ETH Zürich
Chair of Software Engineering

Einführung in die Programmierung

Prof. Dr. Bertrand Meyer

Lektion 19: Von der Programmierung zum Software-Engineering

Software-Engineering (1)

Die Prozesse, Methoden, Techniken, Werkzeuge und Sprachen, um funktionsfähige Qualitätssoftware zu entwickeln.

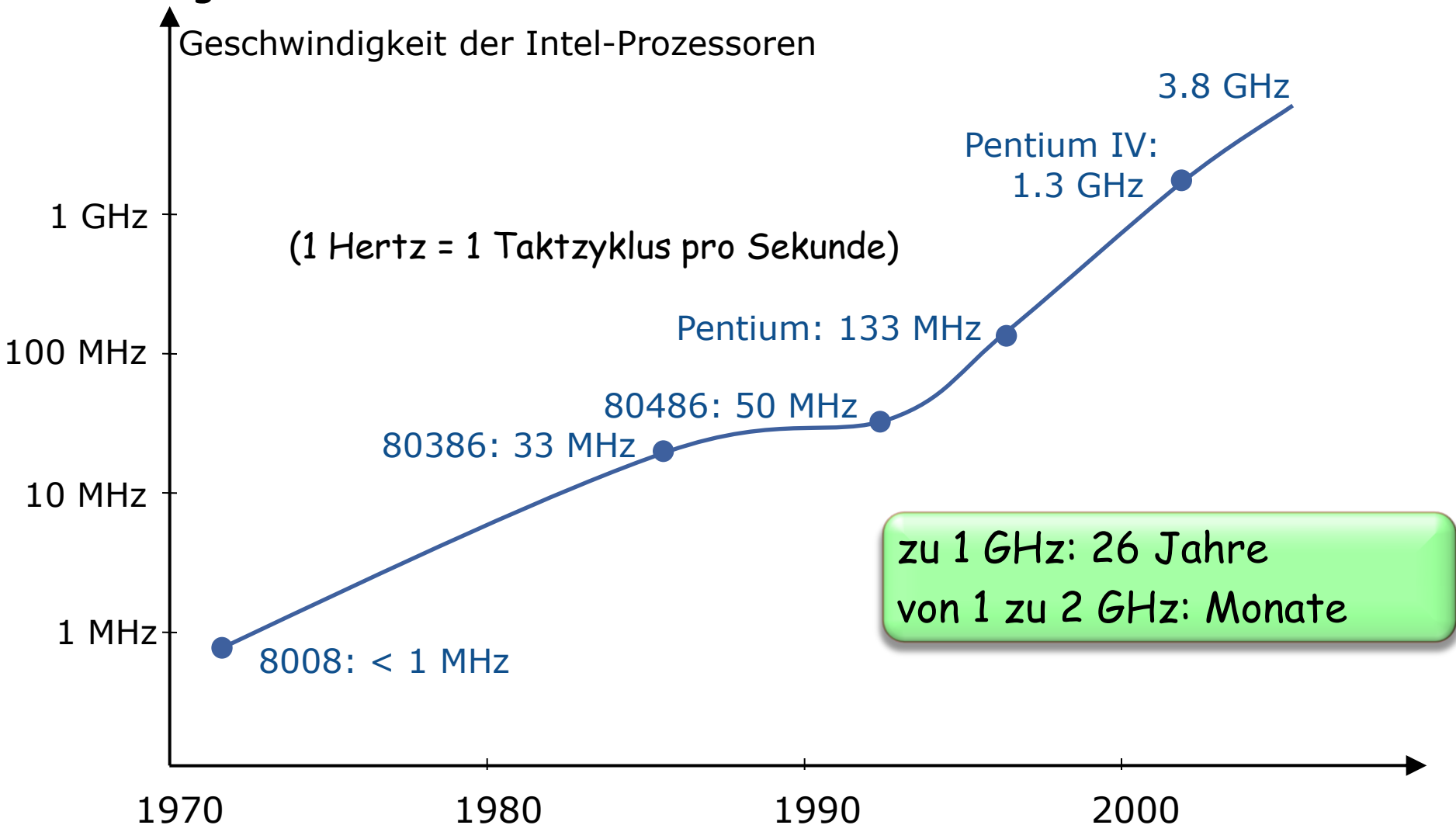
Software-Engineering (2)

Die Prozesse, Methoden, Techniken, Werkzeuge und Sprachen, um funktionsfähige Qualitätssoftware zu entwickeln, die

- sehr gross sein kann
- über einen langen Zeitraum entwickelt und benutzt wird
- viele Entwickler involviert
- oft geändert und verbessert wird.

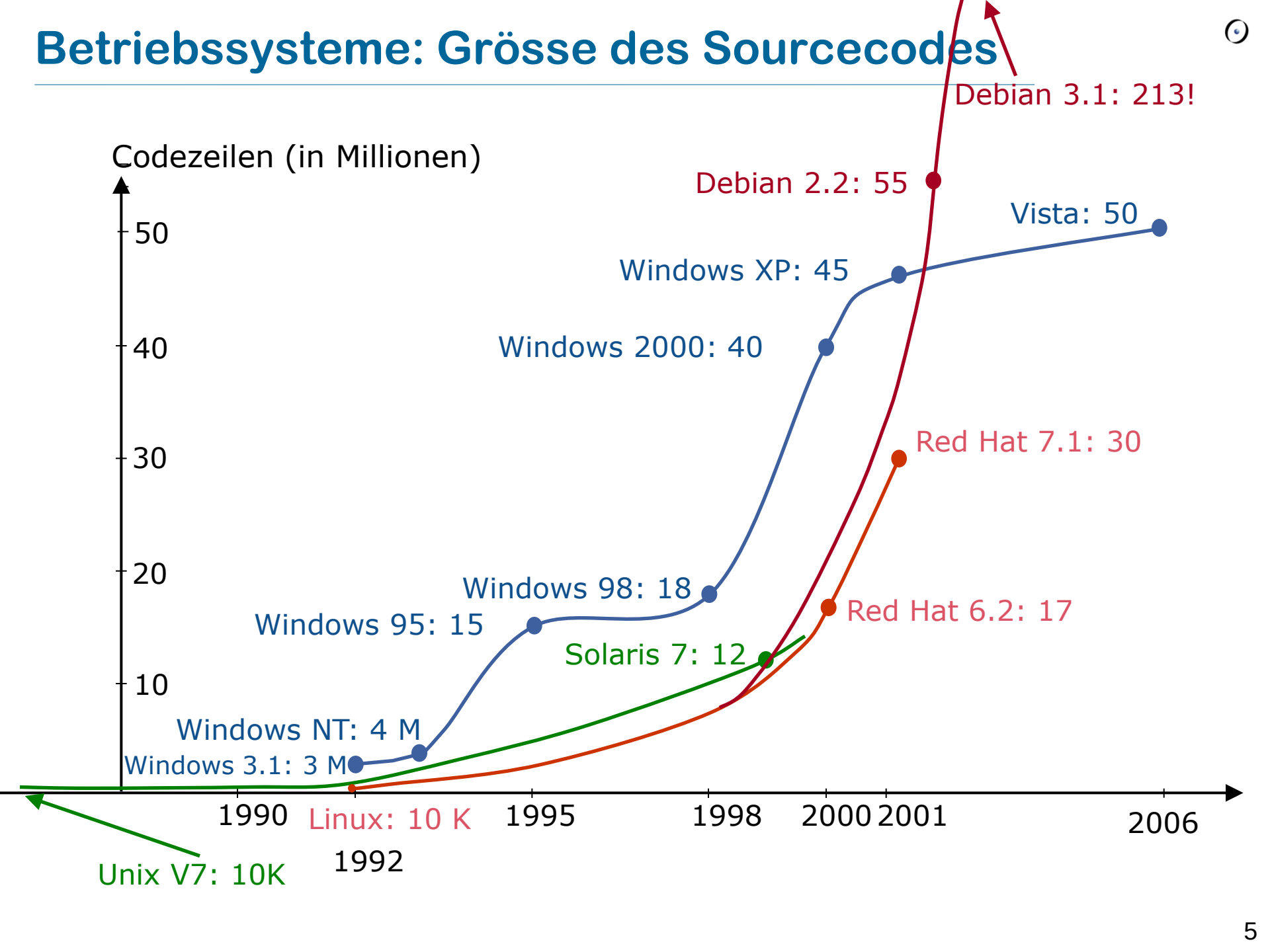
Moore's "Gesetz"

Ungefähre Verdopplung der Rechenleistung alle achtzehn Monate, für vergleichbare Preise



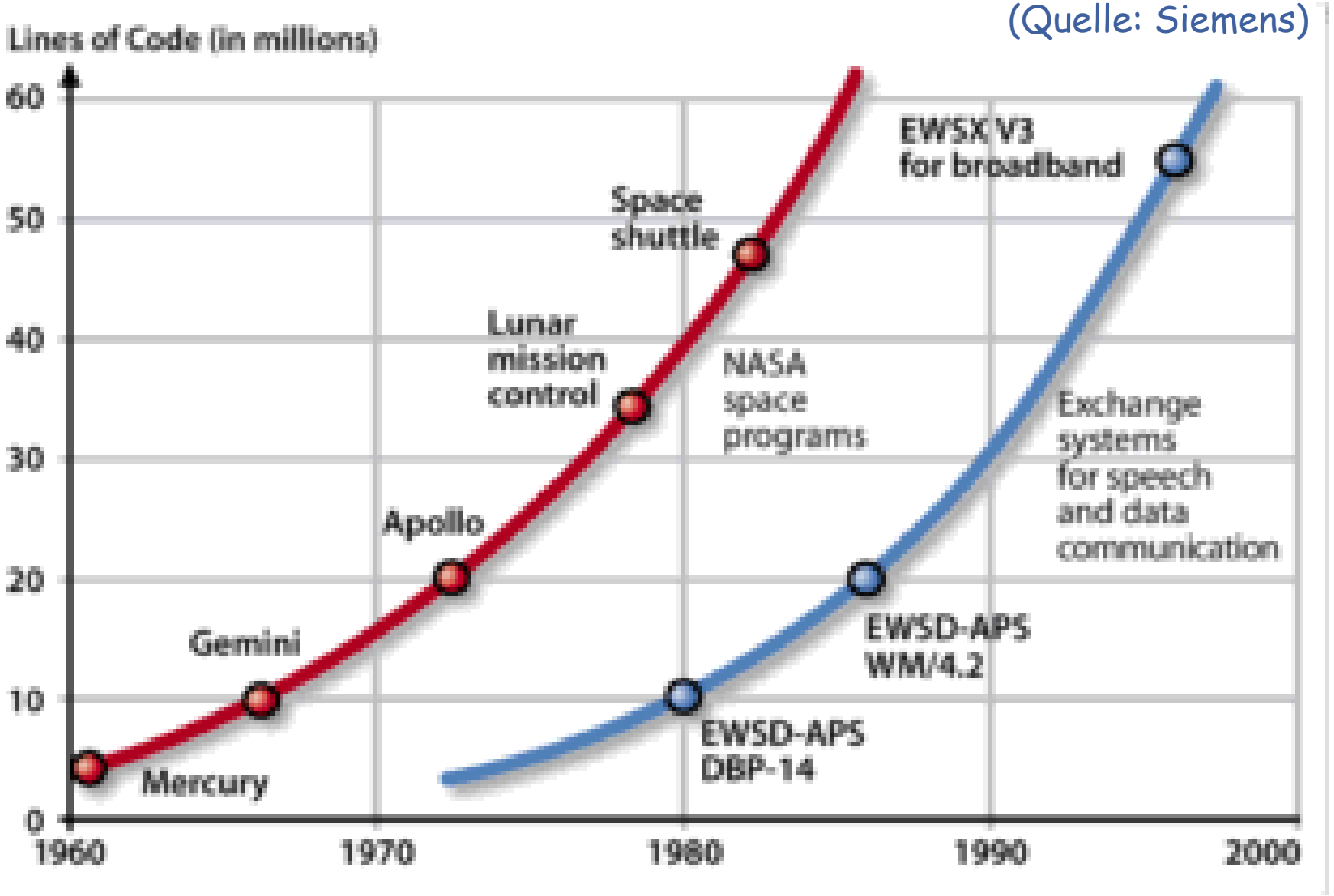
zu 1 GHz: 26 Jahre
von 1 zu 2 GHz: Monate

Betriebssysteme: Grösse des Sourcecodes



In anderen Anwendungsgebieten

(Quelle: Siemens)



