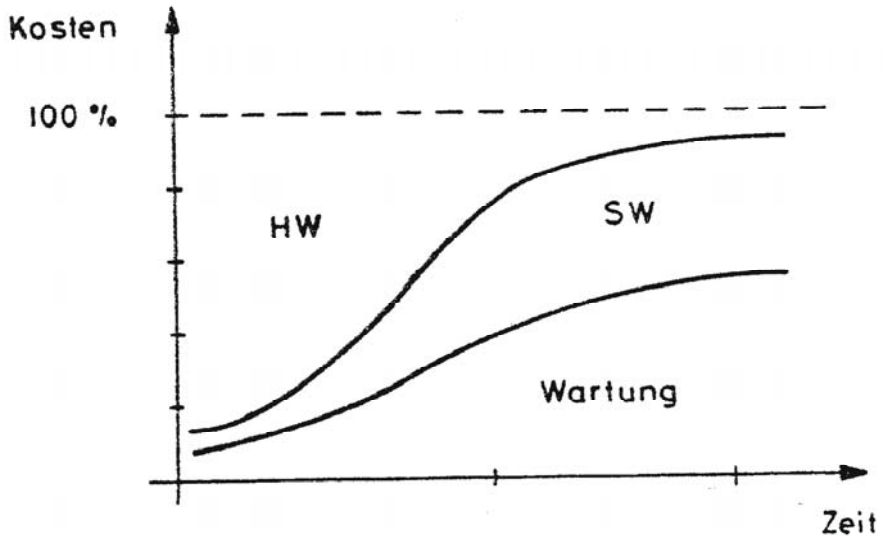

Bild 1.1 Verhältnis von Hardware- und Softwarekosten bei Datenverarbeitungsprojekten

Verschärfend zu diesem Problem kommt hinzu, dass in den letzten Jahren die mittlere

Größe der auf Rechnern installierten Programmsysteme ständig und rapide zugenommen hat. Damit ist der Bereich Software der Engpass bei der Anwendung digitaler Rechner.

Drei charakteristische Größen dieser Entwicklung sollen im folgenden näher besprochen werden: die Entwicklungskosten, die Entwicklungszeitdauer und die Qualität. Die damit verbundenen Probleme sollen zunächst ohne Wertung und Deutungsversuch in Form von "Erfahrungen" aufgelistet werden.

1.3 Softwarekosten steigen rapide

Erfahrung 1:

Innerhalb von 30 Jahren Datenverarbeitungsentwick1ung hat eine Vertauschung von Hardware- und Softwarekosten stattgefunden. Bild 1.1 zeigt die Anteile von Hardware und Software an den Gesamtkosten von Datenverarbeitungsanwendungen.

Erfahrung 2:

Es ist außerordentlich schwer, die Kosten eines Software-Projekts zu dessen Beginn wenigstens einigermaßen exakt abzuschätzen. Erhebliche Fehleinschätzungen verunsichern den späteren Anwender.

Erfahrung 3:

Mit zunehmender Größe der Softwaresysteme steigen ihre Erstellungskosten nichtlinear.

Erfahrung 4:

Ohne zunächst zu spezifizieren, welche Arbeitsabläufe in der Softwarewartung durchzuführen sind, ist festzustellen, dass sie den größten Teil der Systemkosten im laufenden Einsatz ausmacht. Oft wird die Zahl von 200 % der Entwicklungskosten genannt, die zusätzlich an Wartungskosten für jedes Softwaresystem während dessen Lebensdauer aufgebracht werden müssen.

1.4 Softwaretermine gleiten

Erfahrung 1 :

Im Bereich der Softwareerstellung werden wesentlich häufiger falsche Termineinschätzungen gemacht als in anderen Ingenieurdisziplinen. Diesem Problem liegt das objektive Unvermögen zugrunde, zu Beginn des Projekts genaue Angaben zu machen, nicht etwa der triviale Grund der möglichen starken Konkurrenz auf dem Softwaremarkt, indem man durch wissentliche Angaben nicht haltbarer Termine versucht, einen Auftragszuschlag zu erhalten.

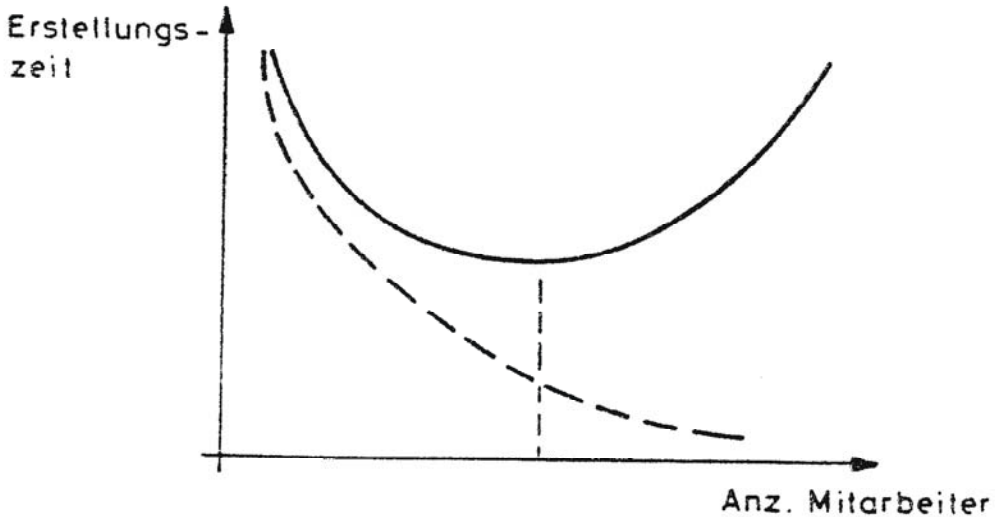

Bild 1.2 Softwareentwicklungszeit im Bezug zur Anzahl der Mitarbeiter

Erfahrung 2:

Bis heute fehlen exakte realistische Verfahren, mit deren Hilfe die Dauer eines Softwareprojekts berechnet werden kann. Derartige Angaben beruhen heute im wesentlichen auf der Schätzung eines erfahrenen Projektmanagers, der sich selten auf eine formal geführte Projektbibliothek abstützen kann. Sie sind damit eine intuitive (oft Über-) Schätzung der eigenen Fähigkeiten und derjenigen des Mitarbeiterteams.

Erfahrung 3:

Die Softwareentwicklungszeit in Abhängigkeit von der Anzahl der Mitarbeiter ist nicht, wie zunächst zu erwarten, eine Hyperbelfunktion. Vielmehr ergibt sich nach Bild 1.2 eine "Badewannenkurve" mit der Aussage, dass mit stetig wachsender Mitarbeiterzahl am gleichen Projekt die Systemerstellungszeit nicht nur stagniert, sondern sogar zunimmt.

