



5. Management der Software-Entwicklung

Themen dieses Kapitels:

- ☐ bessere/modernere Prozessmodelle
- ☐ Verbesserung/Qualität von Softwareprozessmodellen
- ☐ Projektmanagement(-werkzeuge)
- ☐ [optional: Kostenschätzung für Softwareprojekte]

Achtung:

Viele im folgenden vorgestellten Überlegungen sind nicht ausschließlich für **Software-**Entwicklungsprozesse geeignet, sondern werden ganz allgemein für die Steuerung komplexer technischer Entwicklungsprozesse eingesetzt.



Aufgaben des Managements:

- ❑ **Planungsaktivitäten:** Ziele definieren, Vorgehensweisen auswählen, Termine festlegen, Budgets vorbereiten, ...
 - ⇒ Vorgehensmodelle, Kostenschätzung, Projektpläne
- ❑ **Organisationsaktivitäten:** Strukturieren von Aufgaben, Festlegung organisatorischer Strukturen, Definition von Qualifikationsprofilen für Positionen, ...
 - ⇒ Rollenmodelle, Team-Modelle, Projektpläne
- ❑ **Personalaktivitäten:** Positionen besetzen, Mitarbeiter beurteilen, weiterbilden, ...
 - ⇒ nicht Thema dieser Vorlesung
- ❑ **Leistungsaktivitäten:** Mitarbeiter führen, motivieren, koordinieren, ...
 - ⇒ nicht Thema dieser Vorlesung
- ❑ **Kontrollaktivitäten:** Prozess- und Produktstandards entwickeln, Berichts- und Kontrollwesen etablieren, Prozesse und Produkte vermessen, Korrekturen, ...
 - ⇒ Qualitätsmanagement, insbesondere für Software-Entwicklungsprozesse



Ziele des Managements:

Hauptziel des Projektmanagements ist die **Erhöhung der Produktivität!**

Allgemeine Definition von Produktivität:

Produktivität = Produktwert / Aufwand (oder: Leistung / Aufwand)

Für Software-Entwicklung oft verwendete Definition:

Produktivität = Größe der Software / geleistete Mitarbeitertage

Probleme mit dieser Definition:

- ⇒ Maß für Größe der Software
- ⇒ Berücksichtigung der Produktqualität
- ⇒ Aufwand = Mitarbeitertage ?
- ⇒ Nutzen (Return Of Investment) = Größe der Software ?



Einflussfaktoren für Produktivität [ACF97]:

Angabe der Form “+ 1 : X” steht für Produktivitätssteigerung um maximal Faktor X

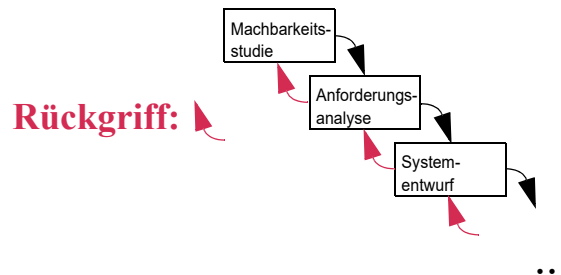
Angabe der Form “- 1 : Y” steht für Produktivitätsminderung um maximal Faktor Y.

- ⇒ Werkzeug-Einsatz:
- ⇒ Geeignete Methoden:
- ⇒ Produktkomplexität:
- ⇒ Hohe Zuverlässigkeitsanforderung:
- ⇒ Firmenkultur (corporate identity):
- ⇒ Arbeitsumgebung (eigenes Büro, abschaltbares Telefon, ...): +
- ⇒ Begabung der Mitarbeiter:
- ⇒ Erfahrung im Anwendungsgebiet:
- ⇒ Bezahlung, Berufserfahrung:



5.1 „Neuere“ Vorgehensmodelle

Die naheliegendste Idee zur Verbesserung des Wasserfallmodells ergibt sich durch die Einführung von **Zyklen** bzw. **Rückgriffen**. Sie erlauben Wiederaufnahmen früherer Phasen, wenn in späteren Phasen Probleme auftreten.

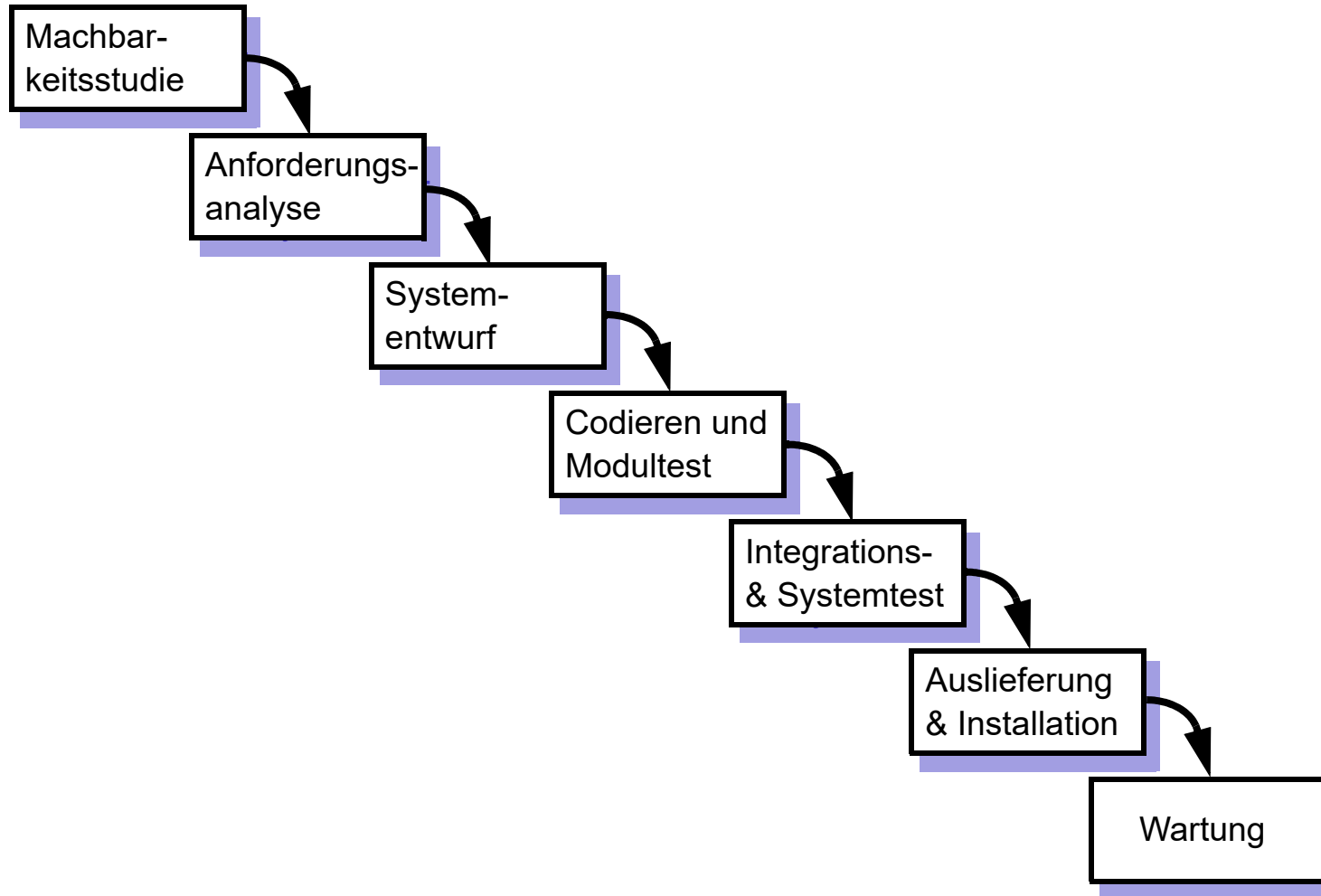


Weitere Vorgehensmodelle:

- ☐ das V-Modell (umgeklapptes Wasserfallmodell)
- ☐ das evolutionäre Modell (iteriertes Wasserfallmodell)
- ☐ Rapid Prototyping (Throw-Away-Prototyping)



Zur Erinnerung - das Wasserfallmodell (nach Royce)



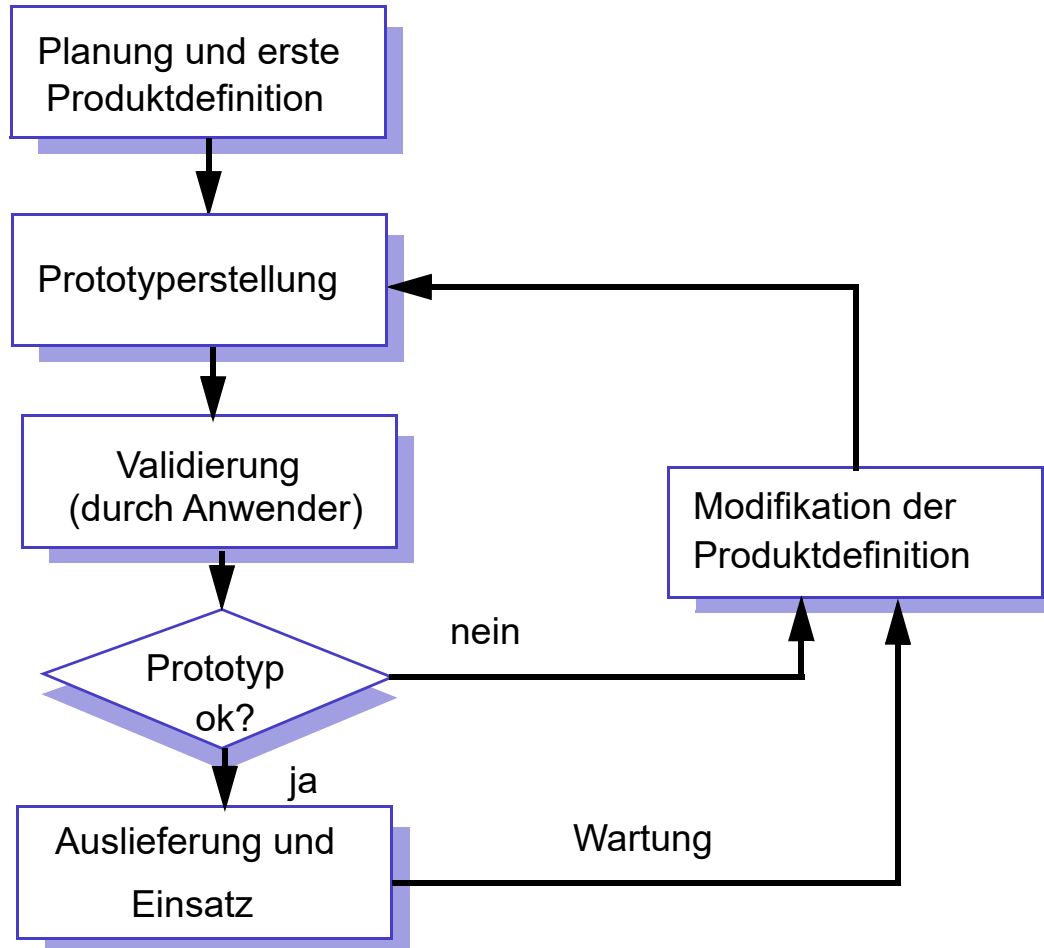


Probleme mit dem Wasserfallmodell insgesamt:

- ☹ Wartung mit ca. 60% des Gesamtaufwandes ist eine Phase
 - ⇒ andere Prozessmodelle mit Wartung als eigener Entwicklungsprozess
- ☹ zu Projektbeginn sind nur ungenaue Kosten- und Ressourcenschätzungen möglich
 - ⇒ Methoden zur Kostenschätzung anhand von Lastenheft (Pflichtenheft)
- ☹ ein Pflichtenheft kann nie den Umgang mit dem fertigen System ersetzen, das erst sehr spät entsteht (Risikomaximierung)
 - ⇒ andere Prozessmodelle mit Erstellung von Prototypen, ...
- ☹ Anforderungen werden früh eingefroren, notwendiger Wandel (aufgrund organisatorischer, politischer, technischer, ... Änderungen) nicht eingeplant
 - ⇒ andere Prozessmodelle mit evolutionärer Software-Entwicklung
- ☹ strikte Phaseneinteilung ist unrealistisch (Rückgriffe sind notwendig)
 - ⇒ andere Prozessmodelle mit iterativer Vorgehensweise



Evolutionäres Modell (evolutionäres Prototyping):





Bewertung des evolutionären Modells:

- 😊 es ist sehr früh ein (durch Kunden) evaluierbarer Prototyp da
- 😊 Kosten und Leistungsumfang des gesamten Softwaresystems müssen nicht zu Beginn des Projekts vollständig festgelegt werden
- 😊 Projektplanung vereinfacht sich durch überschaubarere Teilprojekte
- 😊 Systemarchitektur muss auf Erweiterbarkeit angelegt sein
- 😞 es ist schwer, Systemarchitektur des ersten Prototypen so zu gestalten, dass sie alle später notwendigen Erweiterungen erlaubt
- 😞 Prozess der Prototyperstellung nicht festgelegt
 - ⇒ Spiralmodell von Berry Böhm integriert Phasen des Wasserfallmodells
- 😞 evolutionäre Entwicklung der Anforderungsdefinition birgt Gefahr in sich, dass bereits realisierte Funktionen hinfällig werden
- 😞 Endresultat sieht ggf. wie Software nach 10 Jahren Wartung aus



Rapid Prototyping (Throw-Away-Prototyping):