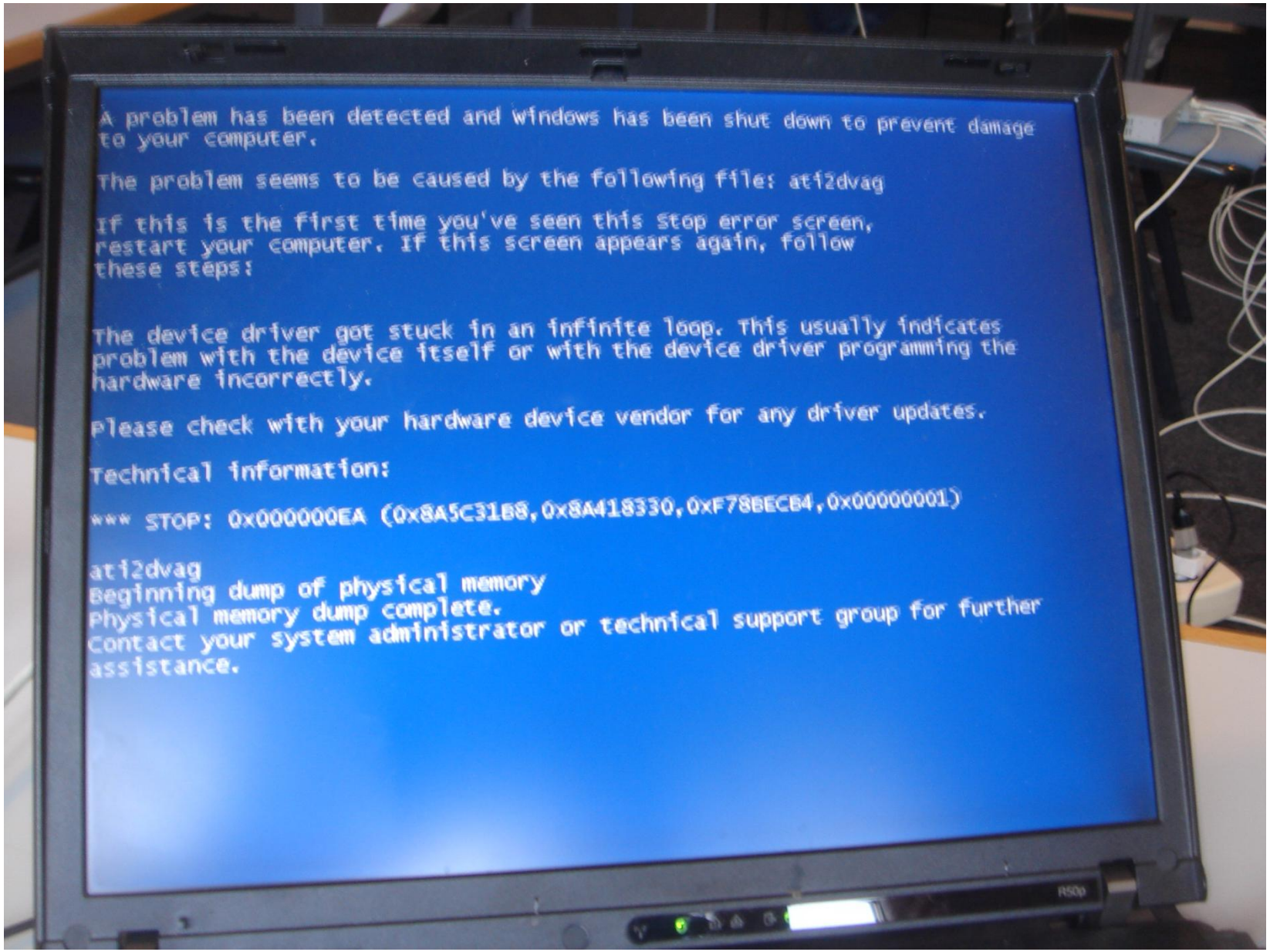
Das Grundproblem

Softwaresysteme zu entwickeln, die

- Rechtzeitig fertig und innerhalb des Budgets sind
- Von grosser unmittelbarer Qualität sind
- Möglicherweise gross und komplex sind
- Erweiterbar sind

Was heisst Qualität bei Software?

Keine Qualität



Associated Press, 21. Oktober 2000

LOS ANGELES. Failure of the Southwest's main air traffic radar system was traced to new software unable to recognize data typed manually by Mexico controllers.

The software installed Wednesday evening at the FAA's Los Angeles Center in the Mojave Desert, which controls aircraft over a 100,000-square-mile area, is the same upgrade completed successfully at 19 other FAA radar centers. But designers didn't allow for information typed in by Mexico controllers, who don't have a computerized system, the FAA spokesman said. "The computer didn't recognize the information from Mexico and it aborted". "A digit out of place could do it."

When controllers at the LA Center switched to the new system Thursday morning, it quickly failed when data from a Mexico controller was received. The radar system instantly switched to backup. The computer with the new software was restarted later, but failed again. The old system was reinstalled and the system returned to operation more than two hours later. Air travel schedules were left in disarray as the FAA ordered a nationwide ground stop for all flights bound for the Southwest, causing cancellations, rerouting, long delays and airport gridlock.

Technicians must now rewrite the software to recognize Mexico controller information. It wasn't clear when a revised program would be installed.

1998 Mars Orbiter Vehicle*

The orbiter was lost due to a miscalculation in trajectory. The miscalculation was caused by an unintended and undetected mismatch between metric and English units of measurement. The use of metric units as well as the data formats to employ were specified in a navigation software interface specification (SIS) published by JPL in 1996. Despite this, the flight operations team at Lockheed Martin provided impulse data in English units of pound-force seconds rather than newton seconds. These values were incorrect by a factor of 4.45 (1 lbf = 4.45 N). The mix-up caused erroneous course corrections that resulting in the orbiter descending too low in Mars atmosphere. The vehicle either burned up or bounced off into space.

*Quelle: Wikipedia

Ariane-5 Jungfernflug, 1996

37 Sekunden nach dem Start wurde eine Ausnahme im Ada-Programm nicht behandelt; es wurde der Befehl erteilt, die Mission abzubrechen. Verlust: ca. 10 Milliarden Dollar.

Die Ausnahme wurde durch eine inkorrekte Konvertierung verursacht: ein 64-Bit Real wurde falsch in einen 16-Bit Integer übersetzt.

Die systematische Analyse hatte „bewiesen", dass diese Ausnahme nicht auftreten kann – es wurde bewiesen, dass der 64-Bit Wert („horizontal bias" des Fluges) immer als 16-Bit Integer repräsentiert werden kann!

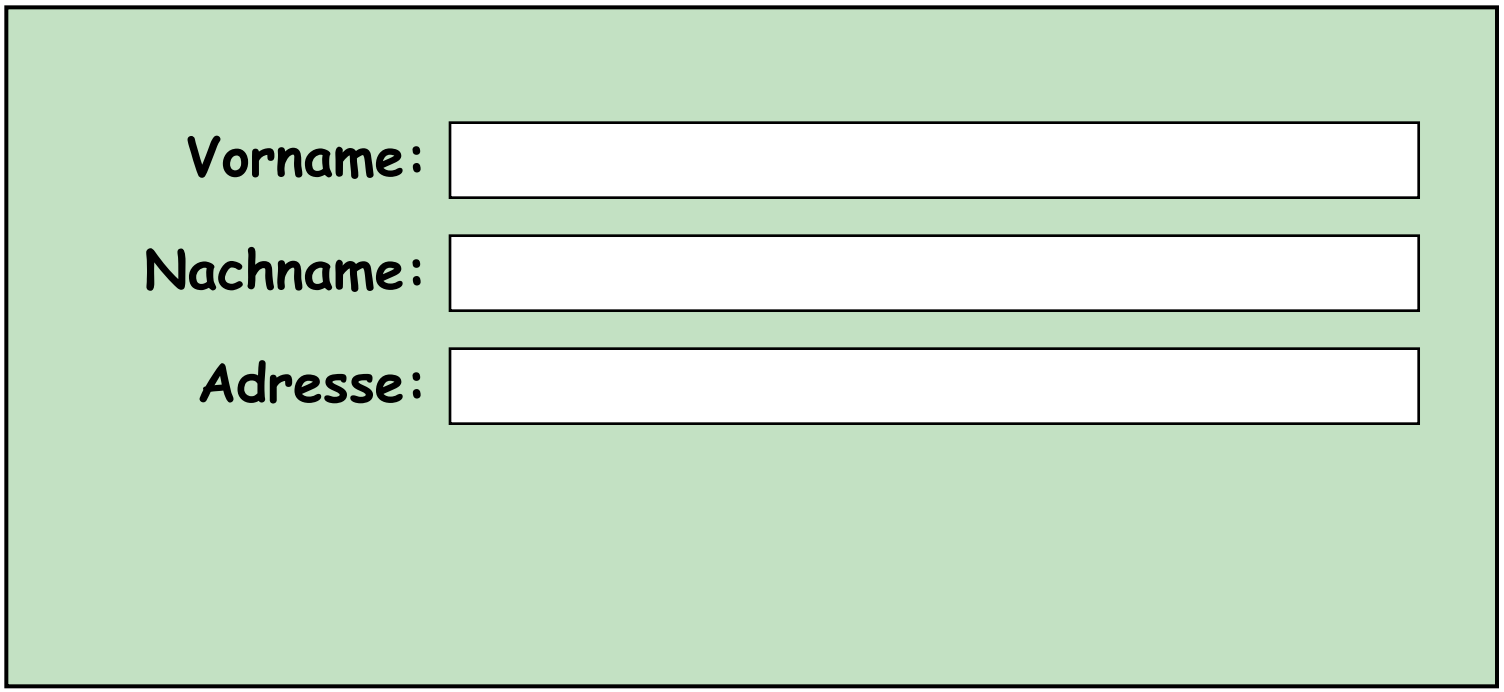
Es war ein WIEDERVERWENDUNGS-Fehler:

- Die Analyse war korrekt – für Ariane 4 !
- Die Annahme wurde dokumentiert – in einem Entwurfsdokument !

Siehe Jean-Marc Jézéquel & Bertrand Meyer, "Design by Contract: The Lessons of Ariane, IEEE *Computer*, January 1997, Link
se.ethz.ch/~meyer/publications/computer/ariane.pdf

Sicherheitsbeispiel: Der Puffer-Überlauf

Das System erwartet eine Eingabe eines externen Benutzers:



Die Eingabe erhalten

```
from i := 1 until
        i > eingabe_länge
loop
        puffer[i] := eingabe[i]
        i := i + 1
end
```

Eine Eigenheit von C

Es ist nicht möglich, eingabe_länge im Voraus zu wissen.

Man muss solange lesen, bis man den String-Terminator, \0 (das null-Zeichen), findet

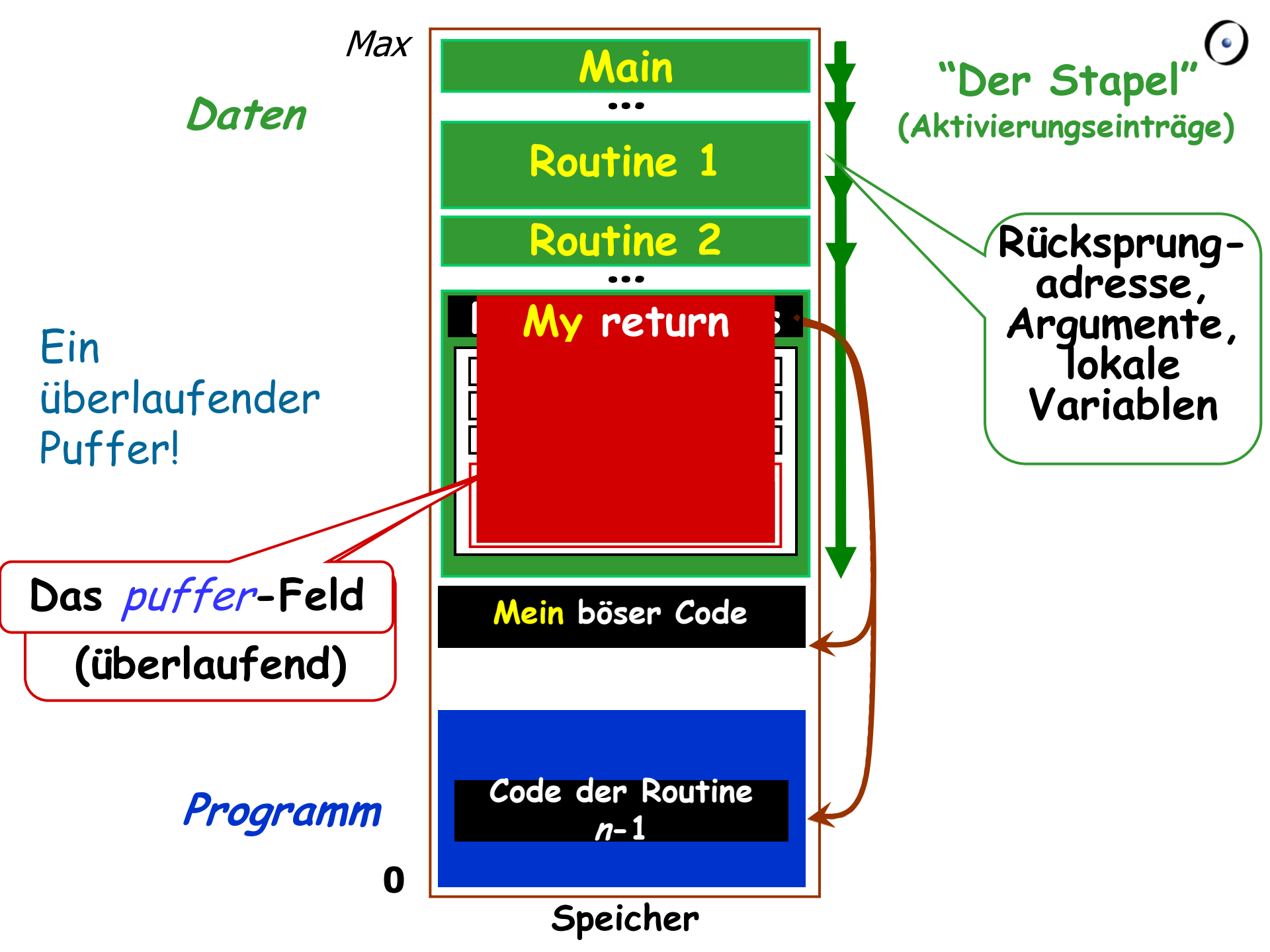
2 Strings besuchen eine Bar. "Was darfs denn sein?" fragt der Barchef.

Der erste String sagt: "Ich hätte gerne ein Bier zdiup tako^jDjftk /. \\134.206.21.02 C#VB.NET 8086%N ~~/~~#@$ Dz @-)))"

"Bitte entschuldigen Sie meinen Freund," sagt der zweite String, "Er ist nicht null-terminiert."

