1. Software-Entwicklung, -Wartung und (Re-)Engineering

„Software-Systeme zu erstellen, die nicht geändert werden müssen, ist unmöglich. Wurde Software erst einmal in Betrieb genommen, entstehen neue Anforderungen und vorhandene Anforderungen ändern sich ... "

[So07]

Lernziele:

- **Wissen** über Entwicklungsprozesse und Qualitätssicherung **auffrischen**
- Probleme mit der **Pflege „langlebiger" Software** kennenlernen
- Begriffe „**Software-Wartung und -Evolution**" definieren können
- **Forward-, Reverse- und Reengineering** unterscheiden können
- Motivation für den Inhalt dieser Lehrveranstaltung

1.1 Einleitung

Zur Erinnerung - die **Geschichte der Software-Technik**:

- Auslöser für Einführung der Software-Technik war die **Software-Krise** von 1968
- Der Begriff "**Software Engineering**" wurde von F.L. Bauer im Rahmen einer Study Group on Computer Science der NATO geprägt
- Bahnbrechend war die NATO-Konferenz

  "**Working Conference on Software Engineering**"

  vom 7. - 10. Oktober 1968 in Garmisch
- Das Gebiet Software-Technik wird der praktischen Informatik zugeordnet, hat aber auch Wurzeln in der theoretischen Informatik
- Informatikübergreifende Aspekte spielen eine wichtige Rolle (wie Projektplanung, Organisation, Psychologie, …)

Definition des Begriffs „Software-Technik”:

Software Engineering = **Software-Technik** ist nach [BEM92]:

- ⇨ die Entwicklung
- ⇨ die Pflege und
- ⇨ der Einsatz

qualitativ hochwertiger Software unter Einsatz von

- ⇨ wissenschaftlichen Methoden
- ⇨ wirtschaftlichen Prinzipien
- ⇨ geplanten Vorgehensmodellen
- ⇨ Werkzeugen
- ⇨ quantifizierbaren Zielen

Bislang in „Software Engineering - Einführung“ kennengelernt:

Entwicklung von Software, aber nicht Pflege (und Einsatz).

Bald 50 Jahre nach Beginn der Software-Krise:

Standish Group (http://www.standishgroup.com) veröffentlicht in regelmäßigen Abständen den sogenannten „Chaos Report“ mit folgenden Ergebnissen (für 2015):

- 19% aller betrachteten IT-Projekte sind gescheitert (früher: 25%)
- 52% aller betrachteten IT-Projekte sind dabei zu scheitern (früher: 50%) (signifikante Überschreitungen von Finanzbudget und Zeitrahmen)
- 29% aller betrachteten IT-Projekte sind erfolgreich (früher: 25%)

Hauptgründe für Scheitern von Projekten:

Nach wie vor unklare Anforderungen und Abhängigkeiten sowie Probleme beim **Änderungsmanagement**!!!

1.2 Software-Qualität

Ziel der Software-Technik ist die effiziente Entwicklung **messbar** qualitativ hochwertiger Software, die

- ⇨ gewünschte Funktionalität anbietet
- ⇨ benutzerfreundliche Oberfläche besitzt
- ⇨ korrekt bzw. zuverlässig arbeitet
- ⇨ ...

Definition des Begriffs „Qualität“ nach [IEEE 610]:

Qualität ist der Grad, in dem ein System, eine Komponente oder ein Prozess die Kundenerwartungen und Kundenbedürfnisse erfüllt.

Definition des Begriffs „Softwarequalität“ nach [ISO 9126]:

Softwarequalität ist die Gesamtheit der Funktionalitäten und Merkmale eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.

Qualitätsmerkmale gemäß [ISO 9126]:

- Funktionalität (Functionality)

Nichtfunktionale Merkmale:

- Zuverlässigkeit (Reliability)
- Benutzbarkeit (Usability)
- Effizienz (Efficiency)
- Änderbarkeit (Maintainability)
- Übertragbarkeit (Portability)

Prinzipien der Qualititätssicherung:

- **Qualitätszielbestimmung**: Auftraggeber und Auftragnehmer legen vor Beginn der Software-Entwicklung gemeinsames Qualitätsziel für Software-System mit nachprüfbarem Kriterienkatalog fest (als Bestandteil des abgeschlossenen Vertrags zur Software-Entwicklung)
- **quantitative Qualitätssicherung**: Einsatz automatisch ermittelbarer Metriken zur Qualitätsbestimmung (objektivierbare, ingenieurmäßige Vorgehensweise)
- **konstruktive Qualitätssicherung**: Verwendung geeigneter Methoden, Sprachen und Werkzeuge (Vermeidung von Qualitätsprobleme)
- **integrierte, frühzeitige analytische Qualitätssicherung**: systematische Prüfung aller erzeugten Dokumente (Aufdeckung von Qualitätsproblemen)
- **unabhängige Qualitätssicherung**: Entwicklungsprodukte werden durch eigen-ständige Qualitätssicherungsabteilung überprüft und abgenommen (verhindert u.a. Verzicht auf Testen zugunsten Einhaltung des Entwicklungszeitplans)

Konstruktives Qualitätsmanagement zur Fehlervermeidung:

- **technische Maßnahmen**:
  - ⇨ Sprachen (wie z.B. UML für Modellierung, Java für Programmierung)
  - ⇨ Werkzeuge (UML-CASE-Tool wie Visual Paradigm oder …)
- **organisatorische Maßnahmen**:
  - ⇨ Richtlinien (Gliederungsschema für Pflichtenheft, Programmierrichtlinien)
  - ⇨ Standards (für verwendete Sprachen, Dokumentformate, Management)
  - ⇨ Checklisten (wie z.B. „bei Ende einer Phase müssen folgende Dokumente vorliegen” oder “Softwareprodukt erfüllt alle Punkte des Lastenheftes”)

Einhaltung von Richtlinien, Standards und Überprüfung von Checklisten kann durch Werkzeugeinsatz = technische Maßnahmen erleichtert (erzwungen) werden.

Beispiel für die Wichtigkeit konstruktiver Qualitätssicherung:

Das Mariner-Unglück: Mariner 1 (Venus-Sonde) musste 4 Minuten nach dem Start wegen unberechenbarem Flugverhalten zerstört werden. Gerüchten zufolge war folgender Softwarefehler in einem Fortranprogramm schuld:

Korrektes Programm:

```
DO 3 i= 1,3
... (irgendwelche Befehle)
3 CONTINUE
```

Fehlerhaftes Programm:

```
DO 3 i=1.3
... (irgendwelche Befehle)
3 CONTINUE
```

Erläuterung:

- korrektes Programm ist eine Schleife, die dreimal ausgeführt wird (für i = 1 bis 3)
- falsches Programm ist eine Zuweisung: DO3i = 1.3
- in Fortran spielen nämlich Leerzeichen keine Rolle
- Variablen brauchen nicht deklariert werden (DO3i ist eine Real-Variable)
- es gibt keine reservierten Schlüsselwörter (wie etwa DO)

Schlussfolgerungen aus Mariner-Vorfall:

- durch systematisches Testen hätte man den Fehler vermutlich bei Testläufen (Simulationen) finden können
- eine „vernünftige“ Programmiersprache hätte die Verwendung eines Dezimal-punktes anstelle von Komma als Syntaxfehler zur Übersetzungszeit gefunden
- das Aufstellen von (durch Werkzeuge überprüfte) Programmierrichtlinien für Fortranprogramme hätte das Unglück auch vermieden:
  - ⇨ alle Variablen werden deklariert (obwohl es nicht notwendig ist)
  - ⇨ die Verwendung von Leerzeichen in Variablennamen wird verboten
  - ⇨ ebenso die Verwendung von Schlüsselwörtern in Variablennamen
- ...

