



# Einführung in die Programmierung

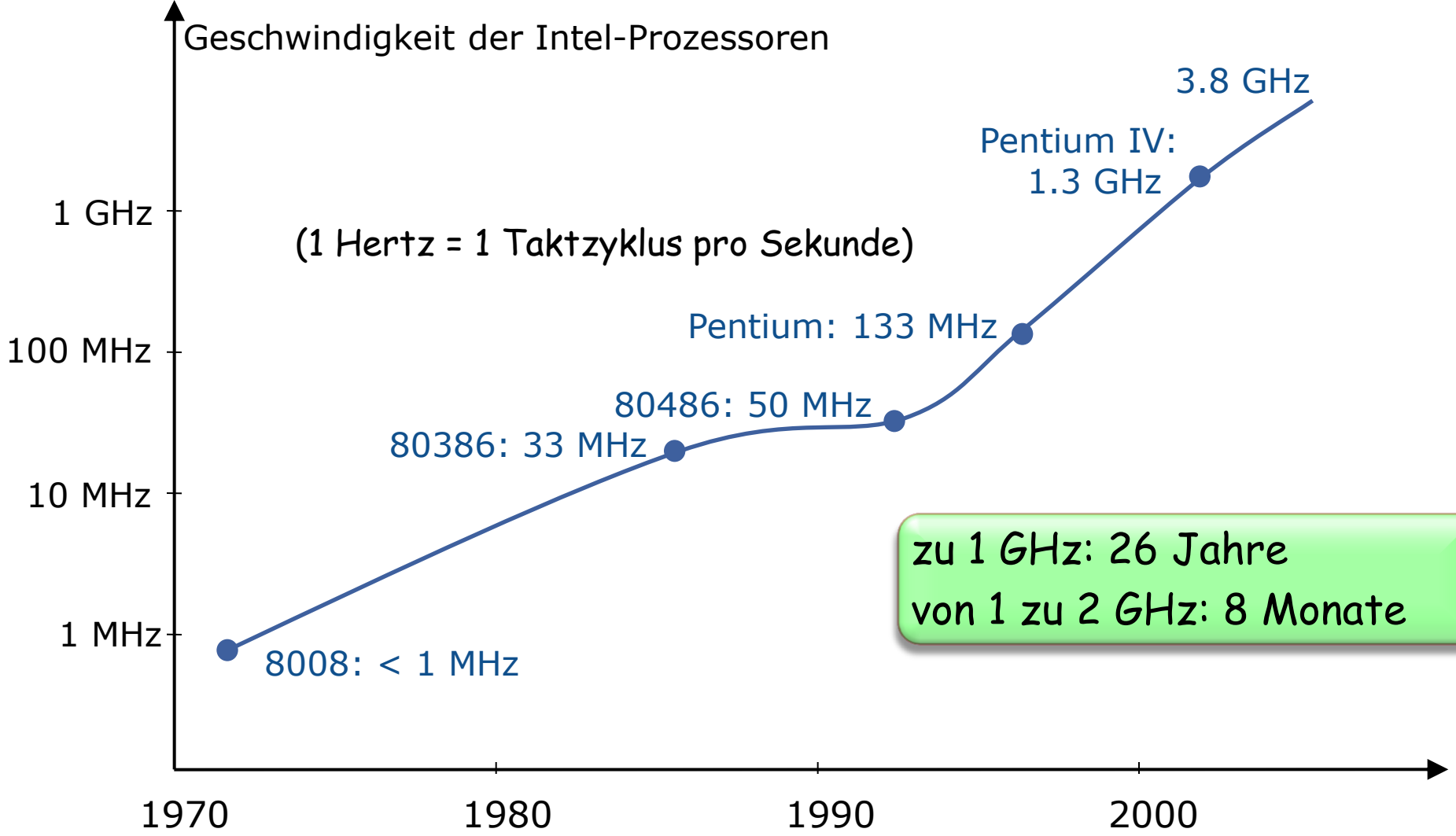
Prof. Dr. Bertrand Meyer

Lektion 19: Von der Programmierung  
zum Software-Engineering

# Moore's "Gesetz"

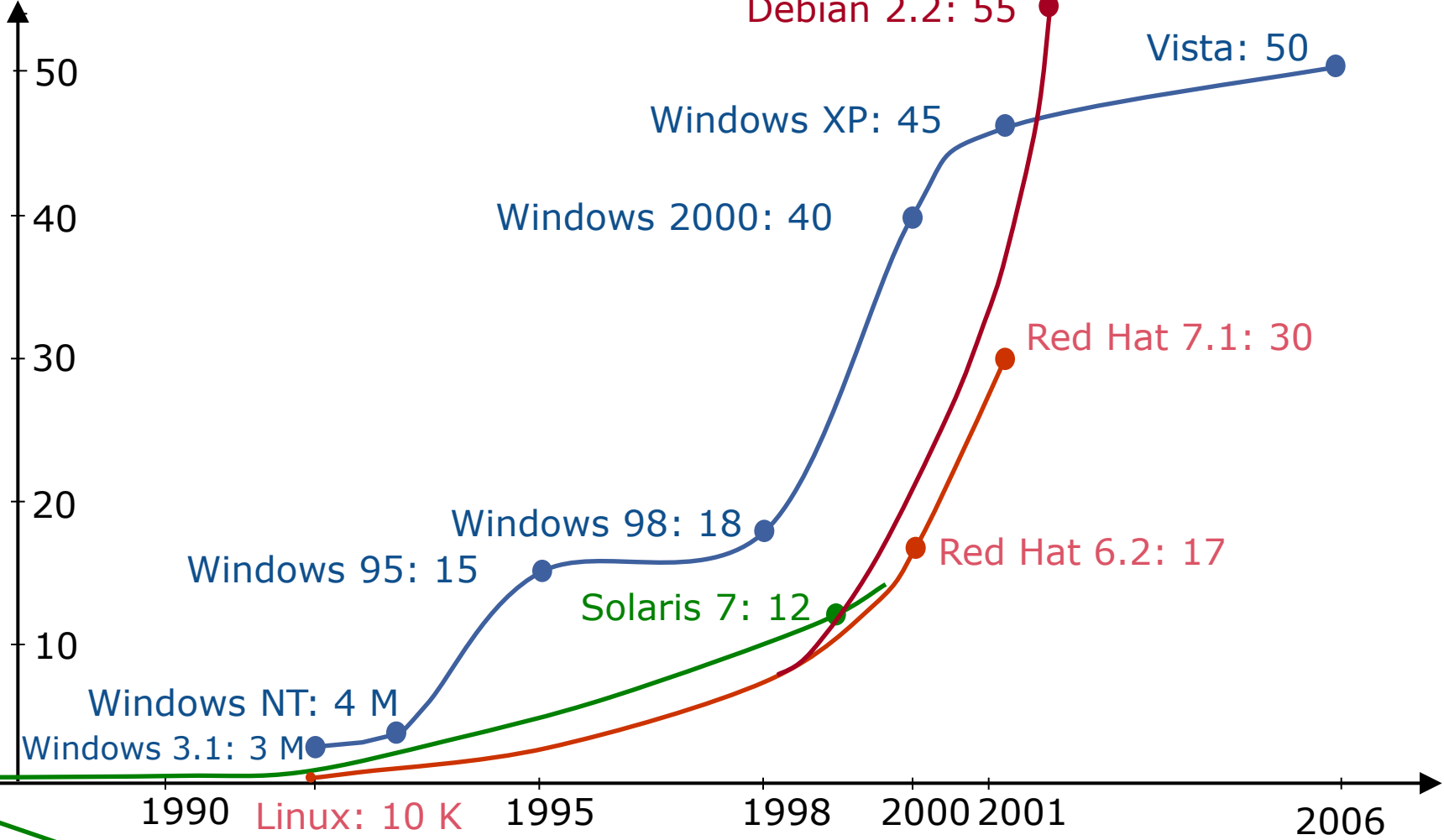


Ungefähre Verdopplung der Rechenleistung alle achtzehn Monate, für vergleichbare Preise



# Betriebssysteme: Grösse des Quellcodes

Codezeilen (in Millionen)



Unix V7: 10K

# Software-Engineering (1)

---



Die Prozesse, Methoden, Techniken, Werkzeuge und Sprachen, um funktionsfähige **Qualitäts**software zu entwickeln

Die Prozesse, Methoden, Techniken, Werkzeuge und Sprachen, um funktionsfähige **Qualitäts**software zu entwickeln, die

- Sehr gross sein kann
- Über einen langen Zeitraum entwickelt und benutzt wird
- Viele Entwickler involviert
- Oft geändert und verbessert wird

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





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 at the FAA's Los Angeles Center, which controls a 100,000-square-mile area, is the same upgrade completed successfully at 19 other centers. But designers didn't allow for information typed in manually by Mexico controllers. A digit out of place could crash the system.