Mit Generatoren, ausführbaren Spezifikationssprachen, Skriptsprachen etc. wird

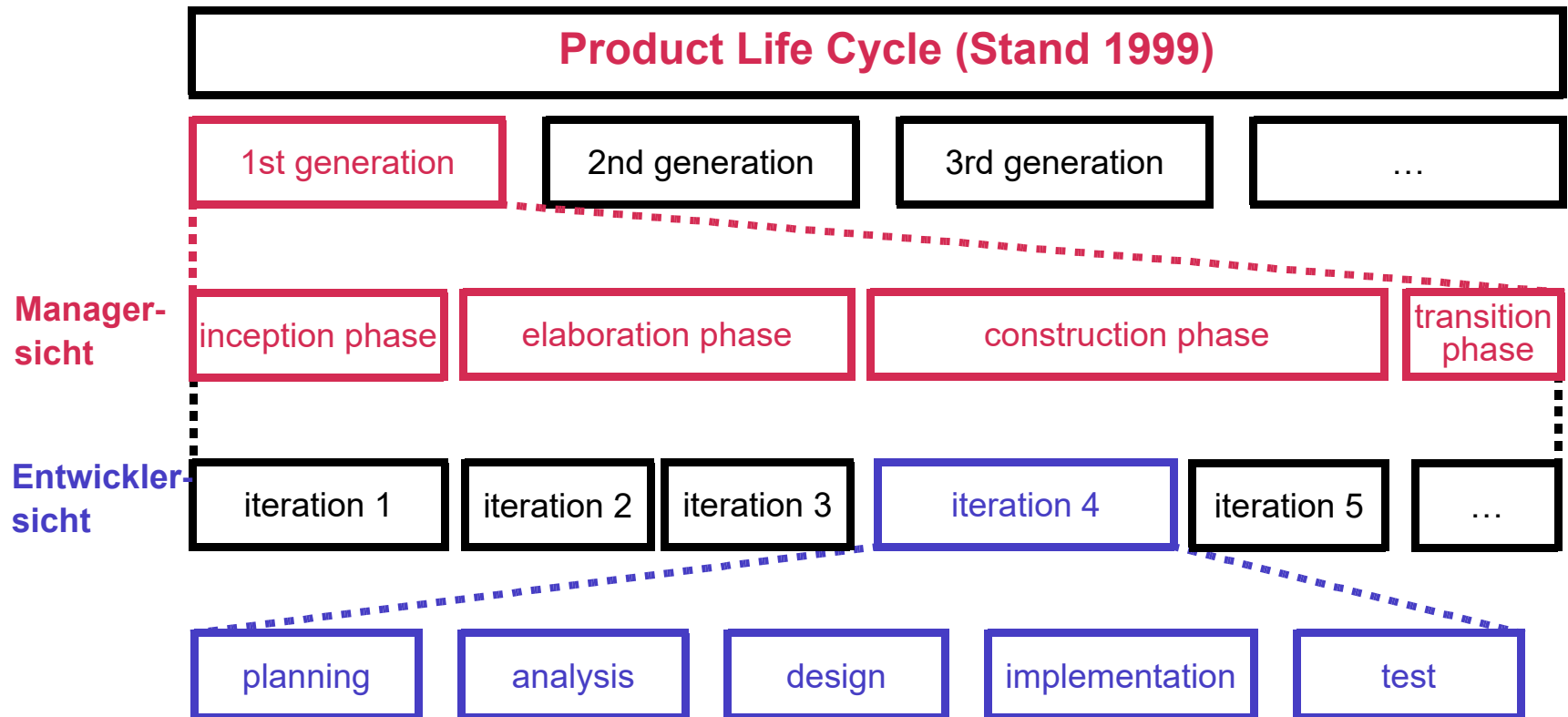
- ☞ Prototyp des Systems (seiner Benutzeroberfläche) realisiert
- ☞ dem Kunden demonstriert
- ☞ und anschließend weggeschmissen

Bewertung:

- 😊 erlaubt schnelle Klärung der Funktionalität und Risikominimierung
- 😊 Vermeidung von Missverständnissen zwischen Entwickler und Auftraggeber
- 😊 früher Test der Benutzerschnittstelle



5.2 Rational Unified Process für UML



- ➡ Firma IBM (ehemals Rational) dominiert(e) Entwicklung der Standard-OO-Modellierungssprache UML und des zugehörigen Vorgehensmodells.



Phasen der Lebenszyklusgenerationen:

- ❑ **Inception (Vorbereitung):** Definition des Problembereichs und Projektziels für Produktgeneration mit Anwendungsbereichsanalyse (Domain Analysis) und Machbarkeitsstudie (für erste Generation aufwändiger)
⇒ bei Erfolg weiter zu ...
- ❑ **Elaboration (Entwurf):** erste Anforderungsdefinition für Produktgeneration mit grober Softwarearchitektur und Projektplan (ggf. mit Rapid Prototyping)
⇒ bei Erfolg weiter zu ...
- ❑ **Construction (Konstruktion):** Entwicklung der neuen Produktgeneration als eine Abfolge von Iterationen mit Detailanalyse, -design, ... (wie beim evolutionären Vorgehensmodell)
⇒ bei Erfolg weiter zu ...
- ❑ **Transition (Einführung):** Auslieferung des Systems an den Anwender (inklusive Marketing, Support, Dokumentation, Schulung, ...)



Eigenschaften des Rational Unified Prozesses (RUP):

- ❑ **modellbasiert**: für die einzelnen Schritte des Prozesses ist festgelegt, welche Modelle (Dokumente) des Produkts zu erzeugen sind
- ❑ **prozessorientiert**: die Arbeit ist in eine genau definierte Abfolge von Aktivitäten unterteilt, die von anderen Teams in anderen Projekten wiederholt werden können.
- ❑ **iterativ und inkrementell**: die Arbeit ist in eine Vielzahl von Iterationen unterteilt, das Produkt wird inkrementell entwickelt.
- ❑ **risikobewusst**: Aktivitäten mit hohem Risiko werden identifiziert und in frühen Iterationen in Angriff genommen.
- ❑ **zyklisch**: die Produktentwicklung erfolgt in Zyklen (Generationen). Jeder Zyklus liefert eine neue als kommerzielles Produkt ausgelieferte Systemgeneration.
- ❑ **ergebnisorientiert**: jede Phase (Iteration) ist mit der Ablieferung eines definierten Ergebnisses meist zu einem konkreten Zeitpunkt (Meilenstein) verbunden

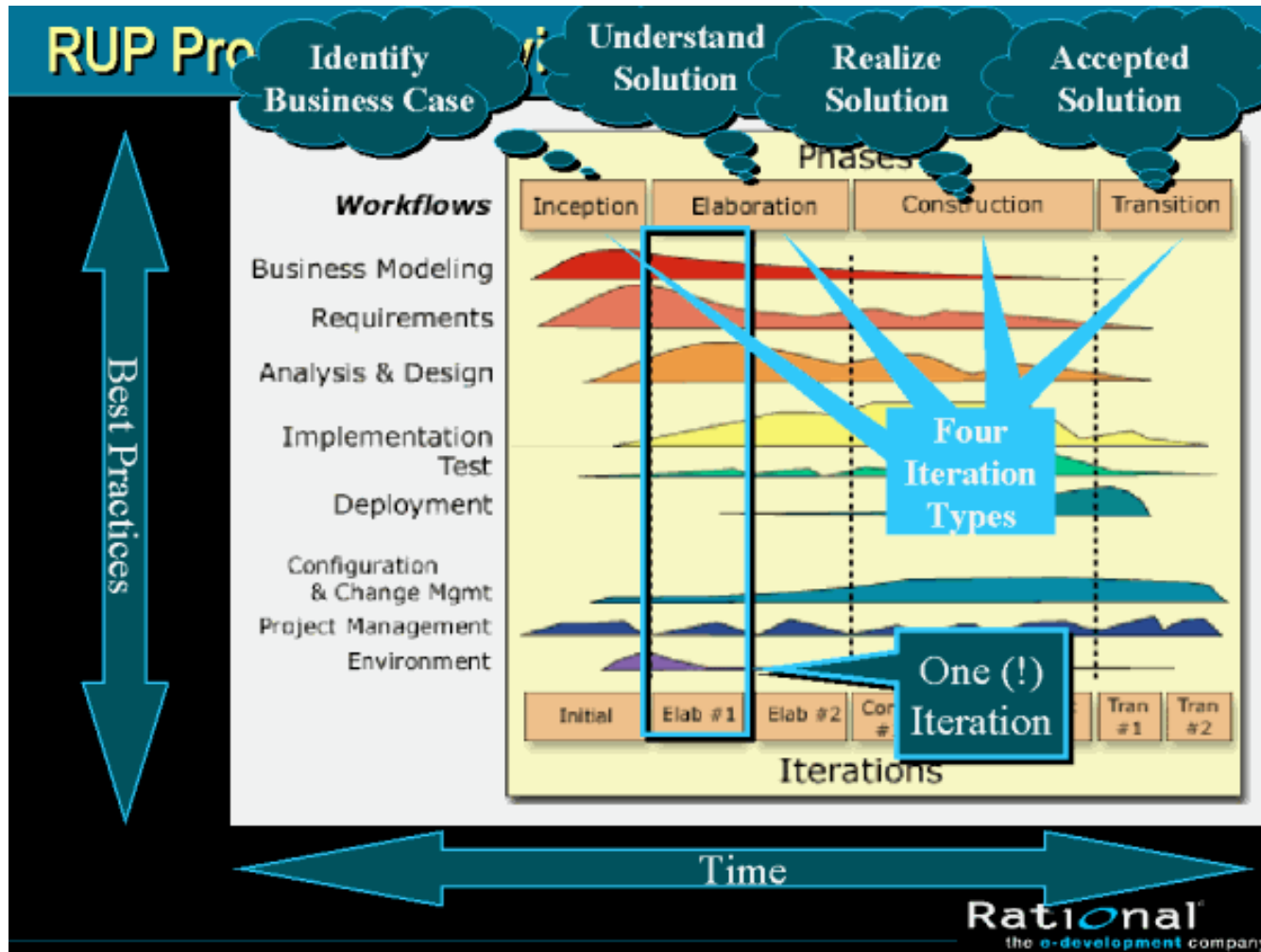


Faustregeln für die Ausgestaltung eines Entwicklungsprozesses:

- ☐ die Entwicklung einer **Produktgeneration** dauert höchstens 18 Monate
- ☐ eine **Vorbereitungsphase** dauert 3-6 Wochen und besteht aus einer Iteration
- ☐ eine **Entwurfsphase** dauert 1-3 Monate und besteht aus bis zu 2 Iterationen
- ☐ eine **Konstruktionsphase** dauert 1-9 Monate und besteht aus bis zu 7 Iterationen
- ☐ eine **Einführungsphase** dauert 4-8 Wochen und besteht aus einer Iteration
- ☐ jede **Iteration** dauert 4-8 Wochen (ggf. exklusive Vorbereitungs- und Nachbereitungszeiten, die mit anderen Iterationen überlappen dürfen)
- ☐ das gewünschte **Ergebnis** (Software-Release) einer Iteration ist spätestens bei ihrem Beginn festgelegt (oft Abhängigkeit von Ergebnissen vorheriger Iterationen)
- ☐ die **geplante Zeit** für eine Iteration wird nie (höchstens um 50%) überschritten
- ☐ innerhalb der Konstruktionsphase wird mindestens im wöchentlichen Abstand ein **internes Software-Release** erstellt
- ☐ mindestens 40% **Reserve** an Projektlaufzeit für unbekannte Anforderungen, ...



Arbeitsbereiche (Workflows) im RUP:



siehe RUP-Resource-Center:

<http://www.rational.net/rupcenter>



Anmerkungen zu den Arbeitsbereichen (Workflows) des RUP:

- ☐ **Business Modeling** befasst sich damit, das Umfeld des zu erstellenden Softwaresystems zu erfassen (Geschäftsvorfälle, Abläufe, ...)
- ☐ **Requirements (Capture)** befasst sich damit die Anforderungen an ein Softwaresystem noch sehr informell zu erfassen
- ☐ **Analysis/Design** präzisiert mit grafischen Sprachen (Klassendiagramme etc.) die Anforderungen und liefert Systemarchitektur
- ☐ **Implementation/Test** entspricht den Aktivitäten in den Phasen „Codierung bis Integrationstest“ des Wasserfallmodells
- ☐ **Deployment** entspricht „Auslieferung und Installation“ des Wasserfallmodells
- ☐ **Configuration Management** befasst sich mit der Verwaltung von Softwareversionen und -varianten
- ☐ **Project Management** mit der Steuerung des Entwicklungsprozesses selbst
- ☐ **Environment** bezeichnet die Aktivitäten zur Bereitstellung benötigter Ressourcen (Rechner, Werkzeuge, ...)



Bewertung des (Rational) Unified Prozesses:

- 😊 Manager hat die grobe “Inception-Elaboration-Construction-Transition”-Sicht
- 😊 Entwickler hat zusätzlich die feinere arbeitsbereichsorientierte Sicht
- 😊 Wartung ist eine Abfolge zu entwickelnder Produktgenerationen
- 😊 es wird endgültig die Illusion aufgegeben, dass Analyse, Design, ... zeitlich begrenzte strikt aufeinander folgende Phasen sind
- 😊 es gibt „Open Unified Process“ im Eclipse-Umfeld
(<http://epf.eclipse.org/wikis/openup/>)
- 😞 sehr komplexes Vorgehensmodell für modellbasierte SW-Entwicklung
- 😞 nicht mit Behördenstandards (V-Modell, ...) richtig integriert
- 😞 Qualitätssicherung ist kein eigener Aktivitätsbereich



5.3 Leichtgewichtige Prozessmodelle

Herkömmlichen Standards für Vorgehensmodelle zur Softwareentwicklung (V-Modell, RUP) wird vorgeworfen, dass

- ⇒ sie sehr starr sind
- ⇒ Unmengen an Papier produzieren
- ⇒ und nutzlos Arbeitskräfte binden

Deshalb werden seit einiger Zeit sogenannte „leichtgewichtige“ Prozessmodelle (**light-weight processes**) unter dem Schlagwort „Agile Prozessmodelle“ propagiert, die sinnlosen bürokratischen Overhead vermeiden wollen.

- ⇒ siehe separater Foliensatz zu dieser Thematik



5.4 Verbesserung der Prozessqualität

Ausgangspunkt der hier vorgestellten Ansätze sind folgende Überlegungen:

- ⇒ Softwareentwicklungsprozesse sind selbst Produkte, deren Qualität überwacht und verbessert werden muss
- ⇒ bei der Softwareentwicklung sind bestimmte Standards einzuhalten (Entwicklungsprozess muss dokumentiert und nachvollziehbar sein)
- ⇒ es bedarf kontinuierlicher Anstregungen, um die Schwächen von Entwicklungsprozessen zu identifizieren und zu eliminieren

Hier vorgestellte Ansätze:

- ⇒ **ISO 9000 Normenwerk** (Int. Standard für die Softwareindustrie)
- ⇒ **Capability Maturity Model** (CMM/CMMI) des Software Engineering Institutes (SEI) an der Carnegie Mellon University
- ⇒ ISO-Norm **SPiCE** (<http://www.sqi.gu.edu.au/SPICE/>) integriert und vereinheitlicht CMM und ISO 9000



Qualitätssicherung mit der ISO 9000:

Das **ISO 9000 Normenwerk** (<http://www.iso-9000.co.uk/>) legt für das Auftraggeber-Lieferantenverhältnis einen allgemeinen organisatorischen Rahmen zur Qualitätssicherung fest.

Das **ISO 9000 Zertifikat** bestätigt, dass die Verfahren eines Unternehmens der ISO 9000 Norm entsprechen.