Analytisches Qualitätsmanagement zur Fehleridentifikation:

- **analysierende Verfahren**:

  der „Prüfling" (Programm, Modell, Dokumentation) wird von Menschen oder Werkzeugen auf Vorhandensein/Abwesenheit von Eigenschaften untersucht

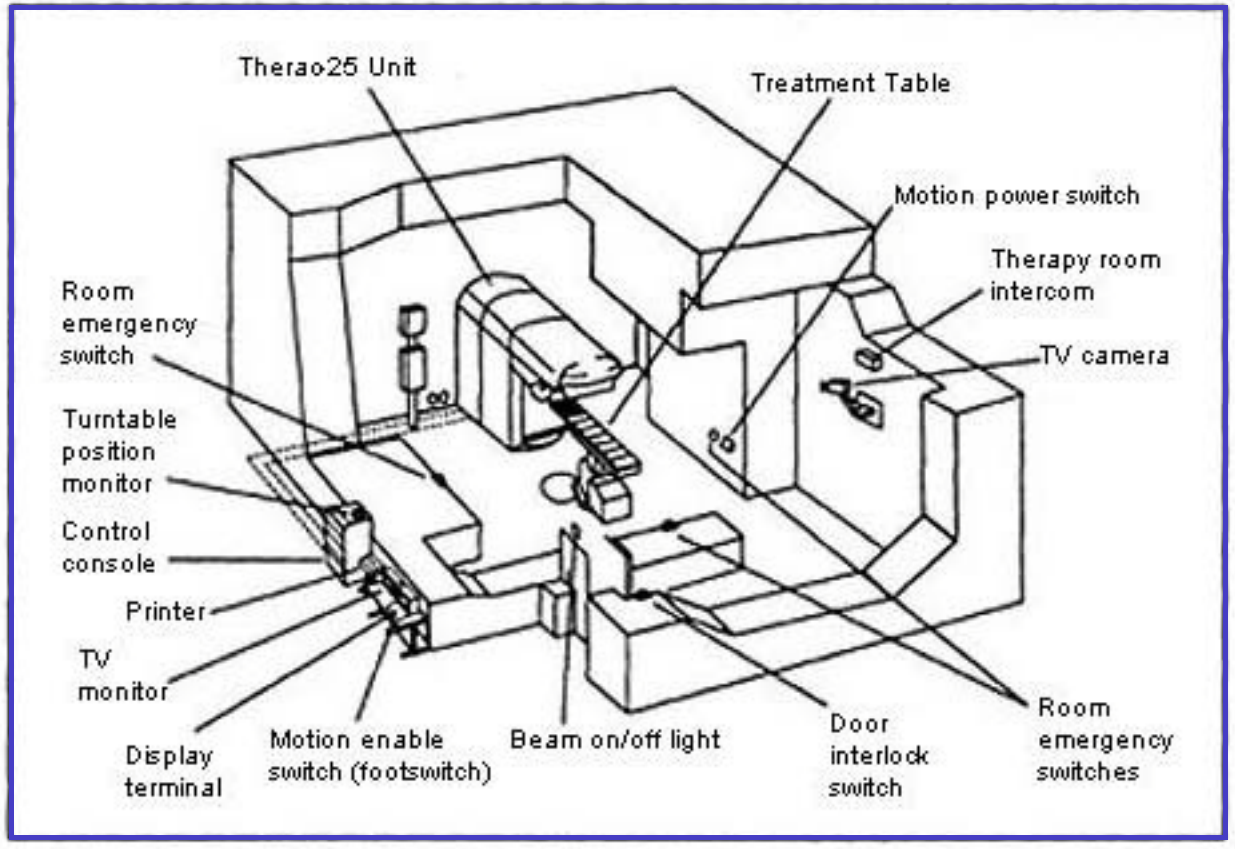
  - ⇨ **Review** (Inspektion, Walkthrough): Prüfung durch (Gruppe v.) Menschen
  - ⇨ **statische Analyse**: werkzeuggestützte Ermittlung von „Anomalien"
  - ⇨ **(formale) Verifikation**: werkzeuggestützter Beweis von Eigenschaften

- **testende Verfahren**:

  der „Prüfling" wird mit konkreten oder abstrakten Eingabewerten auf einem Rechner ausgeführt

  - ⇨ **dynamischer Test**: „normale" Ausführung mit ganz konkreten Eingaben
  - ⇨ **[symbolischer Test**: Ausführung mit symbolischen Eingaben (die oft unendliche Mengen möglicher konkreter Eingaben repräsentieren)]

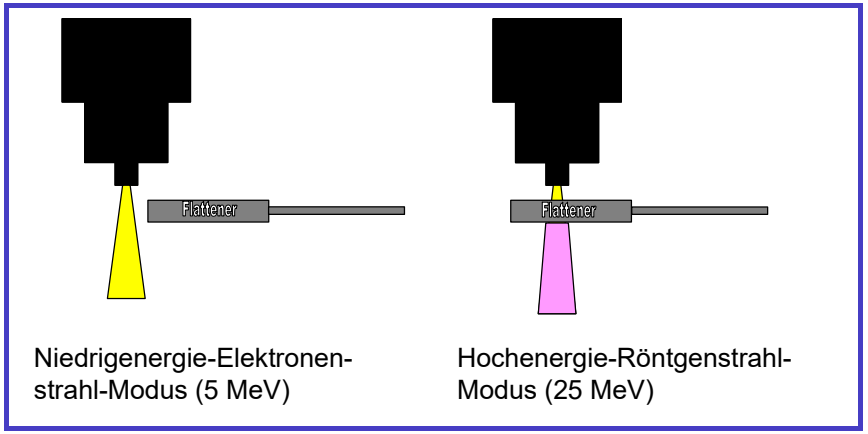
Beispiel für Wichtigkeit analytischer Qualitätssicherung - Therac 25:



Bildquelle: https://www.computingcases.org/images/therac/therac_facility.jpg

Therac 25 - ein rein „softwaregesteuertes“ Bestrahlungsgerät:

Mindestens 6 Personen erhalten zwischen 1985 und 1987 größtenteils tödliche Überdosis bei Krebstherapie mit Strahlenbehandlung, da:



- ☹ Maschine hat zwei Betriebsmodi: Niedrigenergie- und Hochenergiebestrahlung
- ☹ tödliche Bestrahlung im Hochenergiemodus wird allein durch „Flattener“ verhindert, der Strahl modifiziert
- ☹ in mehreren Fällen war „Flattener“ bei Hochenergiebestrahlung falsch positioniert

Externe Sicht auf einen Unglücksfall:

- ☹ Techniker tippt „x“ für „X-Ray-Behandlung ein - anstatt „e“ für „electron mode“
- ☹ Techniker entdeckt Fehler allerdings sofort und korrigiert Fehleingabe sehr schnell und startet Behandlung
- ☹ auf dem Bildschirm wird die Fehlermeldung „Malfunction 54“ ausgegeben
- ☹ ähnliche Fehlermeldungen hatten bislang immer vorzeitigen Abbruch einer Behandlung ohne Bestrahlung angezeigt
- ☹ Techniker wiederholt also den Vorgang (Behandlung) zweimal
- ☹ Patient wird dreimal mit 25 MeV bestrahlt und stirbt 4 Monate später
- ☹ Techniker beharrt hartnäckig darauf, dass Therac-25 eine Fehlfunktion hatte, diese lässt sich aber zunächst nicht reproduzieren
- ☹ Herstellerfirma streitet das wohl zunächst längere Zeit ab; weitere Todesfälle sind die Folge