The new system failed when data from a Mexico controller was received. The radar system switched to backup. The new system was restarted later, and failed again. The old system was reinstalled. The FAA ordered a nationwide ground stop for all flights to the Southwest, causing cancellations, rerouting, delays and airport gridlock.

Technicians must now rewrite the software to recognize Mexico controller information.

# 1998 Mars Orbiter Vehicle

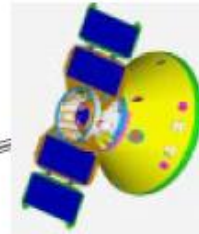


*"The peer review ... indicates that one team used English units (e.g., inches, feet and pounds) while the other used metric units... This information was critical to the maneuvers to place the spacecraft in the proper Mars orbit. "*

## Mars Polar Lander

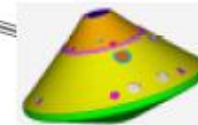
### Cruise

- RCS attitude control
- Four trajectory correction maneuvers, Site Adjustment maneuver 9/1/99, Contingency maneuver up to Entry – 7 hr.
- 11 Month Cruise
- Near-simultaneous tracking w/ Mars Climate Orbiter or MGS during approach



### Entry, Descent, and Landing

- Arrival 12/3/99
- Jettison Cruise Stage
- Microprobes sep. from Cruise Stage
- Hypersonic Entry (6.9 km/s)
- Parachute Descent
- Propulsive Landing
- Descent Imaging [MARDI]

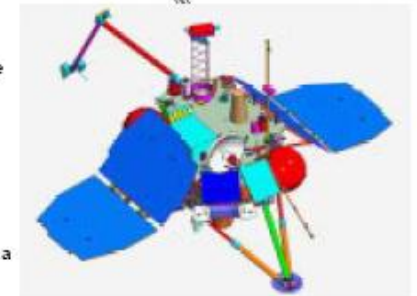


### Launch

- Delta 7425
- Launch 1/3/99
- 576 kg Launch Mass

### Landed Operations

- 76° S Latitude, 195° W Longitude
- Ls 256 (Southern Spring)
- 60–90 Day Landed Mission
- MVACS, LIDAR Science
- Data relay via Mars Climate Orbiter or MGS
- Commanding via Mars Climate Orbiter or direct-to-Earth high-gain antenna



# Ariane-5 Jungfernflug, 1996



37 Sekunden nach dem Start wurde eine Ausnahme im Ada-Programm nicht behandelt; es wurde der Befehl erteilt, die Mission abubrechen. 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](http://se.ethz.ch/~meyer/publications/computer/ariane.pdf)

# Sicherheitsbeispiel: Der Puffer-Überlauf

---



Das System erwartet eine Eingabe eines externen Benutzers:

Vorname:

Nachname:

Adresse:

**from**  $i := 1$  **until**

*$i > \text{eingabe\_länge}$*

**loop**

*$\text{puffer}[i] := \text{eingabe}[i]$*

*$i := i + 1$*

**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 darf's 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."*

```
from  $i := 1$  until  
     $i > \text{eingabe\_länge}$   
loop  
     $\text{puffer}[i] := \text{eingabe}[i]$   
     $i := i + 1$   
end
```

Max

Daten

Main

...

Routine 1

Routine 2

...

