8. Planen und Schätzen

Software Entwicklung ist komplex. Es gibt keine einfache Lösung für die Projektabwicklung! Der Grund liegt in den vielen möglichen Alternativen, die in einem Projekt möglich sind. Jedes Projekt kann bei Nichtbeachtung von Kleinigkeiten zu einem Desaster werden. Projekte verlaufen in der Regel *nicht* problemlos. Daher ist es wichtig, die Hausaufgaben vor dem Absturz gemacht zu haben, den Projektstatus genau zu kennen; der Absturz lässt sich dadurch nicht vermeiden, aber sanfter wird er.

8.1. Planen und der Software Prozess

Im Idealfall würden wir einen Plan erstellen, an den wir uns dann auch peinlich genau halten. Das ist allerdings eine Illusion, da wir nicht genug Informationen über die einzelnen Details haben. Zum Beispiel : nach der Anforderungsphase kann man die ursprüngliche Planung im Wesentlichen vergessen!

Aber auch nach der Erstellung eines Prototypen kann es noch jede Menge Unsicherheiten geben. Die Informationen, die dem Entwickler zur Verfügung stehen, nachdem der Prototyp erstellt sind, sind bestenfalls vergleichbar mit einer Skizze eines Hauses, im Vergleich mit einem detaillierten Bauplan.

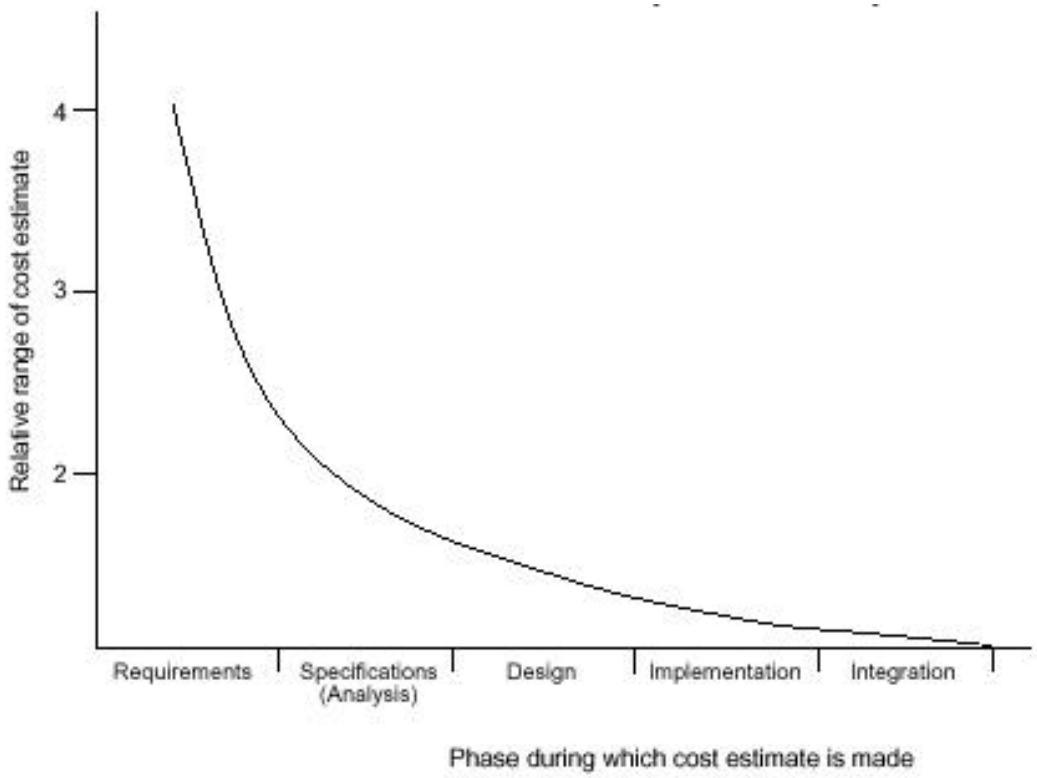
Einigermassen verbindliche Aufwandschätzungen sind erst zum Zeitpunkt möglich, zu dem Kunde die Spezifikation abgenommen hat und schriftlich bestätigt.

Im schlimmsten Fall will der Kunde aber bereits viel früher eine verbindliche Offerte, da er mehrere Offerten eingeholt hat und vergleicht.

Boehm hat im Rahmen verschiedener Projekte für die Luftfahrt und Raumfahrt versucht eine Projektdatenbank aufzubauen und daraus Formeln herzuleiten, mit denen man Abschätzungen erstellen kann. Die ersten Untersuchungen haben grob folgendes Ergebnis gezeigt:

- die ersten Aufwandschätzungen liegen oft bis zu 400% über oder unter den aktuellen Aufwendungen
- je näher das Projekt dem Projektziel kommt, desto genauer werden die Schätzungen

Es ergibt sich also folgendes Bild:



Als Beispiel nehmen wir an, dass ein Produkt insgesamt nach dem Abnahme durch den Kunden 1 Mio USD kostet. Falls in der Anforderungsphase eine Aufwandschätzung abgegeben wurde, dann lag diese vermutlich im Bereich von 0.25 Mio bis 4 Mio USD.
Falls eine verbesserte Aufwandschätzung in der Spezifikationsphase gemacht wurde, dann dürfte diese im Bereich von 0.5 Mio bis 2 Mio USD liegen.
Nach der Spezifikation, also nachdem man die Anforderungen weitestgehend fixiert hat, liegt die Schätzung immer noch im Bereich von 0.67 Mio bis zu 1.5 Mio USD, also um den Faktor auseinander.

Die Kostenschätzung ist also sicher keine exakte Wissenschaft!

8.2. Abschätzen von Dauer und Kosten

Das Budget ist ein integraler Bestandteil des Software Projekt Management Planes. Bevor das Projekt beginnt, möchte der Kunde in der Regel wissen, wieviel das Projekt kosten wird. Nicht zuletzt für eine Kosten / Nutzen Abschätzung ist es wichtig, die Grössenordnung der Kosten und der Projektdauer ermitteln zu können.
Falls das Entwicklungsteam die Kosten (und die Projektdauer) wesentlich falsch einschätzen, muss damit gerechnet werden, dass das Projekt vom potentiellen Kunden und Auftraggeber abgelehnt wird.
Bei den Kosten muss man zwischen internen und externen Kosten unterscheiden. Interne Kosten betreffen unter anderem Gehälter für die am Projekt beteiligten Personen, Produktivitätsverlust in der Einführungsphase, Umstellungsaufwand, Hardware, Software Datenkonversion und viel mehr.

Externe Kosten sind jene Kosten, die nach aussen anfallen, also in der Regel mit einem Aufschlag (Markup) versehen sind. Die Kostenblöcke für externe Kosten sind mit jenen der internen Kosten teilweise identisch, bis auf die Margen. Falls der Entwickler auf Arbeit angewiesen ist, kann es passieren, dass er auf einen Teil der Marge verzichtet.