Interne Sicht auf den Vorfall:

- Intensität des Elektronenstrahls und Position des „Flatteners“ werden allein durch Software kontrolliert
- zu schnelle Eingabe der Daten führte dazu, dass
  - ⇨ „Flattener“ zwar aus Strahl entfernt wurde
  - ⇨ Intensität des Elektronenstrahls aber nicht korrigiert wurde
- erkannter Fehlerzustand führte zu kryptischer Fehlermeldung, aber nicht zur Blockade/Abbruch der Behandlung
- allein (fehlerhaft implementierte) Software-Locks sollen verhindern, dass Elektronenstrahl mit 25 MeV ohne „Flattener“ den Patienten trifft:
  - ⇨ Sperre wird durch Inkrementieren eines Bytes um 1 gesetzt
  - ⇨ Sperre wird durch Dekrementieren des Bytes um 1 rückgesetzt
  - ⇨ Sperrenüberprüfung erfolgt durch Test auf Wert (ungleich) „0“
  - ⇨ ein rekursiver Warteprozess führt zunächst beliebig viele Sperranforderungen aus, bevor er schließlich alle Sperren wieder freigibt

Fehleranalyse und -ursachen aus Managementsicht:

Übliche Sicherheitsanalyse für Therac-25 ging davon aus, dass die verwendete Software immer fehlerfrei funktionieren würde, weil (Zitate aus Bericht):

- ☹ „Programming errors have been reduced by extensive testing on a hardware simulator and under field conditions“
- ☹ „Program software does not degrade due to war, fatigue, or reproduction process“
- ☹ „Computer execution errors are caused by faulty hardware components and ... “

Die Wahrscheinlichkeit für „**Computer selects wrong energy**“ wurde deshalb auf 10^{-11} gesetzt (angenommener Zeitraum ist mir unbekannt).

Schwerpunkte „meiner“ Software Engineering-Vorlesungen:

Schwerpunkte von „Software Engineering - Einführung“:

- ⇨ Sprachen für die Entwicklung/Modellierung von Software
- ⇨ Werkzeuge zur Konstruktion von Software
- ⇨ Vorgehensmodelle für konstruktive Qualitätssicherung
- ⇨ Vorgehensweisen zur Verbesserung/Restrukturierung von Software

Schwerpunkte von „Software Engineering - hier“:

- ⇨ Vorgehensmodelle für analytische Qualitätssicherung
- ⇨ Werkzeuge zur Überprüfung von Software(-Qualität)
- ⇨ konstruktive Maßnahmen zum Management von Software-Entwicklungs- und Änderungsprozessen

1.3 Iterative Software-Entwicklung

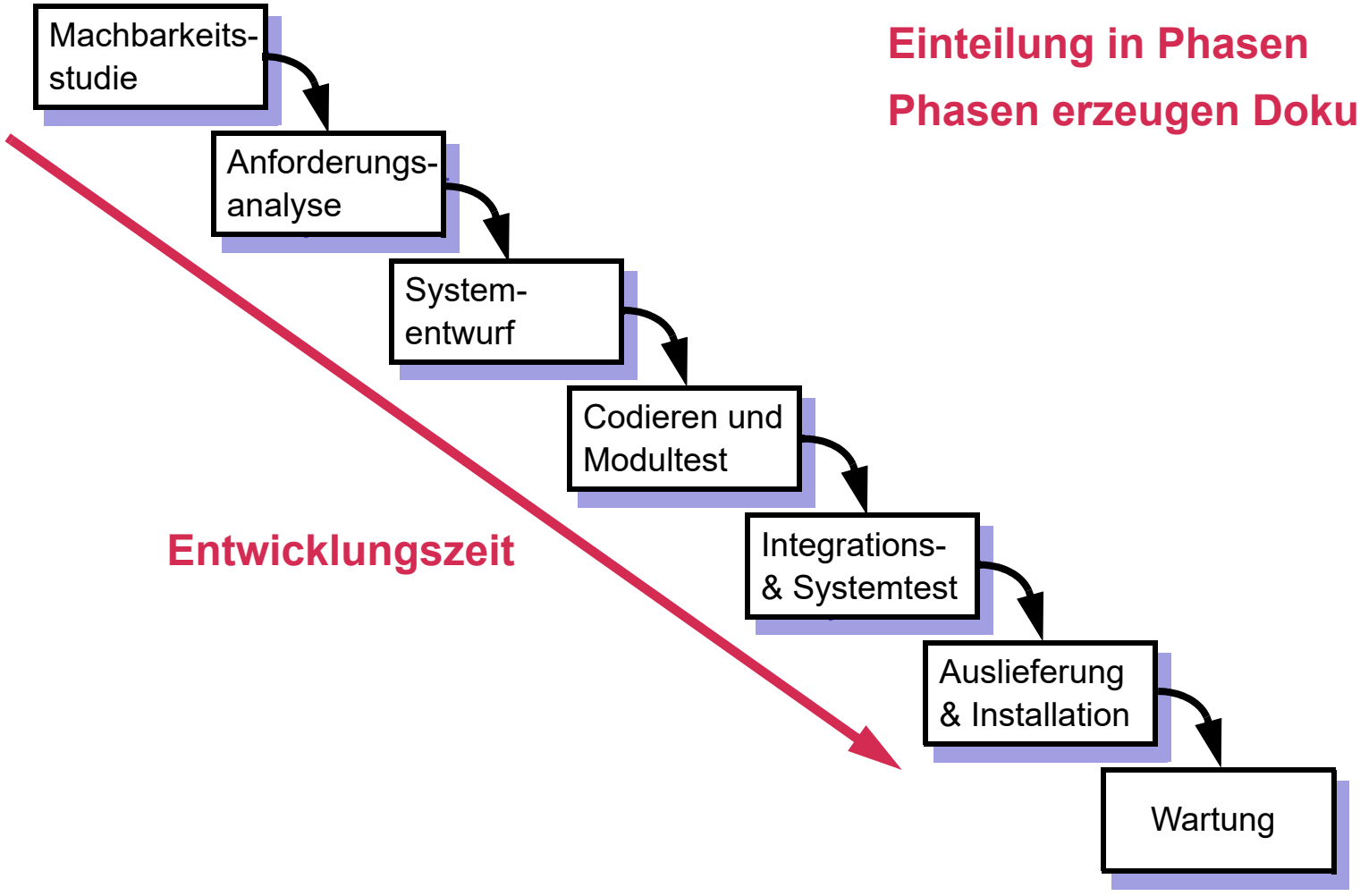
Voraussetzung für den sinnvollen Einsatz von Notationen und Werkzeugen zur Software-Entwicklung ist ein

- **Vorgehensmodell**, das den Gesamtprozeß der Software-Erstellung und -pflege in einzelne Schritte aufteilt.
- Zusätzlich müssen Verantwortlichkeiten der beteiligten Personen in Form von **Rollen** im Software-Entwicklungsprozess klar geregelt sein.

Zur Erinnerung:

- das „Wasserfallmodell“ war lange Zeit das Standardvorgehensmodell zur Erstentwicklung von Software
- in den letzten Jahren wurden für die (Weiter-)Entwicklung von Software bessere **iterative Vorgehensmodelle** entwickelt:
  ⇨ V-Modell, Rational Unified Process, ...