Die Eingabe erhalten

```
from i := 1 until
        i > eingabe_länge
loop
        puffer[i] := eingabe[i]
        i := i + 1
end
```


Max
Daten
Main
...
Routine 1
Routine 2
...
My return
Mein böser Code
Code der Routine n-1
Programm
0
Speicher
"Der Stapel"
(Aktivierungseinträge)
Rücksprung-adresse, Argumente, lokale Variablen
Ein überlaufender Puffer!
Das puffer-Feld
(überlaufend)

Die Eingabe erhalten

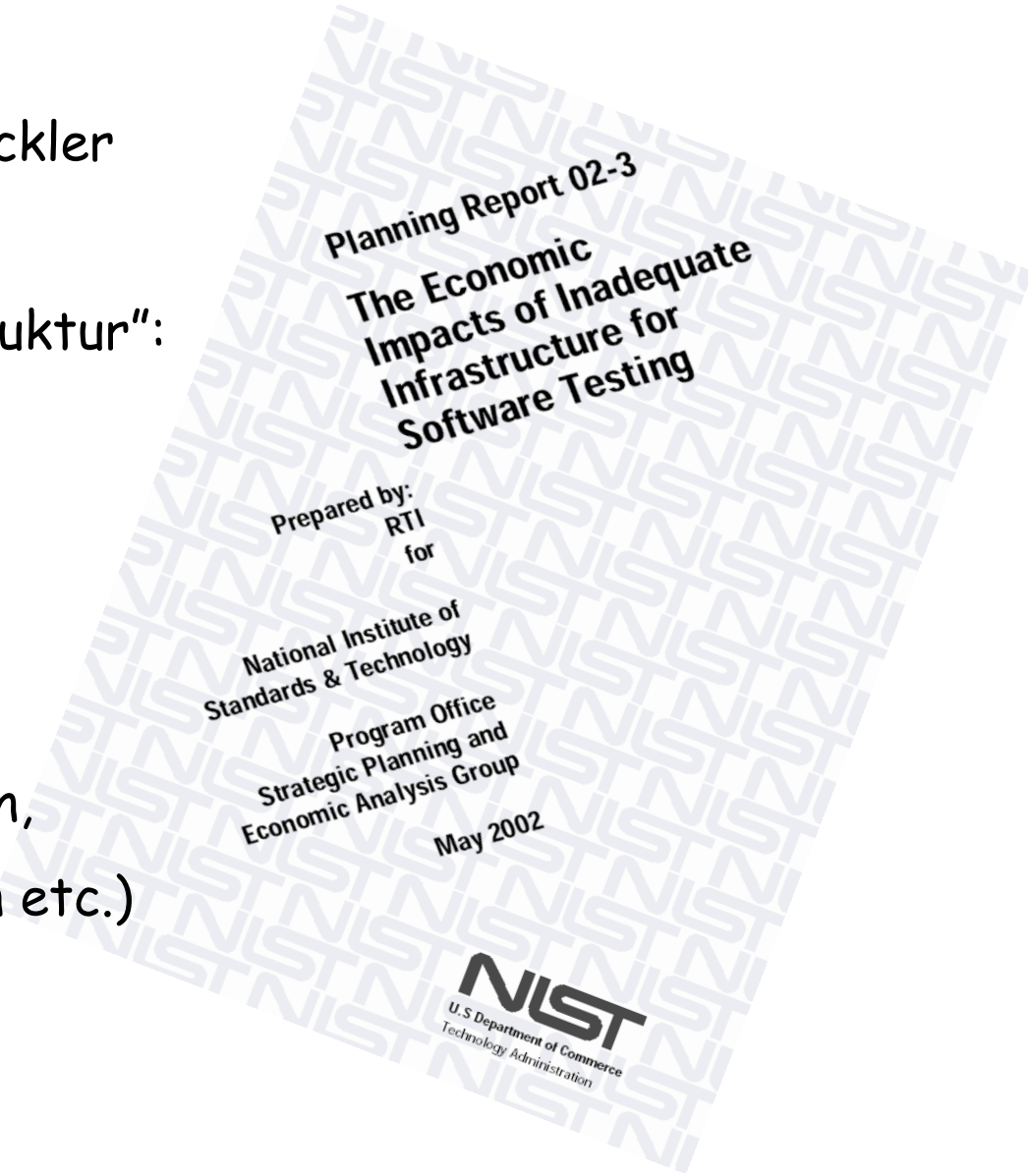
```
from i := 1 until
        i > eingabe_länge or i > puffer_länge
loop
        puffer[i] := eingabe[i]
        i := i + 1
end
```

NIST Report zu “Testen” (Mai 2002)

Finanzieller Effekt für Entwickler und Benutzer aufgrund von “ungenügender Test-Infrastruktur”:

$59.5 Milliarden

(Finanzsektor: $3.3 Milliarden, Auto/Fluzeug: $1.8 Milliarden etc.)



Externe vs. interne Softwarequalität

Externe Faktoren: für den Kunden sichtbar

(Nicht nur Endbenutzer, sondern auch z.B. Käufer)

- *Beispiele*: Benutzerfreundlichkeit, Erweiterbarkeit, Pünktlichkeit

Interne Faktoren: nur für Entwickler ersichtlich

- *Beispiele*: Guter Programmierstil, Geheimnisprinzip

Nur externe Faktoren zählen schlussendlich, aber die internen Faktoren ermöglichen es, diese zu erreichen.

Softwarequalität: Produkt vs Prozess

Produkt: Eigenschaften der resultierenden Software

z.B.: Korrektheit, Effizienz

Prozess: Eigenschaften der Prozeduren, die zur Produktion und Unterhaltung der Software gebraucht werden.

Einige externe Faktoren

Produktqualität (unmittelbar):

- Verlässlichkeit
- Effizienz
- Einfachheit des Gebrauchs
- Einfachheit des Erlernens

Prozessqualität:

- Produktionsgeschwindigkeit (Pünktlichkeit)
- Kosteneffizienz
- Voraussagbarkeit
- Reproduzierbarkeit
- Selbstverbesserung

Produktqualität (langfristig):

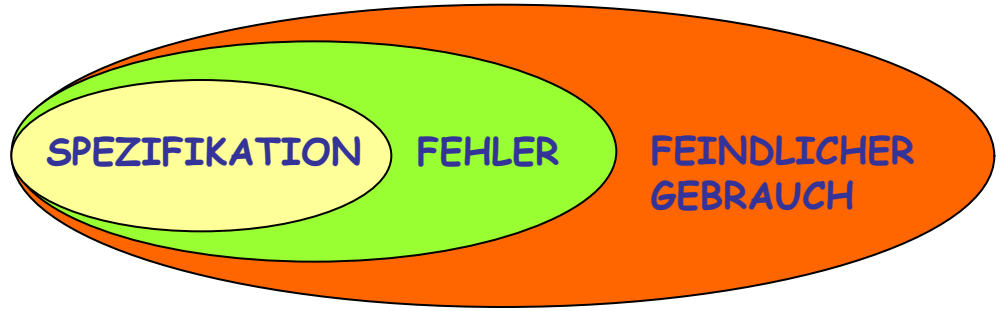
- Erweiterbarkeit
- Wiederverwendbarkeit
- Portabilität

Verlässlichkeit

Korrektheit:
Die Fähigkeit des Systems, in spezifizierten Fällen der Spezifikation entsprechend zu arbeiten.

Robustheit:
Die Fähigkeit des Systems, sich in nicht spezifizierten Fällen angemessen zu verhalten

Sicherheit:
Die Fähigkeit des Systems, sich und seine Daten gegen feindlichen Gebrauch zu schützen



Korrektheit

Robustheit

Sicherheit

Einige interne Faktoren

Modularität

Einhaltung von Stilregeln

Konsistenz

Struktur

...

Externe Faktoren

Produktqualität (unmittelbar):

- Verlässlichkeit
- Effizienz
- Einfachheit des Gebrauchs
- Einfachheit des Erlernens

Prozessqualität:

- Produktionsgeschwindigkeit (Pünktlichkeit)
- Kosteneffizienz
- Voraussagbarkeit
- Reproduzierbarkeit
- Selbstverbesserung

Produktqualität (langfristig):

- Erweiterbarkeit
- Wiederverwendbarkeit
- Portabilität

Software-Aufgaben

Anforderungsanalyse
Spezifikation
Entwurf
Implementation
Validierung und Verifizierung (V&V)
Management
Planen und abschätzen
Messen

Anforderungsanalyse

Die Bedürfnisse der Benutzer verstehen

Die Bedingungen an das System verstehen

- Interne Bedingungen: Klasseninvarianten
- Externe Bedingungen

Aussage zu Anforderungen: Brooks

Quelle*: Brooks 87

The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

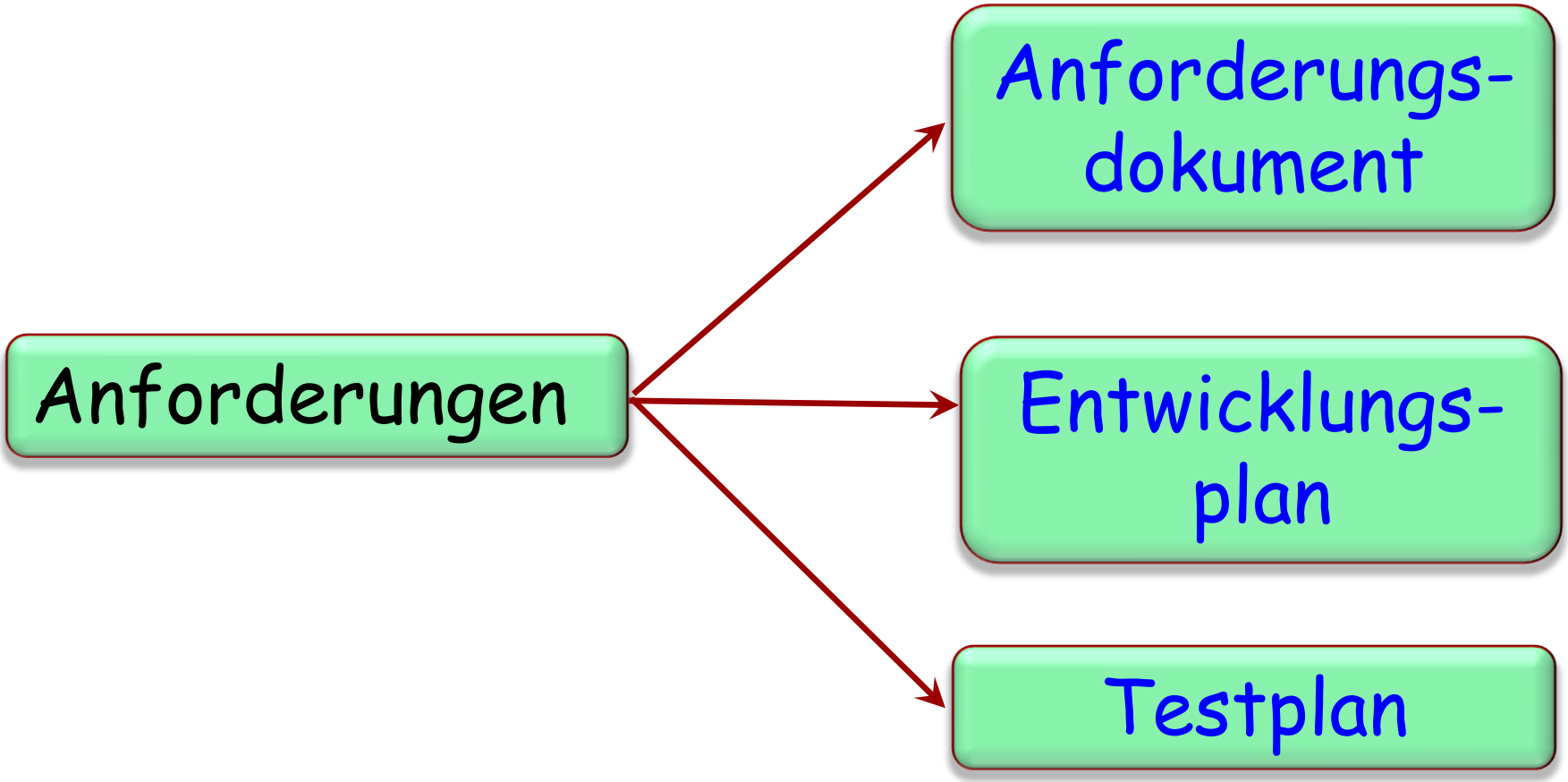
*Siehe Literaturverzeicnis für zitierte Quellen

Ziele einer Anforderungs-Analyse

Quelle: OOSC

- Verstehen Sie das Problem oder die Probleme, die das fertige Softwaresystem lösen soll
- Stellen Sie Fragen über das Problem und das System
- Stellen Sie eine Grundlage zur Verfügung, um Fragen zu spezifischen Eigenschaften des Problems oder Systems zu beantworten
- Entscheiden Sie, was das System tun soll
- Entscheiden Sie, was das System nicht tun soll
- Stellen Sie sicher, dass das System die Bedürfnisse der Akteure befriedigt.
- Stellen Sie eine Grundlage zur Entwicklung des Systems zur Verfügung
- Stellen Sie eine Grundlage für Validierung und Verifikation des Systems zur Verfügung

Produkte von Anforderungen



15 Qualitätsziele für Anforderungen

- Gerechtfertigt
- Korrekt
- Komplett
- Konsistent
- Eindeutig
- Machbar
- Abstrakt
- Verfolgbar

- Begrenzt
- Gekoppelt
- Lesbar
- Modifizierbar
- Verifizierbar
- Priorisiert
- Bestätigt

Schwierigkeiten von Anforderungen

- Natürliche Sprachen und ihre fehlende Präzision
- Formale Techniken und ihre Abstraktion
- Benutzer und ihre Vagheit
- Kunden und ihre Ansprüche
- Der Rest der Welt und seine Komplexität

Schlechte Anforderungen

Quelle: Wiegers

The Background Task Manager shall provide status messages at regular intervals not less than 60 seconds.

Besser:

The Background Task Manager (BTM) shall display status messages in a designated area of the user interface

1. The messages shall be updated every 60 plus or minus 10 seconds after background task processing begins.
2. The messages shall remain visible continuously.
3. Whenever communication with the background task process is possible, the BTM shall display the percent completed of the background task.