R Meine Adresse e



Mein böser Code

Code der Routine  
n-1

0

Speicher

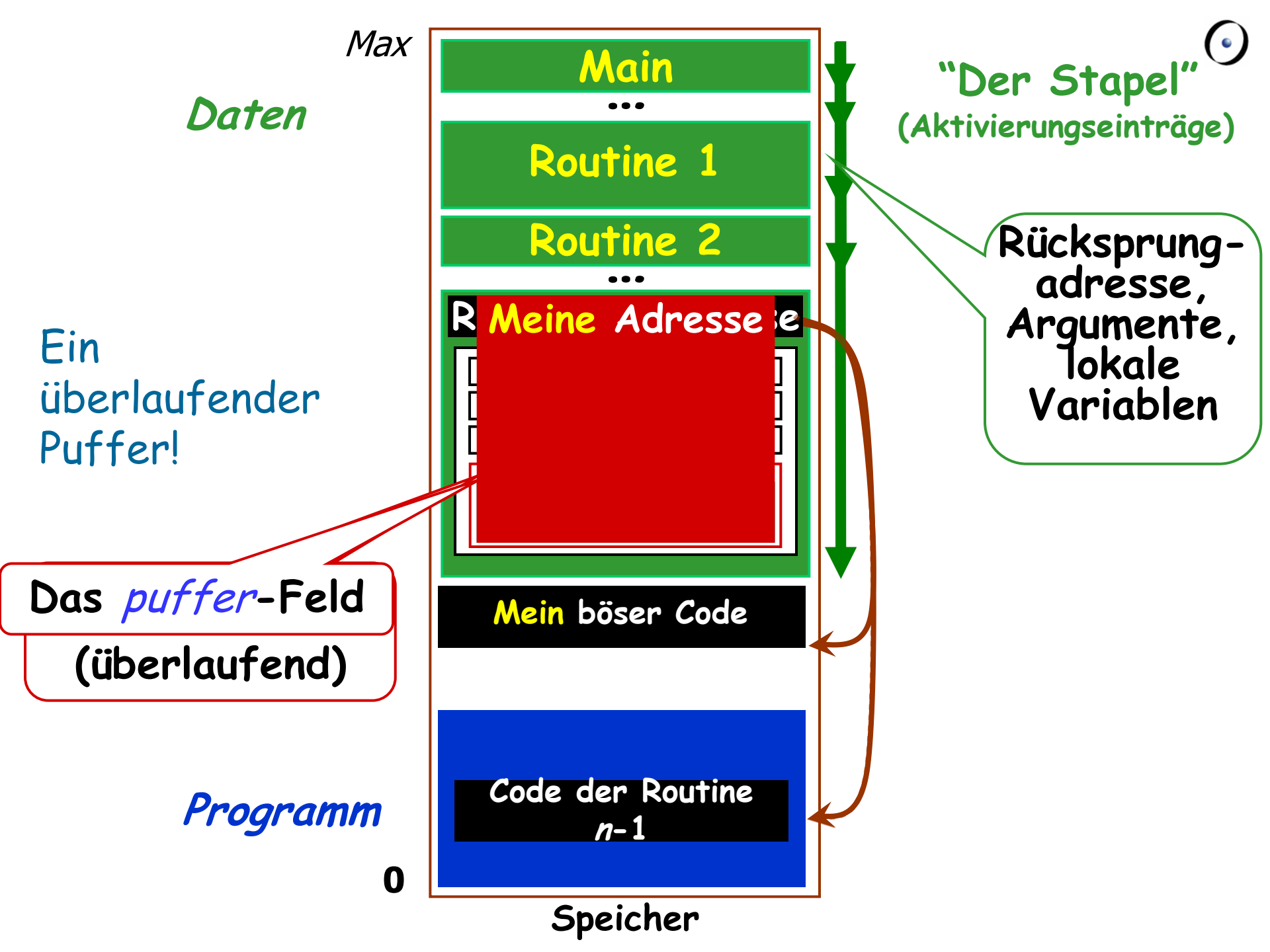
“Der Stapel”  
(Aktivierungseinträge)

Rücksprung-  
adresse,  
Argumente,  
lokale  
Variablen

Ein  
überlaufender  
Puffer!

Das *puffer*-Feld  
(überlaufend)