Die Phasen des Wasserfallmodells im Überblick (nach Royce):



Einteilung in Phasen

Phasen erzeugen Dokumente

Machbarkeitsstudie (feasibility study):

Die Machbarkeitsstudie schätzt **Kosten und Ertrag der** geplanten **Software-Entwicklung** ab. Dazu grobe Analyse des Problems mit Lösungsvorschlägen.

- **Aufgaben**:
  - ⇨ Problem informell und abstrahiert beschreiben
  - ⇨ verschiedene Lösungsansätze erarbeiten
  - ⇨ Kostenschätzung durchführen
  - ⇨ Angebotserstellung
- **Ergebnisse**:
  - ⇨ Lastenheft = (sehr) grobes Pflichtenheft
  - ⇨ Projektkalkulation
  - ⇨ Projektplan
  - ⇨ Angebot an Auftraggeber

Anforderungsanalyse (requirements engineering):

In der Anforderungsanalyse wird exakt festgelegt, **was die Software leisten soll**, aber nicht wie diese Leistungsmerkmale erreicht werden.

- **Aufgaben**:
  - ⇨ genaue Festlegung der Systemeigenschaften wie Funktionalität, Leistung, Benutzungsschnittstelle, Portierbarkeit, … im Pflichtenheft
  - ⇨ Bestimmen von Testfällen
  - ⇨ Festlegung erforderlicher Dokumentationsdokumente
- **Ergebnisse**:
  - ⇨ Pflichtenheft = Anforderungsanalysedokument
  - ⇨ Akzeptanztestplan
  - ⇨ Benutzungshandbuch (1-te Version)

Systementwurf (system design/programming-in-the-large):

Im Systementwurf wird exakt festgelegt, wie die Funktionen der Software zu realisieren sind. Es wird der **Bauplan der Software**, die Software-Architektur, entwickelt.

- **Aufgaben**:
  - ⇨ Programmieren-im-Großen = Entwicklung eines Bauplans
  - ⇨ Grobentwurf, der System in Teilsysteme/Module zerlegt
  - ⇨ Auswahl bereits existierender Software-Bibliotheken, Rahmenwerke, …
  - ⇨ Feinentwurf, der Modulschnittstellen und Algorithmen vorgibt
- **Ergebnisse**:
  - ⇨ Entwurfsdokument mit Software-Bauplan
  - ⇨ detailierte(re) Testpläne

Codieren und Modultest (programming-in-the-small):

Die eigentliche **Implementierungs- und Testphase**, in der einzelne Module (in einer bestimmten Reihenfolge) realisiert und validiert werden.

- **Aufgaben**:
  - ⇨ Programmieren-im-Kleinen = Implementierung einzelner Module
  - ⇨ Einhaltung von Programmierrichtlinien
  - ⇨ Code-Inspektionen kritischer Modulteile (Walkthroughs)
  - ⇨ Test der erstellten Module
- **Ergebnisse**:
  - ⇨ Menge realisierter Module
  - ⇨ Implementierungsberichte (Abweichungen vom Entwurf, Zeitplan, …)
  - ⇨ technische Dokumentation einzelner Module
  - ⇨ Testprotokolle

Integration und Systemtest:

Die einzelnen Module werden schrittweise zum **Gesamtsystem zusammengebaut**. Diese Phase kann mit der vorigen Phase verschmolzen werden, falls der Test isolierter Module nicht praktikabel ist.

- **Aufgaben**:
  - ⇨ Systemintegration = Zusammenbau der Module
  - ⇨ Gesamtsystemtest in Entwicklungsorganisation durch Kunden (α-Test)
  - ⇨ Fertigstellung der Dokumentation
- **Ergebnisse**:
  - ⇨ fertiges System
  - ⇨ Benutzerhandbuch
  - ⇨ technische Dokumentation
  - ⇨ Testprotokolle

Auslieferung und Installation:

Die Auslieferung (Installation) und **Inbetriebnahme** der Software **beim Kunden** findet häufig in zwei Phasen statt.

- **Aufgaben**:
  - ⇨ Auslieferung an ausgewählte (Pilot-)Benutzer (β-Test)
  - ⇨ Auslieferung an alle Benutzer
  - ⇨ Schulung der Benutzer
- **Ergebnisse**:
  - ⇨ fertiges System
  - ⇨ Akzeptanztestdokument

Wartung (Maintenance):

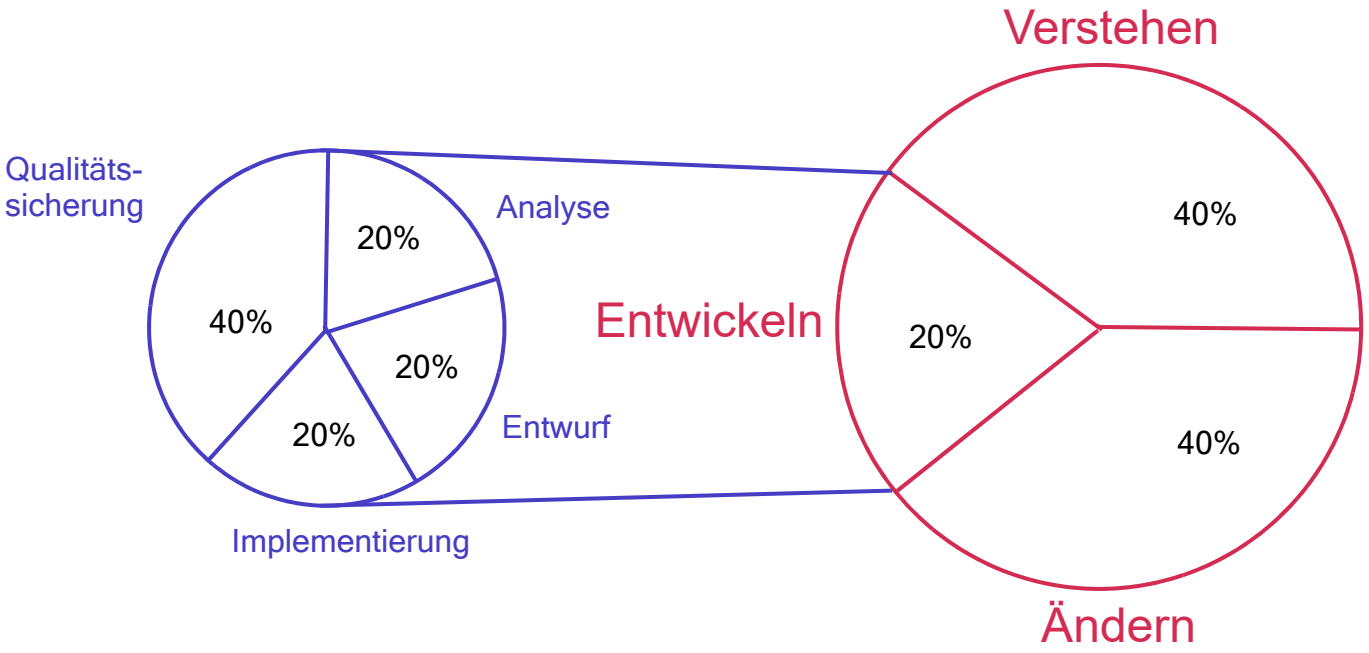
Nach der ersten Auslieferung der Software an die Kunden beginnt das Elend der Software-Wartung, das ca. 60% der gesamten Software-Kosten ausmacht.