Wichtige Teile:

- ☐ **ISO 9000-1**: allgemeine Einführung und Überblick
- ☐ **ISO 9000-3**: Anwendung von ISO 9001 auf Softwareproduktion
- ☐ **ISO 9001**: Modelle der Qualitätssicherung in Design/Entwicklung, Produktion, Montage und Kundendienst
- ☐ **ISO 9004**: Aufbau und Verbesserung eines Qualitätsmanagementsystems



Von ISO 9000-3 vorgeschriebene Dokumente:

- ❑ **Vertrag Auftraggeber - Lieferant:** Tätigkeiten des Auftraggebers, Behandlung von Anforderungsänderungen, Annahmekriterien (Abnahmetest), ...
- ❑ **Spezifikation:** funktionale Anforderungen, Ausfallsicherheit, Schnittstellen, ... des Softwareprodukts
- ❑ **Entwicklungsplan:** Zielfestlegung, Projektmittel, Entwicklungsphasen, Management, eingesetzte Methoden und Werkzeuge, ...
- ❑ **Qualitätssicherungsplan:** Qualitätsziele (messbare Größen), Kriterien für Ergebnisse v. Entwicklungsphasen, Planung von Tests, Verifikation, Inspektionen
- ❑ **Testplan:** Teststrategie (für Integrationstest), Testfälle, Testwerkzeuge, Kriterien für Testvollständigkeit/Testende
- ❑ **Wartungsplan:** Umfang, Art der Unterstützung, ...
- ❑ **Konfigurationsmanagement:** Plan für Verwaltung von Softwareversionen und Softwarevarianten



Von ISO 9000-3 vorgeschriebene Tätigkeiten:

- ❑ **Konfigurationsmanagement** für Identifikation und Rückverfolgung von Änderungen, Verwaltung parallel existierender Varianten
- ❑ **Dokumentenmanagement** für geordnete Ablage und Verwaltung aller bei der Softwareentwicklung erzeugten Dokumente
- ❑ **Qualitätsaufzeichnungen** (Fehleranzahl oder Metriken) für Verbesserungen am Produkt und Prozess
- ❑ **Festlegung** von Regeln, Praktiken und Übereinkommen für ein Qualitätssicherungssystem
- ❑ **Schulung** aller Mitarbeiter sowie Verfahren zur Ermittlung des Schulungsbedarfes

Zertifizierung:

Die Einhaltung der Richtlinien der Norm wird von unabhängigen Zertifizierungsstellen im jährlichen Rhythmus überprüft.



Bewertung von ISO 9000:

- 😊 lenkt die Aufmerksamkeit des Managements auf Qualitätssicherung
- 😊 ist ein gutes Marketing-Instrument
- 😊 reduziert das Produkthaftungsrisiko (Nachvollziehbarkeit von Entscheidungen)
- 😞 Nachvollziehbarkeit und Dokumentation von Prozessen reicht aus
- 😞 keine Aussage über Qualität von Prozessen und Produkten
- 😞 (für kleine Firmen) nicht bezahlbarer bürokratischer Aufwand
- 😞 Qualifikation der Zertifizierungsstellen umstritten
- 😞 oft große Abweichungen zwischen zertifiziertem Prozess und realem Prozess



Das Capability Maturity Model (CMM):

Referenzmodell zur Beurteilung von Softwarelieferanten, vom Software Engineering Institute entwickelt (<http://www.sei.cmu.edu/cmm/cmms.html>).

- ⇒ Softwareentwicklungsprozesse werden in **5 Reifegrade** unterteilt
- ⇒ Reifegrad (**maturity**) entspricht Qualitätsstufe der Softwareentwicklung
- ⇒ höhere Stufe beinhaltet Anforderungen der tieferen Stufen

Stufe 1 - chaotischer initialer Prozess (ihr Stand vor dieser Vorlesung):

- ❑ Prozess-Charakteristika:
 - ⇒ unvorhersehbare Entwicklungskosten, -zeit und -qualität
 - ⇒ kein Projektmanagement, nur „Künstler“ am Werke
- ❑ notwendige Aktionen:
 - ⇒ Planung mit Kosten- und Zeitschätzung einführen
 - ⇒ Änderungs- und Qualitätssicherungsmanagement



Stufe 2 - wiederholbarer intuitiver Prozess (Stand nach dieser Vorlesung?):

- ❑ Prozess-Charakteristika:
 - ⇒ Kosten und Qualität schwanken, gute Terminkontrolle
 - ⇒ Know-How einzelner Personen entscheidend
- ❑ notwendige Aktionen:
 - ⇒ Prozessstandards entwickeln
 - ⇒ Methoden (für Analyse, Entwurf, Testen, ...) einführen

Stufe 3 - definierter qualitativer Prozess (Stand der US-Industrie 1989?):

- ❑ Prozess-Charakteristika:
 - ⇒ zuverlässige Kosten- und Terminkontrolle, schwankende Qualität
 - ⇒ institutionalisierter Prozess, unabhängig von Individuen
- ❑ notwendige Aktionen:
 - ⇒ Prozesse vermessen und analysieren
 - ⇒ quantitative Qualitätssicherung



Stufe 4 - gesteuerter/geleiteter quantitativer Prozess:

- ❑ Prozess-Charakteristika:
 - ⇒ gute statistische Kontrolle über Produktqualität
 - ⇒ Prozesse durch Metriken gesteuert
- ❑ notwendige Aktionen:
 - ⇒ instrumentierte Prozessumgebung (mit Überwachung)
 - ⇒ ökonomisch gerechtfertigte Investitionen in neue Technologien

Stufe 5 - optimierender rückgekoppelter Prozess:

- ❑ Prozess-Charakteristika:
 - ⇒ quantitative Basis für Kapitalinvestitionen in Prozessautomatisierung und -verbesserung
- ❑ notwendige Aktionen:
 - ⇒ kontinuierlicher Schwerpunkt auf Prozessvermessung und -verbesserung (zur Fehlervermeidung)



Stand von Organisationen im Jahre 2000 (2007):

Daten vom Software Engineering Institute (SEI) aus dem Jahr 2000 (2007) unter

<http://www.sei.cmu.edu/appraisal-program/profile/profile.html>

(Die Daten in Klammern betreffen das Jahr 2007 für den Vergleich mit dem Stand im Jahr 2000; die genannte URL existiert nicht mehr):

- ☐ 32,3 % (1,7 %) der Organisationen im Zustand „initial“
- ☐ 39,3 % (32,7 %) der Organisationen im Zustand „wiederholbar“
- ☐ 19,4 % (36,1 %) der Organisationen im Zustand „definiert“
- ☐ 5,4 % (4,2 %) der Organisationen im Zustand „kontrolliert“
- ☐ 3,7 % (16,4 %) der Organisationen im Zustand „optimierend“

Genauere Einführung in CMM findet man in [Dy02]; weitere Informationen zum Nachfolger CMMI von CMM (siehe folgende Seiten) findet man unter:

<http://cmmiinstitute.com/>



ISO 9000 und CMM im Vergleich:

- ❑ Schwerpunkt der **ISO 9001** Zertifizierung liegt auf Nachweis eines Qualitätsmanagementsystems im Sinne der Norm
 - ⇒ allgemein für Produktionsabläufe geeignet
 - ⇒ genau ein Reifegrad wird zertifiziert
- ❑ **CMM** konzentriert sich auf Qualitäts- und Produktivitätssteigerung des gesamten Softwareentwicklungsprozesses
 - ⇒ auf Softwareentwicklung zugeschnitten
 - ⇒ dynamisches Modell mit kontinuierlichem Verbesserungsdruck
- ❑ ISO-Norm **SPiCE** (<http://www.sqi.gu.edu.au/SPICE/>) integriert und vereinheitlicht CMM und ISO 9000 (als ISO/IEC 15504)



SPiCE = Software Process Improvement and Capability dEtermination:

Internationale Norm für **Prozessbewertung** (und Verbesserung). Sie bildet einheitlichen Rahmen für Bewertung der Leistungsfähigkeit von Organisationen, deren Aufgabe Entwicklung oder Erwerb, Lieferung, Einführung und Betreuung von Software-Systemen ist. Norm legt Evaluierungsprozess und Darstellung der Evaluierungsergebnisse fest.

Unterschiede zu CMM:

- ☐ orthogonale Betrachtung von Reifegraden und Aktivitätsbereichen
- ☐ deshalb andere Definition der 5 Reifegrade (z.B. „1“ = alle Aktivitäten eines Bereiches sind vorhanden, Qualität der Aktivitäten noch unerheblich, ...)
- ☐ jedem Aktivitätsbereich oder Unterbereich kann ein anderer Reifegrad zugeordnet werden



Aktivitätsbereiche von SPiCE:

❑ **Customer-Supplier-Bereich:**

- ⇒ Aquisition eines Projektes (Angebotserstellung, ...)
- ⇒ ...

❑ **Engineering-Bereich:**

- ⇒ Software-Entwicklung (Anforderungsanalyse, ... , Systemintegration)
- ⇒ Software-Wartung

❑ **Support-Bereich:**

- ⇒ Qualitätssicherung
- ⇒ ...

❑ **Management-Bereich:**

- ⇒ Projekt-Management
- ⇒ ...

❑ **Organisations-Bereich:**

- ⇒ Prozess-Verbesserung
- ⇒ ...



CMMI = Capability Maturity Model Integration (neue Version von CMM):

CMMI ist die **neue Version des Software Capability Maturity Model**. Es ersetzt nicht nur verschiedene Qualitäts-Modelle für unterschiedliche Entwicklungs-Disziplinen (z.B. für Software-Entwicklung oder System-Entwicklung), sondern integriert diese in einem neuen, modularen Modell. Dieses modulare Konzept ermöglicht zum einen die Integration weiterer Entwicklungs-Disziplinen (z.B. Hardware-Entwicklung), und zum anderen auch die Anwendung des Qualitätsmodells in übergreifenden Disziplinen (z.B. Entwicklung von Chips mit Software).

Geschichte von CMM und CMMI:

- ⇒ 1991 wird Capability Maturity Model 1.0 herausgegeben
- ⇒ 1993 wird CMM überarbeitet und in der Version 1.1 bereitgestellt
- ⇒ 1997 wird CMM 2.0 kurz vor Verabschiedung vom DoD zurückgezogen
- ⇒ 2000 wird CMMI als Pilotversion 1.0 herausgegeben
- ⇒ 2002 wird CMMI freigegeben
- ⇒ Ende 2003 ist die Unterstützung von CMM ausgelaufen



Eigenschaften von CMMI:

Es gibt **Fähigkeitsgrade** für einzelne Prozessgebiete (ähnlich zu SPiCE):

0 - Incomplete:

Ausgangszustand, keine Anforderungen

1 - Performed:

die spezifischen Ziele des Prozessgebiets werden erreicht

2 - Managed:

der Prozess wird gemanagt

3 - Defined:

der Prozess wird auf Basis eines angepassten Standard-Prozesses gemanagt und verbessert

4 - Quantitatively Managed:

der Prozess steht unter statistischer Prozesskontrolle

5 - Optimizing:

der Prozess wird mit Daten aus der statistischen Prozesskontrolle verbessert



Eigenschaften von CMMI - Fortsetzung:

Es gibt **Reifegrade**, die Fähigkeitsgrade auf bestimmten Prozessgebieten erfordern (ähnlich zu CMM):

1- Initial:

keine Anforderungen, diesen Reifegrad hat jede Organisation automatisch

2 - Managed:

die Projekte werden gemanagt durchgeführt und ein ähnliches Projekt kann erfolgreich wiederholt werden

3 - Defined:

die Projekte werden nach einem angepassten Standard-Prozess durchgeführt, und es gibt eine kontinuierliche Prozessverbesserung

4 - Quantitatively Managed:

es wird eine statistische Prozesskontrolle durchgeführt

5 - Optimizing:

die Prozesse werden mit Daten aus statistischen Prozesskontrolle verbessert



Konsequenzen für die „eigene“ Software-Entwicklung:

Im Rahmen von Studienarbeiten, Diplomarbeiten, ... können Sie keinen CMM(I)-Level-5- oder SPiCE-Software-Entwicklungsprozess verwenden, aber:

- ❑ Einsatz von Werkzeugen für **Anforderungsanalyse und Modellierung**
 - ⇒ in der Vorlesung „Software-Engineering - Einführung“ behandelt
- ❑ Einsatz von **Konfigurations- und Versionsmanagement**-Software
 - ⇒ wird in dieser Vorlesung behandelt
- ❑ Einsatz von Werkzeugen für systematisches **Testen, Messen** der Produktqualität
 - ⇒ wird in dieser Vorlesung behandelt
- ❑ Ergänzender Einsatz von „**Extreme Programming**“-Techniken (z.B. „Test first“)
 - ⇒ siehe „Software-Praktikum“ und z.B. [Be99] vom Erfinder Kent Beck
- ❑ Einsatz von Techniken zur Verbesserung des „persönlichen“ Vorgehensmodells
 - ⇒ siehe [Hu96] über den „**Personal Software Process**“
(Buchautor Humphrey ist einer der „Erfinder“ von CMM)



5.5 Projektpläne und Projektorganisation

Am Ende der Machbarkeitsstudie steht die Erstellung eines Projektplans mit

- ⇒ Identifikation der einzelnen **Arbeitspakete**
- ⇒ **Terminplanung** (zeitliche Aufeinanderfolge der Pakete)
- ⇒ **Ressourcenplanung** (Zuordnung von Personen zu Paketen, ...)

Hier wird am deutlichsten, dass eine Machbarkeitsstudie ohne ein grobes Design der zu erstellenden Software nicht durchführbar ist, da:

- ⇒ Arbeitspakete ergeben sich aus der Struktur der Software
- ⇒ Abhängigkeiten und Umfang der Pakete ebenso
- ⇒ Realisierungsart der Pakete bestimmt benötigte Ressourcen

Konsequenz: Projektplanung und -organisation ist ein fortlaufender Prozess.
Zu Projektbeginn hat man nur einen groben Plan, der sukzessive verfeinert wird.



Terminologie:

- ❑ **Prozessarchitektur** = grundsätzliche Vorgehensweise einer Firma für die Beschreibung von Software-Entwicklungsprozessen (Notation, Werkzeuge)
- ❑ **Prozessmodell** = Vorgehensmodell = von einer Firma gewählter Entwicklungsprozess (Wasserfallmodell oder RUP oder ...)
- ❑ **Projektplan** = an einem Prozessmodell sich orientierender Plan für die Durchführung eines konkreten Projektes
- ❑ **Vorgang** = Aufgabe = Arbeitspaket = abgeschlossene Aktivität in Projektplan, die
 - ⇒ bestimmte Eingaben (Vorbedingungen) benötigt und Ausgaben produziert
 - ⇒ Personal und (sonstige) Betriebsmittel für Ausführung braucht
 - ⇒ eine bestimmte Zeitdauer in Anspruch nimmt
 - ⇒ und Kosten verursacht und/oder Einnahmen bringt
- ❑ **Phase** = Zusammenfassung mehrerer zusammengehöriger Vorgänge zu einem globalen Arbeitsschritt
- ❑ **Meilenstein** = Ende einer Gruppe von Vorgängen (Phase) mit besonderer Bedeutung (für die Projektüberwachung) und wohldefinierten *Ergebnissen*



Beispielprojekt (Kundenbeschreibung der Anforderungen):

Motor Vehicle Reservation System (MVRS)

A rental office lends motor vehicles of different types. The assortment comprises cars, vans, and trucks. Vans are small trucks, which may be used with the same driving license as cars. Some client may reserve motor vehicles of a certain category for a certain period. He or she has to sign a reservation contract. The rental office guarantees that a motor vehicle of the desired category will be available for the requested period. The client may cancel the reservation at any time. When the client fetches the motor vehicle he or she has to sign a rental contract and optionally an associated insurance contract. Within the reserved period, at latest at its end, the client returns the motor vehicle and pays the bill.