Neben den Kosten muss auch die Projektdauer abgeschätzt werden : wann steht das Produkt dem Kunden (intern oder extern) zur Verfügung?
Falls das Produkt nicht zur Zeit abgeliefert werden kann, wird der Kunde im schlimmsten Fall einen Schadensersatz verlangen. Falls die Entwicklungszeit überschätzt wird, dann kann es vorkommen, dass sich der Kunde für ein Konkurrenzangebot entscheidet.
Sackman und Mitarbeiter haben 1968 erste Untersuchungen durch geführt und festgestellt, dass Schätzungen bis zu einem Faktor 28 auseinander liegen. Dabei kommt der Fehler nicht auf Grund der unterschiedlichen Erfahrung der Programmierer zu Stande. Die obige Untersuchung verglich ähnliche Projektteams.Die Untersuchung zeigte zudem folgende Ergebnisse:

- Produktgrösse (Programmcode) : Unterschiede im Verhältnis 6:1
- Produktausführungszeit : Unterschiede im Verhältnis 8:1
- Entwicklungszeit : Unterschiede im Verhältnis 9:1
- Programmierzeit : Unterschiede im Verhältnis 18:1
- Debuggingzeit : Unterschiede im Verhältnis 28:1

Selbst nachdem der beste und der schlechteste Programmierer aus der Statistik entfernt wurden, änderten sich die Verhältnisse kaum.

Ein weiterer kritischer Punkt in jedem Projekt ist, dass Entwickler jederzeit aus dem Projekt aussteigen könnten, die Firma verlassen oder einfach etwas anderes machen müssen / wollen.

Kurz und bündig : Kosten und Dauer eines Projektes lassen sich nur sehr schwierig abschätzen, oder besser gesagt einschränken.

8.2.1. Metriken für die Grösse eines Projektes

Die übliche Metrik für die Projekt oder Programm Grösse ist die Anzahl Zeilen des Programmes. Als Massstab haben sich zwei Messlatten eingeführt : Lines of Code (LOC) und Kilo Delivered Source Instructions (KDSI). LOC ist eher problematisch, wie Sie bereits aus der oben zitierten Studie erkennen können: je nach Programmierteam wird das Programm unterschiedlich gross:

1. das Schreiben der Programme ist lediglich ein kleiner Teil der gesamten Projektarbeiten
   Es ist eher unwahrscheinlich, dass alle Aktivitäten in den unterschiedlichen Phasen (Anforderung, Spezifikation, Integration,...) lediglich als Funktion des Programmumfanges dargestellt werden kann.
2. falls das selbe Programm mit Hilfe einer anderen Programmiersprache geschrieben wird, ändert sich die Anzahl Zeilen in der Regel dramatisch. Falls Lisp oder eine 4GL eingesetzt wird, dann ist der Unterschied zu einer Sprache wie COBOL oder C++ besonders krass.
3. oft ist es nicht ganz klar, wie die Programmzeilen gezählt werden sollen, was dazu gehört. Soll man Datenspezifikationen, Kommentare, import und package Anweisungen .... auch zählen?
   Falls Kommentare nicht gezählt werden, dann besteht die Gefahr, dass die Entwickler die Programme nicht mehr dokumentieren.
   Wie sieht es mit Job Control Language (Betriebssystem Befehlen) aus?

4. Oft werden wesentlich mehr Programmzeilen geschrieben als abgeliefert werden. In vielen Fällen sind die Support Tools, die im Rahmen eines Projektes entwickelt werden, fast so umfangreich wie der letztlich abgelieferte Programmcode.
5. Falls der Programmierer einen Code Generator einsetzt, dann wird die Anzahl Programmzeilen noch lächerlicher: man will ja kaum die Produktivität des Generators messen wollen. Speziell im GUI Bereich sind Generatoren üblich.
6. Wie sollen wir den Projektumfang abschätzen, mit Hilfe der Anzahl Programmzeilen, also mit einer Grösse, die wir erst nach dem Projekt kennen!.

Software Science, also die Theoretiker, haben verschiedene Modelle entwickelt, mit dem Ziel, Schätzungen zu ermöglichen und zu verbessern. Diese Verfahren basieren in der Regel auf folgenden Grössen:

- Anzahl Eingaben
- Anzahl Ausgaben
- Anzahl verarbeitender Module

Aber auch hier hat man das Problem, dass diese Grössen letztlich erst bekannt sind, nachdem das Projekt abgeschlossen ist. Natürlich gibt es auch viele Studien, die solche theoretischen Modelle bekämpfen, ob zu Recht oder zu Unrecht sei dahin gestellt.

Alternativ zu den obigen Verfahren könnte man versuchen, Metriken zu definieren, die bei den Schätzungen behilflich sind.
Ein erster Ansatz ist die FFP Metrik, eine einfache Metrik, die auf der Anzahl Dateien (file), Abläufen (flow) und Prozessen (process) beruht. Ablauf wird dabei eher restriktiv definiert als Schnittstelle zwischen dem Produkt und dem Umfeld. Prozesse sind Berechnungen, Manipulationen der Daten, Sortieren und ähnliches.

Beispiel:
bei gegebener Anzahl Dateien Fi , Abläufen Fj und Prozessen Pk ergeben sich die Kosten C (cost) und die Grösse der Programme S (size) zu:

$$\mathbf{S = Fi + Fj + Pk}$$
$$\mathbf{C = d * S}$$

Dabei ist d eine Konstante, die je nach Organisation (also dem Projektumfeld) gewählt, angepasst werden muss.

d ist ein Mass für die Produktivität der Organisation und die Effizienz des Software Entwicklungsprozesses.
Die Grösse S kann man auf Grund der Architektur abschätzen.

Das Modell wurde nur an einfachen Beispielen getestet, ohne Datenbanken oder eher alten Technologien. Generell hat sich das Modell nicht durch gesetzt.

Eine ähnliche aber unabhängig von der obigen entwickelte Metrik wurde 1979 vorgeschlagen. Als Parameter werden dabei die Anzahl Eingaben *Inp* (Inputs), Ausgaben *Out* (Output), Abfragen *Inq* (Inquiries), Stammdaten *Maf* (Masterfiles) und Schnittstellen *Inf* (Interfaces). Daraus wird die Anzahl *Function Points* hergeleitet:

$$\mathbf{FP = 4 * Inp + 5 * Out + 4 * Inq + 10 * Maf + 7 * Inf}$$

Diese Grösse ist ein Mass für die Produktgrösse, also die Anzahl Programmzeilen.

Die obige Formel ist etwas zu stark vereinfacht. Jeder Komponente wird eine bestimmte Anzahl Function Points zugeordnet, jeweils mit einem relevanten Faktor multipliziert.
Die Methode schlägt folgende Schritte vor:

1. jede der Komponenten (Inp, Out, Inq, Maf, Inf) wird bezüglich der Komplexität beurteilt.
   simple
   average
   complex

Component	Level of Complexity		
	Simple	Average	Complex
Input item	3	4	6
Output item	4	5	7
Inquiry	3	4	6
Master file	7	10	15
Interface	5	7	10

   zum Beispiel:
   für eine einfache Eingabe werden 3 Funktionspunkte gegeben
   Falls die Komplexität hoch ist, dann müssen dafür 6 Funktionspunkte vergeben werden.