- **Aufgaben**:
  - ca. 20% Fehler beheben (corrective maintenance)
  - ca. 20% Anpassungen durchführen (adaptive maintenance)
  - ca. 50% Verbesserungen vornehmen (perfective maintenance)
- **Ergebnisse**:
  - Software-Problemberichte (bug reports)
  - Software-Änderungsvorschläge
  - neue Software-Versionen

Probleme mit dem Wasserfallmodell:

- ☹ zu Projektbeginn sind nur ungenaue Kosten- und Ressourcenschätzungen möglich
- ☹ ein Pflichtenheft kann nie den Umgang mit dem fertigen System ersetzen, das erste sehr spät entsteht (Risikomaximierung)
- ☹ es gibt Fälle, in denen zu Projektbeginn kein vollständiges Pflichtenheft erstellt werden kann (weil Anforderungen nicht klar)
- ☹ Anforderungen werden früh eingefroren, notwendiger Wandel (aufgrund organisatorischer, politischer, technischer, … Änderungen) nicht eingeplant
- ☹ strikte Phaseneinteilung ist unrealistisch (Rückgriffe sind notwendig)
- ☹ Wartung mit ca. 60% des Gesamtaufwandes ist eine Phase

Andere Darstellung der Aufwandsverteilung:



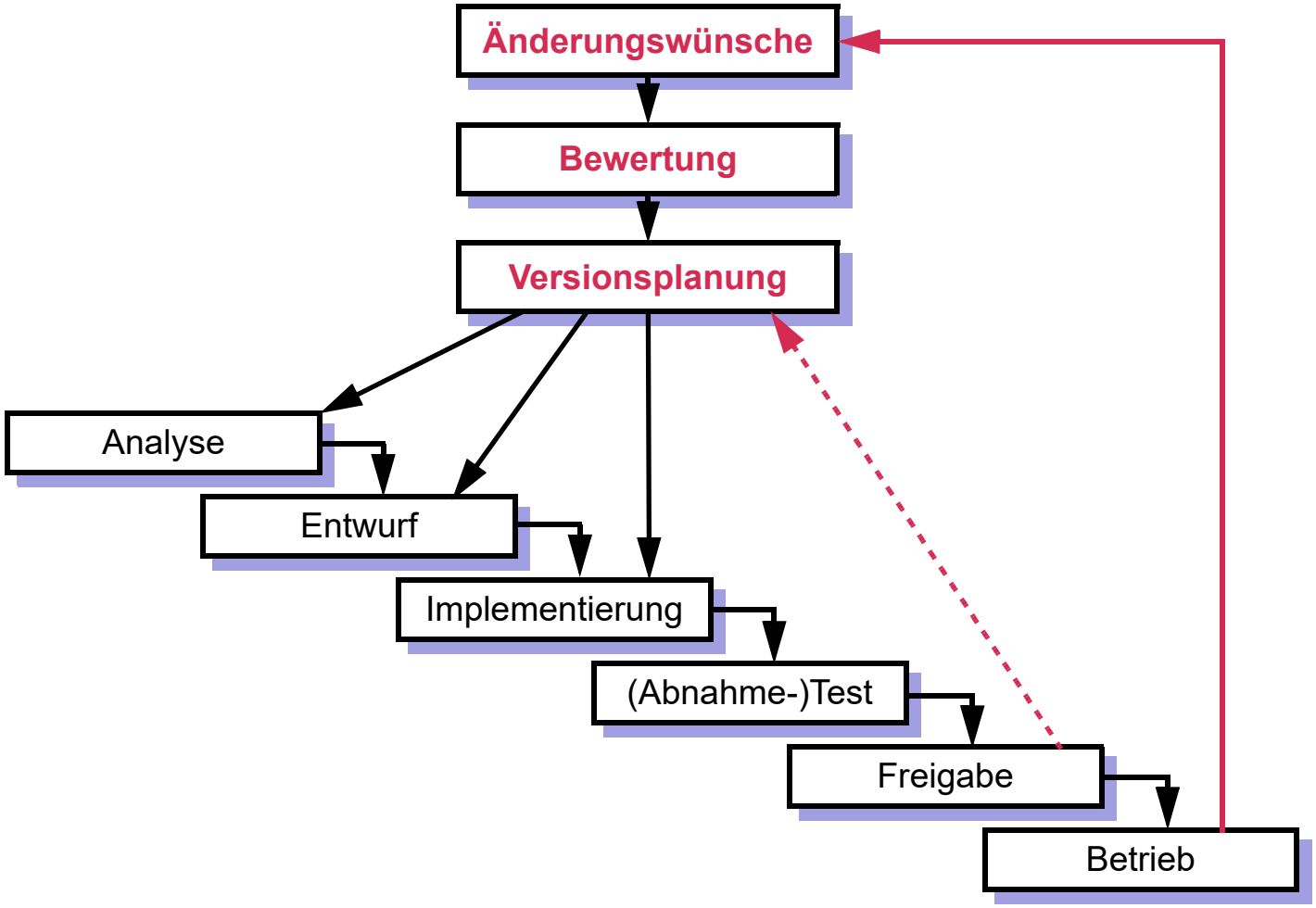
nach Nosek und Palv

⇨ wenn man „Verstehen“ (bestehenden Codes) ganz dem Bereich Software-Wartung zuschlägt, kommt man sogar auf **80% Wartungsaufwand**

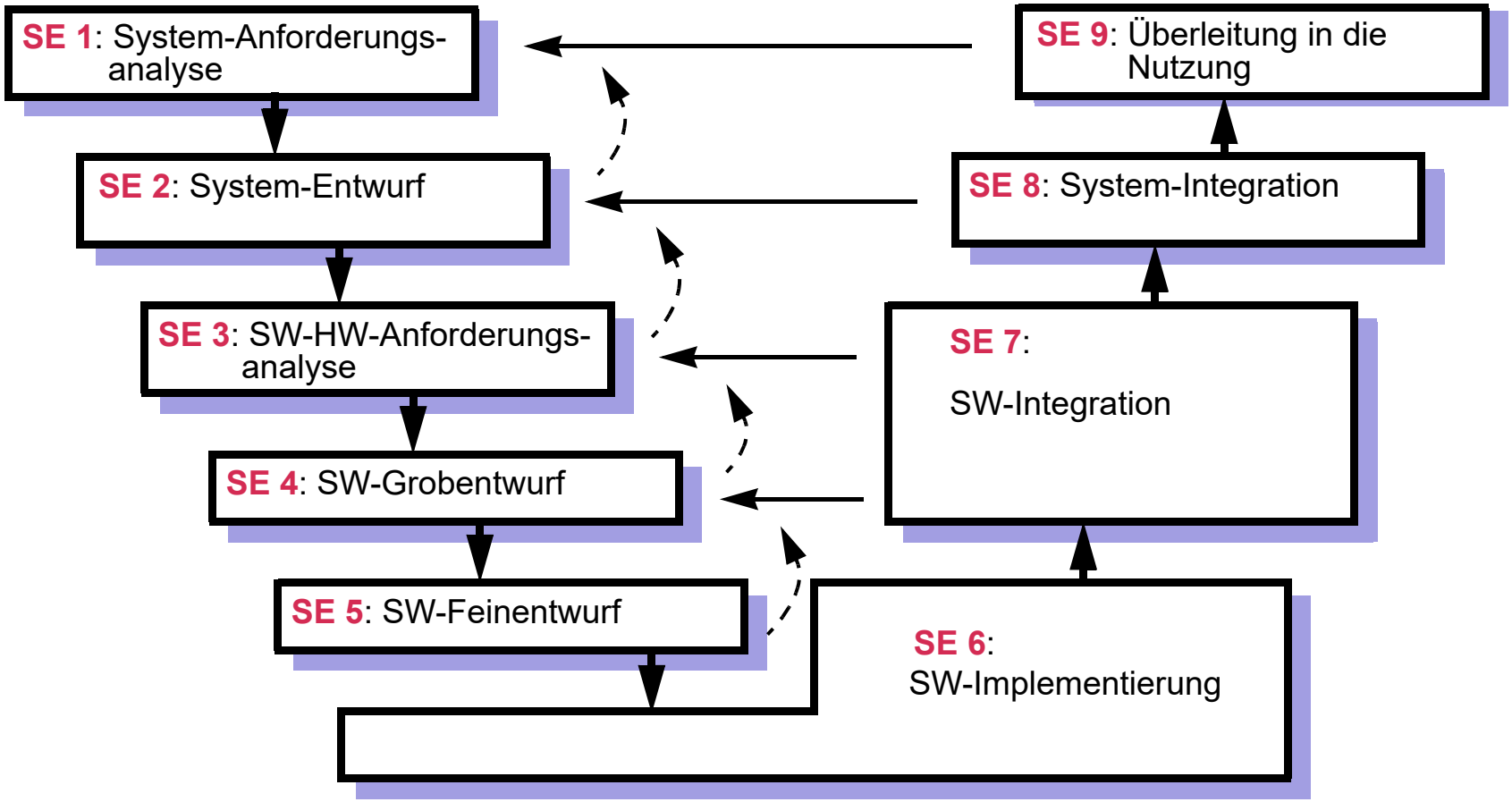
Typische Probleme in der Wartungsphase:

- Einsatz wenig erfahrenen Personals (nicht Entwicklungspersonal)
- Fehlerbehebung führt neue Fehler ein
- stetige Verschlechterung der Programmstruktur
- Zusammenhang zwischen Programm und Dokumentation geht verloren
- zur Entwicklung eingesetzte Werkzeuge (CASE-Tools, Compiler, …) sterben aus
- benötigte Hardware steht nicht mehr zur Verfügung
- Resourcenkonflikte zwischen Fehlerbehebung und Anpassung/Erweiterung
- völlig neue Ansprüche an Funktionalität und Benutzeroberfläche
- ...

Einfaches Software-Lebenszyklus-Prozessmodell für die Wartung:



Das V-Modell (Standard der Bundeswehr bzw. aller Bundesbehörden):



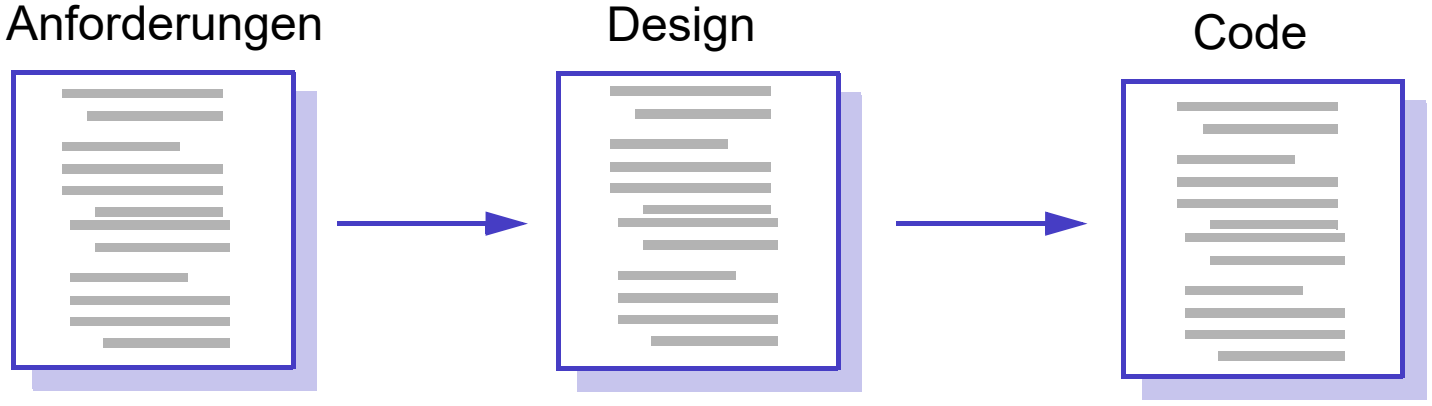
☞ weitere Informationen zu Prozessmodellen in „Kapitel 5 der Vorlesung“

Beschreibung der einzelnen Phasen des V-Modells:

- **Systemanforderungsanalyse**: Gesamtsystem einschließlich aller Nicht-DV-Komponenten wird beschrieben (fachliche Anforderungen und Risikoanalyse)
- **Systementwurf**: System wird in technische Komponenten (Subsysteme) zerlegt, also die Grobarchitektur des Systems definiert
- **Softwareanforderungsanalyse**: technischen Anforderungen an die bereits identifizierten Komponenten werden definiert
- **Softwaregrobentwurf**: Softwarearchitektur wird bis auf Modulebene festgelegt
- **Softwarefeinentwurf**: Details einzelner Module werden festgelegt
- **Softwareimplementierung**: wie beim Wasserfallmodell (inklusive Modultest)
- **Software-/Systemintegration**: schrittweise Integration und Test der verschiedenen Systemanteile
- **Überleitung in die Nutzung**: entspricht Auslieferung bei Wasserfallmodell

1.4 Forward-, Reverse- und Reengineering

Beim **Forward Engineering** ist das fertige Softwaresystem das Ergebnis des Entwicklungsprozesses. Ausgehend von Anforderungsanalyse (Machbarkeitsstudie) wird ein neues Softwaresystem entwickelt:



- ⇨ Anforderungs- und Design-Dokumente für Code existieren (hoffentlich)
- ⇨ alle Dokumente sind - da voneinander abgeleitet - (noch) konsistent
- ⇨ auf Entwickler des Codes kann (noch) zugegriffen werden

Wunsch und Wirklichkeit der Software-Evolution:

„Software-Systeme zu erstellen, die nicht geändert werden müssen, ist unmöglich. Wurde Software erst einmal in Betrieb genommen, entstehen neue Anforderungen und vorhandene Anforderungen ändern sich ... “

[So07]