1.5 Softwarequalität nimmt ab

Erfahrung 1:

Qualität ist ein Merkmal, dass sich am schlechtesten von allen hier aufzuzählenden Fakten in Zahlen ausdrücken lässt oder dass greifbare Merkmale erlaubt, nach denen sie messbar wäre.

Erfahrung 2 :

Die Softwarequalität mangelt oft mehr als nötig in folgenden Punkten:

- **Zuverlässigkeit:** ***P*:** Im Einsatz werden nicht unter allen vorgesehenen

Bedingungen die spezifizierten Leistungen erbracht

- **Benutzerfreundlichkeit:** Das System arbeitet ungenügend mit dem Benutzer zusammen, ist unkomfortabel, setzt dessen Motivation herab.
- **Rechnerunabhängigkeit:** Es erfordert hohen Aufwand, das Softwaresystem auf einem anderen, zunächst nicht vorgesehenen Rechner zu installieren.
- **Effizienz:** Bei seiner Arbeit verschwendet das Softwaresystem mehr als nötig Ressourcen wie Speicherplatz, Rechenzeit, Druckerpapier
- **Erweiterbarkeit:** Es erfordert hohen Aufwand, das Softwaresystem um zusätzliche, zunächst nicht geplante Anforderungen zu erweitern
- **Kompatibilität:** Die Ein- und Ausgabedaten des Softwaresystems sind nicht von anderen Systemen verarbeitbar.
- **Dokumentation:** Die Beschreibungen zum Softwaresystem sind fehlerhaft, unvollständig, schwer verständlich.

Erfahrung 3 :

Es ist unmöglich, alle Qualitätskriterien gleichzeitig zu erfüllen. Vielmehr führt ihre gleichzeitige Priorisierung zu Konflikten; es handelt sich also teilweise um gegensinnige Ziele. So z. B. steht die Effizienz im Gegensatz zur Rechnerunabhängigkeit und zur Benutzerfreundlichkeit. Selbst innerhalb des Merkmals Effizienz führt eine Speicherplatzoptimierung oft zu einer Laufzeitverlängerung (siehe hierzu verschiedene Compileroptimierungseinstellungen).

1.6 Gründe dieser Krise

Um die gezeigten Probleme zu analysieren und gegen sie vorzugehen, sollen nun plakativ Gründe zusammengestellt werden, die zu diesem Dilemma führen. Es handelt sich sowohl um fachliche wie auch um projektororganisatorische Gründe.

Grund 1 :

Der Software-Entwicklungsprozess ist nicht oder ungenügend in abgeschlossene Phasen mit definierten Ergebnissen aufgeteilt.

Grund 2 :

Die Analyse- und Entwicklungsmethoden sind unzureichend und nicht durchgängig für alle Phasen.

Grund 3 :

Regelmäßige Qualitätskontrollen während des Projektfortschritts sind unüblich.

Grund 4 :

Der Wissensstand aller beteiligten Personen ist unzureichend, die Einführung neuer Methoden wird abgelehnt.

Grund 5 :

Die beteiligten Personen entwickeln mangelnde Bereitschaft zur Kooperation.

Grund 6 :

Aufgrund von Schätzfehlern der Produkterstellungszeit stehen alle Mitarbeiter ständig unter hohem Zeitdruck.

Daraus folgt als anzustrebendes Ziel:

Das wirtschaftliche Entwickeln von qualitativ hochwertiger Software setzt Klarheit über Zielsetzung, Vorgehensweise und anzuwendende Methoden und Techniken bei Analyse, Planung und Entwicklung voraus. Software muss **ingenieurmäßig** wie andere technische Produkte entwickelt werden.

1.7 Ingenieurmäßige Softwareentwicklung

Um zu verstehen, warum (neben der Neuheit des Fachgebiets an sich) sich der Software-Entwicklungsprozess so sehr von der althergebrachten ingenieurmäßigen Arbeit unterscheidet und somit so wenig Methoden übernehmbar sind, sollen in der folgenden Zusammenfassung die Unterschiede zwischen der Entwicklung von Software und den anderen altbekannten Ingenieurdisziplinen zusammengestellt werden.

- **Software ist immateriell,** stellt also bis zuletzt reines Gedankengut ihres Entwicklers dar. Das erfordert ein vollständiges Umdenken des Ingenieurs, der bislang gewohnt war, mit materiellen Dingen zu arbeiten und umzugehen.
- **Software altert nicht in dem Sinne,** wie etwa elektronische Geräte oder Bauteile altern. Das Ende der Softwarenutzungsdauer ist vielmehr dann erreicht, wenn sich die äußeren Anforderungen so verschieben, dass das alte System unmodern geworden ist.
- **Software unterliegt keinem Verschleiß** und benötigt aus diesem Grund auch keine Ersatzteilhaltung. Die Softwarewartung im engeren Sinne beschränkt sich vielmehr auf die Beseitigung von Fehlern, die seit Anbeginn, also seit der Produktion, im System steckten. Im weiteren Sinne wird der Begriff Softwarewartung auch für Erweiterungen und Modifikationen während des Betriebs verwendet.
- **Software ist kein Serienprodukt** wie etwa ein Auto oder ein Fernsehgerät. Der gesamte Entwicklungsaufwand steckt in der Produktion des Prototyps. Der Zeit-, Arbeits- und Kostenaufwand für die Vervielfältigung ist vernachlässigbar.
- **Software besitzt einen hohen Komplexitätsgrad.** Gerade dieses Problem stellt

eine hohe Anforderung für den menschlichen Geist dar. Da Gedankengänge sequentiell ablaufen und der Mensch nur über kleine abgeschlossene Probleme intensiv nachdenken kann, ist es außerordentlich schwierig, alle Tätigkeiten und Entscheidungen, die innerhalb eines großen Softwaresystems ablaufen, gleichzeitig zu überblicken.

Kaum eine andere ingenieurmäßige Tätigkeit hat in einer ähnlich kurzen Zeit derartige Dimensionen erreicht wie die Softwareentwicklung. Das Schreiben eines Programms war zunächst nur ein Nebeneffekt bei Rechnerinstallationen, wurde nicht recht ernst genommen. Der Schwerpunktwechsel ging zu schnell vonstatten, überrollte die Entwickler.

Bild 1.3 verdeutlicht den Übergang von der individuellen zur industriellen Softwareerstellung, wie er von Brooks /1/ dargestellt wird.

Ausgangspunkt ist das Programm (links oben), das nur für einen Anwendungsfall und für einen Anwender, oft für den Ersteller selbst, geschrieben wird. Die Anforderungen an die Allgemeingültigkeit, an die Dokumentation und an die Handhabung sind somit minimal.