Programm





# Die Eingabe erhalten

---



from  $i := 1$  until

$i > \text{eingabe\_länge}$  or  $i > \text{puffer\_länge}$

loop

$\text{puffer}[i] := \text{eingabe}[i]$

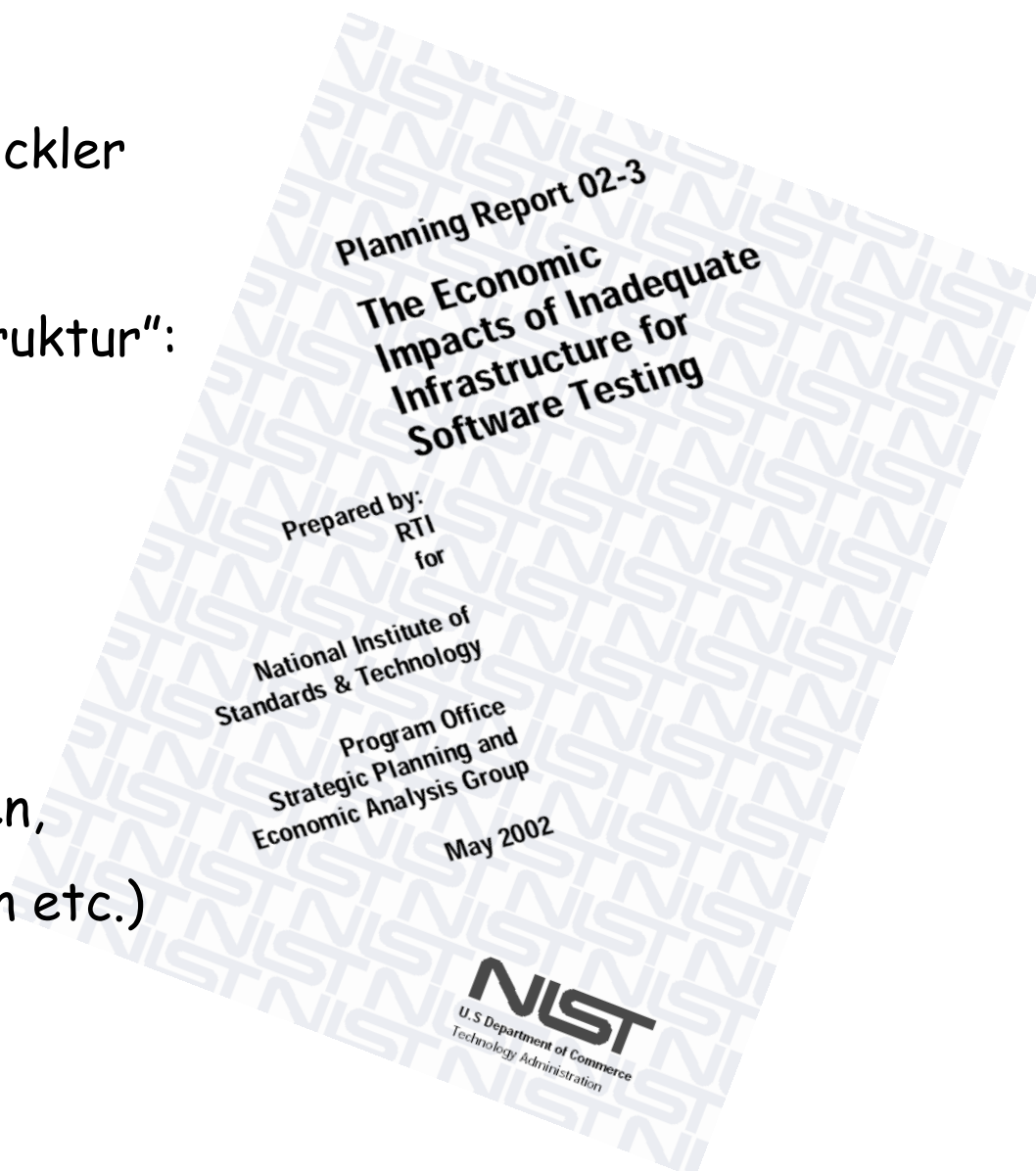
$i := i + 1$

end

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** 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.

# Einige interne Faktoren

---



Modularität, Geheimnisprinzip

Einhaltung von Stilregeln

Klare Kommentare

Konsistenz

Struktur

Gebrauch von Entwurfsmustern

...

**Produkt:** Eigenschaften der resultierenden Software

z.B.: Korrektheit, Effizienz

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

## Produktqualität (unmittelbar):

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

## Produktqualität (langfristig):

- Erweiterbarkeit
- Wiederverwendbarkeit
- Portabilität

## Prozessqualität:

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

## Korrektheit:

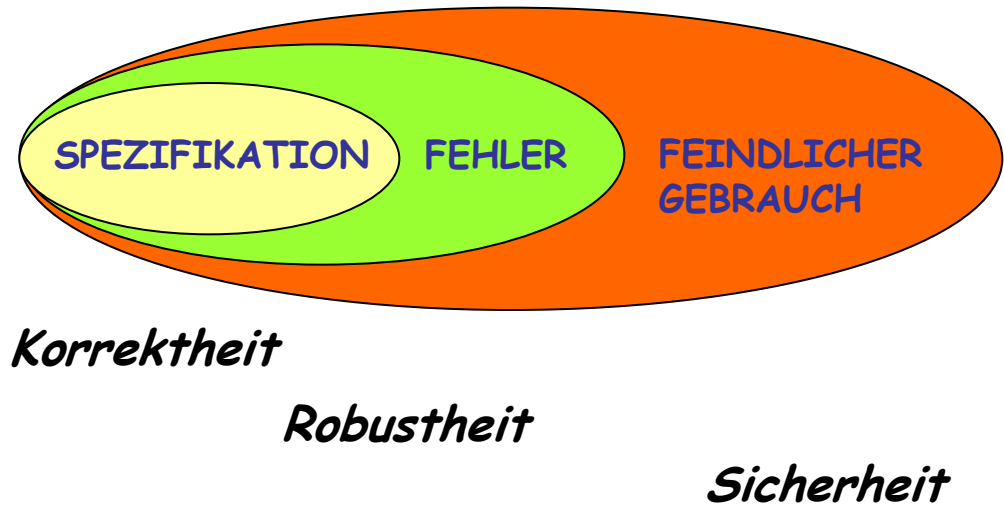
Die Fähigkeit des Systems, in spezifizierten Fällen der Spezifikation entsprechend zu arbeiten