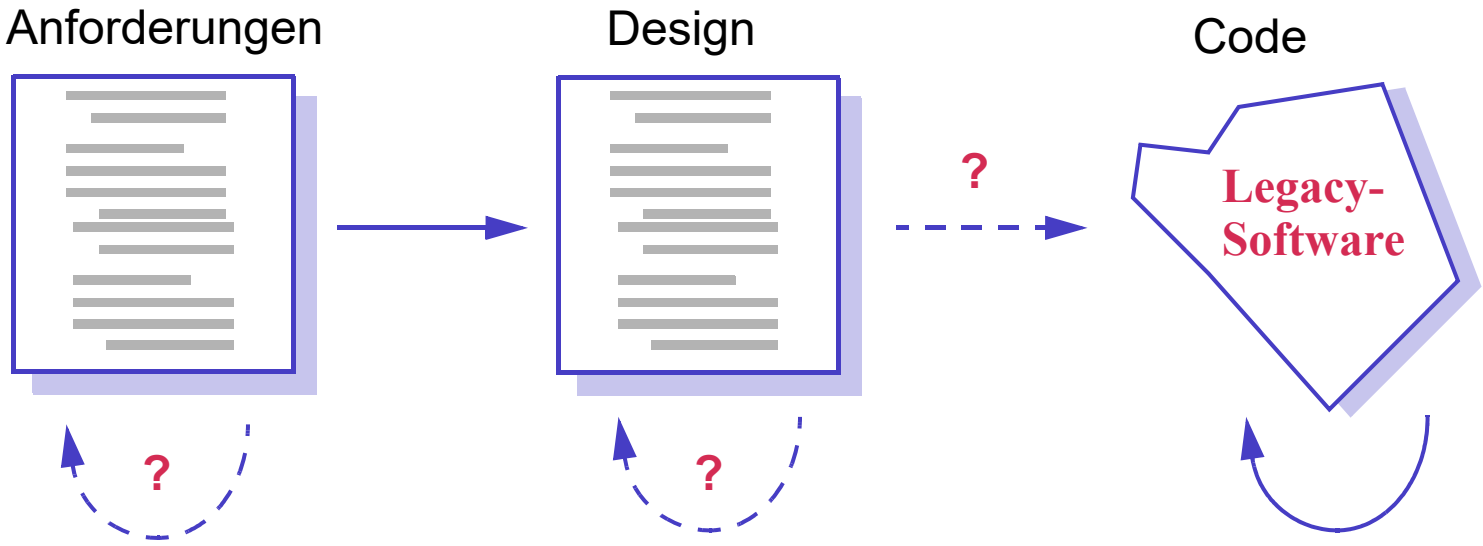
Software-Evolution - Wünsche:

- ☺ Wartung ändert Software kontrolliert ohne Design zu zerstören
- ☺ Konsistenz aller Dokumente bleibt erhalten

Software-Evolution - Wirklichkeit:

- ☹ ursprüngliche Systemstruktur wird ignoriert
- ☹ Dokumentation wird unvollständig oder unbrauchbar
- ☹ Mitarbeiter verlassen Projekt

Ergebnis unkontrollierter Software-Evolution:

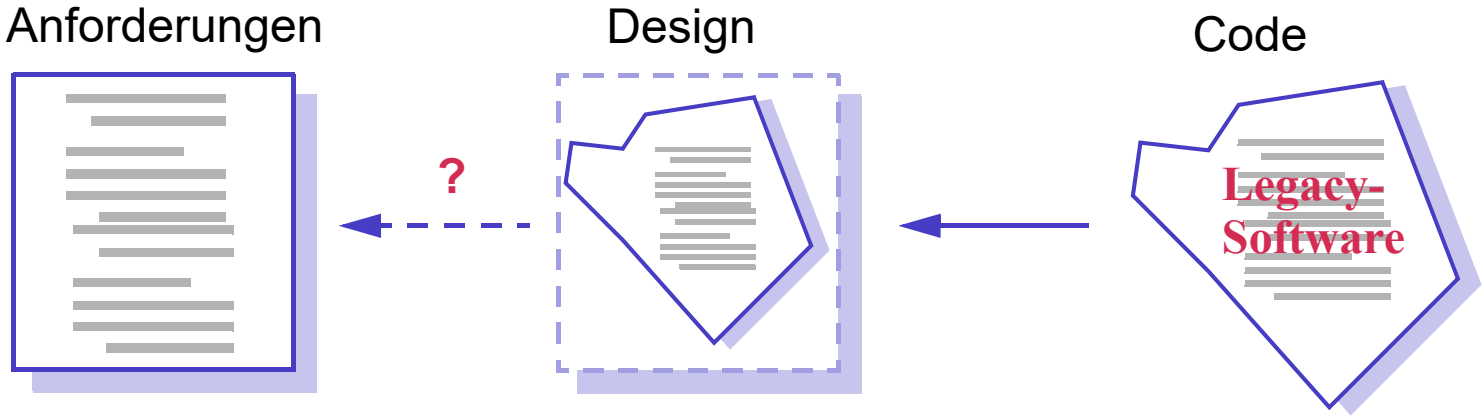


Legacy = „wertvolles“ Erbe

- ☹ der Code ändert sich mit Sicherheit, seine Struktur bleibt meist erhalten (auch wenn sie neuen Anforderungen eigentlich nicht gewachsen ist)
- ☹ das Design wird nach einer gewissen Zeit nicht mehr aktualisiert
- ☹ die Anforderungsdokumente werden erst recht nicht mehr gepflegt

Reverse Engineering von „Legacy Software“:

Beim **Reverse Engineering** ist das vorhandene Software-System der Ausgangspunkt der Analyse. Ausgehend von existierender Implementierung wird meist „nur“ das Design rekonstruiert und dokumentiert. Es wird (noch) nicht das betrachtete System modifiziert.



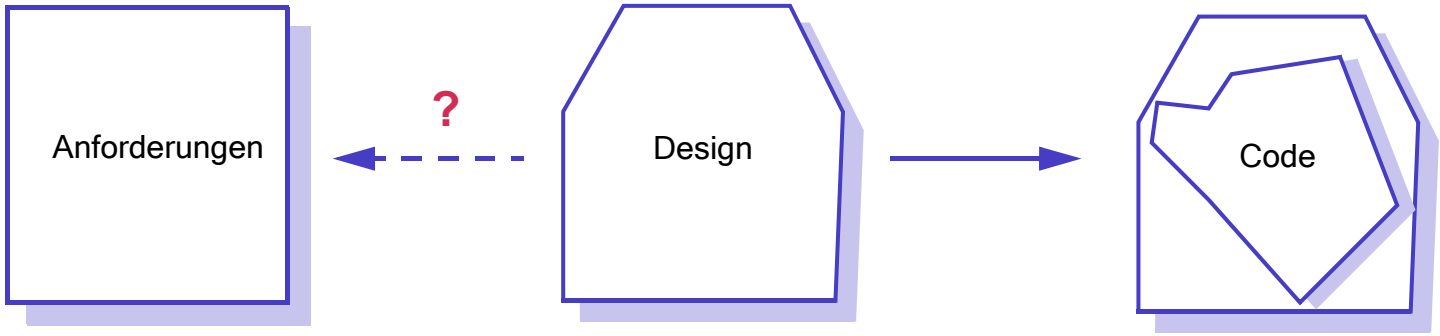
Fragen:

- Lässt sich das Software-System noch restrukturieren/sanieren?
- Oder kann man sein Innenleben „verkapseln“ und zunächst weiterverwenden?