Der Übergang zum Programmprodukt ist gekennzeichnet durch die Weitergabe des Programms an mehrere Anwender. Der Aufwand in die Allgemeingültigkeit, d. h. die Berücksichtigung aller in diesem Anwendungsgebiet möglichen Fälle, steigt jetzt erheblich. Des weiteren sind Beschreibungen und Handbücher zu erstellen.

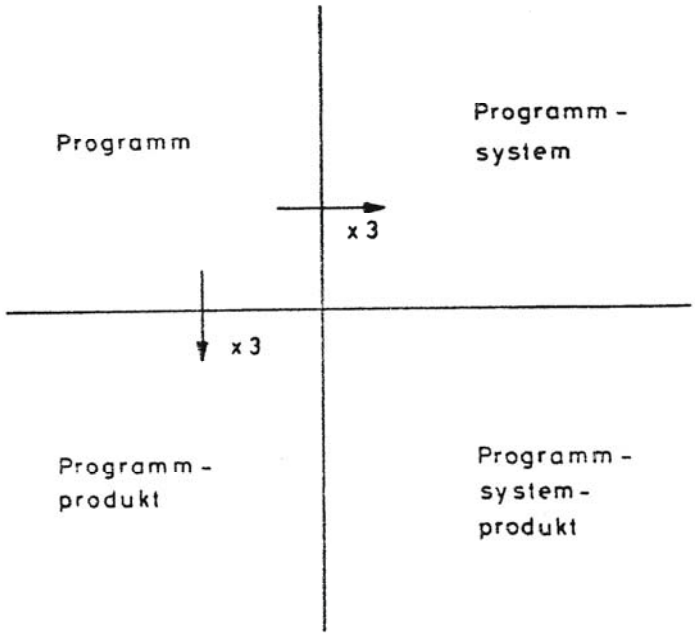

Bild 1.3: Programmarten und Aufwand

Ein Programmsystem ist die Zusammenfassung und das Zusammenwirken mehrerer Programme. Obwohl, wie beim Programm, nur ein Anwender vorausgesetzt wird, steigt der Aufwand erheblich, da die Programme in allen Einzelheiten aufeinander abzustimmen sind.

Die ingenieurmäßige Softwareerstellung hat nach dieser Nomenklatur die Produktion von Programmsystemprodukten zum Ziel.

Mehrere, oft viele Teilprogramme, werden für mehrere, oft viele Anwender erstellt. Im Vergleich zum Programm steigt nach Brooks der Aufwand (an Kosten und Zeit) etwa um den Faktor 10. Dies resultiert aus der vollständigen Berücksichtigung aller Anwendungsfälle für alle Programme, sowie aus der vollständigen Erzeugung des gesamten Dokumentations- und Benutzermaterials.

Gefordert ist also im Bereich der Software der Übergang von der adhoc-Lösung zur Softwareproduktion, mit dem vorrangigen Ziel der Einhaltung der spezifizierten Funktionen sowie der Einhaltung von Kosten und Termine.

Grundlage der Softwareproduktion ist der Einsatz allgemeiner Prinzipien und Methoden.

Darunter erfasst ist auch eine straffe Organisation der Softwareproduktion und damit eine Spezialisierung der Mitarbeiter.

Seit etwa 1970 gibt es eine eigenständige Ingenieurdisziplin mit dem Namen Software Engineering. Wie weit diese eigenständige Ingenieurdisziplin zunächst zu sich selbst finden musste, zeigen zwei Definitionen des Begriffs Software Engineering:

Software Engineering today is a collection of
prescriptions, techniques and disciplines
that show promise of bringing same order
into the chaos that is computer programming.

Software Engineering ist
die genaue Kenntnis und gezielte Anwendung
von Prinzipien, Methoden und Werkzeugen
für die Technik und das Management
der Software-Entwicklung und -Wartung

auf Basis wissenschaftlicher Erkenntnisse
und praktischer Erfahrungen,
sowie unter Berücksichtigung des
jeweiligen ökonomisch-technischen Zielsystems.

Diese zweite Definition beinhaltet drei grundsätzliche Begriffe, die der näheren Spezifikation bedürfen.

Ein **Prinzip** ist ein Grundsatz, den man seinem Handeln zugrunde legt.

Prinzipien bilden sich aus Erfahrung und aus Erkenntnis und bilden eine theoretische allgemein gültige Grundlage.

Eine **Methode** ist eine auf einem Prinzip aufbauende begründete Vorgehensweise zur Erreichung eines festgelegten Ziels. Methodisch vorgehen bedeutet also den Gegensatz zum Herumprobieren. Methoden sind konkreter Natur, etwa durch genaue Vorgabe von Arbeitsschritten.

Ein Werkzeug stellt ein Hilfsmittel zur Methodenunterstützung dar. Dieses Hilfsmittel wird eingeführt, um den Einsatz der Methode zu erleichtern, zu beschleunigen oder abzusichern. Ein Werkzeug ist oft mit Hilfe eines digitalen Rechners automatisiert.

Die im folgenden beschriebenen Prinzipien, Methoden und Werkzeuge haben sich mit der Zeit für die Herstellung von Programmsystemprodukten bewährt. Es schadet jedoch nichts, wenn auch der einzelne Programmierer sie zur Grundlage der Herstellung seiner privaten Programme verwendet. Eines Tages, wenn er sein Programm doch einmal weitergibt, oder wenn er sich nach längerer Pause in sein eigenes Programm neu einarbeiten will, wird er spätestens den Nutzen dieses Tuns erkennen.

Wohlauf, lasset uns hernieder fahren
und ihre Sprache daselbst verwirren
dass keiner des anderen Sprache verstehe.
Also mussten sie aufhören, die Stadt zu bauen.
Daher heißt ihr Name Babel,
dass der Herr daselbst verwirrt hatte aller Länder Sprache
und sie zerstreut von dort in alle Länder.

Die Bibel, 1. Mose 11

2. Der Software-Life-Cycle

2.1 Der allgemeine ingenieurmäßige Entwicklungsprozess

Bevor speziell auf das Vorgehen bei der ingenieurmäßigen Erstellung von Software eingegangen wird, soll zunächst eine Betrachtung des allgemeinen ingenieurmäßigen Vorgehens stattfinden, das in der Regel bei der Entwicklung eines neuen Produkts durchlaufen wird. Dieser Entwicklungsprozess ist schematisch in Bild 2.1 dargestellt.

Nach diesem Modell werden die Aktivitäten des Ingenieurs initialisiert durch die Erkennung eines Problems und natürlich durch den Wunsch und/oder den Auftrag, dieses Problem zu lösen. Der Ablauf startet dann mit einer Phase, in der das Problem formuliert wird. Vorrangig ist also hier das Verständnis; denn ohne die Sicherheit, das Problem vollkommen verstanden zu haben, ist es sinnlos, eine Lösung anzustreben. In diesem Stadium des Bemühens um Verständlichkeit besteht die wichtigste Aufgabe in der Sondierung der eintreffenden Informationen in die Kategorien

- zum Problem beitragend
- irrelevant
- irreführend.