   Die obige Formel führt in diesem Falle zu den sogenannten unadjusted function points UFP
2. als Nächstes wird die Komplexität berücksichtigt.
   Dies führt zum technical complexity factor (TCF). Für die Berechnung dieser Grösse werden insgesamt 14 Parameter berücksichtigt, wie Transaktionsrate, Leistungsanforderungen und ähnliches. Jeder der technischen Faktoren wird wie folgt berücksichtigt:

   falls der Faktor im Projekt eine Rolle spielt, dann erhält er eine Gewichtung zwischen o (nicht vorhanden) und 5 (wesentlicher Faktor im Projekt).
   Die 14 Zahlen werden addiert und liefern den Einflussfaktor "total degree of influence" (DI):

   1. Data communications
   2. Distributed data processing
   3. Performance criteria
   4. Heavily utilized hardware
   5. High transaction rates
   6. Online data entry
   7. End-user efficiency
   8. Online updating
   9. Complex computations
   10. Reusability
   11. Ease of installation
   12. Ease of operation
   13. Portability
   14. Maintainability

   Daraus ergibt sich der technical complexity factor TCF als:

   TCF = 0.65 + 0.01 * DI

   DI liegt im Bereich zwischen 0 und 70. Daraus ergibt sich, dass TCF im Bereich von 0.65 bis 1.35 liegt.

3. Die Function Points ergeben sich nun mit Hilfe der Gleichung

   FP = UDP * TCF

Test mit dieser Metrik haben gezeigt, dass dies wesentlich bessere Resultate liefert als die FFP Metrik oder KDSI. Eine Studie zeigt, dass unter den in der Studie gegebenen Bedingungen, KDSI Methoden bis zu 800 Prozent falsch waren, während die FP Methode noch im Bereich von 200% lag.

Ein anderes Beispiel zeigt die nebenstehende Tabelle: darin wird ein Projekt in Assembler mit einem Projekt in Ada verglichen.

	Assembler Version	Ada Version
Source code size	70 KDSI	25 KDSI
Development costs	$1,043,000	$590,000
KDSI per person-month	0.335	0.211
Cost per source statement	$14.90	$23.60
Function points per person-month	1.65	2.92
Cost per function point	$3,023	$1,170

Das Problem der LOC oder KDSI basierten Methoden erkennt man bereits an der Zeile "KDSI per peron-month":
gemäss dieser Zahl müsste die Produktivität eines Assembler Programmierers 60% höher sein als jene des Ada Programmierers, was offensichtlich nicht stimmt! Dritt Generation Sprachen wie Ada sind Assembler bei weitem überlegen.
Als nächstes vergleichen wir die Kosten pro Anweisung (cost per source statement). Eine Ada Anweisung entspricht im Schnitt 2.8 Assembler Anweisungen. Aber auch hier wäre gemäss obiger Tabelle Assembler Ada überlegen!
Die Funktionspunkt basierten Grössen entsprechen jedoch der Realität weitest gehend.

Aber FFP und Function Point Methoden haben die Schwäche, dass sie die Wartungsphase in der Regel nicht berücksichtigen können. Die Wartung hat aber unter Umständen grossen Einfluss auf die Anzahl Funktionspunkte, Eingaben und Ausgaben. Die KDSI und LOC Metriken versagen in der Wartung zwangsweise: es kann sein, dass in der Wartung ganze Teile des Programmes ersetzt werden, aber die Anzahl Programmzeilen die selbe bleibt.

In der Zwischenzeit wurden verschiedene Vorschläge zur Verbesserung der Function Point Methode vorgeschlagen. Als erstes wurde UDP genauer berechnet. Die daraus hergeleitete FP ist die wohl am meisten verwendete Metrik weltweit.

Neben den Function Points wurde um 1986 eine Metrik basierend auf *Feature Points* definiert. Die entsprechende Formel:

$$\textit{Feature Point} = FP - 3 * Maf + 3 * Alg$$

wobei FP wie bisher berechnet wird. *Maf* ist die Anzahl Masterfiles, Stammdateien; *Alg* steht für die Anzahl Algorithmen.

Diese Formel wurde und wird speziell im Bereich Echtzeit Software angewandt.

8.2.2. Techniken zur Kostenschätzung

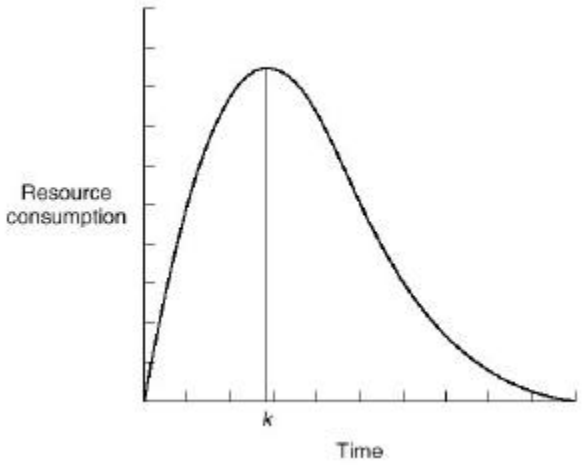
Die Probleme der Kosten und Aufwand Schätzung sind wahrscheinlich in keinem anderen Entwicklungsprojekt so gross wie in der Informatik.
Faktoren wie Teamzusammensetzung, Komplexität der Aufgabe und viel mehr sind in anderen Projekten auch kaum in dieser ausgeprägten Form anzutreffen.
Einige der gängigen Methoden sind:

1. Experten Meinung
   Expert(en) versuchen auf Grund bereits abgeschlossener Projekte, also der Erfahrung, eine möglichst genaue Aufwandschätzung zu machen. Da die Umgebung sich hoffentlich nicht wesentlich geändert hat und die Entwickler immer noch die selben sind und die selben Technologien einsetzen, ist der Experte in der Lage eine einigermassen zuverlässige Aussage machen zu können, sagen wir im Bereich von 15%.

   In einer Untersuchung wurden mehrere Experten zum selben Projekt befragt. Die Aufwandschätzung schwankte dabei zwischen 15 Monaten, 13.5 Monaten und 16 Monaten. Dies lässt sich verallgemeinern :
   mehrere Experten geben ihre Schätzungen pro Teilgebiet ab. Dann werden die Daten ausgetauscht und jeder Experte hat die Möglichkeit seine Schätzung zu korrigieren. Anschliessend wird gemittelt und mit diesem Wert wird weiter gearbeitet.

2. Bottum-Up Approach
   Ein guter Ansatz zur Bestimmung des Aufwandes besteht in der Aufteilung des zu entwickelnden Systems in mehrere überschaubare Subsysteme. Diese lassen sich besser analysieren, da die Komplexität geringer ist. In der Regel wird man versuchen Top Down und Bottom Up vorgehen und die zwei resultierenden Zahlen vergleichen. Je dichter diese Zahlen beieinander sind, desto hoffnungsvoller kann man in die Zukunft blicken.
   Speziell im Rahmen Objekt orientierter Systeme, möglichst entkoppelt und coherent, ist die Bottom Up Methode die nahe liegende.

3. Algorithmische Kostenschätzung Methoden
   In den algorithmischen Methoden versucht man mit Hilfe der Function Points (oder KDSI) und einem Modell die Produktkosten zu berechnen.
   Algorithmische Methoden scheinen einer Experten Meinung überlegen zu sein. Dies braucht aber nicht der Fall zu sein: schlechte algorithmische Modelle oder falsch verstandene Modelle liefern auch nicht gerade zuverlässige Schätzungen.

   Das SLIM Modell geht davon aus, dass die Resourcen in einem Projekt abhängig von der Phase eingesetzt werden. Am Anfang eines Projektes werden nur wenige Resourcen aktiv eingesetzt, genauso am Ende des Projektes bei der Abnahme.

Das wohl am Besten fundierte Modell stammt von Boehm und nennt sich COCOMO für COnstructive COst MOdel.