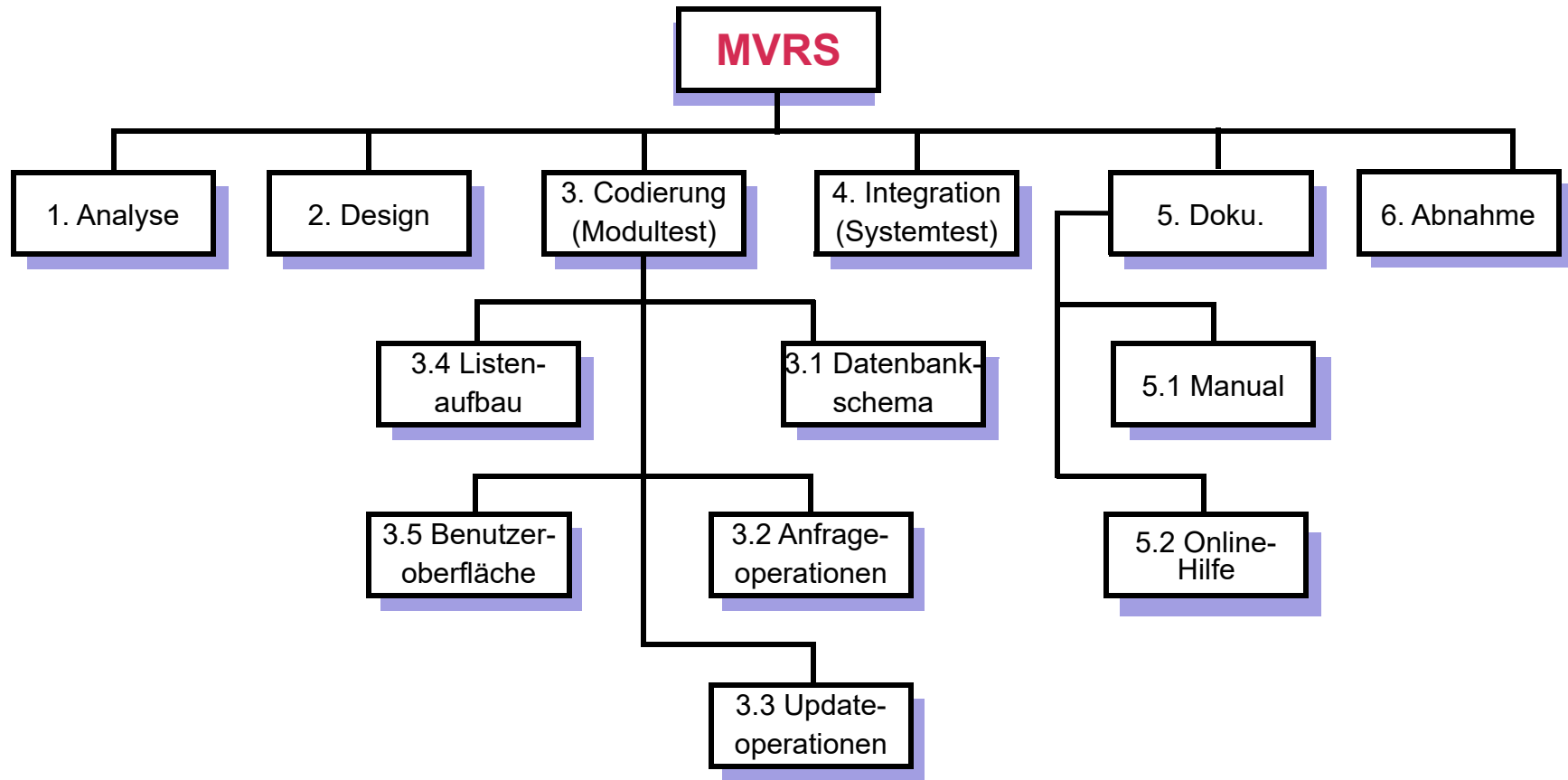


Annahmen für MVRs-Projektplanung über (Grob-)Design der Software:

- ❑ es wird eine (sehr einfache) Dreischichtenarchitektur verwendet mit Teilsystemen für Benutzeroberfläche, fachliche Funktionalität und Datenhaltung
- ❑ alle Daten werden in einer Datenbank gespeichert
 - ⇒ Datenbankschema muss entworfen werden
 - ⇒ Anfrageoperationen setzen auf Schema auf und geben Ergebnisse in Listenform aus
 - ⇒ Update-Operationen setzen auf Datenbankschema auf und sind unabhängig von Anfrageoperationen
- ❑ die Realisierung der Benutzeroberfläche ist von der Datenbank entkoppelt, für den sinnvollen Modultest der Operationen braucht man aber die Oberfläche
- ❑ um das Beispiel nicht zu kompliziert zu gestalten, wird das Wasserfallmodell mit Integration der einzelnen Teilsysteme im „Big Bang“-Testverfahren verwendet
- ❑ gedruckte Manuale und Online-Hilfe enthalten Screendumps der Benutzeroberfläche (teilweise parallele Bearbeitung trotzdem möglich)



Aufteilung des MVRS-Projekts in Arbeitspakete (Aufgaben):



Im obigen Bild fehlen noch die Angaben für die geschätzten Arbeitszeiten zur Bearbeitung der einzelnen Teilaufgaben des „Motor Vehicle Reservation System“-Projekts



Organisation von Aufgaben mit MS Project:

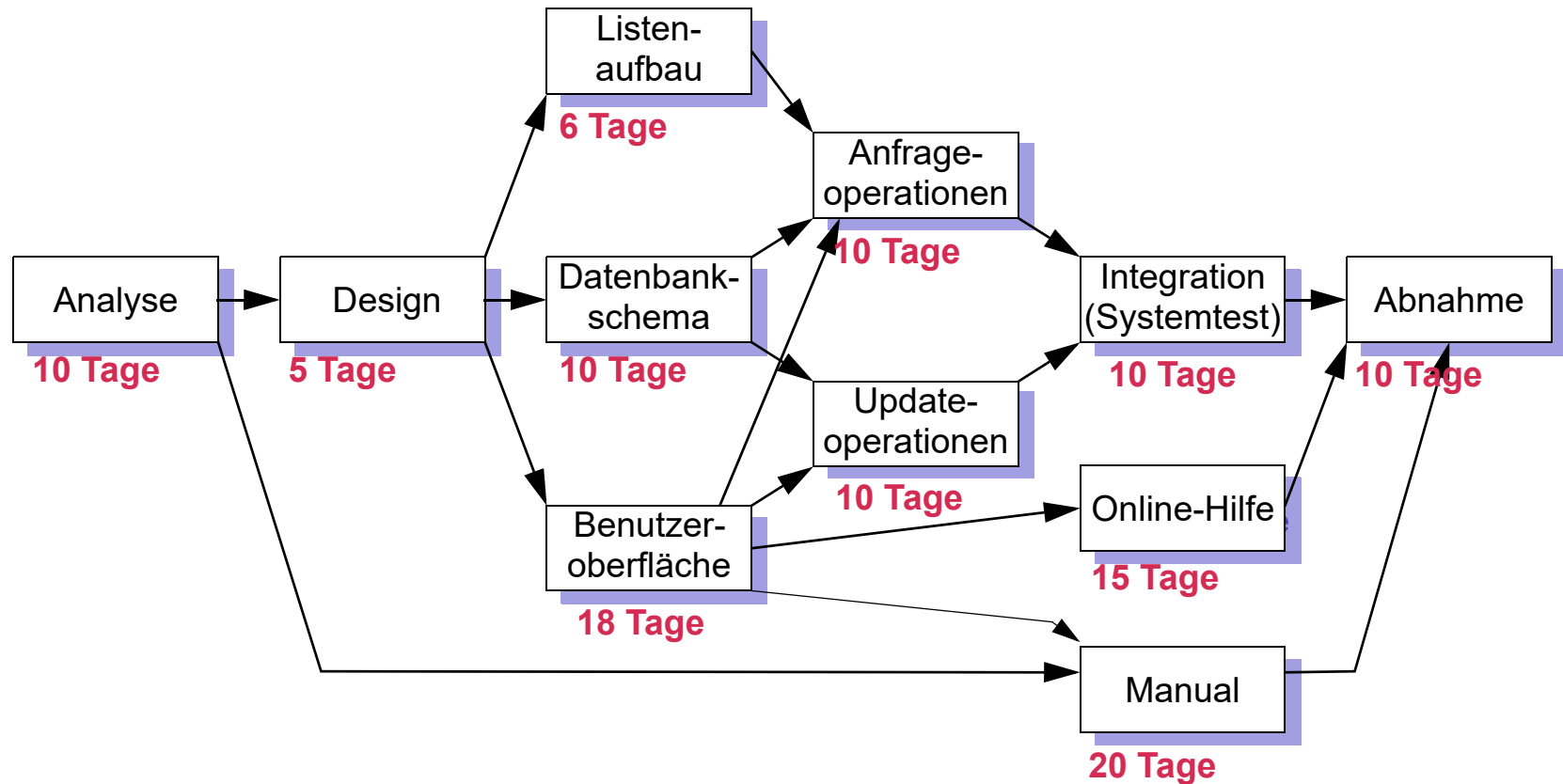
ID	WBS	Task Name	Duration	Start	Finish	Predecessors
1	1	Analyse	10 days	Mon 14.04.03	Fri 25.04.03	
2	2	Design	5 days	Mon 28.04.03	Fri 02.05.03	1
3	3	Codierung	30 days	Mon 05.05.03	Fri 13.06.03	2
4	3.1	Code-Datenbankschema	10 days	Mon 05.05.03	Fri 16.05.03	
5	3.2	Code-Liste wartbar	6 days	Mon 19.05.03	Mon 26.05.03	
6	3.3	Code-Benutzerschnittstelle	18 days	Mon 05.05.03	Wed 28.05.03	
7	3.4	Code-Update-Operationen	10 days	Mon 02.06.03	Fri 13.06.03	6;4
8	3.5	Code-Antragsoperationen	10 days	Mon 02.06.03	Fri 13.06.03	4;5
9	4	Dokumentation	37 days	Thu 01.05.03	Fri 20.06.03	1
10	4.1	Doku-Manual	20 days	Thu 01.05.03	Wed 28.05.03	61
11	4.2	Doku-Online-Hilfe	15 days	Mon 02.06.03	Fri 20.06.03	6
12	5	Integration	10 days	Mon 16.06.03	Fri 27.06.03	8;7;3
13	6	Abnahme	10 days	Mon 30.06.03	Fri 11.07.03	10;11;12

Aufgabenhierarchie Dauer Start/Ende
Datum Abhängigkeiten

Achtung: Start und Ende von Aufgaben werden aus Dauer der Aufgaben und Zeitpunkt für Projektbeginn automatisch errechnet (siehe folgende Folien).



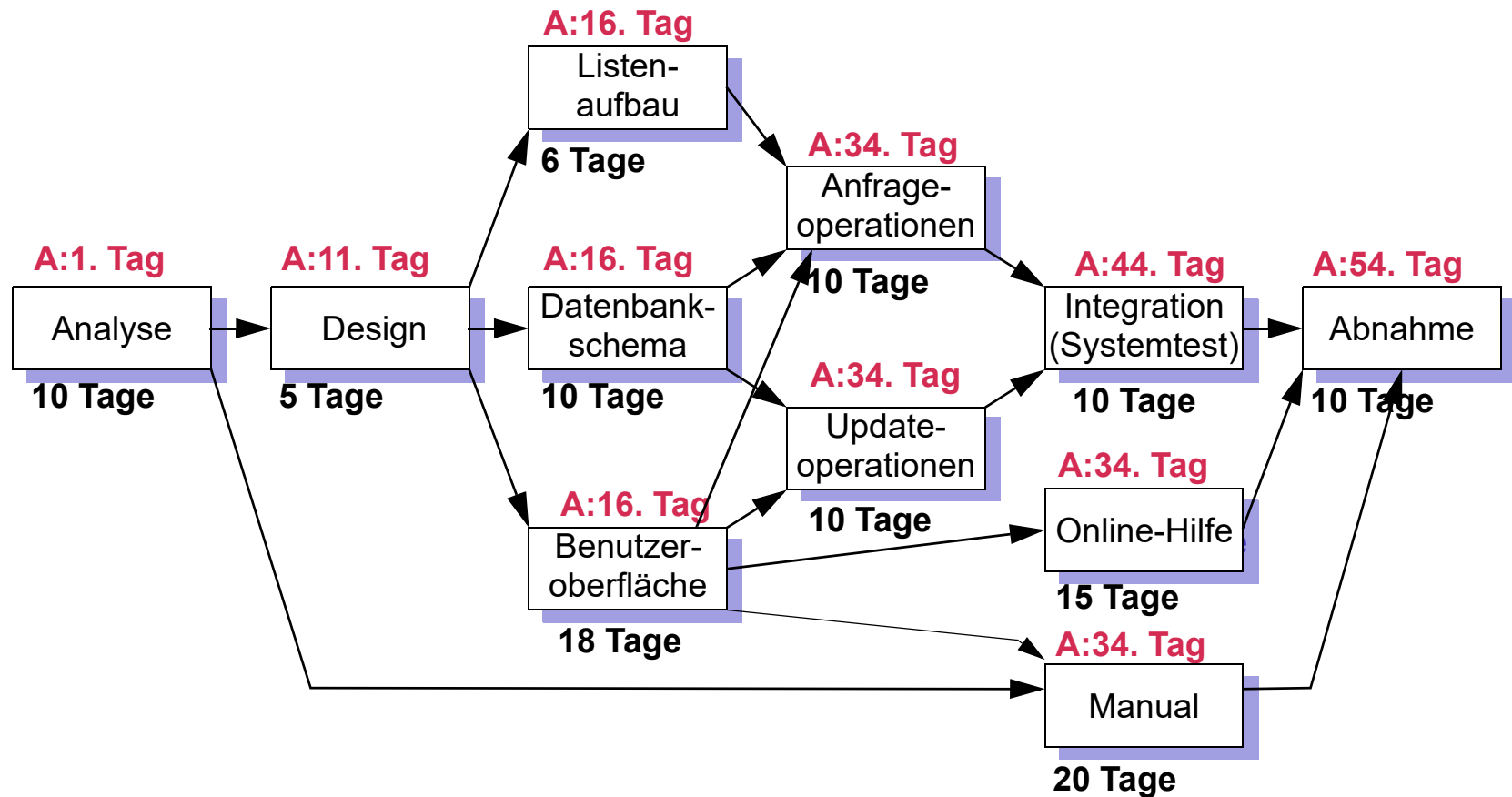
Planung von Arbeitspaketabhängigkeiten (mit PERT-Charts) - Idee:



PERT-Chart = **P**roject **E**valuation and **R**eview **T**echnique



Planung von Arbeitspaketabhängigkeiten - früheste Anfangszeiten:

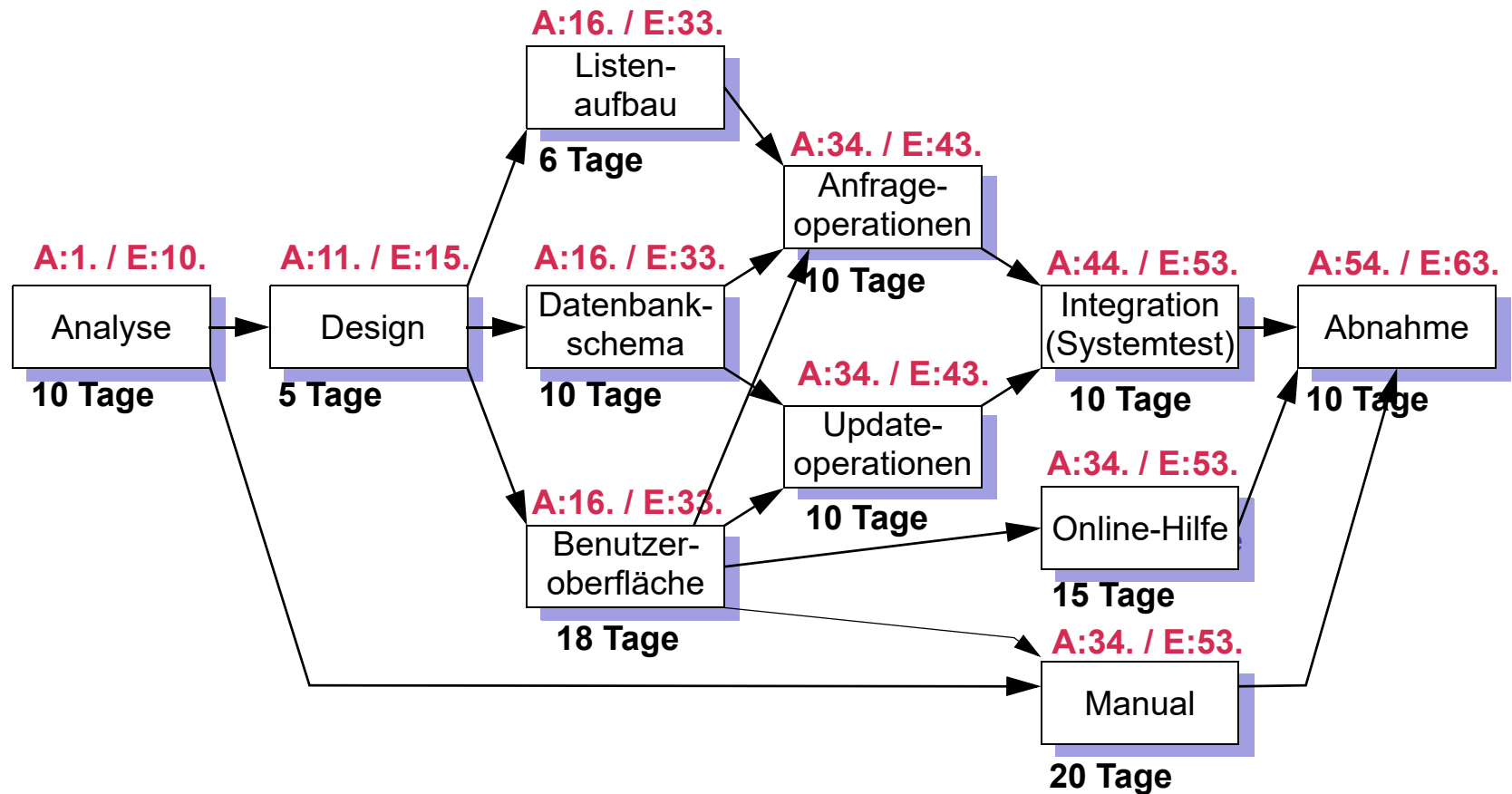


Vorwärtsberechnung von Startdatum für Aufgabe:

früheste Anfangszeit = $\text{Max}(\text{früheste Vorgängeranfangszeit} + \text{Vorgängerdauer})$



Planung von Arbeitspaketabhängigkeiten - späteste Endzeiten:

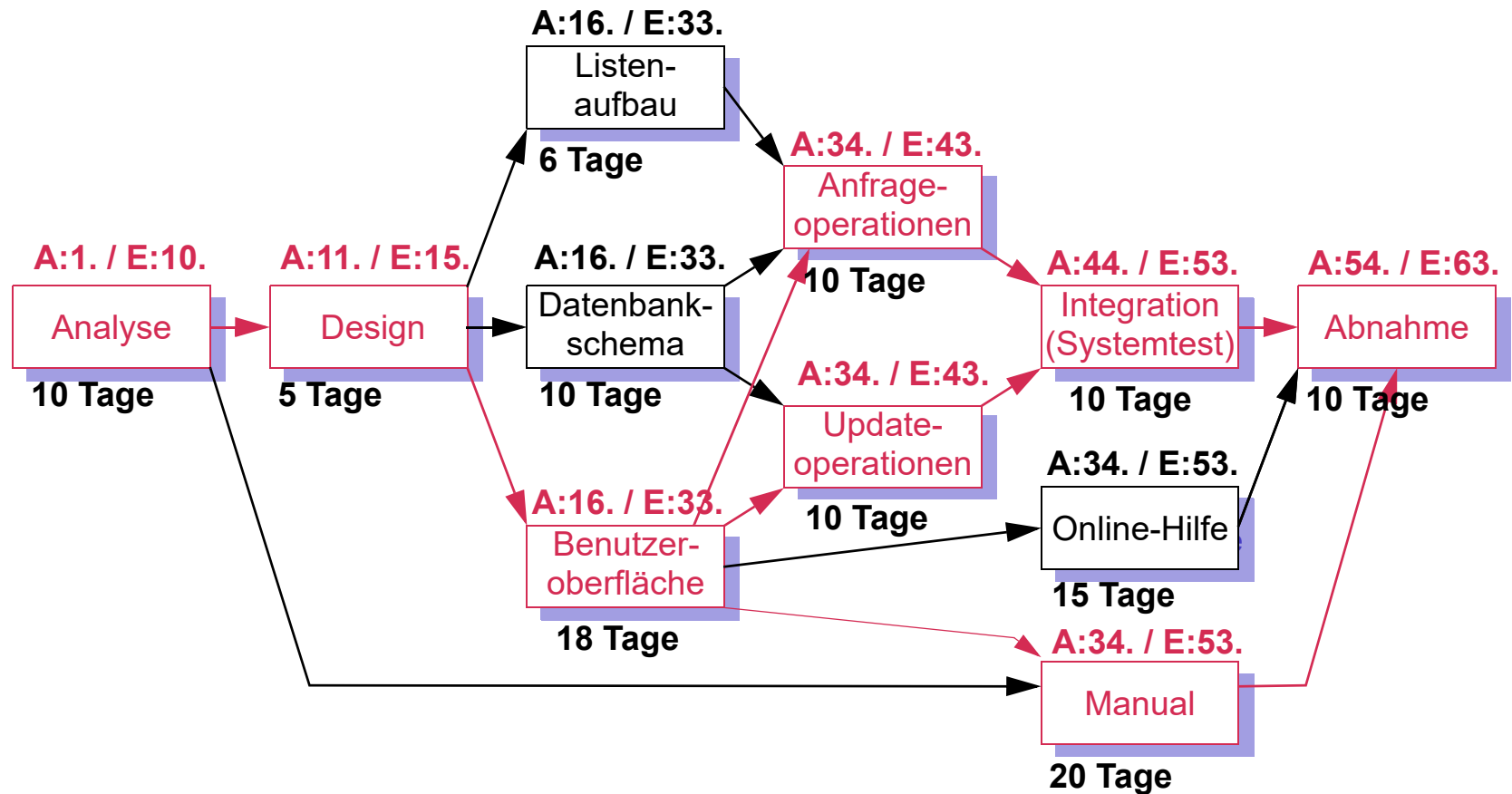


Rückwärtsberechnung von Enddatum für Aufgabe:

späteste Endzeit = $\text{Min}(\text{späteste Nachfolgerendzeit} - \text{Nachfolgerdauer})$



Planung von Arbeitspaketabhängigkeiten - kritische Pfade u. Aufgaben:



Berechnung kritischer Pfade mit kritischen Aufgaben:

für **kritische Aufgabe** gilt: früheste Anfangszeit + Dauer = späteste Endzeit + 1



Zusammenfassung der Berechnung:

- ❑ **Vorwärtsberechnung** frühester Anfangszeiten für Aufgaben:
 - ⇒ früheste **A**nfangszeit erster Aufgabe = 1
 - ⇒ früheste **A**nfangszeit von Folgeaufgabe =
Maximum(früheste Anfangszeit + Dauer einer Vorgängeraufgabe)
- ❑ **Rückwärtsberechnung** spätester Endzeiten für Aufgaben:
 - ⇒ späteste **E**ndzeit letzter Aufgabe = früheste Anfangszeit + Dauer - 1
 - ⇒ späteste **E**ndzeit von Vorgängeraufgabe =
Minimum(späteste Endzeit - Dauer einer Nachfolgeraufgabe)
- ❑ **kritische Aufgabe** auf kritischem Pfad (Anfangs- und Endzeiten liegen fest):
 - ⇒ früheste Anfangszeit = späteste Anfangszeit := späteste Endzeit - Dauer + 1
 - ⇒ späteste Endzeit = früheste Endzeit := früheste Anfangszeit + Dauer - 1
- ❑ **kritische Kante** auf kritischem Pfad (zwischen zwei kritischen Aufgaben):
 - ⇒ früheste Endzeit Kantenquelle + 1 = späteste Anfangszeit Kantensenke
- ❑ **Pufferzeit** nichtkritischer Aufgabe (Zeit um die Beginn verschoben werden kann):
 - ⇒ Pufferzeit = späteste Anfangszeit - früheste Anfangszeit = ...



Probleme mit der Planung des Beispiels:

- ❑ zu viele Aufgaben liegen auf kritischen Pfaden (wenn kritische Aufgabe länger als geschätzt dauert, schlägt das auf Gesamtprojektlaufzeit durch)
 - ⇒ an einigen Stellen zusätzliche Pufferzeiten einplanen (um Krankheiten Fehlschätzungen, ... auffangen zu können)
- ❑ einige eigentlich parallel ausführbare Aufgaben sollen von der selben Person bearbeitet werden
 - ⇒ Ressourcenplanung für Aufgaben durchführen
 - ⇒ Aufgaben serialisieren um „Ressourcenkonflikte“ aufzulösen
 - ⇒ (oder: weitere Personen im Projekt beschäftigen)
- ❑ Umrechnung auf konkrete Datumsangaben fehlt noch
 - ⇒ Berücksichtigung von Wochenenden (5 Arbeitstage pro Woche)
 - ⇒ ggf. auch Berücksichtigung von Urlaubszeiten



Verfeinerung von Abhängigkeiten und Zeitplanung:

Bislang gilt für abhängige Aktivitäten A1 und A2 (mit A2 hängt von A1 ab):

- ⇒ **„Finish-to-Start“-Folge (FS) = Normalfolge:**
Nachfolgeaktivität A2 kann erst bearbeitet werden, wenn Vorgänger A1 vollständig abgeschlossen ist: $A1.Finish + 1 + lag \leq A2.Start$

Manchmal lassen sich aber abhängige Aktivitäten auch parallel bearbeiten:

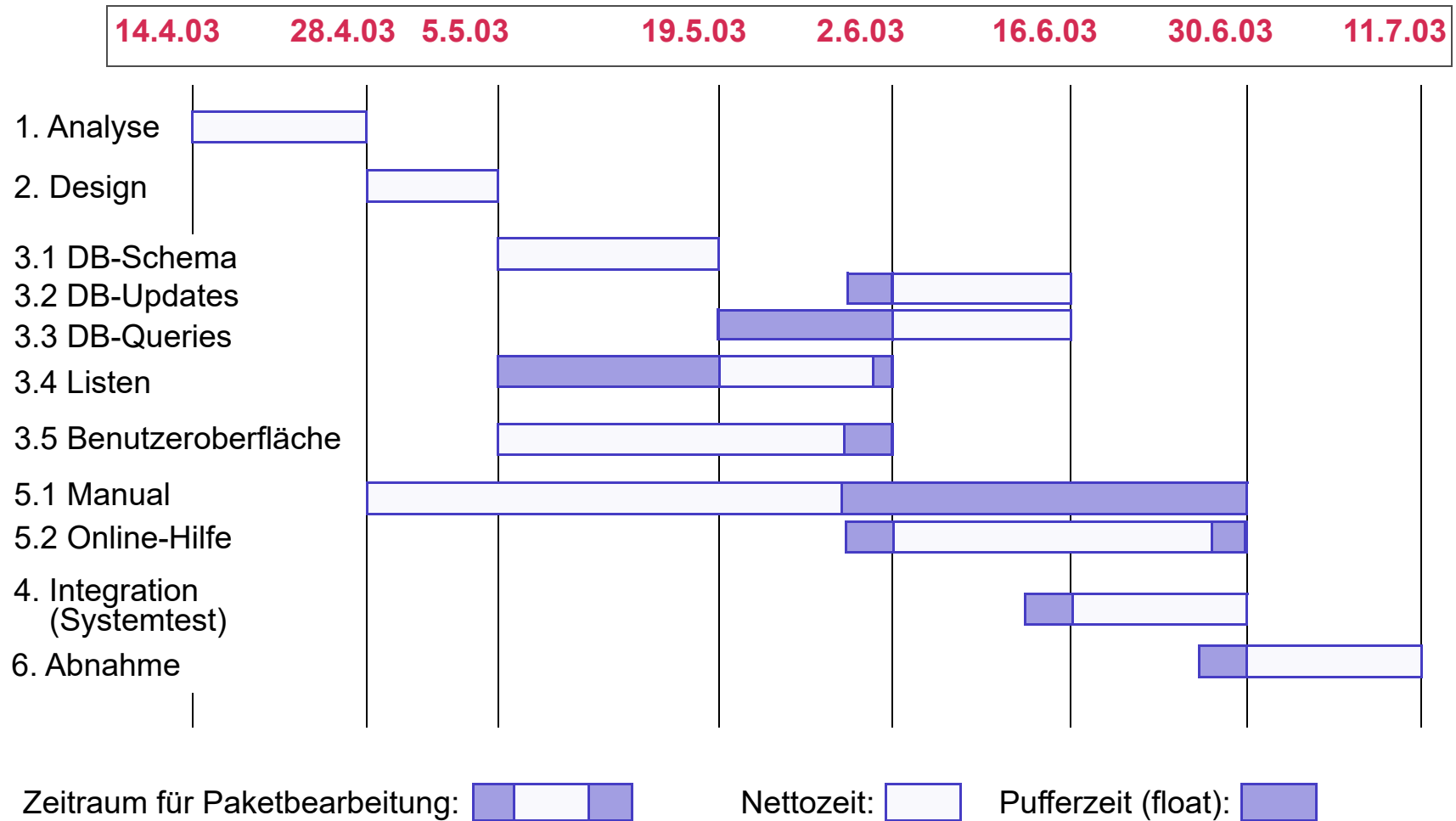
- ⇒ **„Start-to-Start“-Folge (SS) = Anfangsfolge:**
Aktivitäten können gleichzeitig beginnen, also: $A1.Start + lag \leq A2.Start$
- ⇒ **„Finish-to-Finish“-Folge (FF) = Endfolge:**
Aktivitäten können gleichzeitig enden, also: $A1.Finish + lag \leq A2.Finish$
- ⇒ **„Start-to-Finish“-Folge (SF) = Sprungfolge (nicht benutzen):**
 $A1.Start + lag \leq A2.Finish$ (früher für Rückwärtsrechnungen eingesetzt)