Schlechte Anforderungen

Quelle: Wiegers

Der Hintergrund Task-Manager wird Statusmeldungen in regelmäßigen Zeitspannen von nicht weniger als 60 Sekunden anzeigen

Besser:

Der Hintergrund Task-Manager (HTM) wird Statusmeldungen in einem bestimmten Bereich der Benutzeroberfläche anzeigen

1. Die Meldungen werden alle 60 plus-minus 10 Sekunden nach dem Anfang der Hintergrund-Verarbeitung aktualisiert.
2. Die Meldungen müssen kontinuierlich sichtbar bleiben.
3. Wenn die Kommunikation mit dem Hintergrund-Task Prozess möglich ist, wird der HTM der abgeschlossene Anteil des Hintergrund-Task anzeigen.

Schlechte Anforderungen

Quelle: Wiegers

The XML Editor shall switch between displaying and hiding non-printing characters instantaneously.

Besser:

The user shall be able to toggle between displaying and hiding all XML tags in the document being edited with the activation of a specific triggering mechanism. The display shall change in 0.1 second or less.

Schlechte Anforderungen

Quelle: Wiegers

The XML parser shall produce a markup error report that allows quick resolution of errors when used by XML novices.

Besser:

1. After the XML Parser has completely parsed a file, it shall produce an error report that contains the line number and text of any XML errors found in the parsed file and a description of each error found.
2. If no parsing errors are found, the parser shall not produce an error report.

Schlechte Anforderungen

Quelle: Wiegers

Der XML-Parser wird einen Bericht erstellen, der eine schnelle Lösung von Markup-Fehlern erlaubt, wenn der Benutzer mit XML unerfahren ist.

Besser:

1. Nachdem der XML-Parser eine Datei völlig analysiert hat, wird er einen Fehlerbericht anzeigen, der die Zeilennummer, den Wortlaut und eine Beschreibung der einzelnen XML-Fehler enthält, die in der bearbeiteten Datei gefunden wurden.
2. Wenn keine Parsing-Fehler gefunden wurden, wird der Parser keinen Fehlerbericht anzeigen.

Verifizierbare Anforderungen

Angepasst von: IEEE

Nicht verifizierbar:

- Das System soll zufriedenstellend arbeiten
- Die Schnittstelle soll benutzerfreundlich sein
- Das System soll in Echtzeit reagieren

Verifizierbar:

- Die Ausgabe soll in allen Fällen innerhalb von 30 Sekunden nach dem Eingabeereignis ersichtlich sein. Sie soll in 80% der Eingabefälle innerhalb von 10 Sekunden erscheinen.
- Professionelle Zugführer erreichen Level 1-Professionalität (*in den Anforderungen definiert*) nach zwei Trainingstagen.

Praktischer Tipp

Lieber präzise, falsifizierbare Sprache als angenehme Allgemeinheiten

IEEE 830-1998

"IEEE Recommended Practice for Software Requirements Specifications"

Zugestimmt am 25. Juni 1998 (Eine Revision eines früheren Standards)

Beschreibung des Inhalts und der Qualitäten einer guten Software-Anforderungsspezifikation (SRS)

IEEE Standard

Vorgeschlagene Struktur des Dokumentes

1. Introduction

- 1.1 Purpose
- 1.2 Scope
- 1.3 Definitions, acronyms, and abbreviations ← Glossary!
- 1.4 References
- 1.5 Overview

2. Overall description

- 2.1 Product perspective
- 2.2 Product functions
- 2.3 User characteristics
- 2.4 Constraints
- 2.5 Assumptions and dependencies

3. Specific requirements

Appendixes

Index

Praktischer Tipp

Benutzen Sie die vorgeschlagene IEEE-Struktur

Praktischer Tipp:

Schreiben Sie ein Glossar

Ein paar Rezepte für gute Anforderungen

Management-Aspekte:

- Alle Akteure involvieren
- Vorgehensweise für kontrollierte Änderungen entwerfen
- Mechanismen für die Rückverfolgbarkeit etablieren
- Behandeln Sie das Anforderungsdokument als eines der grössten Posten des Projektes; Fokus auf Klarheit, Präzision, Vollständigkeit

Technische Aspekte: Wie Präzision erlangen?

- Formale Methoden?
- Design by Contract

Validierung & Verifikation

Verifikation: überprüft interne Konsistenz

Beispiele: Überprüfung der Typen; Uberprüfung, dass die Ausführung kein Crash hervorbringt

"Überprüfen, dass wir das System richtig entwickelt haben" (nach allen Regeln)

Validierung: Überprüfung auf Beschreibungen in einer höheren Ebene

Beispiel: Validierung eines Programmes gegen seine Spezifikation

"Überprüfen, dass wir das richtige System entwickelt haben"

(Bedürfnisse der Benutzer befriedigt)

Modelle der Lebenszyklen von Software

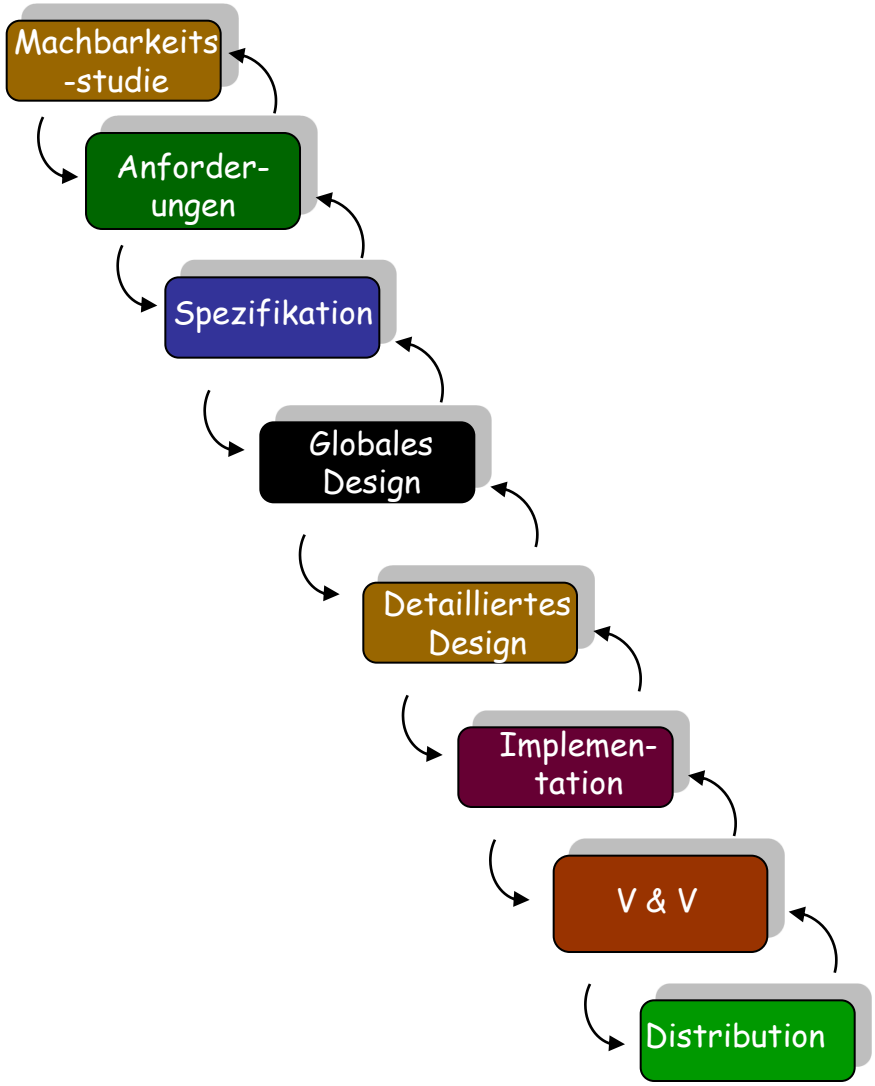
Beschreiben Sie eine allgemeine Verteilung der Softwarekonstruktion in Aufgaben und Reihenfolgen dieser Aufgaben.

Diese sind Modelle in zwei Arten:

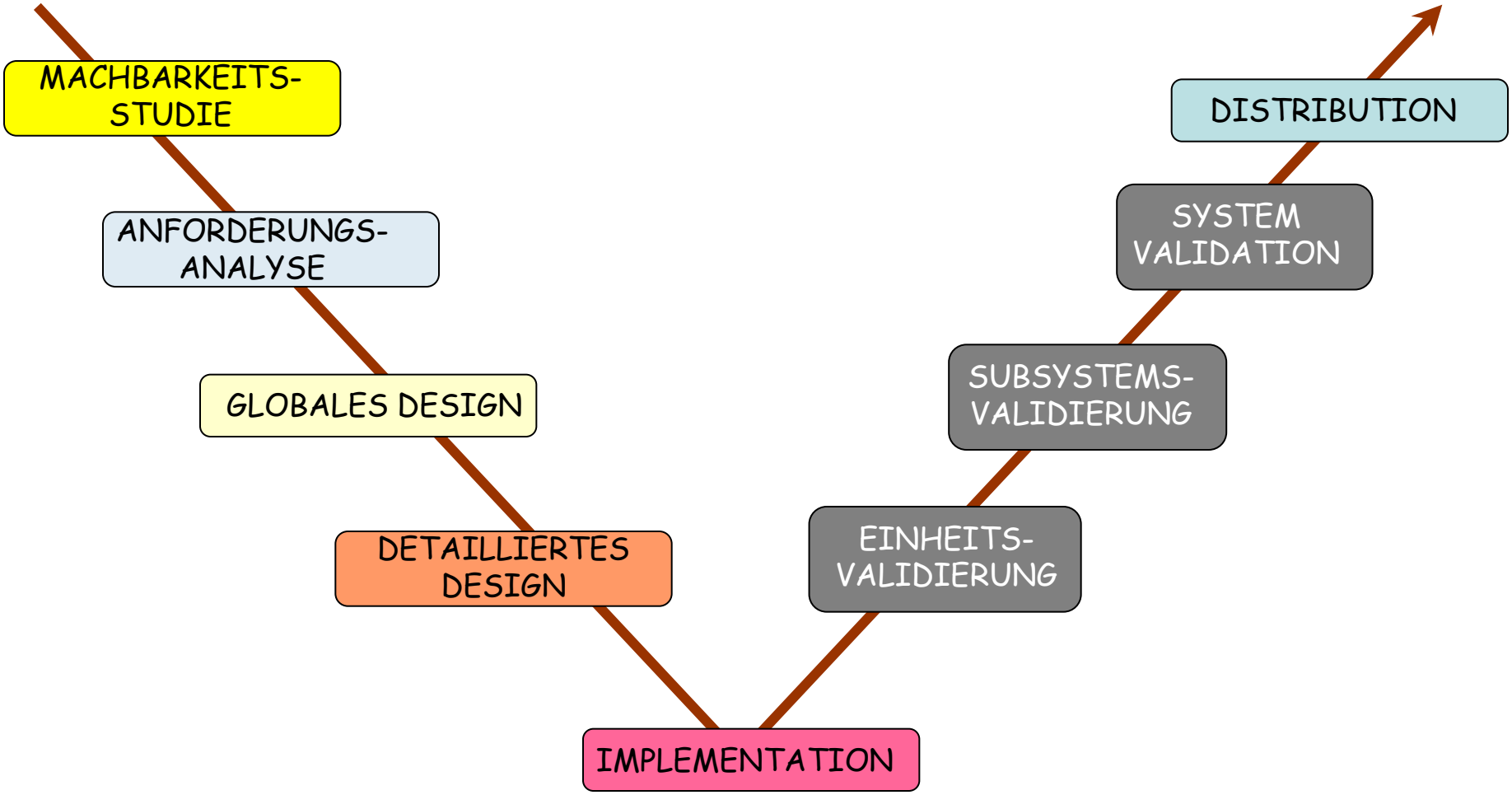
- Eine abstrahierte Version der Realität

- Beschreibung eines idealen Systems, in der Praxis nicht immer eingehalten.