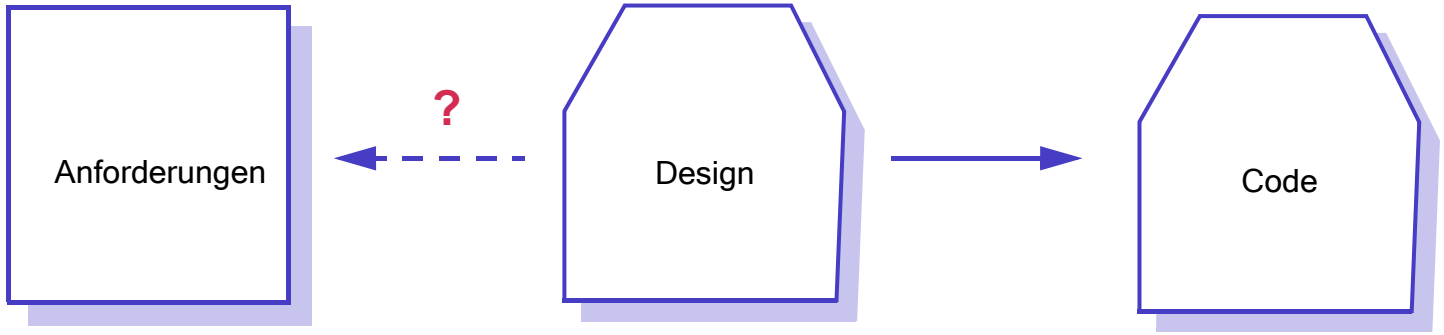
Strategien des Reengineerings:

Reengineering befaßt sich mit der Sanierung eines vorhandenen Software-Systems bzw. seiner Neuimplementierung. Dabei werden die Ergebnisse des Reverse Engineerings als Ausgangspunkt genommen

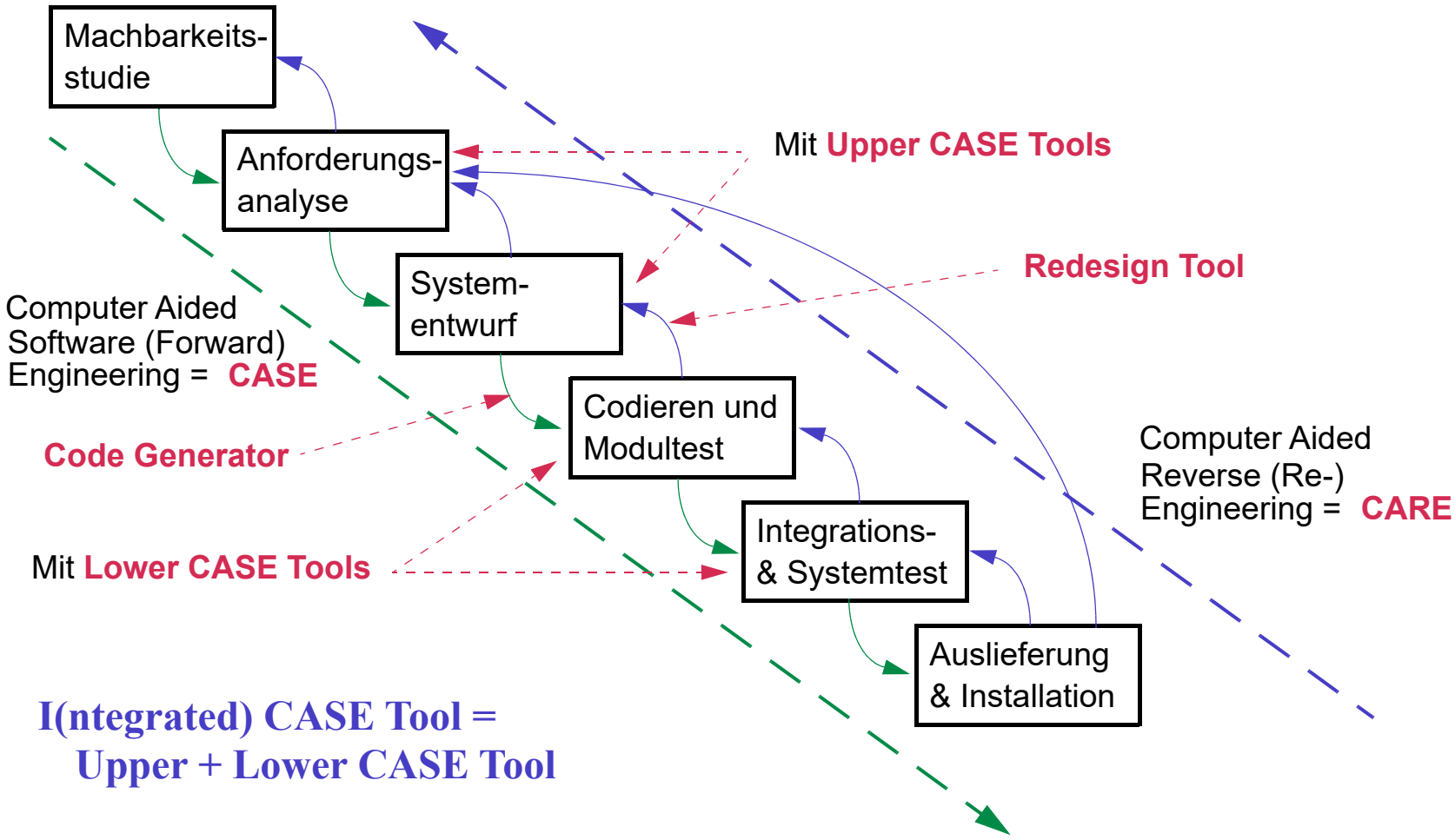
Restrukturierung des Designs, Kapselung des Codes:



Restrukturierung von Design und Code:



Forward- und Re(-verse)-engineering = Round Trip Engineering:



I(ntegrated) CASE Tool =
Upper + Lower CASE Tool

1.5 Zusammenfassung

Die Lehrveranstaltung „Software Engineering - Einführung“ und das „SW-Praktikum“ haben sich nur mit dem **Forward Engineering** von Software-Systemen befasst, also nur mit ca. 20% - 40% des Software-Lebenszyklus. Das Thema „**Software-Qualitäts-sicherung**“ wurde zudem nur kurz angerissen.

In dieser Vorlesung befassen wir uns deshalb mit:

- Kapitel 2: Management von Software-Änderungsprozessen
- Kapitel 3: Analyse und Überwachung von Software(-Qualität)
- Kapitel 4: Qualitätssicherung durch systematisches Testen
- Kapitel 5: Management der Software-Entwicklung

1.6 Zusätzliche Literatur

[BD00] B. Bruegge, A.H. Dutoit: *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*, Prentice Hall (2000)

[BEM92] A.W. Brown, A.N. Earl, J.A. McDermid: *Software Engineering Environments: Automated Support for Software Engineering*, McGraw-Hill (1992)

[Di72] E.W. Dijkstra: *The Humble Programmer*, Communications of the ACM, Vol. 15, No. 10 (1972)

[Fu93] A. Fugetta: *A Classification of CASE Technology*, Computer, Vol. 26, No. 12, S. 25-38, IEEE Computer Society Press (1993)

[GJ96] P.K. Garg, M. Jazayeri (Eds.): *Process-Centered Software Engineering Environments*, IEEE Computer Society Press (1996)

[Hr00] P. Hruschka: *Mein Weg zu CASE: von der Idee über Methoden zu Werkzeugen*, Hanser Verlag (1991)

[IEEE83] IEEE: *Standard Glossar of Software Engineering Terminology - IEEE Standard 729*, IEEE Computer Society Press (1983)

[Jo92] C. Jones: *CASE's Missing Elements*, IEEE Spektrum, Juni 1992, S. 38-41, IEEE Computer Society Press (1992)

[BEM92] B. Kahlbrandt: *Software-Engineering: Objektorientierte Software-Entwicklung mit der Unified Modeling Language*, Springer Verlag (1998)

[Na96] M. Nagl (ed.): *Building Thightly Integrated Software Development Environments: The IPSEN Approach*, LNCS 1170, Springer Verlag (1996)

[Ro92] Ch. Roth: *Die Auswirkungen von CASE*, Proc. GI-Jahrestagung 1992, Karlsruhe, Informatik aktuell, S. 648-656

[Sn87] H.M. Sneed: *Software-Management*, Müller GmbH (1987)