Von Vorteil oder wünschenswert ist hier sicher die bei vorangegangenen ähnlichen Problemen gewonnene Erfahrung in Form von Aufzeichnungen oder auch Erinnerungen eines der damaligen Teilnehmer. Die damaligen Lösungen können bei Bedarf in das Verständnis des neuen Problems einfließen.

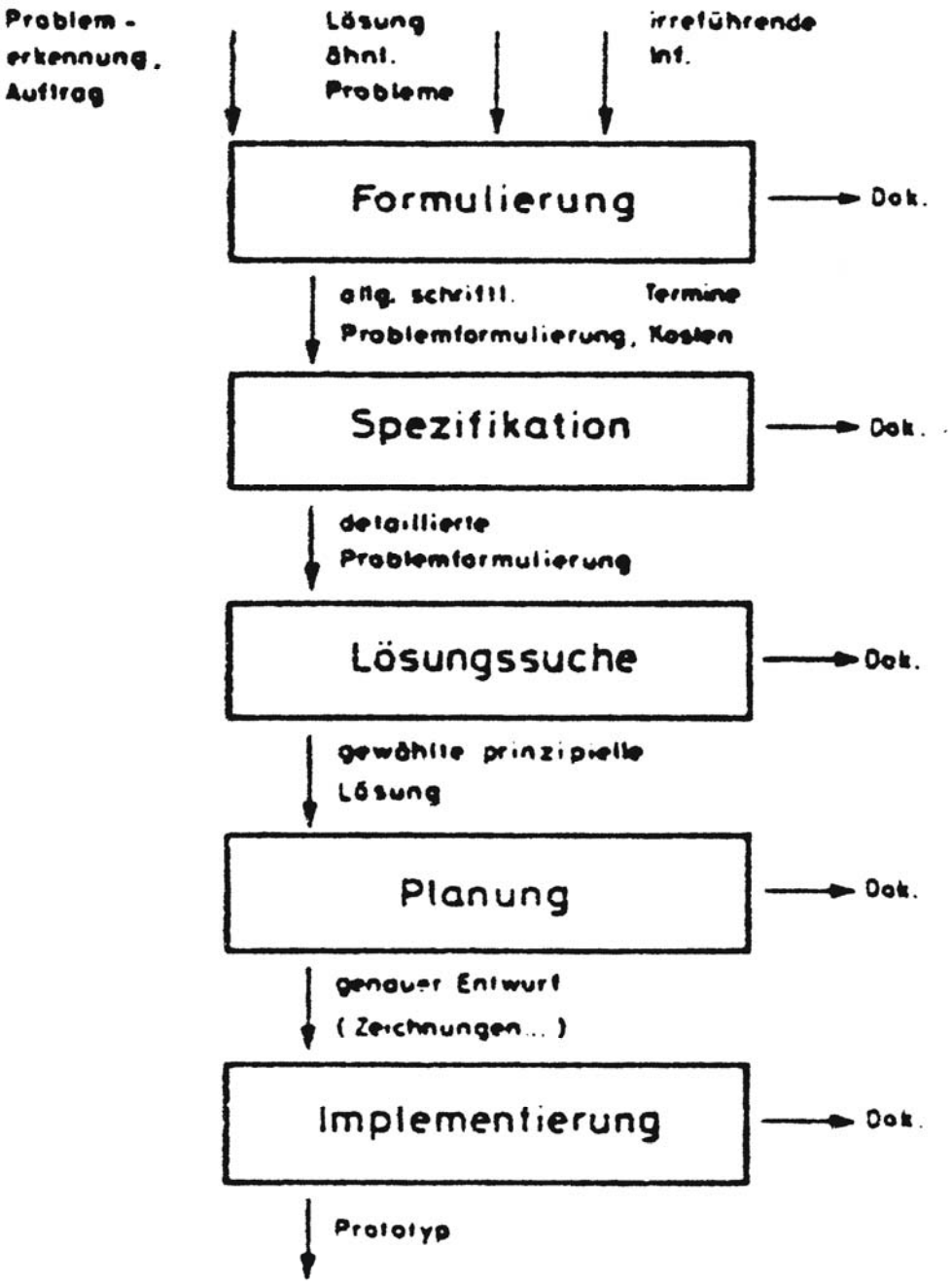

Bild 2.1: Allgemeiner ingenieurmäßiger Entwicklungsprozess

Ziel der Problemformulierung ist eine schriftliche Zusammenfassung, aus der die generellen Zusammenhänge hervorgehen, die jedoch nicht mit in dieser frühen Phase

unnötigen Details belastet ist. Zu drei grundsätzlichen Ergebnissen sollte jede Problemformulierung gelangen:

- **Durchführbarkeit der Problemlösung:** Eine Aussage, ob die Lösung des Problems generell und mit den vorhandenen Mitarbeiter- und Maschinenkapazitäten durchführbar ist

- **Projektdauer:** Eine Information Über die voraussichtliche Dauer des Projekts, mit der Festlegung der vorläufigen Termine für Zwischenergebnisse, im Fachjargon **Meilensteine** genannt

- **Projektkosten:** eine Information über die voraussichtlichen Kosten des Projekts, möglichst aufgeteilt in Material kosten und Lohnkosten.

Die Fortsetzung in die nächste Phase des Entwicklungsprozesses, die Problemanalyse, erfolgt nur dann, wenn aus der Problemformulierung eine positive Entscheidung für die Durchführung des Projekts getroffen wurde. Das eigentliche vollständige Projekt startet also erst zu diesem Zeitpunkt.

Während bei der Problemformulierung ein Beschreibungsumfang entstanden ist, der die drei oben genannten Ergebnisse mit genügender Sicherheit festlegbar machte, gilt es jetzt, das Problem in allen Einzelheiten zu spezifizieren. Ziel dieser Phase ist also eine detaillierte Problemdefinition, die eine Festlegung aller möglichen Zustände des zu entwickelnden Objekts bis in alle Feinheiten enthält und auch explizit die Grenzen der Lösung, also die per Definition nicht berücksichtigten Einflüsse charakterisiert. Das Ergebnis dieser Phase wird als **Pflichtenheft** bezeichnet.