Der Grund für die starke Verbreitung des Modells liegt zum einen daran, dass das Modell im Umfeld der Armee und der Nasa entwickelt wurde; zum andern werden verschiedene Software Pakete angeboten (COSTAR, ein kommerzielles Produkt; aber auch Applets und gratis Software).

Randbemerkung:
Oft spricht man vom COCOMO Modell. Eigentlich ist dies ein Overkill, da das MO im COCOMO schon für Modell steht.

8.2.3. Intermediate COCOMO

COCOMO besteht aus mehreren Modellen. Das Basismodell besteht aus drei Modellen, je einem für Makroabschätzungen und einem für Mikroabschätzungen, bei dem jedes Detail berücksichtigt werden kann.
Intermediate COCOMO wurde entwickelt für Projekte mittlerer Komplexität und Detaillierung. Boehm hat seine Erkenntnisse in einem umfangreichen Buch
Barry W. Boehm : Software Engineering Economics, Prentice Hall, 1981, mit 700+ Seiten, veröffentlicht. Das Buch gilt als Standardwerk für dieses Spezialgebiet und sollte von jedem Projektleiter zu mindestens diagonal gelesen worden sein.
Die Berechnung der Entwicklungszeit mit Hilfe des Intermediate Modells geschieht in zwei Schritten.

1. Als erstes versucht man die KDSI abzuschätzen. Diese Zahl wird als Parameter für das Modell benutzt.
2. Als zweites versucht man die interne Komplexität des Projektes abzuschätzen. Dabei unterscheidet man drei Modi :
   a) organic : klein und ohne wesentliche Komplexität
   b) semidetached : mittlere Grösse
   c) embedded : komplex

Mit Hilfe dieser zwei Parameter berechnet man dann den *nominal effort*.

Beispiel:
Für ein einfaches Projekt (organic) wird der nominale Effort durch folgende Formel wieder gegeben:

$$\textbf{Nominaler Effort} = 3.2 * (\text{KDSI})^{1.05} \textbf{ [Personenmonate]}$$

Die Konstanten (3.2 beziehungsweise 1.05) wurden von Boehm mittels seiner Projektdatenbank ermittelt.

Beispiel:
Für die Entwicklung eines 12'000 Zeilen Programmes (12 KDSI) benötigt man

NE = 3.2 * (12) ** 1.05 = 43 Personenmonate

Der nominale Effort muss nun noch mit weiteren Faktoren multipliziert werden. Boehm definiert insgesamt *15 Software Entwicklungs- Effort Multiplier*.

Software development effort multipliers

Cost Drivers	Very low	Low	Nominal	High	Very high	Extra high
Product Attributes						
Required software reliability	0.75	0.88	1.00	1.15	1.40	
Database size		0.94	1.00	1.08	1.16	
Product complexity	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
Execution time constraint			1.00	1.11	1.30	1.66
Main storage constraint			1.00	1.06	1.21	1.56
Virtual machine volatility*		0.87	1.00	1.15	1.30	
Computer turnaround time		0.87	1.00	1.07	1.15	
Personnel Attributes						
Analyst capabilities	1.46	1.19	1.00	0.86	0.71	
Applications experience	1.29	1.13	1.00	0.91	0.82	
Programmer capability	1.42	1.17	1.00	0.86	0.70	
Virtual machine experience*	1.21	1.10	1.00	0.90		
Programming language experience	1.14	1.07	1.00	0.95		
Project Attributes						
Use of modern programming practices	1.24	1.10	1.00	0.91	0.82	
Use of software tools	1.24	1.10	1.00	0.91	0.83	
Required development schedule	1.23	1.08	1.00	1.04	1.10	

*For a given software product, the underlying virtual machine is the complex of hardware and software (operating system, database management system) it calls on to accomplish its task.

Der obige Auszug aus dieser Tabelle zeigt, dass jeder der Multiplier bis zu sechs Werten haben kann, je nach Komplexität des Projektes.

Die Einschätzung der Komplexität ist den Programmierern überlassen.

Boehm gibt auch Hinweise darauf, wie man am Besten diese Parameter bestimmt.

Falls unser Programm nur einfache Kontrollanweisungen besitzt, wie if...then, do...while, case ... dann ist die Komplexität tief.

Randbemerkung
Das Beispiel weiter oben zeigt, dass man für ein 12'000 Zeilen Programm eine recht lange Zeit benötigt.
Dividiert man die Anzahl Zeilen durch den Aufwand, dann erhält man etwa 300 Programmzeilen pro Monat.
Eine Reaktion könnte sein:
Soviel schaffe ich ja pro Nacht!

Man darf aber nicht vergessen, dass die 300 Zeilen produktiver Programmcode sind.

Nur etwa 15% des totalen Entwicklungsaufwandes sind Modul Programmierung.

Diese Tabelle wird im Buch von Boehm im Detail besprochen.

Betrachten wir ein komplexeres Beispiel:
Boehm beschreibt die Entwicklung eines Kontrollsoftware Systems, basierend auf Mikroprozessoren, mit Echtzeitanforderungen.

Es wird geschätzt, dass das Programm 10'000 Zeilen umfassen wird als 10 KDSI.
Die Komplexität ist gross ; Boehm schlägt dafür die Konstanten 2.8 und 1.2 als Exponent vor.
Durch Einsetzen in die Formel erhalten wir

NE = 2.8 * (10) ** 1.2 = 44 Personenmonate

Als nächstes müssen wir mit Hilfe einer Komplexitätsabschätzung einen Komplexitätsfaktor bestimmen.

Software development effort multipliers

Effort Cost Drivers	Situation	Rating	Multiplier
Required software reliability	Serious financial consequences of software fault	High	1.15
Data base size	20,000 bytes	Low	0.94
Product complexity	Communications processing	Very high	1.30
Execution time constraint	Will use 70% of available time	High	1.11
Main storage constraint	45K of 64K store (70%)	High	1.06
Virtual machine volatility	Based on commercial microprocessor hardware	Nominal	1.00
Computer turnaround time	Two hour average turnaround time	Nominal	1.00
Analyst capabilities	Good senior analysts	High	0.86
Applications experience	Three years	Nominal	1.00
Programmer capability	Good senior programmers	High	0.86
Virtual machine experience	Six months	Low	1.10
Programming language experience	Twelve months	Nominal	1.00
Use of modern programming practices	Most techniques in use over one year	High	0.91
Use of software tools	At basic minicomputer tool level	Low	1.10
Required development schedule	Nine months	Nominal	1.00

Der Komplexitätsfaktor ergibt sich als Produkt dieser Multiplier, in unserem Beispiel ergibt dies 1.35. Damit erhalten wir als Projektaufwand:

1.35 * 44 = 59 Personenmonate

Mit dieser Zahl werden nun weitere Berechnungen durchgeführt:

- Wartungskosten
- Aufwand pro Phase
- ...

Das zeigt aber auch , dass eines der Probleme des Modells offensichtlich die Anzahl Programmzeilen ist, also eine Zahl, die nur sehr schwierig zu bestimmen ist beziehungsweise schwierig abzuschätzen sind vor dem Projekt.

8.2.4. COCOMO II

COCOMO wurde im Laufe der Zeit neueren Phasenmodellen und Entwicklungstechniken angepasst. COCOMO basierte noch auf dem Wasserfall Modell, welches ebenfalls von Boehm entwickelt wurde.