## 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



Anforderungsanalyse

Spezifikation

Entwurf

Implementation

Validierung und Verifizierung (V&V)

Management

Planen und abschätzen

Messen



Die Bedürfnisse der Benutzer verstehen

Die Bedingungen an das System verstehen

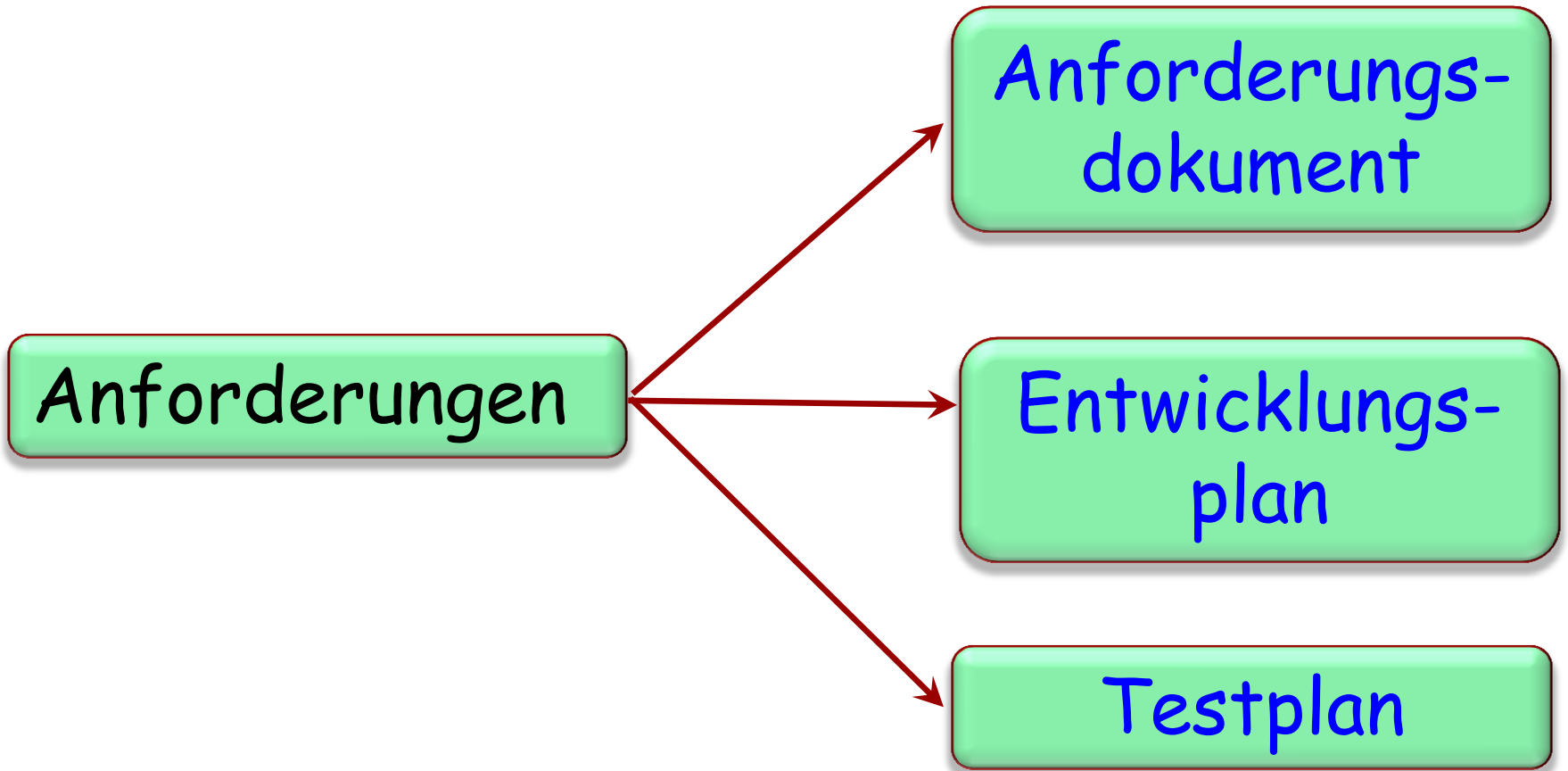
- Interne Bedingungen: Klasseninvarianten
- Externe Bedingungen

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 Literaturverzeichnis für zitierte Quellen

- **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



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

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

- 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

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 den abgeschlossenen Anteil des Hintergrund-Tasks anzeigen.