Mit der darauf folgenden Phase der **Lösungssuche und Planung** beginnt der eigentliche kreative Prozess des Ingenieurs. Auf Basis seiner Ausbildung und seiner Erfahrung aus vorangegangenen ähnlichen Projekten entwickelt er im Idealfall mehrere Lösungsvorschläge. Die Wahl der weiter zu verfolgenden und zu entwickelnden Lösung geschieht anhand von Auswahlkriterien und deren Gewichtung durch den Entwickler und den Kunden. Der derart bevorzugte Lösungsweg, d. h. **das entstehende Produkt, wird in allen Details entworfen.**

Dieser Entwurf entsteht soweit irgend möglich auf vorgeschriebenen bewährten Wegen (Prinzipien und Methoden) und unter Nutzung vorgeschriebener Werkzeuge. Die Entwurfsschritte stellen sich in einer Form dar, die gleichzeitig als Dokumentation verwendbar ist, etwa in technischen Zeichnungen, Storni aufplanen oder Ansichtsskizzen.

Die Implementation hat dann die **Realisierung des fertigen Produkts zum Ziel**. Bei später geplanter Massenproduktion entsteht in dieser Phase der Prototyp.

Dieses idealisierte Modell wird jedoch in der Praxis selten in konsequenter Form durchlaufen. Trotzdem diene es jetzt als Vorbild, den Software-Entwicklungsprozess aus ihm abzuleiten und dabei die besonderen Belange der Software zu berücksichtigen.

2.2 Der Software-Entwicklungsprozess

Ausgehend von der idealisierten Form des allgemeinen ingenieurmäßigen Entwicklungsprozesses soll nun der allgemein anerkannte Software- Entwicklungsprozess vorgestellt werden, auf dessen Ablauf die weiteren Ausführungen aufbauen und dessen einzelne Phasen in den folgenden Kapiteln genau beschrieben werden.

Der vorzustellende Software-Entwicklungsprozess besteht nach Bild 2.2 aus fünf Phasen. **Alle Phasen werden begleitet durch**

- **die Projektführung und Überwachung**, die Sollvorgaben macht, den Projektfortschritt überprüft, und den Kontakt zum Auftraggeber hält. Die damit verbundenen Managementprobleme und die Aspekte der Software-Teamorganisation werden im letzten Kapitel dieses Skripts genauer untersucht.

- **das Produktarchiv,** einem (möglichst automatisierten) Speichermedium, in dem alle projektrelevanten Daten gesammelt werden. Da dieses Archiv übergreifend aufgebaut sein soll, dient es in den einzelnen Phasen auch der Entnahme von **Erfahrung** und von konkreten wieder verwendbaren Teilen, die aus alten Projekten existieren.

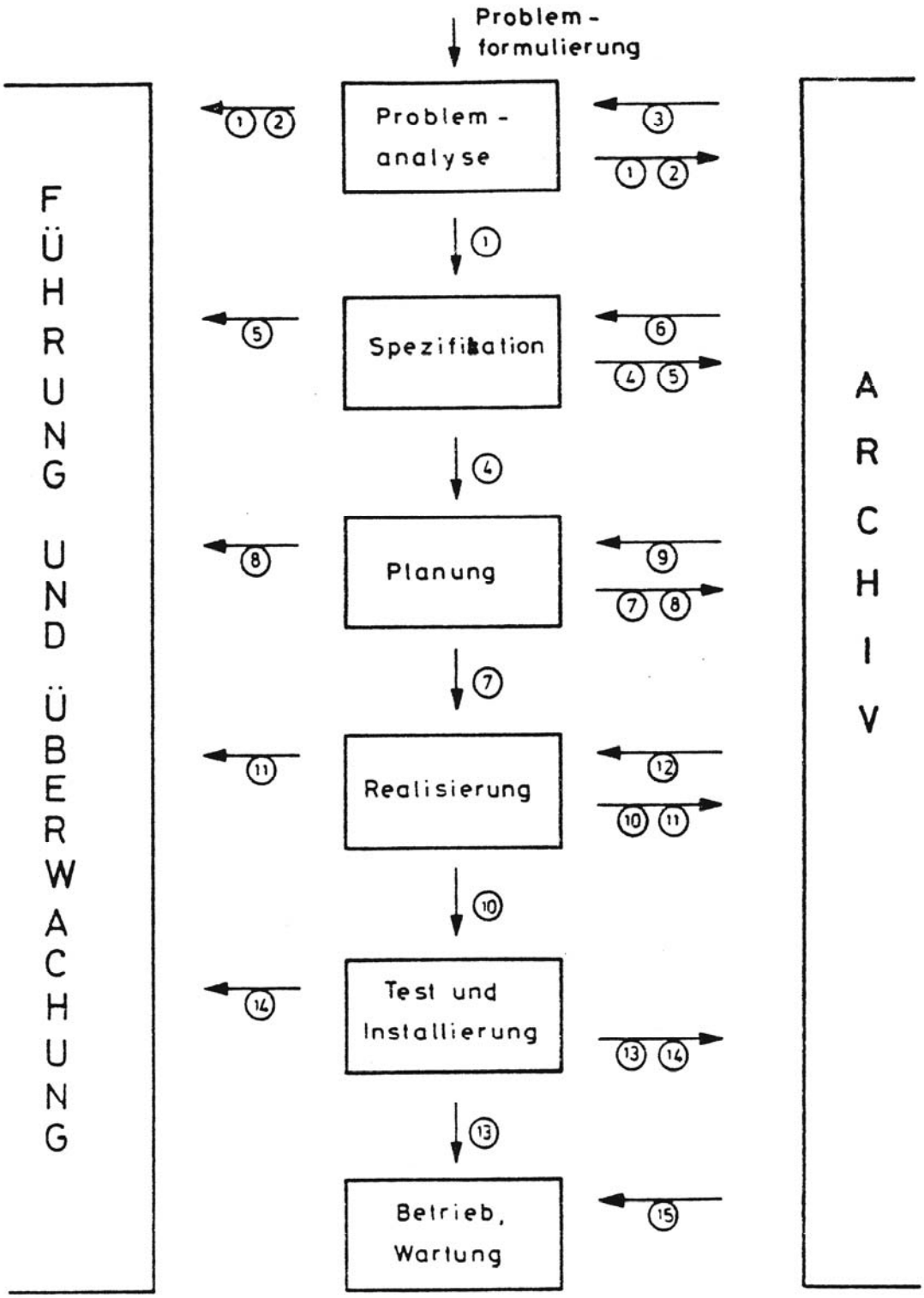

Bild 2.2: Der Software-Life-Cycle

Erläuterungen:

1 Pflichtenheft I, Funktionshandbuch I, Benutzerhandbuch I

2 Projektplan I

3 Erfahrung

4 Pflichtenheft II, Funktionshandbuch II, Benutzerhandbuch II, Begriffslexikon

5 Projektplan II

6 existierende Teilsspezifikationen

7 Funktionshandbuch III, Benutzerhandbuch III, Systemdokumentation

8 Zustandsberichte

9 existierende Module

10 syntaxfehlerfreie Module, Testdaten und Hinweise

11 Zustandsberichte

12 fertig programmierte Standardmodule

13 fertiges Softwareprodukt

14 Zustandsberichte

15 Dokumentation aller Art über alle Systemversionen

Die in Bild 2.2 eingezeichnete sechste Phase, Betrieb und Wartung, gehört im engen Sinne nicht zum Softwareentwicklungsprozess und hätte im Idealfall keinen Kontakt zum Produzenten. Die zugehörigen Anmerkungen aus Kapitel 1 zeigen jedoch, dass die Realität anders aussieht.