COCOMO II berücksichtigt Prototyping, 4GL, Objekt orientiertes Entwickeln von Systemen. Der Nachteil ist die wesentlich höhere Komplexität des neuen Modells.
Während COCOMO noch ein einziges Modell war, basierend auf KDSI als Parameter, umfasst COCOMO II im wesentlichen drei Grundmodelle:

1. *Application Composition Model*
   Dieses basiert auf Objekt Punkten, die analog zu Funktionspunkten definiert werden. Dieses Modell wird in den frühen Phasen der Entwicklung eingesetzt, also zu einem Zeitpunkt, zu dem noch wenig genaue Erkenntnisse zum Projekt vorliegen.
2. *Early Design Model*
   Dieses basiert auf dem FP Modell und wird erst eingesetzt, nach dem man bereits genauere Informationen über das zu entwickelnde Produkt gewonnen hat.
3. *Post-Architecture Model*
   Dieses wird eingesetzt, sobald die Architektur festgelegt worden ist, also zu einem späteren Zeitpunkt im Projekt.

Der zweite grosse Modell Unterschied ist die mathematische Form:

effort = a * (size) **b

wobei im COCOMO Modell die Grössen a und b je nach Projekttyp bestimmt sind:

- *organic* *b=1.05*
- *semi-detached* *b= 1.12*
- *embedded* *b=1.2*

In COCOMO II variiert der Wert von b zwischen 1.01 und 1.26, abhängig von anderen Grössen.

Der dritte Unterschied der Modelle resultiert aus der Wiederverwendung. In COCOMO wurde angenommen, dass die Einsparungen durch Wiederverwendung proportional zu den Anzahl Zeilen wiederverwendeter Programmcode ist. In COCOMO II werden andere Faktoren mit berücksichtigt, wie etwa Testaufwand.

Als viertes wurden die Komplexitätsfaktoren überholt: statt 15 sind es nun 17, davon 7 neue.

COCOMO II wurde mit Hilfe von 83 Projekten validiert. Aber es ist noch zu früh, abschliessende Kommentare zum Modell abzugeben.
Sie finden im Internet diverse Produkte, wie zum Beispiel j_cocomo, eine Implementierung von COCOMO (II) in Java.

8.2.5. Verfolgen der Aufwände und Kostenschätzungen

Die Planung alleine nützt einiges, reicht aber nicht aus. In jedem Projekt wird eine übergeordnete Stelle, sei es ein Geldgeber, ein Abteilungsleiter, ..., periodisch einen Statusbericht erwarten. Daher ist es wichtig, den Projektstatus laufend zu aktualisieren und allfällige Abweichungen frühzeitig zu dokumentieren, um Gegenmassnahmen ergreifen zu können.

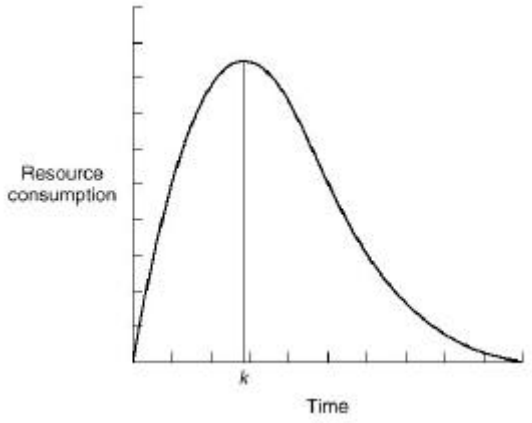
8.3. Komponenten eines Software Projekt Management Planes

Ein Software Projekt Management Plan besteht aus drei grundlegenden Komponenten:
1. dem offenen Aufwand
2. den Resourcen, mit denen das Projekt abgewickelt wird
3. finanzielle Resourcen (Projektkosten)

Die Software Entwicklung benötigt *Resourcen.* Die wichtigste Resource sind Personen, welche die Software entwickeln werden. Weitere Resourcen sind die Hardware und die System Software sowie Support Tools, mit deren Hilfe das Projekt abgewickelt wird.

Der Einsatz der Resourcen variiert je nach Projektphase. Wir haben weiter vorne dieses Phänomen bereits beschrieben. Es zeigt sich, dass die Rayleigh Verteilung

$$\mathbf{R_c = t/k^2 * exp(-t^2/(2k^2))\ \ 0 <= t < Infinity}$$



Beispiel:
zu Beginn des Projektes arbeiten 2 Programmierer am Projekt. Nach dem das Projekt gestartet wurde und anläuft, werden zusätzlich ein GUI Programmierer und ein Datenbank Programmierer in das Projekt aufgenommen. Diese werden nach Fertigstellung der entsprechenden Module, und Abnahme durch den Auftraggeber, nicht mehr benötigt. Das Projekt wird ab diesem Zeitpunkt nur noch durch zwei Programmierer weiter betreut.

Bei den Arbeiten, die in einem Projekt anfallen, unterscheiden wir zwei Kategorien:
1. *Projektfunktionen*
   Dies sind laufende Aktivitäten wie Projektmanagement Aufgaben. Auch die Qualitätskontrolle ist eine Projektfunktion
2. **Phasenbezogene Funktionen**
   Je nach Phase werden unterschiedliche Resourcen benötigt. Diese werden in einzelnen Aufgaben, *Tasks* und *Aktivitäten*, Activities eingesetzt. Aktivitäten sind die kleinsten Einheiten; Tasks bestehen also aus verschiedenen Aktivitäten.
   Resourcen müssen Resultate produzieren, die messbar sind (LOC, KDSI, ...)

Da Projekte zeitlich beschränkte Aktivitäten sein sollten, ist es wichtig periodisch *Meilensteine* zu definieren. Ein Meilenstein ist im Wesentlichen ein Kontrollpunkt, ein Zeitpunkt, zu dem zum Beispiel eine Komponente fertig gestellt sein muss.

Mit Hilfe periodischer *Reviews* wird in der Regel versucht, sicher zu stellen, dass das Projekt nicht aus dem Kurs läuft : das das produziert wird, was erwartet wird, zu den Kosten, die vertretbar und vertraglich fixiert wurden ,....

Falls ein Produkt oder Teilprodukt abgeschlossen und abgenommen ist, darf es nur noch nach festgelegten Regeln (Änderungsanforderungen, Erweiterungsanforderungen) geändert werden. Man sagt dazu auch : das Produkt wird zur *Baseline*.

8.4. Software Projekt Management Plan Framework

Verschiedene internationale Organisationen haben sich bemüht Raster für den Projektplan zu definieren. Die IEEE (Institut for Electric and Electronics Engineers), eine Berufsvereinigung der Elektroingenieure und Informatiker in den USA, hat folgendes Raster vorgeschlagen:

IEEE SPMP

1. Introduction
   - 1.1 Project Overview
   - 1.2 Project Deliverables
   - 1.3 Evolution of the Software Project Management Plan
   - 1.4 Reference Materials
   - 1.5 Definitions and Acronyms
2. Project Organization
   - 2.1 Process Model
   - 2.2 Organizational Structure
   - 2.3 Organizational Boundaries and Interfaces
   - 2.4 Project Responsibilities
3. Managerial Process
   - 3.1 Management Objectives and Priorities
   - 3.2 Assumptions, Dependencies, and Constraints
   - 3.3 Risk Management
   - 3.4 Monitoring and Controlling Mechanisms
   - 3.5 Staffing Plan
4. Technical Process
   - 4.1 Methods, Tools, and Techniques
   - 4.2 Software Documentation
   - 4.3 Project Support Functions
5. Work Packages, Schedule, and Budget
   - 5.1 Work Packages
   - 5.2 Dependencies
   - 5.3 Resources Requirements
   - 5.4 Budget and Resource Allocation
   - 5.5 Schedule

Additional Components

Dieser Aufbau ist im Standard IEEE 1058.1 festgehalten worden.