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 ganz 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.



## 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 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)

## Vorgeschlagene Struktur des Dokuments:

### 1. Introduction

1.1 Purpose

1.2 Scope

1.3 Definitions, acronyms, and abbreviations ← Glossar!

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.

## Management-Aspekte:

- Alle Akteure einbinden
- Vorgehensweise für kontrollierte Änderungen entwerfen
- Mechanismen für die Rückverfolgbarkeit etablieren
- Behandeln sie das Anforderungsdokument als wertvolles Kapital eines jeden Projekts; Fokus auf Klarheit, Präzision, Vollständigkeit

## Technische Aspekte: Wie Präzision erlangen?

- Formale Methoden?
- Design by Contract

**Verifikation:** überprüft interne Konsistenz

Beispiele: Überprüfung der Typen; Überprüfung, dass die Ausführung keinen Crash verursacht

*"Überprüfen, dass wir das **System richtig** entwickelt haben"* (alle Regeln eingehalten)

**Validierung:** Überprüfung bezüglich einer Beschreibung auf einer höheren Ebene

Beispiel: Validierung eines Programmes gegen seine Spezifikation

*"Überprüfen, dass wir das **richtige System** entwickelt haben"*

(Bedürfnisse der Benutzer befriedigt)

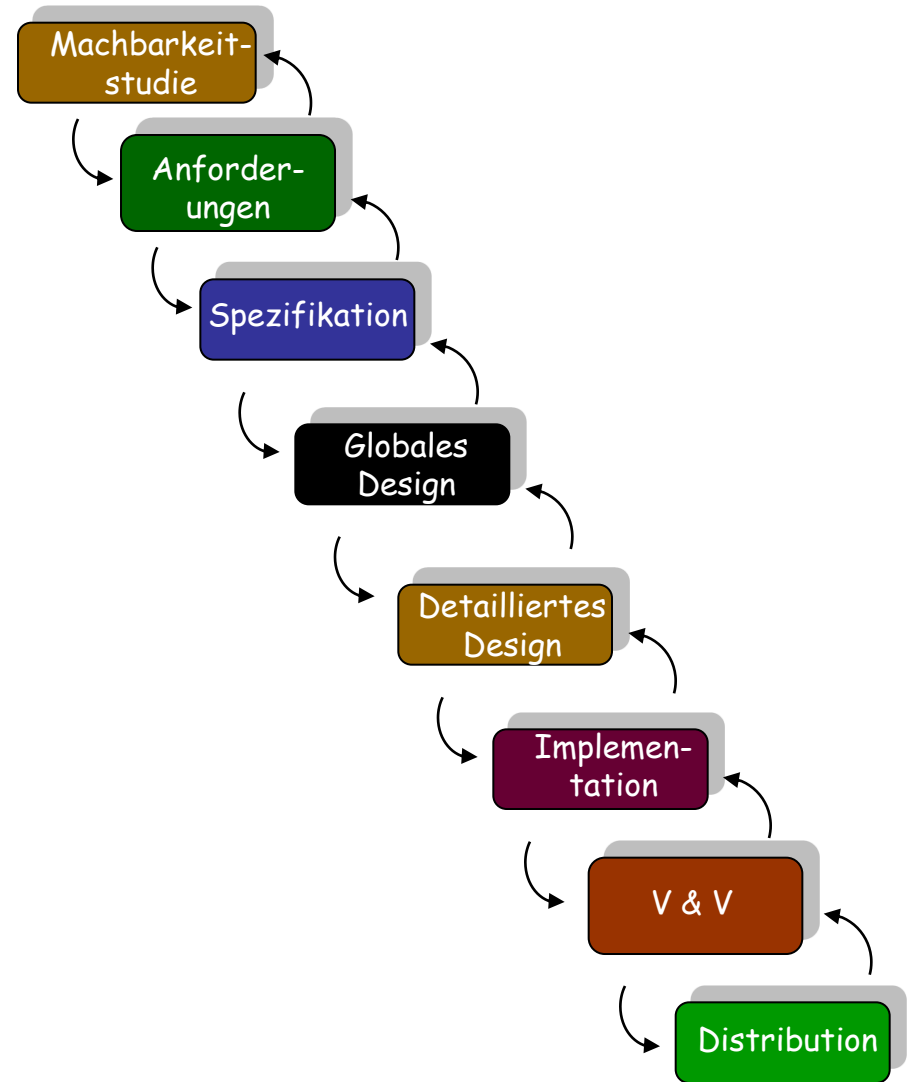


Beschreiben eine allgemeine Aufteilung der Softwarekonstruktion in Aufgaben und die Reihenfolge dieser Aufgaben.