Die Gesamtheit aller sechs Phasen wird als Software-Life-Cycle bezeichnet. Dieser beschreibt also ein Softwareprodukt von seinem Entstehen bis zu seiner Unbrauchbarkeit.

Eine Verfeinerung des Life-Cycle-Modell ist das sog. „Wasserfall“-Modell, was in Bild 2.2-1 gezeigt wird. Das Wasserfallmodell führt zwischen aufeinander folgenden Phasen zusätzliche Rückkopplungen ein. Ziel ist es, teuere und zeitintensive Nacharbeit bei notwendigen Iterationsschritten über mehrere Phasen hinweg zu vermeiden.

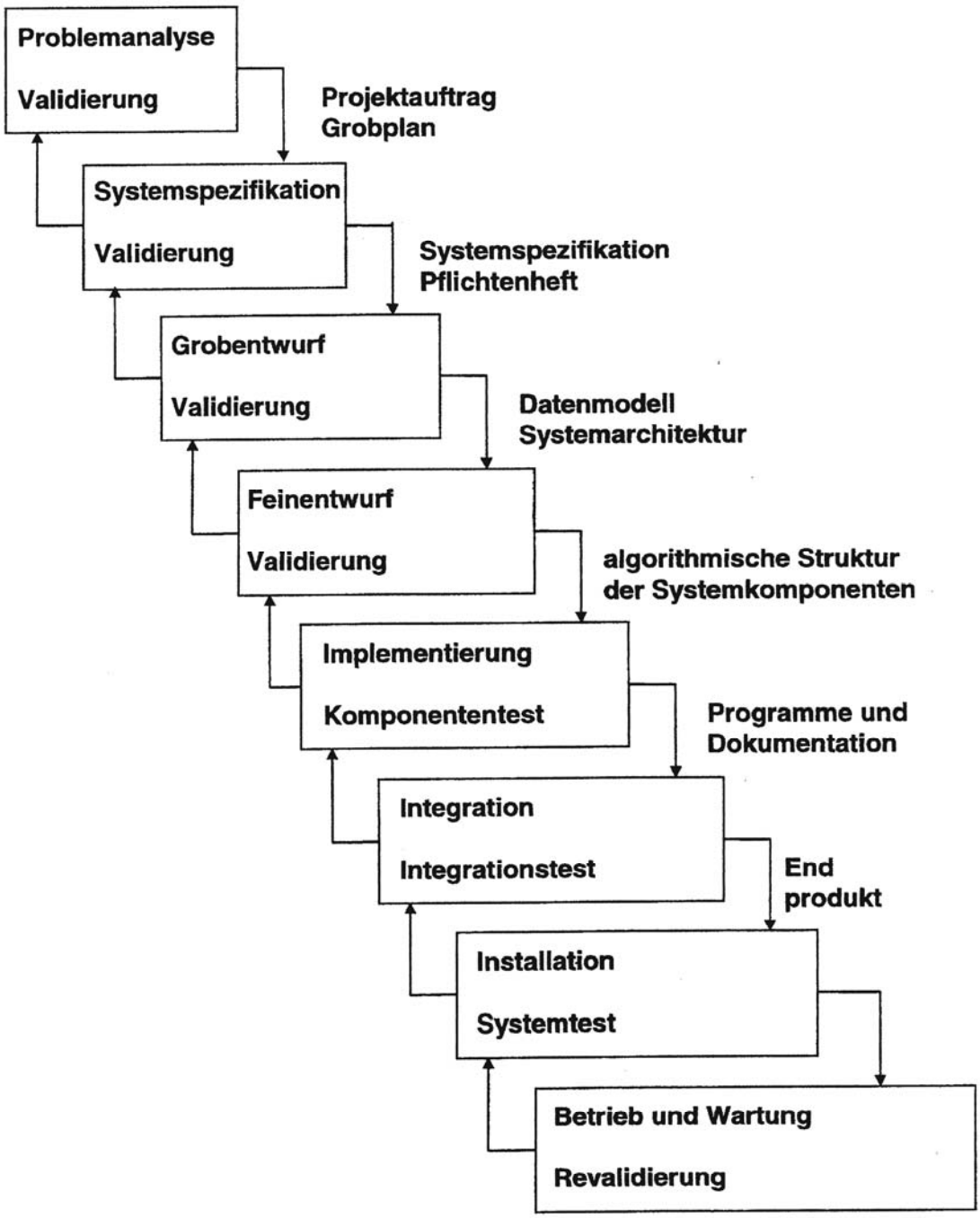

Bild 2.2-1: Der verfeinerte Software-Life-Cycle - Das Wasserfallmodell

Ein Vergleich mit dem allgemeinen ingenieurmäßigen Entwicklungsprozess, der ja per Definition als Vorbild für den Softwareentwicklungsprozess dienen sollte, zeigt tatsächlich auch viele Gemeinsamkeiten, d. h. das dort beschriebene Vorgehensprinzip gilt generell auch hier.

Auch in der Softwareentwicklung wird in der ersten Phase eine **Problemanalyse** durchgeführt. Das Verständnis der an den Softwareingenieur herangetragenen Probleme hat herausragende Bedeutung, da der Einsatz digitaler Rechner inzwischen in weiten industriellen Bereichen erfolgt und sie als Hilfsmittel für Dienstleistungen eingesetzt werden, die für den Entwicklungsingenieur oft ein vollkommen fachfremdes Gebiet darstellen. Somit ist schon bei der schriftlichen Problembeschreibung, mit dem sich ein möglicher Auftraggeber an den Produzenten wendet und auf Basis derer die weitere Analyse stattfindet, besonders auf eine einheitliche und in allen Begriffen von beiden Seiten geklärte Terminologie zu achten.

Führen die Ergebnisse der Problemanalyse zu einer positiven Entscheidung über den Projektfortschritt, (vergleiche die Ergebnisse der Problemformulierung aus Kap. 2.1) so dient die folgende **Spezifikation** zu einer möglichst detaillierten Beschreibung des Problems. Schon hier erfolgt aus Sicht des Benutzers eine Aufspaltung in Teilfunktionen. Es werden Prozess- und Rechenabläufe festgelegt und Dialoge mit dem Benutzer in ihrem Verlauf beschrieben. Die Ergebnisse dienen als Grundlage für alle weiteren Phasen und im Extremfall auch als Rechtfertigung gegenüber späteren Kundenreklamationen.