Der Einsatz eines Standards ist aus verschiedenen Gründen sinnvoll:
1. die einzelnen Projektpläne werden irgend wie vergleichbar
2. das Raster ist universell einsetzbar, nicht nur für kommerzielle Software

In der Schweiz kennt man auf diesem Gebiet einen anderen Standard: das Pflichtenheft. dieses wurde von der Schweizerischen Informatikervereinigung in Buchform veröffentlicht; auch der Ablauf einer Evaluation wird im Detail beschrieben.

Pflichtenheft
für eine Informatiklösung
Inhalt
1 Ausgangslage
2 Ist-Zustand
3 Ziele
4 Anforderungen
5 Mengengerüst, Verarbeitungshäufigkeiten
6 Aufbau der Offerte
7 Administratives
8 Fragenkatalog
Interner Verteiler (alphabetisch):...

Schauen wir uns einmal den IEEE Standard etwas genauer an:

8.5. IEEE Software Projekt Management Plan

Wir möchten kurz erläutern, was die einzelnen Punkte in der obigen Abbildung des IEEE Standards beinhalten.

8.5.1. Introduction

In den fünf Abschnitten dieses Kapitels wird eine Übersicht über das Projekt präsentiert und über das Produkt, welches entwickelt werden soll.

8.5.1.1. Project Overview

In diesem Abschnitt werden die Ziele beschrieben, die mit dem Projekt erreicht werden sollen. Die Aktivitäten werden aufgelistet inklusive den Resultaten, die erwartet werden. Meilensteine werden aufgeführt, benötigte Resourcen aufgelistet und eine Grobplanung für Budget und Projektplan präsentiert.

8.5.1.2. Project Deliverables

Die Ergebnisse, die an den Kunden abgeliefert werden, müssen festgehalten werden und beschrieben werden.

8.5.1.3. Evolution of the Software Project Management Plan

Da die Planung laufend den neusten Erkenntnissen angepasst werden muss, besteht die Notwendigkeit, den Plan laufend auf dem neusten, aktuellen Stand zu halten.
In diesem Abschnitt des Standards werden die formalen Abläufe beschrieben.

8.5.1.4. Reference Material

Alle Dokumente, auf die verwiesen wird, werden hier aufgeführt.

8.5.1.5. Definitions and Acronyms

In vielen Projekten werden Begriffe eingeführt, die für mehrere Projektmitglieder neu oder unklar sind. Dieser Abschnitt dient dem Festhalten klarer Begriffe.

8.5.2. Project Organization

In den vier Abschnitten des zweiten Kapitels wird die Projektorganisation beschrieben. Die Beschreibung umfasst die Struktur des Entwicklungsteams also auch der Gesamtorganisation, in der entwickelt wird.

8.5.2.1. Process Model

Das Prozessmodell beschreibt den Projektablauf in Form von Aktivitäten, die beim Testen, Konfigurationsmanagement, .. ausgeführt werden.
Die Beschreibung umfasst auch Meilensteine, Baselines, Reviews, Ergebnisse.

8.5.2.2. Organizational Structure

Die Managementstruktur der Entwicklungsorganisation wird beschrieben. verantworten werden festgehalten und Stellen gegen einander abgegrenzt.

8.5.2.3. Organizational Boundaries and Interfaces

Da die Projektmitglieder mit dem Kunden kommunizieren müssen, ist es wichtig, die Kommunikationsbeziehungen und -Linien zu fixieren. Wer ist wo für was wann in welchem Umfange verantwortlich?

8.5.2.4. Project Responsibilities

Für jede im Projekt benötigte Aktivität müssen Personen, Verantwortliche festgelegt werden.

8.5.3. Managerial Process

In diesem Kapitel wird beschrieben, wie das Projekt geleitet, gemanaged wird.

8.5.3.1. Management Objectives and Priorities

Zuerst werden die übergeordneten Ziele festgehalten, die Philosphie beschrieben, im Sinne einer allgemeinen Orientierung.
Auf der konkreteren Ebene wird beschrieben, wie und wann Berichte abgeliefert werden müssen, wie die Budget-, Risikomanagement-, Zeitplanungs- Abläufe aussehen.

8.5.3.2. Assumptions, Dependencies, and Constraints

Falls spezielle Voraussetzungen gegeben sind, werden diese hier aufgeführt.

8.5.3.3. Risk Management

Risikofaktoren werden soweit wie bekannt, aufgelistet und es wird aufgezeigt, wie die Risikofaktoren verfolgt werden können.

8.5.3.4. Monitoring and Controlling Mechanisms

Reporting Abläufe werden im Detail erklärt, ebenso der Ablauf von Audits, Reviews, ...

8.5.3.5. Staffing Plan

Die Projektmitglieder werden aufgelistet, zusammen mit allfällig bereits bekanntem Einsatz in bestimmten Phasen.

8.5.4. Technical Process

Technische Aspekte werden in drei Abschnitten festgehalten.

8.5.4.1. Methods, Tools and Techniques

Technische Aspekte der Hardware und Software werden im Detail beschrieben.
Systemsoftware, Datenbanksysteme, ... werden aufgelistet. Dies umfasst sowohl die Entwicklungsplatformen als auch die Zielplatformen.
Allfällig einzusetzende CASE Tools werden festgehalten.
Technische Standards werden beschrieben oder es wird darauf verwiesen, zum Beispiel Dokumentationsstandards.

8.5.4.2. Software Documentation

In diesem Abschnitt werden die Dokumentionsstandards und Anforderungen festgelegt für laufende Tätigkeiten, Software Reviews und Baselines.

8.5.4.3. Project Support Functions

Dieser Abschnitt legt fest, wie die Qualitätssicherung und Testpläne aussehen.

8.5.5. Work Packages, Schedule, and Budget

In fünf Abschnitten werden Arbeitspakete festgehalten, deren gegenseitige Abhängigkeiten, Resourcenbedarf und Budgets.

8.5.5.1. Work Packages

Arbeitspakete werden festgehalten und das Produkt der Arbeitspakete wird in Aktivitäten und Tasks zerlegt.

8.5.5.2. Dependencies

Da die einzelnen Arbeitspakete voneinander abhängen, können sich daraus bestimmte Konflikte ergeben. In diesem Abschnitt werden die Abhängigkeiten aufgezeichnet.

8.5.5.3. Resource Requirements

Die Resourcen werden auf der Zeitachse aufgelistet. Das ist in der Regel Teil eines ProjektmanagementTools (MS Project erlaubt die Eingabe und Auswertung dieser Informationen).

8.5.5.4. Budget and Resource Allocation

Die Resourcen, welche weiter vorne beschrieben wurden, werden monetär bewertet. Dabei werden die Finanzzahlen als Funktion der Zeit dargestellt: für die Finanzplanung ist es wichtig zu wissen, wann welche Zahlungen fällig sind, beziehungsweise wann mit welchen Einnahmen gerechnet werden kann.
Die Resourcen werden den einzelnen Aktivitäten und Aufgaben zugeteilt. Sie erhalten diese Darstellung, wenn auch mit einigem Aufwand, aus WinProject.