Des weiteren sind manchmal zusätzliche Verzögerungszeiten sinnvoll:

- ⇒ frühestmöglicher Beginn einer Aktivität wird um **Verzögerungszeit (lag)** nach vorne oder hinten verschoben (für feste Puffer, Überlappungen)

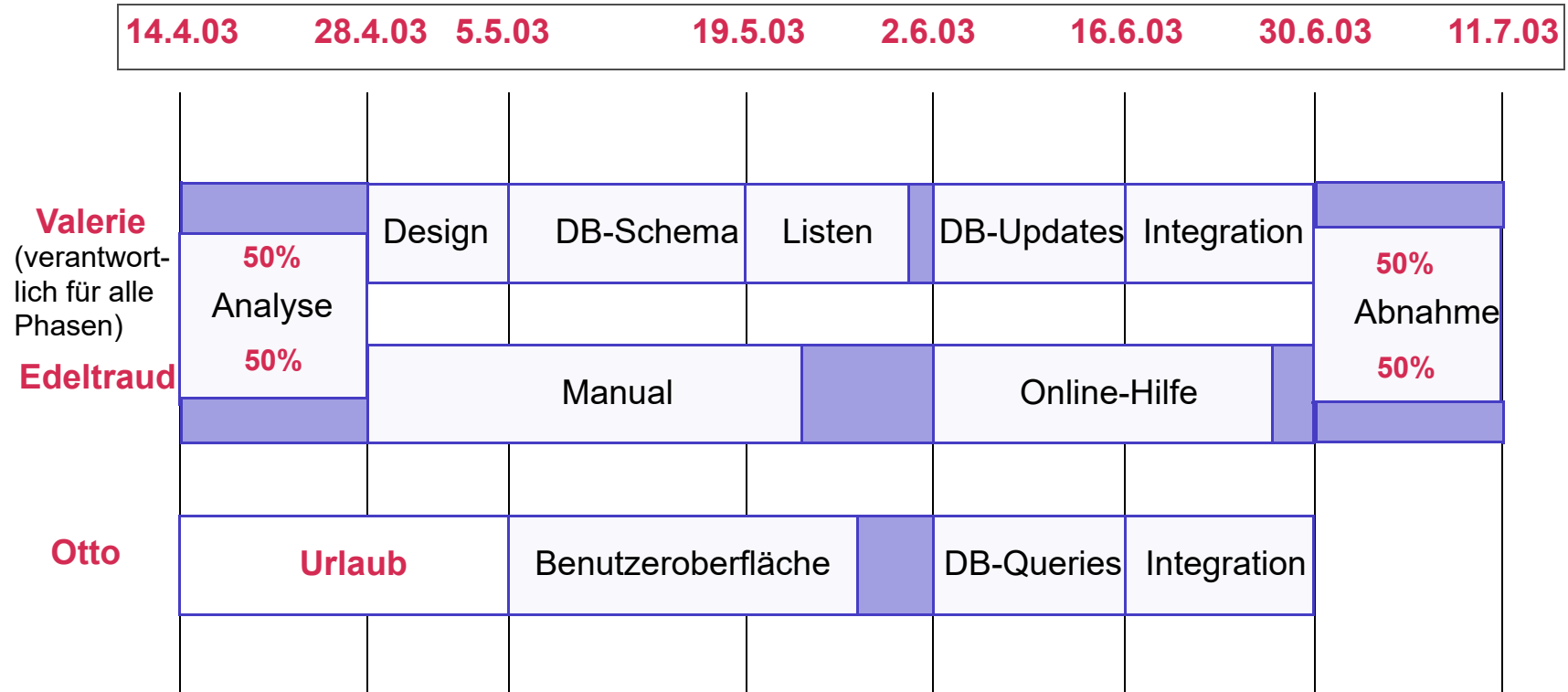


Balkendiagramm (Gantt Chart, 1917 von Henry Gantt erfunden) - Idee:






Balkendiagramm mit Personalplanung:



Nettozeit: 

Pufferzeit: 
(float)

Achtung:

Durch Ressourcenzuteilung entstehen zusätzliche Zeitrestriktionen (z.B. Listen nach DB-Schema)



5.6 Aufwands- und Kostenschätzung

Die **Kosten** eines Softwareproduktes und die **Entwicklungsdauer** werden im wesentlichen durch den personellen Aufwand bestimmt. Bislang haben wir vorausgesetzt, dass der personelle Aufwand bekannt ist, hier werden wir uns mit seiner Berechnung bzw. Schätzung befassen.

Der **personelle Aufwand** für die Erstellung eines Softwareproduktes ergibt sich aus

- ⇒ dem „**Umfang**“ des zu erstellenden Softwareprodukts
- ⇒ der geforderten **Qualität** für das Produkt

Übliches Maß für Personalaufwand:

Mitarbeitermonate (MM) oder Mitarbeiterjahre (MJ):

$1 \text{ MJ} \approx 10 \text{ MM}$ (wegen Urlaub, Krankheit, ...)

Übliches Maß für Produktumfang:

„Lines of Code“ (LOC) = Anzahl Zeilen der Implementierung ohne Kommentare



Schätzverfahren im Überblick:

- ❑ **Analogiemethode:** Experte vergleicht neues Projekt mit bereits abgeschlossenen ähnlichen Projekten und schätzt Kosten „gefühlsmäßig“ ab
 - ⇒ Expertenwissen lässt sich schwer vermitteln und nachvollziehen
- ❑ **Prozentsatzmethode:** aus abgeschlossenen Projekten wird Aufwandsverteilung auf Phasen ermittelt; anhand beendeter Phasen wird Projektrestlaufzeit geschätzt
 - ⇒ funktioniert allenfalls nach Abschluss der Analysephase
- ❑ **Parkinsons Gesetz:** die Arbeit ist beendet, wenn alle Vorräte aufgebraucht sind
 - ⇒ praxisnah, realistisch und wenig hilfreich ...
- ❑ **Price to Win:** die Software-Kosten werden auf das Budget des Kunden geschätzt
 - ⇒ andere Formulierung von „Parkinsons Gesetz“, führt in den Ruin ...
- ❑ **Gewichtungsmethode:** Bestimmung vieler Faktoren (Erfahrung der Mitarbeiter, verwendete Sprachen, ...) und Verknüpfung durch mathematische Formel
 - ⇒ LOC-basierter Vertreter: COConstructive COst MOdel (COCOMO)
 - ⇒ FP-basierte Vertreter: Function-Point-Methoden in vielen Varianten



Softwareumfang = Lines of Code?

Die „**Lines of Code**“ als Ausgangsbasis für die Projektplanung (und damit auch zur Überwachung der Produktivität von Mitarbeitern) zu verwenden ist **fragwürdig**, da:

- ⇒ Codeumfang erst mit Abschluss der Implementierungsphase bekannt ist
- ⇒ selbst Architektur auf Teilsystemebene noch unbekannt ist
- ⇒ Wiederverwendung mit geringeren LOC-Zahlen bestraft wird
- ⇒ gründliche Analyse, Design, Testen, ... zu geringerer Produktivität führt
- ⇒ Anzahl von Codezeilen abhängig vom persönlichen Programmierstil ist
- ⇒ Handbücher schreiben, ... ungenügend berücksichtigt wird

Achtung:

Die starke **Abhängigkeit** der LOC-Zahlen **von** einer **Programmiersprache** ist zulässig, da Programmiersprachenwahl (großen) Einfluss auf Produktivität hat.



Einfluss von Programmiersprache auf Produktivität:

	Analyse	Design	Codierung	Test	Sonstiges
C	3 Wochen	5 Wochen	8 Wochen	10 Wochen	2 Wochen
Smalltalk	3 Wochen	5 Wochen	2 Wochen	6 Wochen	2 Wochen

Konsequenzen für die Gesamtproduktivität:

	Programmgröße	Aufwand	Produktivität
C	2.000 LOC	28 Wochen	70 LOC/Woche
Smalltalk	500 LOC	18 Wochen	27 LOC/Woche

Fazit:

- ☞ Produktivität kann **nicht** in „Lines Of Code pro Zeiteinheit“ sinnvoll gemessen werden (sonst wäre Programmieren in Assembler die beste Lösung)
- ☞ also: Vorsicht mit Einsatz von Maßzahlen (keine sozialistische Planwirtschaft)



Softwareumfang = Function Points!

Bei der **Function-Point-Methode** zur Kostenschätzung wird der Softwareumfang anhand der Produktanforderungen aus dem Lastenheft geschätzt. Es gibt inzwischen einige Spielarten; hier wird (weitgehend) der Ansatz der **International Function Point Users Group (IFPUG)** vorgestellt, siehe auch <http://www.ifpug.org>.

Jede Anforderung wird gemäß IFPUG einer von 5 Kategorien zugeordnet [ACF97]:

1. **Eingabedaten** (über Tastatur, CD, externe Schnittstellen, ...)
2. **Ausgabedaten** (auf Bildschirm, Papier, externe Schnittstelle, ...)
3. **Abfragen** (z.B. SQL-Queries auf einem internen Datenbestand)
4. **Datenbestände** (sich ändernde interne Datenbankinhalte)
5. **Referenzdateien** (im wesentlichen unveränderliche Daten)

Dann werden **Function-Points (FPs)** berechnet, bewertet,



Datenbestände = Internal Logical File (ILF) = Interne Entitäten:

Unter einer **internen (Geschäfts-)Entität** definiert die IFPUG eine aus Anwendersicht logisch zusammengehörige Gruppe vom Softwaresystem verwalteter Daten, also z.B.:

- ⇒ eine Gruppe von Produktdaten des Lastenheftes in der Machbarkeitsstudie
- ⇒ Klassen mit Attributen u. Beziehungen eines Paketes aus Modellen im Pflichtenheft in der Analysephase

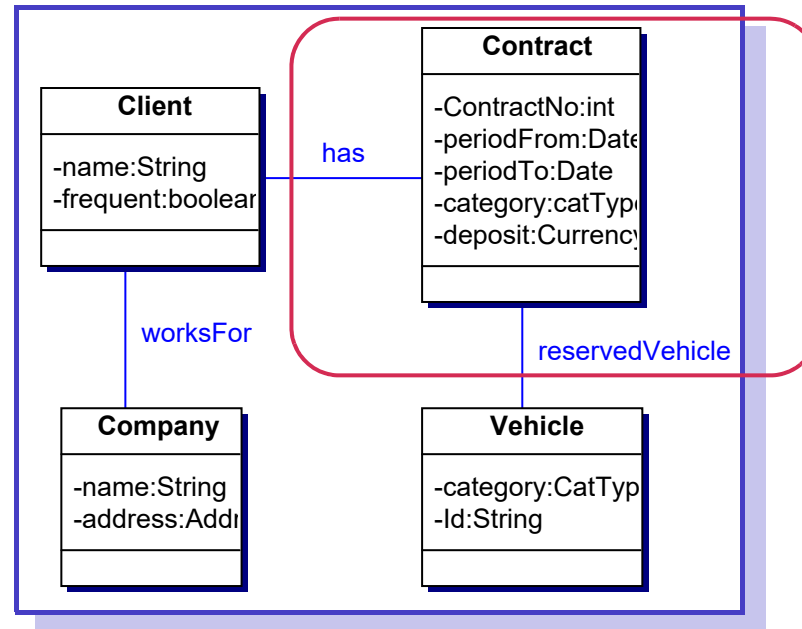
Es werden Datenelementtypen (Attribute) sowie Entitätstypen (Klassen, Sätze) und zusätzlich Beziehungstypen (Assoziationen) gezählt. Anhand dieser Zählung wird Komplexität eines Datenbestandes wie folgt bestimmt:

einfach = 7 FPs, mittel = 10 FPs oder komplex = 15 FPs

Interne Entitäten	Anzahl Attribute ≤ 19	$19 < \text{Anzahl Attribute} \leq 50$	Anzahl Attribute > 50
Klassen+Assoz. ≤ 1	einfache Komplexität	einfache Komplexität	mittlere Komplexität
$2 \leq \text{Klassen+Assoz.} \leq 5$	einfache Komplexität	mittlere Komplexität	hohe Komplexität
Klassen+Assoz. > 5	mittlere Komplexität	hohe Komplexität	hohe Komplexität



Beispiel für Bewertung eines internen Datenbestandes:



**Datenbasis mit
Verträgen**

Die Datenbasis besteht aus



Referenzdateien = External Interface File = (EIF) = Externe Entitäten:

Unter einer **externen (Geschäfts-)Entität** definiert die IFPUG eine aus Anwendersicht logisch zusammengehörige Gruppe vom System benutzter aber nicht selbst verwalteter Daten.

Wieder werden Datenelementtypen (Attribute) sowie Entitätstypen (Klassen, Sätze) und zusätzlich Beziehungstypen (Assoziationen) gezählt. Anhand dieser Zählung wird Komplexität eines Datenbestandes wie folgt bestimmt:

einfach = 5 FPs, mittel = 7 FPs oder komplex = 10 FPs

Externe Entitäten	Anzahl Attribute ≤ 19	$19 < \text{Anzahl Attribute} \leq 50$	Anzahl Attribute > 50
Klassen+Assoz. ≤ 1	einfache Komplexität	einfache Komplexität	mittlere Komplexität
$2 \leq \text{Klassen+Assoz.} \leq 5$	einfache Komplexität	mittlere Komplexität	hohe Komplexität
Klassen+Assoz. > 5	mittlere Komplexität	hohe Komplexität	hohe Komplexität

Es werden weniger FPs als bei internen Entitäten vergeben, da die betrachteten Datenbestände nur eingelesen aber nicht verwaltet werden müssen.