In der **Planungsphase** erfolgen die ersten konstruktiven Schritte in Richtung auf die Lösung des jetzt vollständig spezifizierten Problems. Das in der Regel sehr mächtige Gesamtproblem wird unter Berücksichtigung bestimmter Kriterien in kleine überschaubare Teilprobleme derart aufgeteilt, dass ihre Realisierung unabhängig voneinander durch verschiedene Mitarbeiter in Parallelarbeit erfolgen kann. Die Berücksichtigung von Methoden für eine gute Aufteilung des Problems und die gute Wahl, Festlegung und Beschreibung der Schnittstellen zwischen den einzelnen Teilproblemen sind die Grundlage für den Erfolg der weiteren Arbeiten.

Erst die folgende **Realisierungsphase** führt zum direkten Kontakt mit dem digitalen Rechner. Die einzelnen Teilprobleme werden getrennt voneinander und in Parallelarbeit so weit verfeinert, dass sie schließlich in eine Anweisungsfolge der gewählten Programmiersprache umgesetzt werden können. Die meist zunächst handschriftlich erstellten Programme werden in den Rechner eingegeben und mit Hilfe eines Compilers oder Assemblers in die jeweilige Maschinensprache übersetzt.

Auch die **Testphase** basiert auf der Unterteilung in Einzelprobleme. Wurde die Planungsphase richtig durchgeführt, so ist jedes einzeln realisierte Teilproblem auch einzeln testbar. Funktionieren dann alle Teilprobleme für sich, so werden sie in einem zweiten Schritt zusammengefügt. Bei guter Planung der Schnittstellen sollte man dann im Idealfan davon ausgehen, dass das Gesamtsystem zufrieden stellend läuft. Jedoch zeigt die Praxis, dass dieses Ergebnis selten im ersten Anlauf erreicht wird. Vielmehr ist der Regelfall, dass gerade bei der Installierung des Gesamtsystems Fehler entdeckt werden, die in früheren Phasen gemacht wurden.

Der Vollständigkeit halber wird auch die **Wartungsphase** kurz besprochen. Ihr primäres Ziel ist die Beseitigung von Fehlern, die während der Testphase nicht entdeckt wurden, und die erst nach einiger Betriebszeit das erste Mal auftreten. Das ist nur aus dem Grunde möglich, weil komplexe Probleme und die dazugehörigen Programmsysteme sehr viele unterschiedliche Zustände, Zustandskombinationen und damit verbundene Entscheidungen beinhalten. Bestimmte Programmzweige werden vielleicht nur in den seltensten Fällen durchlaufen, weil die zugehörigen äußeren Zustände entsprechend selten auftreten. In ihnen enthaltene Fehler sind vielleicht beim Test nicht aufgefallen, oder - schlimmer - bestimmte Zustände wurden nicht berücksichtigt, und das System reagiert falsch, unsinnig oder auch gar nicht.

Mehrere so aufgetretene Systemfehler werden dann mehr oder weniger formal beschrieben und gesammelt, und in bestimmten Zeitabständen findet eine mehr oder weniger formal durchgeführte Prozedur statt, die als Maintenance bezeichnet wird. Ziel dieser Prozedur ist die Einführung einer neuen Systemversion, die dann hoffentlich fehlerfreier ist als die alte.

Eins mag zunächst überraschen: Es gibt keine eigene Dokumentationsphase. Vielmehr sollen die nötigen Dokumentationen sozusagen automatisch als Teilergebnis der einzelnen Phasen erstellt werden.

Man hat sich bis heute im Bereich des Software Engineering weder auf eine einheitliche Phasendarstellung des Software-Life-Cycle, noch auf eine einheitliche Nomenklatur geeinigt. Bild 2.3 zeigt den Software-Life-Cycle, so wie er von Balzert /2/ dargestellt wird. Andere Firmen und Institutionen haben intern ihren "eigenen" Software-Life-Cycle definiert, der oft die einzelnen Phasen genauer unterteilt, so etwa die Planung in Grobplanung, Feinplanung, Schnittstellenbeschreibung und Planungskontrolle. Der inhaltliche Ablauf ist jedoch prinzipiell überall gleich. Deshalb soll in den weiteren Ausführungen das einfache Modell nach Bild 2.2 weiterverfolgt werden.

Bild 2.4 stellt die heute gängige Zeitaufteilung für die fünf Entwicklungsphasen auf die gesamte Produktionszeit dar. Besonders interessant ist, dass nach diesem Modell erst nach ca. 60 % der Gesamtzeit der erste Rechnerkontakt stattfindet, d.h. die ersten Teilprobleme codiert werden. Die Testphase mit 25 % der Gesamtzeit erscheint zunächst als recht hoch angesetzt. In der Praxis zeigt sich jedoch sehr schnell, dass es sinnvoll ist, gerade in diesem Bereich einen hohen Zeitbedarf einzuplanen, denn Fehlersuche und Fehlerbeseitigung mit Rückgriffen und Aufarbeitung früherer Phasen ist äußerst zeitaufwendig.