Das Wasserfall-Modell



V-Form

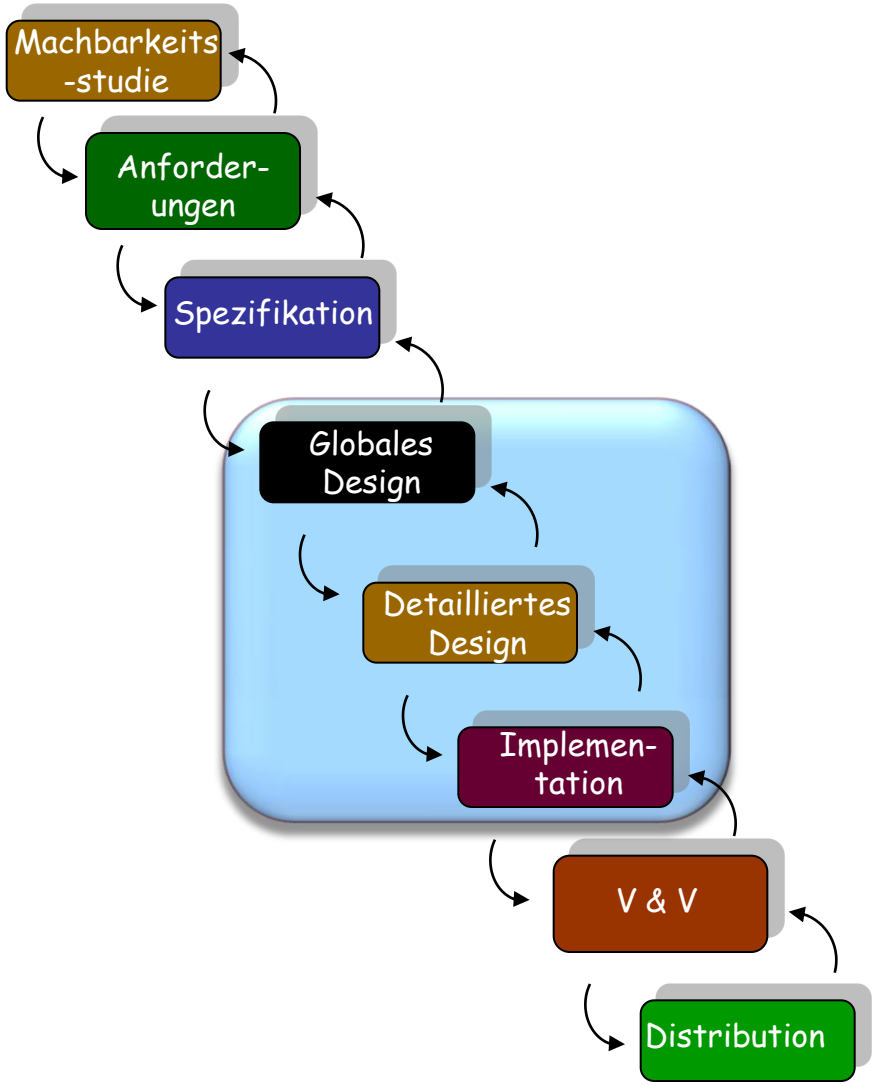

MACHBARKEITS-
STUDIE
ANFORDERUNGS-
ANALYSE
GLOBALES DESIGN
DETAILLIERTES
DESIGN
IMPLEMENTATION
EINHEITS-
VALIDIERUNG
SUBSYSTEMS-
VALIDIERUNG
SYSTEM
VALIDATION
DISTRIBUTION

Argumente für den Wasserfall

(nach B.W. Boehm: *Software engineering economics*)

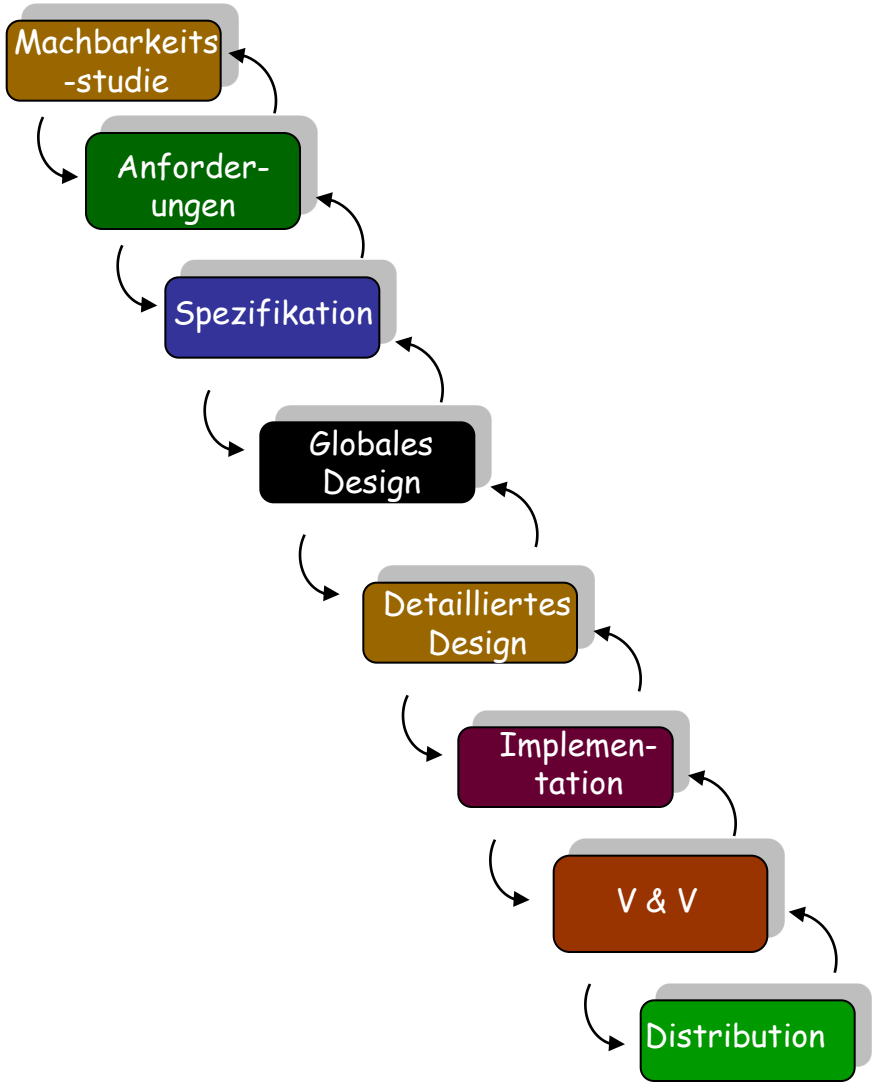
- Die Aktivitäten sind nötig
  - (Aber: Verschmelzen von mittleren Aktivitäten)
- Die Reihenfolge ist die Richtige.

Verschmelzen von mittleren Aktivitäten

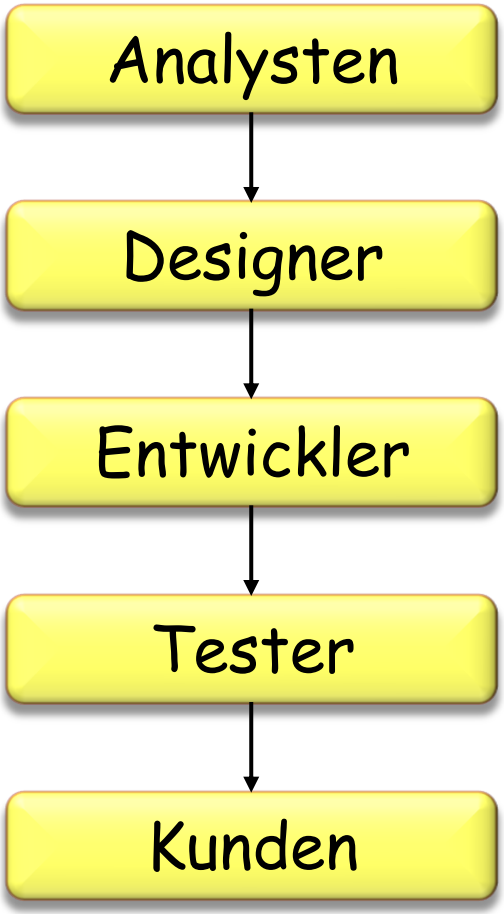


Probleme mit dem Wasserfall

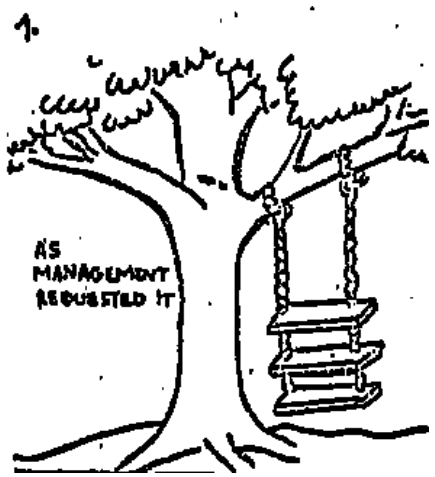
- Richtiger Code erst spät.
- Keine Unterstützung für Änderungen der Anforderungen – und allgemeiner für Erweiterbarkeit und Wiederverwendbarkeit
- Keine Unterstützung für Unterhaltungsaktivitäten (70% der Softwarekosten?)
- Unterteilung der Arbeit behindert „Total Quality Management“
- Sehr synchrones Modell



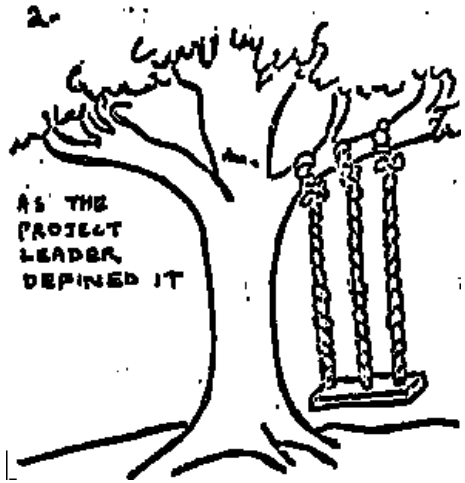
Qualitätskontrolle?



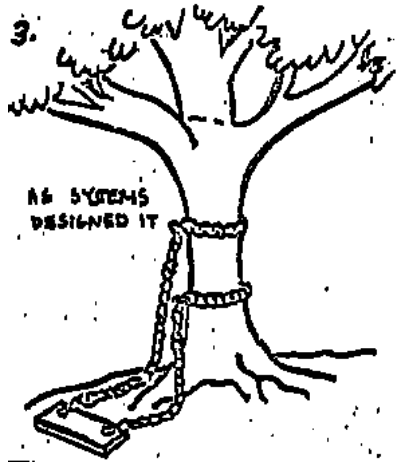
Lebenszyklus: “impedance mismatches”