(Externe) Eingabedaten = External Input (EI):

Eingabedaten für **Elementarprozess**, der Daten oder Steuerinformationen des Anwenders verarbeitet, aber keine Ausgabedaten liefert. Es handelt sich dabei um den kleinsten selbständigen Arbeitsschritt in der Arbeitsfolge eines Anwenders, als etwa:

- ⇒ Produktfunktionen des Lastenheftes in der Machbarkeitsstudie
- ⇒ „Use Cases“ aus Pflichtenheft in der Analysephase

Gezählt werden für jeden Elementarprozess die Anzahl seiner als Eingabe verwendeten Entitätstypen (Klassen, Sätze) und deren Datenelementtypen (Attribute, Felder). Anhand dieser Zählung wird Komplexität des Elementarprozesses wie folgt bestimmt:

einfach = 3 FPs, mittel = 4 FPs oder komplex = 6 FPs

Externe Eingabe	Anzahl Attribute ≤ 4	$4 < \text{Anzahl Attribute} \leq 15$	Anzahl Attribute > 15
Anzahl Klassen ≤ 1	einfache Komplexität	einfache Komplexität	mittlere Komplexität
Anzahl Klassen = 2	einfache Komplexität	mittlere Komplexität	hohe Komplexität
Anzahl Klassen > 2	mittlere Komplexität	hohe Komplexität	hohe Komplexität



Beispiel für die Bewertung externer Eingabedaten:

- ☐ Reservierungsvertrag **neu erstellen** mit Beginn- und Endedatum, gewünschter Fahrzeugkategorie, Kautions, Kunde und Fahrzeug (eine eindeutige Vertragsnummer wird automatisch angelegt)
- ☐ Reservierungsvertrag mit Vertragsnummer **verändern** (alle Werte ausser Kunde)
- ☐ Reservierungsvertrag mit Vertragsnummer **löschen**
- ☐ Daten zu einem Reservierungsvertrag mit Vertragsnummer **anzeigen**

Es wird wie folgt gezählt:



Externe Ausgaben = External Output (EO):

Ausgabedaten eines **Elementarprozesses** (Produktfunktion, Use Case), der Anwender Daten oder Steuerinformationen liefert. Achtung: der Elementarprozess darf keine Eingabedaten benötigen; ansonsten handelt es sich um eine „Externe Abfrage“ oder

Gezählt werden für jeden Elementarprozess die Anzahl seiner als Ausgabe verwendeten Entitätstypen (Klassen, Sätze) und deren Datenelementtypen (Attribute, Felder). Anhand dieser Zählung wird Komplexität des Elementarprozesses wie folgt bestimmt:

einfach = 4 FPs, mittel = 5 FPs oder komplex = 7 FPs

Externe Ausgaben	Anzahl Attribute ≤ 5	$5 < \text{Anzahl Attribute} \leq 19$	Anzahl Attribute > 19
Anzahl Klassen ≤ 1	einfache Komplexität	einfache Komplexität	mittlere Komplexität
$2 \leq \text{Anzahl Klassen} \leq 3$	einfache Komplexität	mittlere Komplexität	hohe Komplexität
Anzahl Klassen > 3	mittlere Komplexität	hohe Komplexität	hohe Komplexität



Beispiel für die Bewertung externer Ausgabedaten:

- ☐ Daten zu einem Reservierungsvertrag mit Vertragsnummer **anzeigen** (mit Name des Kunden und tatsächlicher Fahrzeugkategorie zusätzlich zu Kundennummer und Fahrzeugnummer)
- ☐ **alle Reservierungsverträge** zu einem Kunden mit Kundennummer anzeigen
- ☐ die **Kosten** für alle Reservierungsverträge eines Kunden anzeigen

Es wird wie folgt gezählt:



Externe Abfragen = External Inquiry (EQ):

Betrachtet werden **Elementarprozesse** (Produktfunktion, Use Case), die anhand von Eingaben Daten des internen Datenbestandes ausgeben (ohne auf diesen Daten komplexe Berechnungen durchzuführen).

Nach den Regeln für „Externe Eingaben“ werden die Eingabedaten bewertet, nach den Regeln für „Externe Ausgaben“ die Ausgabedaten; anschließend wird die höhere Komplexität übernommen und wie folgt umgerechnet:

einfach = 3 FPs, mittel = 4 FPs oder komplex = 6 FPs

Achtung: ein Elementarprozess, der Eingabedaten zur Suche nach intern gespeicherten Daten benötigt und vor der Ausgabe **komplexe Berechnungen** durchführt, wird anders behandelt. In diesem Fall wird nicht das Maximum gebildet, sondern die **Summe** der FPs von „Externe Eingabe“ und „Externe Ausgabe“.



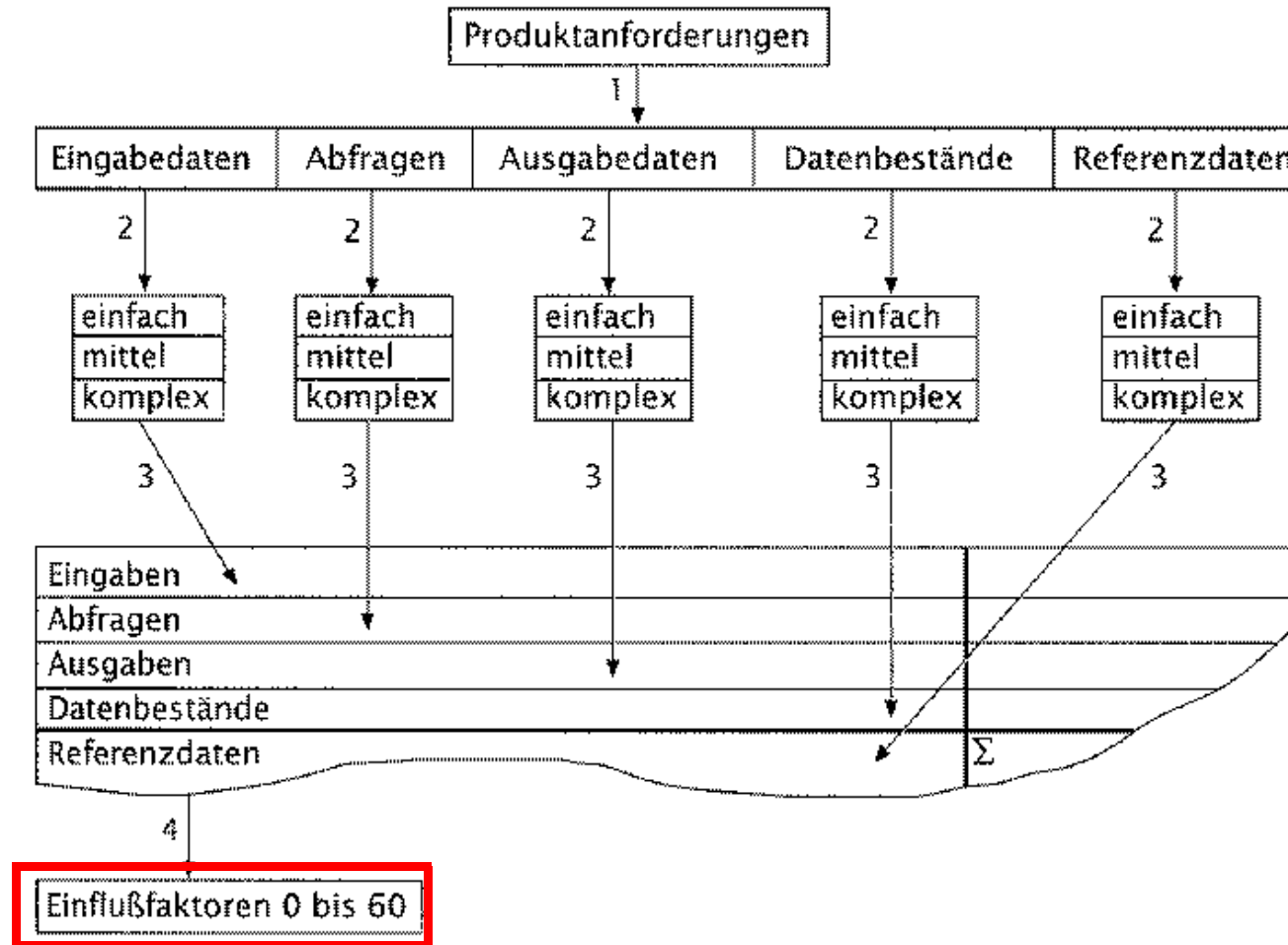
Beispiel für die Bewertung einer externen Abfrage:

- ❑ **alle Reservierungsverträge** zu einem Kunden mit Kundennummer anzeigen (mit Kundenname und -nummer sowie Fahrzeugkategorie und -nummer)
- ❑ die **Kosten** für alle Reservierungsverträge eines Kunden anzeigen, der über seine Kundennummer festgelegt wird.

Es wird wie folgt gezählt:



Überblick über die FP-Methode - 1 [Ba98]:



1. Schritt:
Kategorisierung für
jede Anforderung

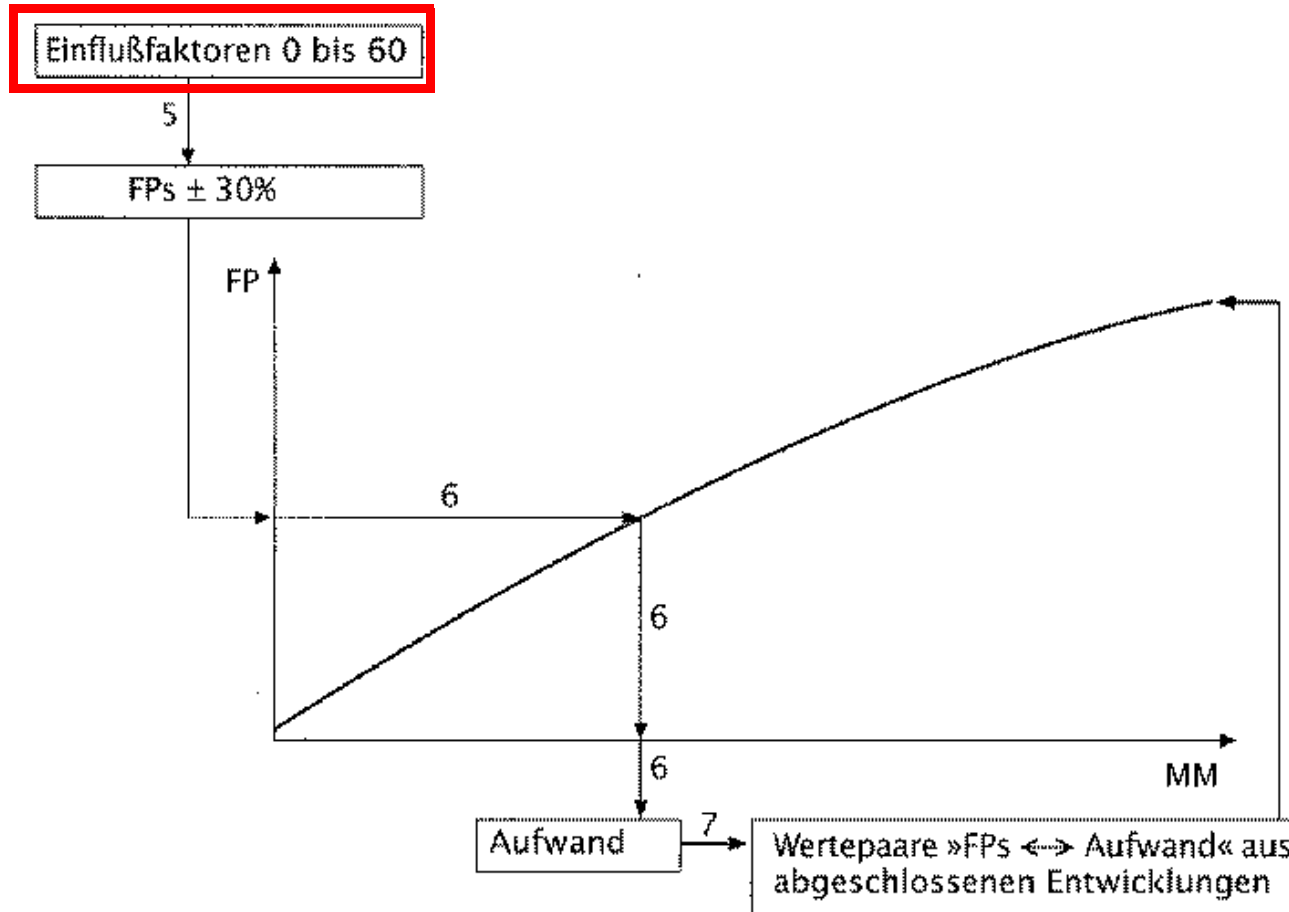
2. Schritt:
Klassifizierung jeder
Anforderung

3. Schritt: Eintrag
der jeweiligen
Anzahl in das
Berechnungs-
formular und
Ermittlung der
unbewerteten FPs

4. Schritt:
Bewertung der
Einflußfaktoren



Überblick über die FP-Methode - 2:



5. Schritt:
Berechnung der
bewerteten FPs

6. Schritt: Ablesen
des Aufwandes

7. Schritt:
Aktualisierung der
Wertepaare, Neu-
berechnung der
Aufwandskurve



Berechnungsformel für die FP-Methode:

Berechnungsformel für die FP-Methode:

Kategorie	Anzahl	Klassifizierung	Gewichtung	Zellensumme
Eingabedaten		einfach	x 3	=
		mittel	x 4	=
		komplex	x 6	=
Abfragen		einfach	x 3	=
		mittel	x 4	=
		komplex	x 6	=
Ausgaben		einfach	x 4	=
		mittel	x 5	=
		komplex	x 7	=
Datenbestände		einfach	x 7	=
		mittel	x 10	=
		komplex	x 15	=
Referenzdaten		einfach	x 5	=
		mittel	x 7	=
		komplex	x 10	=
Summe			E1	=
Einflußfaktoren (ändern den <i>Function</i> <i>Point</i> -Wert um $\pm 30\%$)	1 Verflechtung mit anderen Anwendungssystemen (0-5)			=
	2 Dezentrale Daten, dezentrale Verarbeitung (0-5)			=
	3 Transaktionsrate (0-5)			=
	4 Verarbeitungslogik			=
	a Rechenoperationen (0-10)			=
	b Kontrollverfahren (0-5)			=
	c Ausnahmeregelungen (0-10)			=
	d Logik (0-5)			=
	5 Wiederverwendbarkeit (0-5)			=
	6 Datenbestands-Konvertierungen (0-5)			=
	7 Anpaßbarkeit (0-5)			=
Summe der 7 Einflüsse	E2			=
Faktor Einflußbewertung = $E2 / 100 + 0,7$	E3			=
Bewertete <i>Function</i> <i>Points</i> : $E1 * E3$				=



Zusätzliche Einflussfaktoren:

Die vorige Tabelle unterscheidet sieben Einflussfaktoren; andere Quellen nennen 14 bzw. **19 verschiedene Faktoren**, die Werte von 0 bis 5 erhalten (siehe [Hu99]):

1. Komplexität der Datenkommunikation
2. Grad der verteilten Datenverarbeitung
3. geforderte Leistungsfähigkeit
4. Komplexität der Konfiguration (Zielplattform)
5. Anforderung an Transaktionsrate
6. Prozentsatz interaktiver Dateneingaben
7. geforderte Benutzerfreundlichkeit
8. interaktive bzw. Online-Pflege des internen Datenbestandes
9. Komplexität der Verarbeitungslogik



Zusätzliche Einflussfaktoren - Fortsetzung:

10. geforderter Grad der Wiederverwendbarkeit
11. benötigte Installationshilfen
12. leichte Bedienbarkeit (Grad der Automatisierung der Bedienung)
13. Mehrfachinstallationen (auf verschiedenen Zielplattformen)
14. Grad der gefoderten Änderungsfreundlichkeit
15. Randbedingungen anderer Anwendungen
16. Sicherheit, Datenschutz, Prüfbarkeit
17. Anwenderschulung
18. Datenaustausch mit anderen Anwendungen
19. Dokumentation