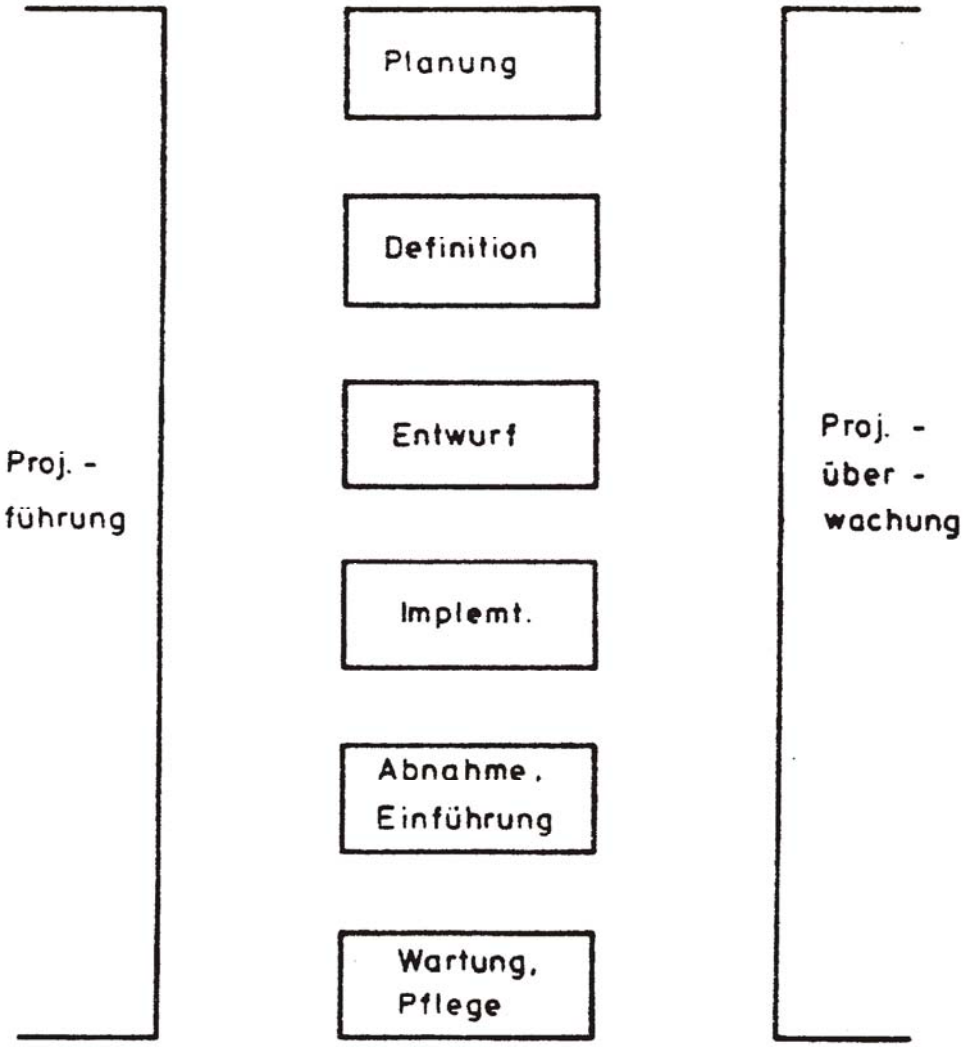

Bild 2.3: Software-Life-Cycle nach Balzert

In Bild 2.5 dargestellt ist der Fertigstellungsgrad eines Softwareprojekts, gemessen als Menge produzierten Codes (oder Zeilen einer Programmiersprache) über der Entwicklungszeit. Die Ideallinie zeigt analog zu Bild 2.4, wie zunächst über 50 % der Systementwicklungszeit in "Vorbereitungen" für die Codierung besteht. Diese Vorbereitungen sind optimal so effektiv, dass dann als Teil der Realisierungshase in ca. 10 % der Gesamtzeit die eigentliche Programmierung erfolgen kann und das System nach einem ausführlichen Test zur rechten Zeit fertig gestellt ist.

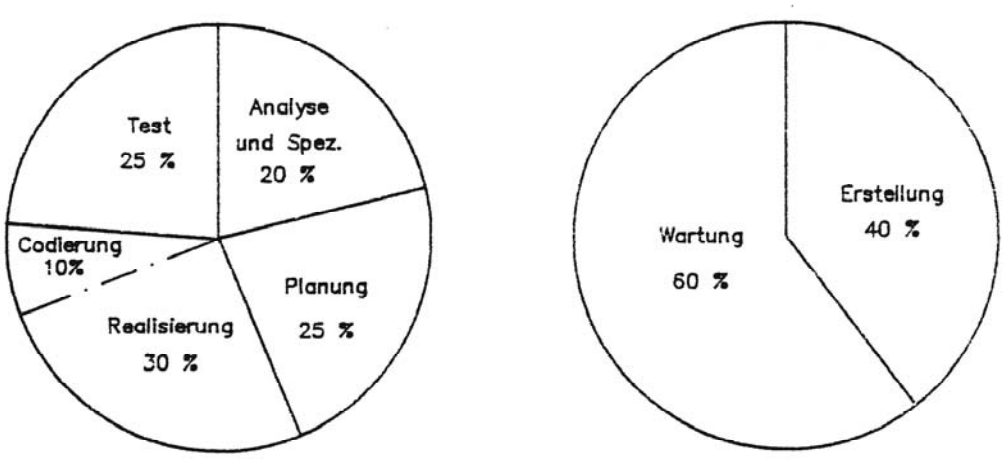

Bild 2.4: Zeitaufteilung der Entwicklungsphasen

Die heute oft durchlaufene Linie bei ungenügender Phasenaufteilung des Projekts zeigt einen zunächst beruhigenden gleichmäßig linearen Anstieg der Codeproduktion, der jedoch erfahrungsgemäß bei 80 - 90 % des Gesamtsystems stagniert.

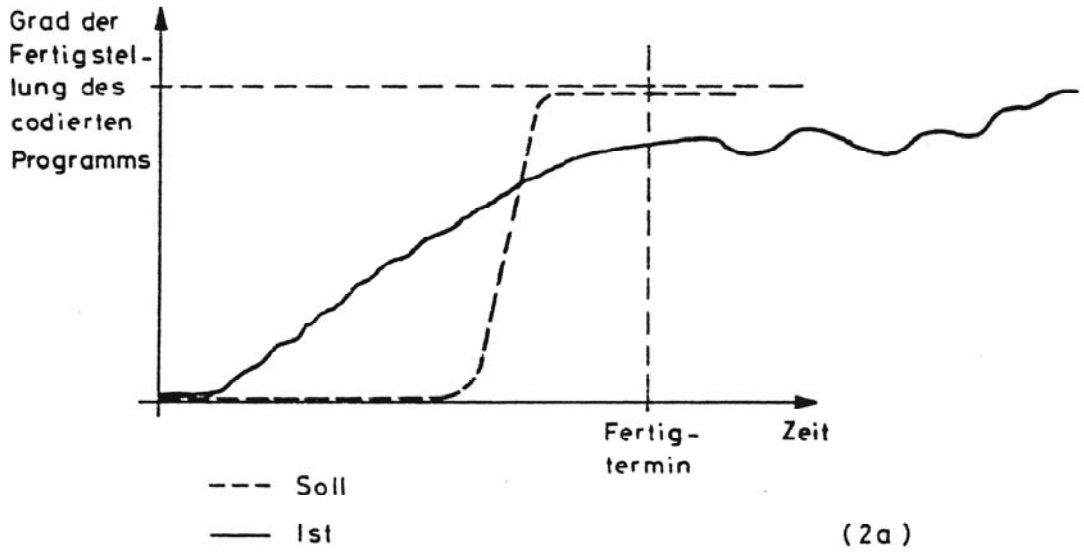

Bild 2.5: Menge produzierten Codes zu Projektphasen

Infolge ungenügender Vorbereitungen ergeben sich jetzt Inkompatibilitäten, die dazu zwingen, Programmteile neu zu schreiben und die den weiteren Verlauf des Projekts unüberblickbar und unabwägbar machen.

In den folgenden Abschnitten sollen nun die einzelnen vorgestellten Entwicklungsphasen eines Softwareprojekts mit ihren bekanntesten Methoden und Werkzeugen genauer betrachtet werden.