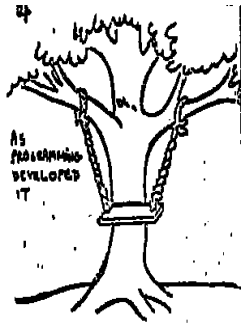
Von Management angefordert



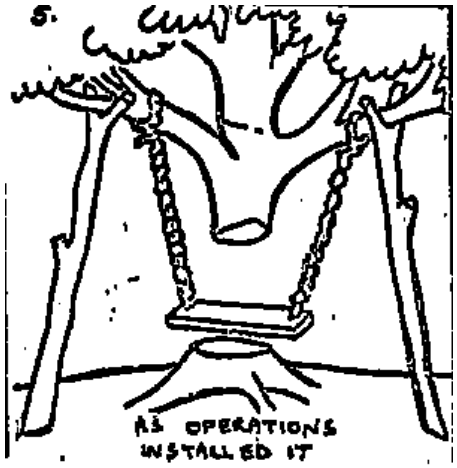
Von dem Projektleiter definiert



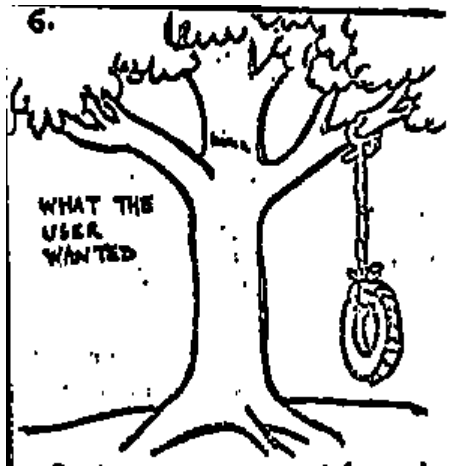
Von Systems gestaltet



Von Programming implementiert



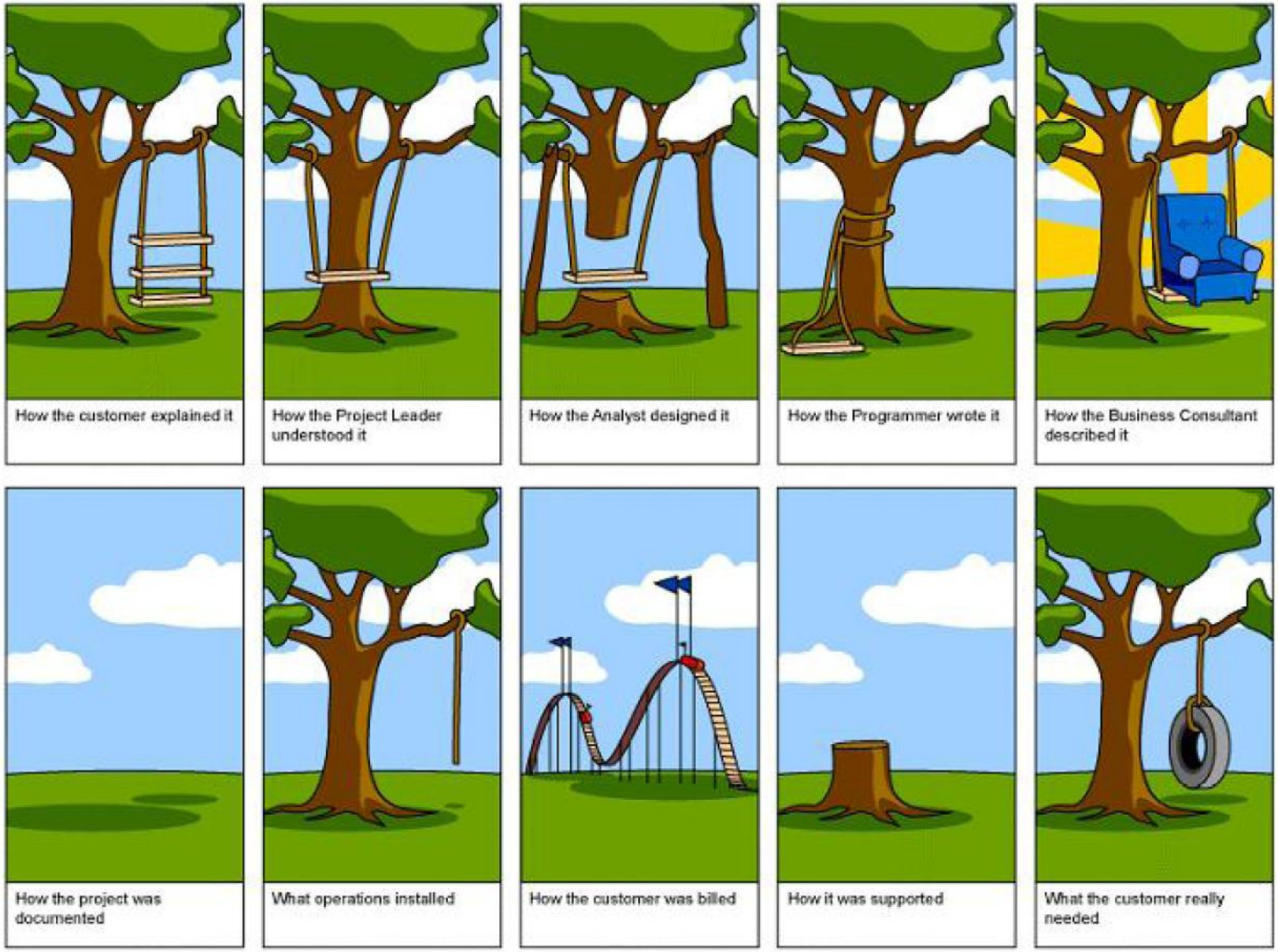
Von Operations installiert



Was der Benutzer wollte

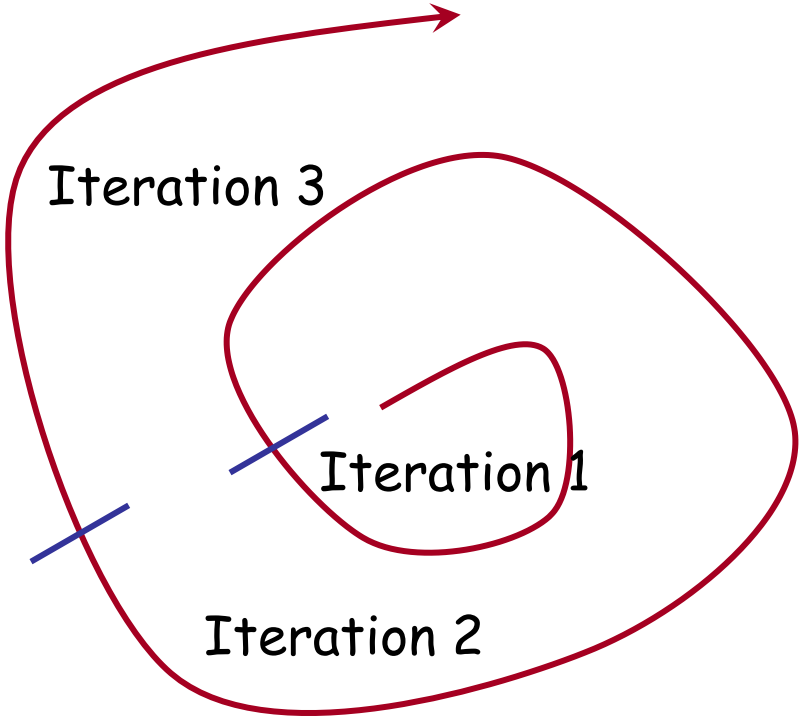
(Pre-1970 cartoon; Quelle unbekannt)

Eine modernere Variante



Das Spiralenmodell (Boehm)

Ein Wasserfall-ähnlicher Ansatz auf aufeinanderfolgende Prototypen



Das Spiralenmodell

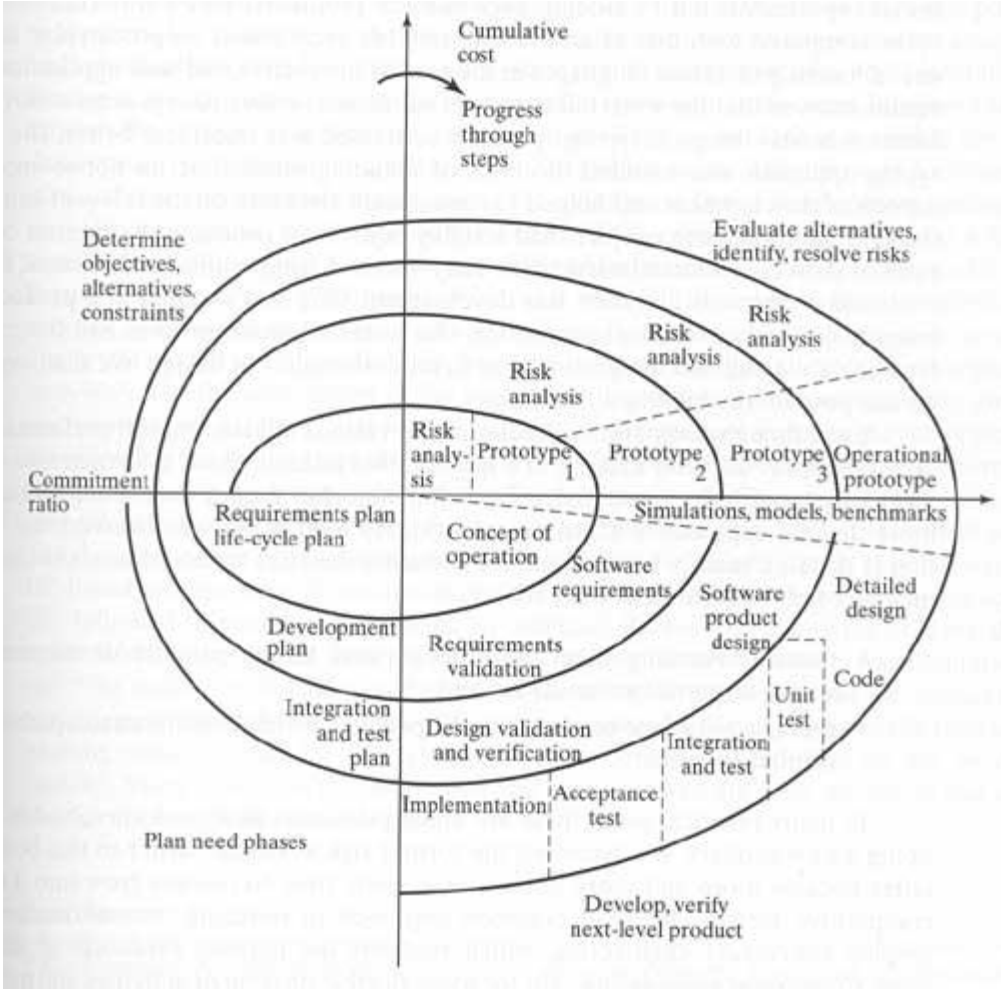


Abbildung aus: Ghezzi, Jazayeri, Mandrioli, *Software Engineering*, 2nd edition, Prentice Hall

“Prototyping” in Software

Der Begriff hat mehrere Bedeutungen

- Experimentieren:
  - Erfassen der Anforderungen
  - Spezifische Techniken ausprobieren: GUI, Implementation (“Informationen kaufen”)
- Pilotprojekt
- Schrittweise Entwicklung
- Wegwerf-Entwicklung

(Fred Brooks, *The Mythical Man-Month*: “Plan to throw one away, you will anyhow”).

Das Problem mit der Wegwerf-Entwicklung

Software-Entwicklung ist schwierig, da man Konflikte in Kriterien (z.B. Protabilität und Effizienz) abstimmen muss.

Ein Prototyp opfert meist einige dieser Kriterien

Risiko, den Prototyp auszuschiffen

Nahtlose, schrittweise Entwicklung

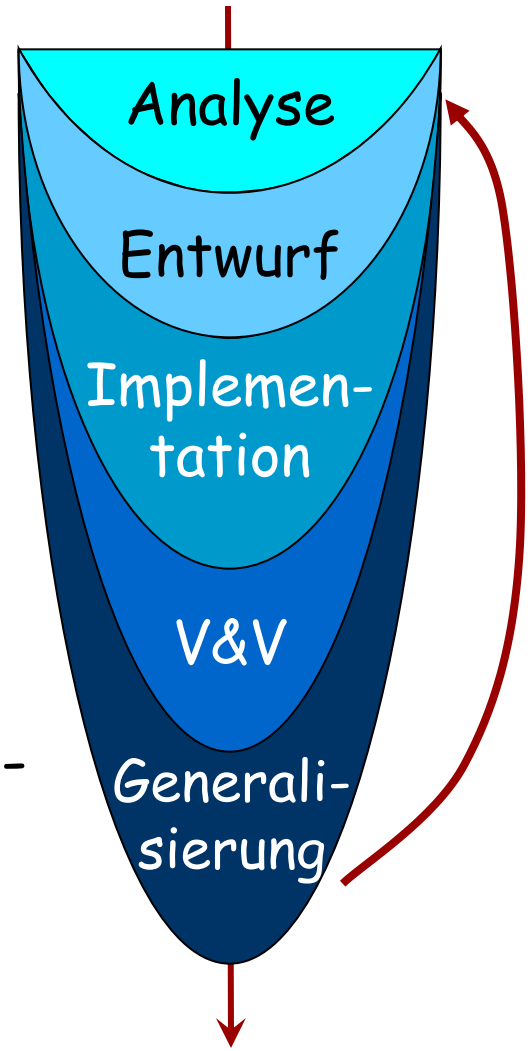
Die Eiffel-Sicht

- Nur eine einzige Garnitur von Notation, Werkzeugen, Konzepte, Prinzipien
- Beständige, schrittweise Entwicklung
- Modell, Implementation und Dokumentation konsistent halten

Umkehrbarkeit: Man kann vor- und zurückgehen

Nahtlose Entwicklung

- Eine Notation, Werkzeuge, Konzepte, Prinzipien
- Beständige, schrittweise Entwicklung
- Modell, Implementation und Dokumentation konsistent halten
- Umkehrbarkeit: Man kann vor- und zurückgehen



Beispielklassen:

PLANE, ACCOUNT, TRANSACTION…

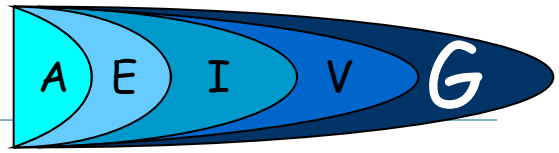
STATE, COMMAND…

HASH_TABLE…

TEST_DRIVER…

TABLE…

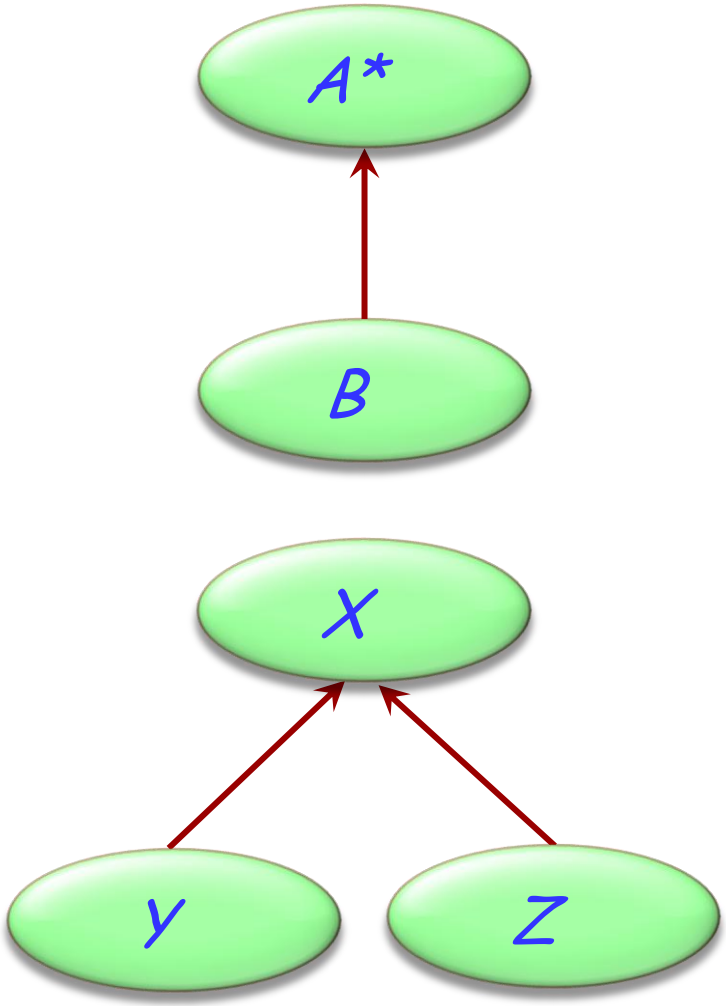
Generalisierung



Für Wiederbenutzung vorbereiten.
Zum Beispiel:

- Eingebaute Grenzen entfernen
- Abhängigkeiten von den Details des Projektes entfernen
- Dokumentation, Verträge verbessern…
- Abstrakt
- Ähnlichkeiten extrahieren und Vererbungshierarchie ausbessern