Ermittlung der FP-Aufwandskurve:

- ❑ beim ersten Projekt muss man auf **bekannte Kurven** (für ähnliche Projekte) zurückgreifen (IBM-Kurve mit $FP = 26 * MM^{0,8}$, VW AG Kurve, ...)
- ❑ alternativ kann man eigene abgeschlossene Projekte **nachkalkulieren**, allerdings:
 - ⇒ Nachkalkulationen sind aufwändig
 - ⇒ Dokumentation von Altprojekten oft unvollständig
 - ⇒ oft gibt es nur noch den Quellcode (keine Lasten- oder Pflichtenhefte)
 - ⇒ Kosten (Personenmonate) alter Projekte oft unklar (wurden Überstunden berücksichtigt, welche Aktivitäten wurden mitgezählt, ...)
- ❑ das Verhältnis von MM zu FP bei abgeschlossenen eigenen Projekten wird zur nachträglichen „**Kalibrierung**“ der Kurve benutzt:
 - ⇒ neues Wertepaar wird hinzugefügt oder neues Wertepaar ersetzt ältestes Wertepaar
 - ⇒ Frage: was für eine Funktion benutzt man für Interpolation von Zwischenwerten (meist nicht linear, sondern eher quadratisch oder gar exponentiell)



Nachkalkulation von Projekten mit „Backfiring“-Methode:

Bei alten Projekten gibt es oft nur noch den Quellcode und keine Lasten- oder Pflichtenhefte, aus denen FPs errechnet werden können. In solchen Fällen versucht man FPs aus Quellcode wie folgt nach [Jo92] rückzurechnen:

Sprache	Lines of Code / Function Points		
Komplexität	niedrig	mittel	hoch
Assembler	200	320	450
C	60	128	170
Fortran	75	107	160
COBOL	65	107	150
C++	30	53	125
Smalltalk	15	21	40

Achtung:

LOC in Programmiersprache X pro MM Codierung angeblich nahezu konstant

⇒ damit ist z.B. Produktivität beim Codieren in Smalltalk 4 mal höher als in C



Vorgehensweise bei Kostenschätzung (mit FP-Methode):

1. Festlegung des verwendeten **Ansatzes**, Bereitstellung von Unterlagen
2. **Systemgrenzen** festlegen (was gehört zum Softwaresystem dazu)
3. **Umfang** des zu erstellenden Softwaresystems (in FPs) **messen**
4. Umrechnung von Umfang (FPs) in **Aufwand** (MM) mit Aufwandskurve
5. **Zuschläge** einplanen (für unvorhergesehene Dinge, Schätzungenauigkeit)
6. Aufwand auf Phasen bzw. Iterationen **verteilen**
7. Umsetzung in **Projektplan** mit Festlegung von **Teamgröße**
8. Aufwandschätzung **prüfen** und dokumentieren
9. Aufwandschätzung für Projekt während Laufzeit regelmäßig **aktualisieren**
10. Datenbasis für eingesetztes Schätzverfahren aktualisieren, Verfahren **verbessern**



Einplanung von Zuschlägen (Faustformel nach Augustin):

$$\text{Korrekturfaktor } K = 1,8 / (1 + 0,8 F^3)$$

für Zuschläge mit F als geschätzter Fertigstellungsgrad der Software.

Problem mit der Berechnung des Fertigstellungsgrades der Software:

Der Wert F ist vor Projektende unbekannt, muss also selbst geschätzt werden als

$$F = \text{bisheriger Aufwand} / (\text{bisheriger Aufwand} + \text{geschätzter Restaufwand})$$

Modifizierte Formel für korrigierte Aufwandsschätzung:

$$MM_g = MM_e + MM_k = MM_e + MM_r * 1,8 / (1 + 0,8 (MM_e / (MM_e + MM_r))^3)$$

MM_g = korrigierter geschätzter Gesamtaufwand in Mitarbeitermonaten

MM_e = bisher erbrachter Aufwand in Mitarbeitermonaten

MM_k = korrigierter geschätzter Restaufwand in Mitarbeitermonaten

MM_r = geschätzter Restaufwand in Mitarbeitermonaten



Erläuterungen zu Korrekturfaktor für Kostenschätzung:

- ❑ zu **Projektbeginn** ist $F = 0$, da noch kein Aufwand erbracht wurde; damit wird der geschätzte Aufwand um 80% nach oben korrigiert
- ❑ am **Projektende** ist $F = 1$, da spätestens dann die aktuelle Schätzung mit tatsächlichem Wert übereinstimmen sollte, es gibt also keinen Aufschlag mehr
- ❑ Unsicherheiten in der Schätzung nehmen nicht **nicht linear** ab, da Wissenszuwachs über zu realisierende Softwarefunktionalität und technische Schwierigkeiten im Projektverlauf keinesfalls linear ist
- ❑ im Laufe des Projektes wird Fertigstellungsgrad F nicht immer zunehmen, sondern ggf. auch abnehmen, wenn Schätzungen sich als **zu optimistisch** erwiesen haben
- ❑ Auftraggeber wird mit Zuschlag von 80% auf geschätzte Kosten nicht zufrieden sein, deshalb werden inzwischen manchmal Verträge geschlossen, bei denen nur **Preis je realisiertem FP** vereinbart wird:
 - ⇒ Risiko für zu niedrige Schätzung von FPs liegt bei Auftraggeber
 - ⇒ Risiko für zu niedrige Umrechnung v. FPs in MM liegt bei Auftragnehmer



Aufwandsverteilung auf Phasen bzw. Entwicklungsaktivitäten:

Hat man den Gesamtaufwand für ein Softwareentwicklungsprojekt geschätzt, muss man selbst bei einer ersten Grobplanung schon die ungefähre **Länge einzelner Phasen** oder Iterationen festlegen:

- ❑ für die Aufteilung des Aufwandes auf Phasen bzw. Aktivitätsbereiche gibt es die **Prozentsatzmethode**, hier in der Hewlett-Packard-Variante aus [Ba98]:
 - ⇒ Analyseaktivitäten: 18% des Gesamtaufwandes
 - ⇒ Entwurfsaktivitäten: 19% des Gesamtaufwandes
 - ⇒ Codierungsaktivitäten: 34% des Gesamtaufwandes
 - ⇒ Testaktivitäten: 29% des Gesamtaufwandes
- ❑ für die Aufwandsberechnung **einzelner Iterationen** einer Phase wird die Zuordnung von FPs zu diesen Iterationen herangezogen oder es wird bei festgelegter Projektlänge und fester Länge von Iterationen (z.B. 4 Wochen) die Anzahl der FPs, die in einer Iteration zu behandeln sind, festgelegt



Bestimmung optimaler Entwicklungsdauer (Faustformel nach Jones):

für geringen Kommunikationsoverhead und hohen Parallelisierungsgrad:

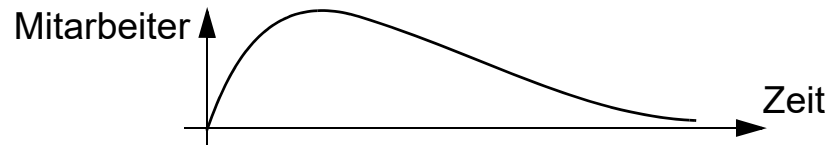
$$\text{Dauer} = 2,5 * (\text{Aufwand in MM})^s$$

$s = 0,38$ für Stapel-Systeme
 $s = 0,35$ für Online-System
 $s = 0,32$ für Echtzeit-Systeme

durchschnittliche **Teamgröße** = **Aufwand / Dauer**

Überlegungen zu obiger Formel:

- ⇒ Anzahl der maximal sinnvoll parallel arbeitenden Mitarbeiter hängt ab von Projektart
- ⇒ große Projekte dürfen nicht endlos lange laufen (also mehr Mitarbeiter)
- ⇒ mit der Anzahl der Mitarbeiter wächst aber der Kommunikations- und der Verwaltungsaufwand überproportional (also weniger Mitarbeiter)
- ⇒ Anzahl sinnvoll parallel zu beschäftigender Mitarbeiter während Projektlaufzeit (Putnam-Kurve):





Rechenbeispiele für Faustformel:

Aufwand in MM	Projektart	Projektdauer	Mitarbeiterzahl
20 MM	Stapel (Batch)	7,8 Monate	2,6 Mitarbeiter
	Echtzeit (Realtime)	6,5 Monate	3,1 Mitarbeiter
200 MM	Stapel	18,7 Monate	10,7 Mitarbeiter
	Echtzeit	13,6 Monate	14,7 Mitarbeiter
2000 MM	Stapel	45,0 Monate	44,4 Mitarbeiter
	Echtzeit	28,5 Monate	70,2 Mitarbeiter

Achtung:

- ☞ geschätzter Aufwand in Mitarbeitermonaten enthält bereits organisatorischen **Overhead** für Koordination von immer mehr Mitarbeitern
- ☞ in 45,0 Monaten mit 44,4 Mitarbeitern = 2.000 MM werden also nicht 100 mal mehr FPs oder LOCs als in 7,8 Monaten mit 2,6 Mitarbeitern = 20 MM erledigt (eher nur 30 bis 40 mal mehr FPs)



Bewertung der FP-Methode:

- ☐ lange Zeit wurde LOC-basierte Vorgehensweise propagiert
- ☐ inzwischen: FP-Methode ist wohl einziges halbwegs funktionierendes Schätzverfahren
- ☐ Abweichungen trotzdem groß (insbesondere bei Einsatz „fremder“ Kurven)
- ☐ Anpassung an OO-Vorgehensmodelle, moderne Benutzeroberflächen notwendig
- ☐ moderne Varianten in Form von „**Object-Point-Methode**“, ... sind noch nicht standardisiert und haben sich wohl noch nicht durchgesetzt
- ☐ Schätzungsfehler in der Machbarkeitsstudie sind nicht immer auf fehlerhafte Schätzmethode zurückzuführen, sondern ggf. auch auf **nicht** im Lastenheft **vereinbarte** aber **realisierte Funktionen** oder zusätzliche Umbaumaßnahmen
- ☐ bisher geschilderte Vorgehensweise **nur für Neuentwicklungen** geeignet (ohne umfangreiche Umbaumaßnahmen im Zuge iterativer Vorgehensweise)



Problematik der FP-Berechnung bei iterativer Vorgehensweise:

Bei Projekten zur **Sanierung oder Erweiterung** von Softwaresystemen bzw. bei einer stark iterativ geprägten Vorgehensweise (mit Umbaumaßnahmen) werden einem System nicht nur Funktionen hinzugefügt, sondern auch Funktionen verändert bzw. entfernt. Damit ergibt sich der Aufwand für Projektdurchführung aus:

$$\text{Aufwand in MM} = \text{Aufwand für hinzugefügte Funktionen} + \text{Aufwand für gelöschte Funktionen} + \text{Aufwand für geänderte Funktionen}$$

Vorgehensweise:

- ❑ man benötigt modifizierte Regeln für die Berechnung von FPs für **gelöschte** Funktionen (Löschen etwas einfacher als Hinzufügen, deshalb weniger FPs?)

man benötigt modifizierte Regeln für die Berechnung von FPs für **geänderte** Funktionen (Ändern = Löschen + Hinzufügen?)



5.7 Weitere Literatur

- [ACF97] V. Ambriola, R. Conradi, A. Fugetta: *Assessing Process-Centered Software Engineering Environments*, ACM TOSEM, Vol. 6, No. 3, ACM Press (1997), S. 283-328

Nicht zu alter Aufsatz mit Überblickscharakter zum Thema dieses Kapitels. Beschränkt sich allerdings im wesentlichen darauf, die drei Systeme OIKOS (Ambriola), EPOS (Conradi) und SPADE (Fugetta) der drei Autoren miteinander zu vergleichen. Es handelt sich dabei um Systeme der zweiten Generation, die in diesem Kapitel nicht vorgestellt wurden.

- [Ba98] H. Balzert: *Lehrbuch der Softwaretechnik (Band 2): Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*, Spektrum Akademischer Verlag (1998)

Hier findet man (fast) alles Wissenswerte zum Thema Management der Software-Entwicklung.

- [BP84] V.R. Basili, B.T. Perricone: *Software Errors and Complexity: An Empirical Investigation*, Communications of the ACM, Vol. 27, No. 1, 42-52, ACM Press (1984)

Eine der ersten Publikationen zu empirischen Untersuchungen über den Zusammenhang von Softwarekomplexität und Fehlerhäufigkeit

- [Be99] K. Beck: *Extreme Programming Explained - Embrace the Change*, Addison Wesley (1999)

Eins der Standardwerke zum Thema XP, geschrieben vom Erfinder. Mehr Bettlektüre mit Hintergrundinformationen und Motivation für XP-Techniken, als konkrete Handlungsanweisung.

- [Dy02] K.M. Dymond: *CMM Handbuch*, Springer Verlag (2002)

Einfach zu verstehende schrittweise Einführung in CMM (Levels und Upgrades von einem Level zum nächsten) in deutscher Sprache.



- [Hu99] R. Hürten: Function-Point-Analysis - Theorie und Praxis (Die Grundlage für ein modernes Software-Management), expert-verlag (1999), 177 Seiten
Kompaktes Buch zur Kostenschätzung mit Function-Point-Methode, das Wert auf nachvollziehbare Beispiele legt. Zusätzlich gibt es eine Floppy-Disc mit entsprechenden Excel-Sheets.
- [Hu96] W.S. Humphrey: *Introduction to the Personal Software Process*, SEI Series in Software Engineering, Addison Wesley (1996)
Das CMM-Modell für den „kleinen Mann“. Das Buch beschreibt wie man als einzelner Softwareentwickler von bescheidenen Anfängen ausgehend die Prinzipien der Prozessüberwachung und -verbesserung einsetzen kann. Zielsetzungen sind zuverlässigere Zeit- und Kostenschätzungen, Fehlerreduktion,
- [JBR99] I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*, Addison Wesley (1999)
Präsentation des auf die UML zugeschnittenen Vorgehensmodells der Softwareentwicklung, eine Variante des hier vorgestellten Rational Unified (Objectory) Prozesses.
- [Jo92] C. Jones: *CASE's Missing Elements*, IEEE Spektrum, Juni 1992, S. 38-41, IEEE Computer Society Press (1992)
Enthält u.a. ein vielzitiertes Diagramm über Produktivitäts- und Qualitätsverlauf bei Einsatz von CASE.
- [OHJ99] B. Oestereich (Hrsg.), P. Hruschka, N. Josuttis, H. Kocher, H. Krasemann, M. Reinhold: *Erfolgreich mit Objektorientierung: Vorgehensmodelle und Managementpraktiken für die objektorientierte Softwareentwicklung*, Oldenbourg Verlag (1999)
Ein von Praktikern geschriebenes Buch mit einer Fülle von Tipps und Tricks. Es enthält eine kurze Einführung in den “Unified Software Development Process”, sowie in das V-Modell.