8.5.5.5. Schedule

Für jede Komponente wird ein detaillierter Zeitplan präsentiert. Dabei muss immer beachtet werden dass in der ersten Runde die Planung notwendigerweise unsicher ist.
Aber als Arbeitsgrundlage muss man davon ausgehen, dass mit dieser Planung gearbeitet wird.

8.5.5.6. Additional Components

Unter "zusätzliche Komponenten" werden weitere Punkte aufgeführt, die für den Projekterfolg und den Projektablauf wesentlich sind. Beispiele sind Unterlieferanten und die Planung der Zusammenarbeit mit diesen, aber auch Sicherheitsdispositive, Testpläne, Trainingspläne, Hardware Beschaffungspläne, Installationsplanungen und eventuell bereits ein Wartungsplan.

8.6. Testplanung

Obschon der Testplan eigentlich zum SPMP gehört, wird er oft vernachlässigt. Beim Einsatz neuerer Methoden, wie RUP (Rational Unified Process) für die Entwicklung eines Informatik Produktes werden bereits sehr früh mit Hilfe der Anwendungsfälle, Use Cases genannt, Grundlagen geschaffen, die für sinnvolle Tests eingesetzt werden können: die Use Cases bilden die Basis für Tests, denn darin steht, was der Benutzer vom neuen System erwartet. Damit wird auch die "Traceability", also der Bezug des Egebnisses zu den Anforderungen nicht verloren gehen.

Ähnlich ist es beim eXtreme Programming: dort wird versucht dauernd mit Tests zu arbeiten, dauernd ein lauffähiges System zu haben , wenn auch am Anfang mit wenig bis gar keiner Funktion; aber dadurch, dass das System immer läuft, fühlen sich die Entwickler wesentlich wohler und sicherer als im Fall einer Entwicklungsmethodik, bei der erst am Schluss beim Integrationstest festgestellt wird, dass doch etwas prinzipielles nicht so funktioniert, wie man sich das gedacht hatte.

In vielen Firmen wird für die Tests eine Software Qualitätssicherung SQA eingesetzt, bestehend aus hoch qualifizierten, verantwortlichen Mitgliedern des Teams.

Die SQA wird in der Regel versuchen statistische Daten zu sammeln: Anzahl (gefundene) Fehler pro Phase, Gewicht der Fehler (wie gravierend ist der Fehler), und eventuell Korrekturmassnahmen (Programmierstandard ändern, eventuell Fehler in einem Tool beseitigen, ...).

Beim Black Box Testing werden Tests durch geführt, ohne dass man sich um den inneren Aufbau der Module kümmert (diese sind black, undurchsichtig). Dabei stützt man sich auf die Spezifikation. Diese Tests sind also nicht "invented here" behaftet: der Tester kennt das Programm nur als Ganzes, er wird nicht wie der Programmierer eine bestimmte Vorauswahl treffen ("das funktioniert sowieso...").

Zusammenfassend:
der Testplan muss Teil des SPMP sein und bereits zu einem frühen Zeitpunkt erstellt werden . Nach der Spezifikationsphase ist eigentlich klar, was das System einmal alles können muss, was also zu testen ist.

8.7. Planung in Objekt Orientierten Projekten

Falls strukturierte Methoden angewendet werden, dann wird man die Planung im Sinne eines Stepwise refinement schrittweise verfeinern, vom Groben zum Detail.
Im Falle eines Objekt Orientierten Systems kennt man bereits recht früh eine bestimmte Granularität des System, seine Klassen oder Klassenkandidaten.

Im Falle von OO Techniken wird man also sehr schnell kleine Einheiten zur Verfügung haben, die auch testbar sein können. Dadurch kann die Planung auch verbessert werden, speziell bei der Wiederverwendung von Komponenten, sprich Klassen.
Dabei muss immer beachtet werden, dass die Summe der Bauteile noch kein Produkt ausmacht: diese müssen auch noch richtig zusammengesetzt werde, sonst funktioniert das System nicht!

Falls Komponenten von Anfang an als wieder verwendbar geplant werden, dann muss man beachten, dass diese Komponenten in der Regel sorgfältiger geplant und entwickelt werden müssen. Es hat sich gezeigt, dass man mit einem bis zu drei mal höheren Aufwand rechnen muss.

8.8. Trainingsanforderungen

Unter Training verstehen wir hier unter anderem:

- die Schulung des zukünftigen Anwenders
- die Schulung der Projektmitarbeiter

Oft wird, fälschlicherweise, behauptet, dass der zukünftige Anwender erst nach Fertigstellung geschult werden muss. Das stimmt nur bedingt. Oft werden im Projekt bestimmte Funktionen durch den Benutzer getestet werden, wie zum Beispiel ein GUI, eine bestimmte Ablaufsteuerung,...

Die Projektmitarbeiter werden speziell beim Einsatz neuster Technologien gezwungen sein, sich mit diesen vertraut zu machen und Schulungskurse zu besuchen.
Neue Betriebsysteme, GUI Builder, Programmiersprachen, neue Dokumentationsstandards oder Metoden, Entwicklungsmethoden, ... bedingen entweder ein Coaching oder der Besuch externer Kurse, Selbststudien-Kursen, die Beschaffung von Schulungsunterlagen.

8.9. Dokumentations-Standards

Software Dokumentation ist ein Übel. IBM hat einmal in den eigenen Reihen einige typische Zahlen zusammen gestellt:

- kommerzielle Software für den Eigengebrauch
  28 Seiten Dokumentation / KDSI
  Projektumfang : um die 50 KDSI
- kommerzielle Software für externe
  66 Seiten Dokumentation / KDSI
  Projektumfang : um die 50 KDSI
- Datenbanksystem IMS/360
  157 Seiten Dokumentation / KDSI
  Projektumfang : um die 166 KDSI

Offensichtlich gibt es riesige Unterschiede. Eine generelle Bemerkung ist angebracht: oft sieht man in Projektplänen, dass die Dokumentation erst nach der Entwicklung der Software erstellt wird. Das ist grundlegend falsch. Ein Besitzer eines Software Hauses erklärte einmal an einer Tagung zum Thema Software Qualität, dass die Dokumentation als erstes erstellt werden muss, damit der Kunde gleich entscheiden kann, ob er das Produkt möchte oder nicht. Falls nicht, dann wurden nur einige Resourcen für die Erstellung der Dokumentation "verbraucht".

Eine Untersuchung (63 Entwicklungs- und 25 Wartungs- Projekte) zeigt, dass pro 100 Stunden Entwicklung etwa 150 Stunden für die Dokumentation verwendet werden müssen (die Untersuchung stammt von Boehm).
Eine andere Studie, in der Boehm TRW Projekte untersuchte, zeigte, dass etwa 200 Stunden pro 100 Stunden Kodierung aufgewendet werden müssen.
Standards für die Dokumentation werden in vielen Projekten durch den Auftraggeber vorgegeben:
im Falle der US Army werden diese durch sogenannte MIL Standards festgelegt.
Auch falls die NASA der Auftraggeber ist, werden die Dokumentationsstandard in der Regel vorgeschrieben.
IEEE hat ebenfalls einen Standard definiert, der sich an die MIL Standards anlehnt. Dies ist bei IEEE Standards nicht unüblich, da das Militär einer der grössten Kunden vieler Ingenieurunternehmen ist. Der entsprechende IEEE Standard ist IEEE 1063 Software User Documentation.