Nur wenige Firmen haben das Budget für diesen Ansatz

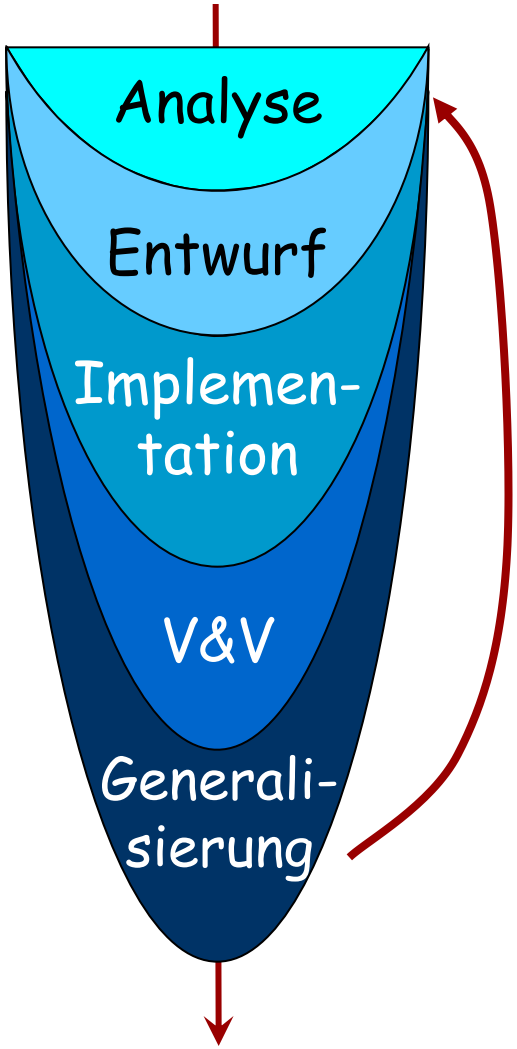


Antoine de Saint-Exupéry

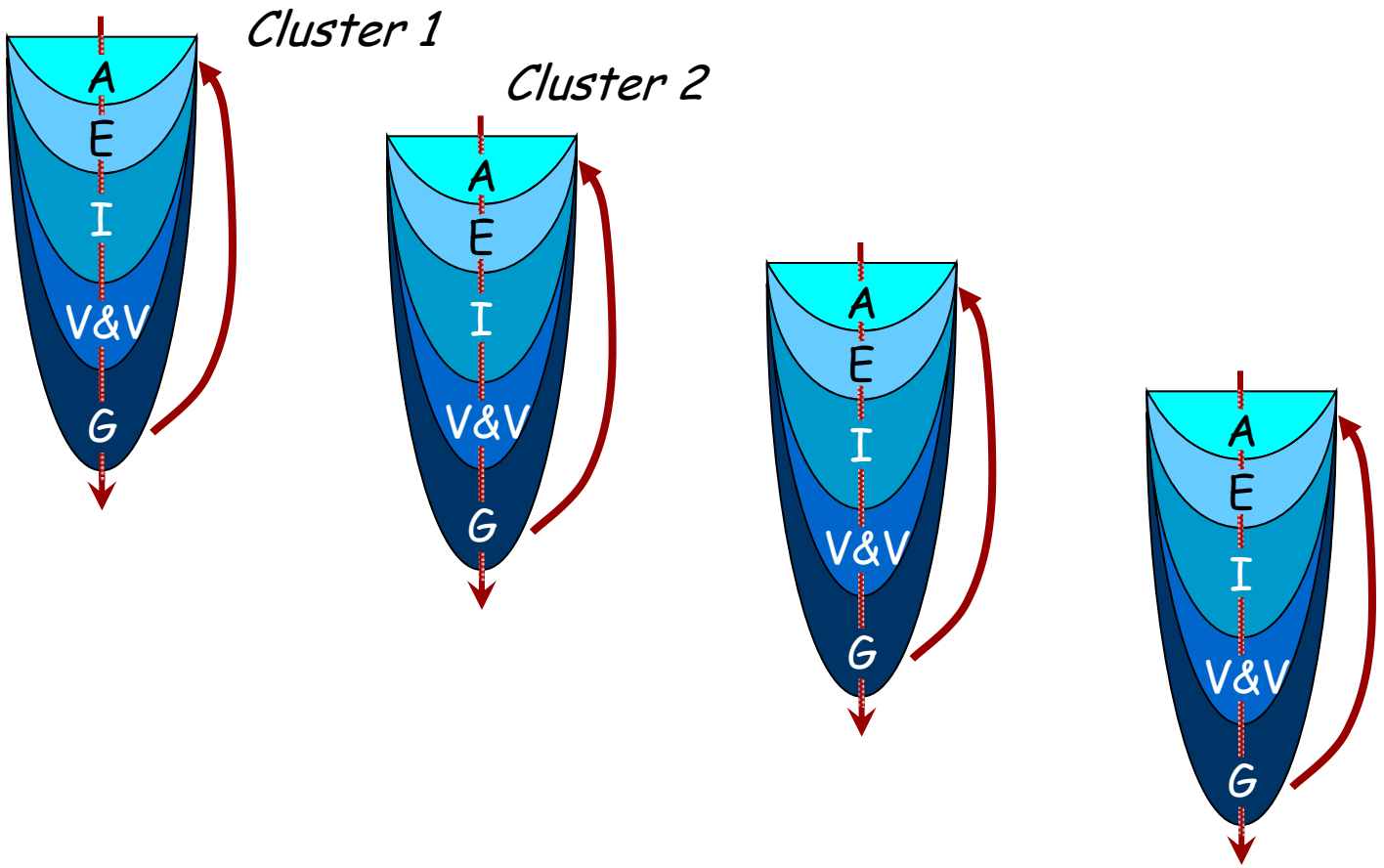
Es scheint folglich, dass die Arbeit der Ingenieure, der Designer und der Kalkulatoren in den Konstruktionsbüros darin besteht, zu verwischen und zu polieren, jenes Verbindungsstück leichter zu machen, diesen Flügel auszubalancieren bis man ihn nicht mehr wahrnimmt, bis er nicht mehr ein am Rumpf befestigter Flügel ist, sondern eine perfekt abgestimmte Form, die endlich freigelegt wurde von seiner Schicht, gleich einem von Geisterhand zusammengehaltenen Ganzen und von derselben Beschaffenheit wie die eines Gedichts. Es scheint, dass Vollkommenheit nicht erreicht ist, wenn es nichts mehr hinzuzufügen gibt, sondern wenn es nichts mehr wegzulassen gibt. Am Ende ihrer Entwicklung, versteckt sich die Maschine selbst.

(Terre des Hommes, 1937)

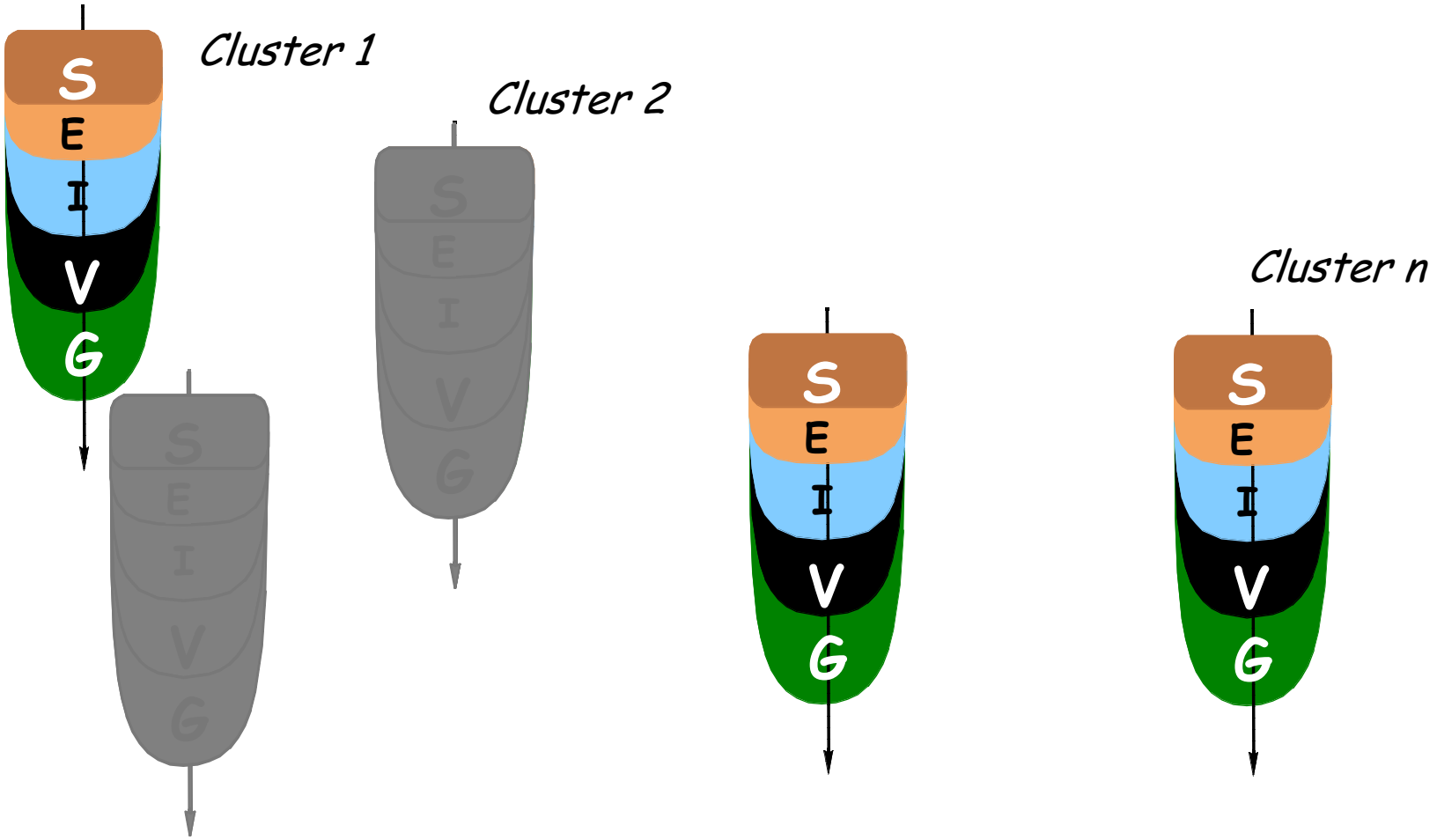
Umkehrbarkeit



Das Cluster-Modell



Das Cluster-Modell



Nahtlose Entwicklung am Beispiel von Eiffel

Diagrammwerkzeug

- Systemdiagramme können automatisch aus dem Softwaretext produziert werden.
- Funktioniert auch umgekehrt: Diagramme oder Text aktualisieren – Die andere Sicht wird ebenfalls sofort aktualisiert.

Kein Bedarf nach separaten UML-Werkzeugen

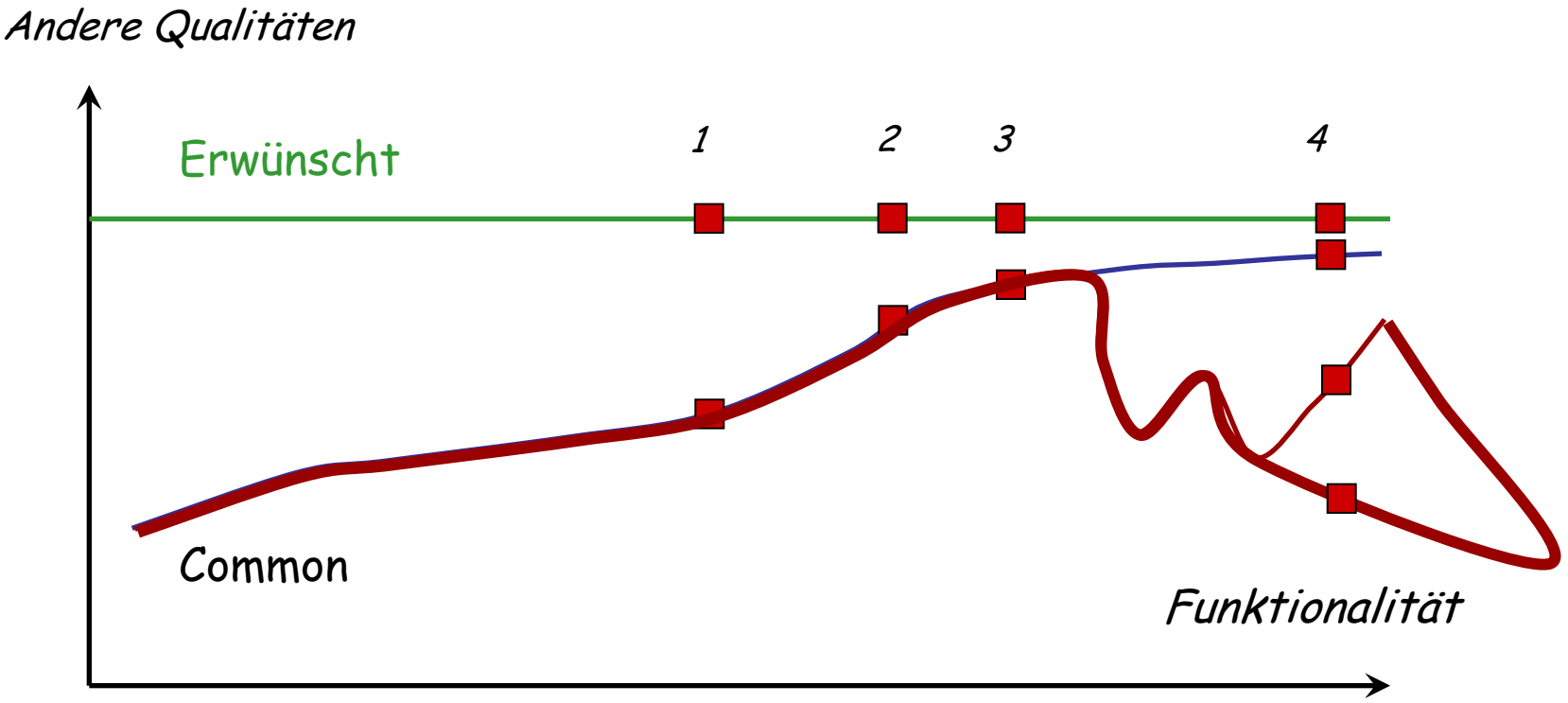
Metrisches Werkzeug

Profiler-Werkzeug

Werkzeug zur Generierung einer Dokumentation

...

Qualitätsziele: Die Osmond-Kurve



- Vorausgesehen
- Abgabe

Tipp: Hinzufügen von Funktionalität mit konstanter Qualität

Agile/schlanke Methoden und „extreme Programming“

Weniger Wert auf formale Prozesse

Betonung von kurz-zyklischer, zeitlich begrenzter, iterativer Entwicklung

Mehr Wert auf Testen zur Leitung der Entwicklung
("TDD", Test-Driven Development)

Nutzen eines zweiten Augenpaars:
Paarprogrammierung

Betonung der Rolle des Refactoring

Selbst-organisierte Teams

Mehr Wert auf die Miteinbeziehung der Kunden

Open-source Prozesse

Kollaborative, verteilte Entwicklungen

Konzentrische Vertrauenskreise

Erfolg mit starkem Projektleiter (z.B. Linux)

"Mit genügend Augen sind alle Bugs sichtbar"

Validierung und Verifikation

Nicht nur Testen:

- Statische Analysie-Werkzeuge durchsuchen den Code nach möglichen Schwächen, z.B. uninitialisierte Variablen
- Korrektheitsbeweise werden immer realistischer
- Die Überprüfung des Modells erkundet den Zustandsraum einer abstrahierten Version des Programmes

Qualitätssicherung sollte während des ganzen Prozesses durchgeführt werden, nicht nur am Ende

Werkzeuge des Software-Engineering

Entwicklungsumgebungen (Compiler, Browser, Debugger, …): "IDE"

Dokumentationswerkzeuge

Werkzeuge, um Anforderungen zu sammeln

Analyse- und Entwurfswerkzeuge

Konfigurations- & Versionsmanagement (CVS, Source Safe…) (auch "make" etc.)