Die Tests werden oft bereits beim erfassen der Benutzeranforderungen definiert. Der IEEE Standard für die Test Dokumentation ist ANSI/IEEE 829 Standard for Software Test Documentation.

8.10. CASE Tools für die Planung und Schätzung

Neben den üblichen Werkzeugen Excel und Word, sind für diese Aufgabe auch spezialisierte Werkzeuge erhältlich und üblich :

- COCOMO Software (COSTAR, J_COCOMO, ...)
- Spreadsheets (Excel, Lotus 1-2-3,...)
- Projekt Management Tools (WinProject, ...)

Bei komplexeren Projekten werden häufig Schnittstellen zu bestehenden Finanz- und Rechnungswesen verlangt, damit die Projektabrechnung vereinfacht werden kann. Dies bedingt komplexere Projektplanungs-Werkzeuge als WinProject.

Techniken sind in der Regel PERT und CPM.

Randbemerkung

PERT wurde 1957 durch die US Navy entwickelt. Das System wurde im Zusammenhang mit dem Bau der Polaris Missiles zum ersten Male eingesetzt.
Im Verlaufe des Projektes zeigte es sich, dass in komplexeren Fällen die PERT Netzwerke mit mehreren Varianten (statistisch) durchgerechnet werden müssen, um zuverlässige Aussagen über kritische Pfade machen zu können.

Wir wollen uns hier mit diesem Thema nicht näher beschäftigen, da die verschiedenen Techniken inklusive Tools in einem anderen Zusammenhang vorgestellt werden (Projektmanagement).

8.11. Testen des Software Management Planes

Der gesamte SPMP muss bevor er als verbindlich erklärt wird, von der SQA der Software Quality Assurance abgenommen werden.
Es muss allen Projektmitgliedern klar sein, was im Plan steht, was der Plan festlegt, was er empfiehlt und welche Konsequenzen er hat, zum Beispiel in Bezug auf Reporting, Dokumentation, Kommunikation, Ergebnisprüfung, Meilensteinen und anderes mehr.

8.12. Zusammenfassung

Das Hauptthema dieses Kapitels ist die Planung im Software Prozess. Wichtige Teile dieser Projektplanung sind die Abschätzung der Aufwände (Kosten, Arbeitsvolumen).

Spezielle Techniken und Modelle wie COCOMO werden besprochen und an Beispielen erläutert.

Verschiedene IEEE Standards werden im Kontext erwähnt und der IEEE SPMP wird in einigem Detail erläutert.

8.13. Aufgaben

1. Warum bezeichnen einige zynische Organisationen Milestones als Millstones?
   Schauen Sie in einem Wörterbuch nach, was Millstone bedeutet.
2. Sie sind Software Entwickler bei der KWV Software Firma. Vor einem Jahr wurde Ihnen mitgeteilt, dass das nächste Produkt
   18 Dateien, 59 Flows und 93 Prozesse
   besitzen werde.
   (i) bestimmen Sie mit Hilfe der FFP die Grösse des Projektes
   (ii) für die KWV ist d=812 USD. Wie teuer wird das Projekt gemäss FFP?
   (iii) das Projekt wurde een abgeschlossen. Es hat 124'000 USD gekostet.
   Welche Rückschlüsse erlauben Ihnen diese Aussagen in Bezug auf die Produktivität des Teams der KWV?
3. Ein geplantes Produkt hat
   7 einfache Eingaben
   8 durchschnittlich komplexe Eingaben
   11 Komplexe Eingaben
   5 durchschnittlich komplexe Ausgaben
   40 einfache Abfragen
   12 durchschnittlich komplexe Stammdateien
   18 komplexe Schnittstellen
   Bestimmen Sie UDP = unadjusted function points.
4. Im obigen Projekt sei der Influence Factor 52. Bestimmen Sie die Anzahl Functionspunkte.
5. Was denken Sie: warum ist LOC und KDSI trotz heftiger Kritik immer noch sehr verbreitet?
6. Sie haben sie Aufgabe ein 83 KDSI Produkt zu entwickeln, welches in Steuerungen (embedded system) eingesetzt werden soll. Die Datenbank ist recht gross und der Einsatz von Support Software Tools sehr gering. Schätzen Sie den Aufwand in Personalmonaten für das Projekt mit Hilfe von COCOMO Intermediate.
7. Sie sind für zwei organisches Projekt verantwortlich. Grösse des Projektes (geschätzt) 38 KDSI. Beide Projekte sind nominal. Projekt P1 zeigt extra grosse Komplexität und P2 extra tiefe Komplexität.
   Ihnen stehen zwei Teams zur Verfügung. Team A ist sehr erfahren, mit virtuellen Maschinen und Programmiersprachen. Team B hat tiefe Werte für alle fünf Attribute.
   (i) Berechnen Sie den Aufwand in Personenmonaten für Projektteam A in Projekt P1 und Projektteam B für Projekt P2.
   (ii) Berechnen Sie den Aufwand (Personenmonate) falls Team A Projekt P2 und Team B

Projekt P1 bearbeiten.
(iii) Welche der beiden Kombinationen ist sinnvoller?

8. Sie müssen ein 45 KDSI Produkt (organic) entwickeln. Das Projekt ist nominal in allen Aspekten:
   (i) falls der Personenmonat 8'400 USD kostet, wieviel kostet das Projekt?
   (ii) das Projektteam verlässt die Firma. Sie finden ein neues Team, sehr erfahren und hoch qualifiziert. Aber die Kosten pro Personenmonat steigen dadurch auf 11'200 USD. Wieviel Geld müssen Sie mehr oder weniger ausgeben? Verdienen oder verlieren Sie dadurch Geld im Projekt, falls Sie dieses gemäss der ersten Schätzung abrechnen müssen?
9. Sie sind für ein Projekt verantwortlich, in dem zum ersten Mal ein neuer Algorithmus eingesetzt werden soll (Wegoptimierung für ein Logistikunternehmen).
   Mit Hilfe des Intermediate COCOMO wurden die Kosten auf 430'000 USD geschätzt. Die Teammitglieder berechnen die Kosten mit Hilfe der Function Point Methode und erhalten Gesamtkosten von 890'000 USD, also zweimal soviel wie in der ersten Schätzung. Was machen Sie nun?
10. Zeigen Sie, als kleine Übung in Mathematik: die Rayleigh Distribution erreicht ihr Maximum für t=k. Welche Resourcen werden dann benötigt?
11. Gemäss IEEE SPMP ist ein Wartungsplan eine zusätzliche Komponente. Falls wir davon ausgehen, dass alle nicht trivialen Produkte gewartet werden müssen und die Wartung etwa zweimal soviel kostet wie die Entwicklung des Produktes, wie lässt sich dies rechtfertigen?
12. Lesen Sie dir Beschreibung des Air Gourmet Projektes. Warum ist eine Aufwandschätzung für dieses Projekt auf Grund der vorliegenden Angaben nicht möglich?
13. Lesen Sie den Artikel von Boehm:
    Barry W. Boehm, et al
    "Cost Models for Future Life Cycle Processes", Annals of Software Engineering Vol 1, (1995), pp. 57-94.
    sofern Sie den Artikel beschaffen können.