Formale Entwicklungs- und Beweiswerkzeuge

Integrierte CASE (Computer-Aided Software Engineering) -Umgebungen

Konfigurationsmanagement

Ziel: Sicherstellen, dass die Versionen, die für verschiedene Komponenten des Systems gebraucht werden, kompatibel sind

Zwei prinzipielle Varianten:

- Build-Management
- Versionsmanagement

Build-Management

Make (späte 70er Jahre): automatische Rekonstruktion eines Systems von einem "makefile", das Abhängigkeiten auflistet

Beispiel

```
make program
```

Mit dem makefile

```
program: main.o module1.o module2.o
        cc main.o module1.o module2.o
%.c: %.o
        cc $<
```

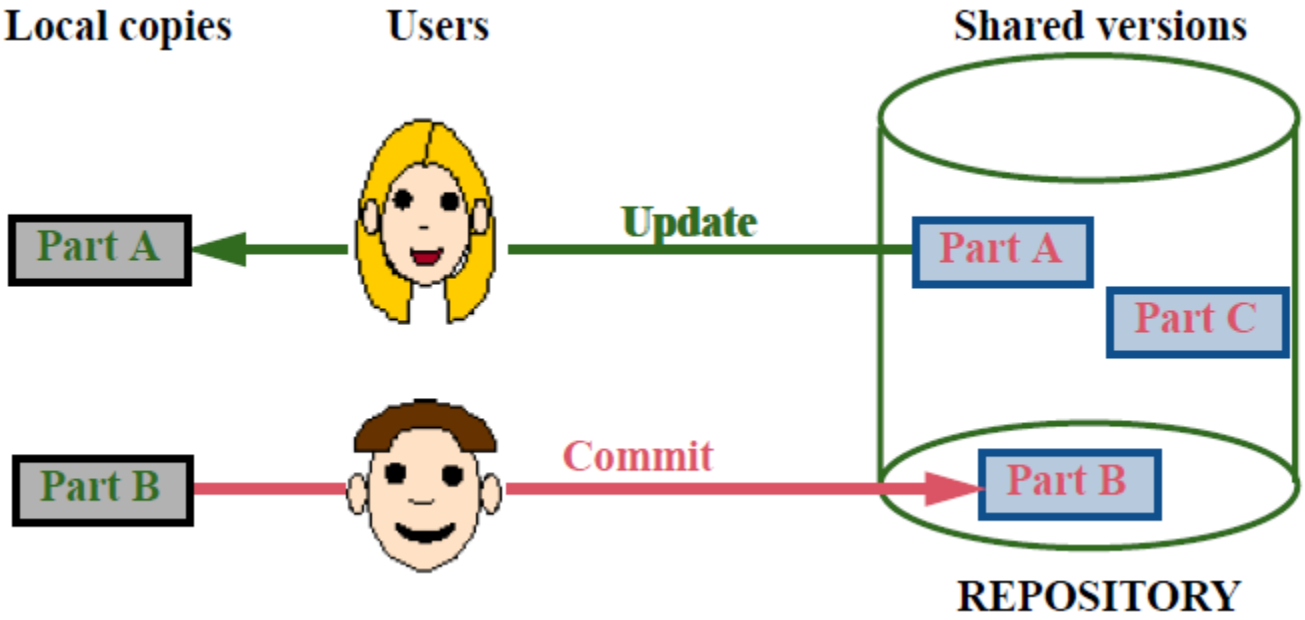
Grösste Limitierung: Die Abhängigkeiten müssen manuell erfasst werden

Versionsmanagement

Beispiele:

RCS,

Subversion



Haupt-operationen:

- Commit
- Update

Speichert die "diffs" zwischen Versionen

Tipp: Verzweigungen vermeiden; früh und oft abgleichen

Konfigurationsmanagement

Diese Werkzeuge sind erhältlich und einfach zu gebrauchen

Kein Projekt kann es sich leisten, diese nicht zu gebrauchen

Formale Methoden

Mathematik als Basis für die Softwareentwicklung

Ein Software-System wird als mathematische Theorie betrachtet und stufenweise verbessert, bis es direkt implementierbar ist.

Jede Variante der Theorie und jeder Verbesserungsschritt ist bewiesen.

Beweise werden durch rechnergestützte Werkzeuge unterstützt.

Beispiel: *Atelier B*, das Sicherheitssystem der neusten Metrolinie in Paris

Metrik

Dinge, die gemessen werden müssen:

Produktattribute: Anzahl Codezeilen, Anzahl Klassen, Komplexität der Kontrollstrukturen ("zyklomatische Zahl"), Komplexität und Tiefe der Vererbungsstruktur, Präsenz von Verträgen…

Projektattribute: Anzahl Personen, Kosten, Zeit bis zur Fertigstellung, Zeit von verschiedenen Aktivitäten (Analyse, Entwurf, Implementation, V&V, etc.)

„Taking good measurements helps take good measures"

Kostenmodelle

Versuch, die Kosten einer Softwareentwicklung vor dem Projekt abzuschätzen, gestützt auf Parameter

Beispiel: *COCOMO* (Constructive Cost Model), Barry Boehm

L: 1000 * Delivered Source Instructions (KDSI)

Programmtyp	*Aufwand (pm)*	*Zeit*
Applikation	2.4 * L * 1.05	2.5 * *pm* * 0.38
Utility	3.0 * L * 1.12	2.5 * *pm* * 0.35
System	3.6 * L * 1.20	2.5 * *pm* * 0.32

Modelle der Software-Verlässlichkeit

Anzahl der Bugs abschätzen durch

- Charakteristiken eines Programmes
- Anzahl bisher gefundener Bugs

Variante: "Fault injection"

Projektmanagement

Teamspezialisierungen: Kundenrelationen, Analyst, Designer, Entwickler, Tester, Manager, Dokumentierer...

Welche Rolle hat der Manager: nur führend, oder auch technisch?

"Chief Programmer teams"

Schlussendlich ist es Code

Unterschätzen Sie nicht die Rolle von Werkzeugen, Sprachen, oder allgemeiner: Technologien

Ein schlechtes Management tötet Projekte
Gute Technologien machen ein Projekt erfolgreich

Programmiersprachen

Nicht nur, um mit Ihrem Computer zu sprechen!

Eine Programmiersprache ist ein Denkwerkzeug

Ein bisschen Geschichte

"Plankalkül", Konrad Zuse, 1940er

Fortran (FORmula TRANSlator), John Backus, IBM, 1954

Algol, 1958/1960

Einige Zeilen FORTRAN

```
        I = 0
        SUM = 0
100     I = I + 1
        READ ("I6") N
        IF (N) 150, 170, 160
150     A (I) = A (I) ** 2
        GOTO 100
C       THE NEXT ONE IS THE TOUGH CASE
160     A (I) = A (I) + 1
        GOTO 100
170     DO 200 I=1,N
        SUM = SUM + A (I)
200     CONTINUE
        END
```

Algol

Internationales Komitee, Europäer und Amerikaner, geführt von IFIP. Algol 58, Algol 60.

Beeinflusst von (und eine Reaktion auf) FORTRAN; ebenfalls beeinflusst von LISP (gleich). Rekursive Prozeduren, dynamische Felder, Blockstrukturen, dynamisch alloziierte Variablen

Neuer Mechanismus zur Sprachenbeschreibung: BNF (für Algol 60).

Algol W und Pascal

Nachfahren von Algol 60, entworfen von Niklaus Wirth an der ETH

Algol W führte den Struktursatz ein

Pascal legt Wert auf Einfachheit, Datenstrukturen (Strukturen, Zeiger). Kleine Sprache, oft zu Lernzwecken verwendet.

Half, die PC-Revolution auszulösen (durch Turbo Pascal von Borland (Philippe Kahn))

C

1968: Brian Kernighan und Dennis Richie, AT&T Bell Labs

Zu Beginn eng mit Unix verbunden

Betonung auf Maschinenzugriff auf tiefer Ebene: Zeiger, Adressarithmetik, Umwandlungen

Von der Industrie in den 80ern und 90ern schnell aufgenommen

Lisp und funktionale Sprachen

LISt Processing, 1959, John McCarthy, MIT, danach Stanford

Der fundamentale Mechanismus ist die rekursive Funktionsdefinition

Automatische Speicherbereinigung (1959!)

Viele Nachfolger, z.B. Scheme (MIT)

Funktionale Sprachen: Haskell, Scheme, ML

LISP “lists”

Eine Liste hat die Form (x1 x2...) , wobei jedes x_i entweder ein Atom (Zahl, Bezeichner etc.), oder (rekursiv) wieder eine Liste ist.

Beispiele:

()
(x1 x2)
(x1 (x2 x3) x4 (x5 (x6 () x7)))

((x1 x2)) ist nicht dasselbe wie (x1 (x2))

LISP Funktionsanwendung und Definition

Die Anwendung (der Aufruf) einer Funktion f auf die Argumente a, b, c wird wie folgt geschrieben:

```
(f a b c)
```

Beispielfunktion (Scheme):

```
(define (factorial n)
        (if (eq? n 0)
                1
                (* n (factorial (- n 1)))))
```

Um eine solche Andwendung zu vermeiden, kann mein eine Quote benutzen:

`(f (a b c))` → wendet f dem Resultat der Anwendung von f auf die liste (a b c) an

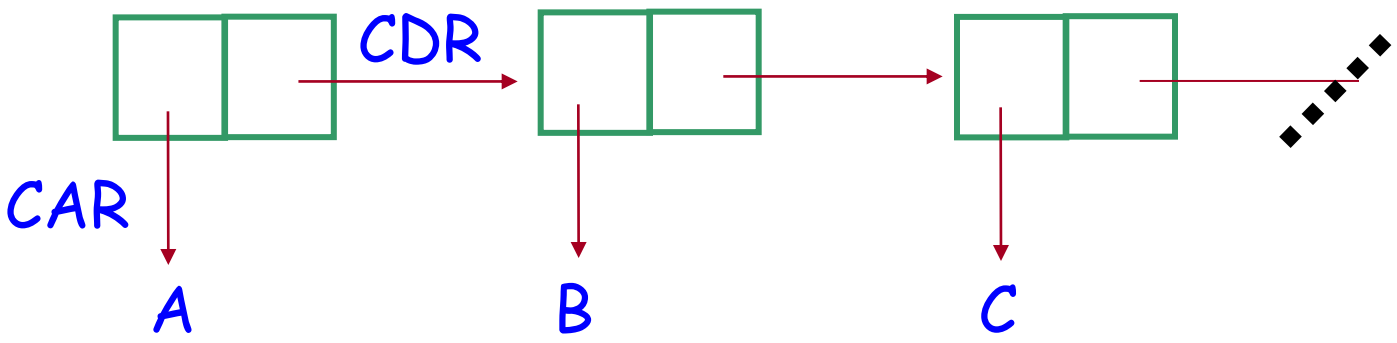
`(f '(a b c))` → wendet f auf die liste (a b c) an

Grundfunktionen

Sei my_list = (A B C)

(CAR my_list) = A
(CDR my_list) = (B C)

(CONS A (B C)) = (A B C)



Funktionen auf Listen

```
(define double-all (list)
        (mapcar
            '(lambda (x) (* 2 x)) list))

(define (mapcar function f)
    (if (null? ls) '()
    (cons
        (function (car ls))
        (mapcar function (cdr ls))) ) )
```

Objekt-orientierte Programmierung

Simula 67: Algol 60 Erweiterungen für Simulationen, University of Oslo, 1967 (nach Simula 1, 1964).
Kristen Nygaard, Ole Johan Dahl

Wurde zu einer vollwertigen Programmiersprache

Smalltalk (Xerox PARC) fügte Ideen von Lisp hinzu und hatte innovative Ideen für Benutzerschnittstellen.
Alan Kay, Adele Goldberg, Daniel Bobrow

“Hybrid”-Sprachen

Objective-C, ca 1984: Smalltalk-Layer auf C

C++, ca 1985: “C mit Klassen”

Machten O-O akzeptabel für die Mainstream-Industrie

Schlüsselmoment: erste OOPSLA, 1986

Java und C#

C++ mit genügend Restriktionen, um Typ-Sicherheit und Speicherbereinigung zu ermöglichen

Java wurde zuerst als „Applets“ im Zusammenhang mit der Explosion des Internets 1995 bekannt.

C# hat “delegates” (Agenten-ähnlicher Mechanismus)

Eiffel

Erste Version geht in die Mitte der 80er-Jahre zurück.
Erste Vorstellung an der OOPSLA 86

Legt Wert auf Prinzipien des Software-Engineerings: Geheimnisprinzip, Design by Contract, statische Typisierung (durch Generik), vollständige Anwendung von O-O-Prinzipien

Anwendungen: auftragsentscheidende Projekte in der Industrie

(noch) 30 Sekunden

Andere Vorlesungen von unserem Lehrstuhl

Software Architecture, 2. Jahr

(Warscheinlich neuer Name von nächsten Jahr:

Introduction to Software Architecture &

Software Engineering)

Bachelor's/Masters:

Distributed and Outsourced Software Engineering (DOSE)

Languages in depth: Java & C#

Languages in depth: Eiffel

Concepts of Concurrent Computation

Software Verification

Software Engineering Seminar

+ Gelegentliche Gast-Vorlesungen