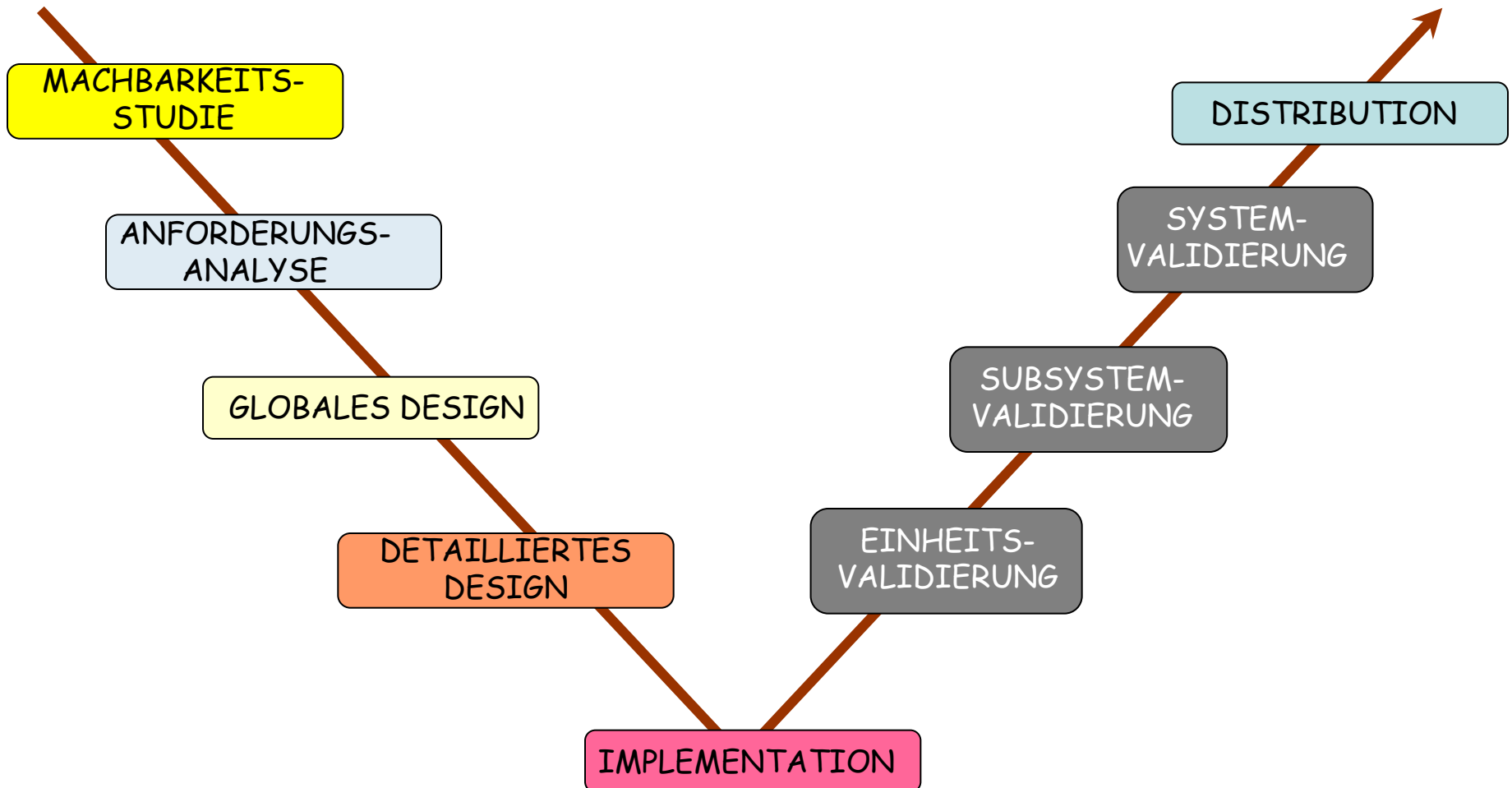
Sie sind Modelle auf zwei Arten:

- Bilden eine abstrahierte Version der Realität
- Beschreibung eines idealen Systems, in der Praxis nicht immer eingehalten.

# Das Wasserfall-Modell



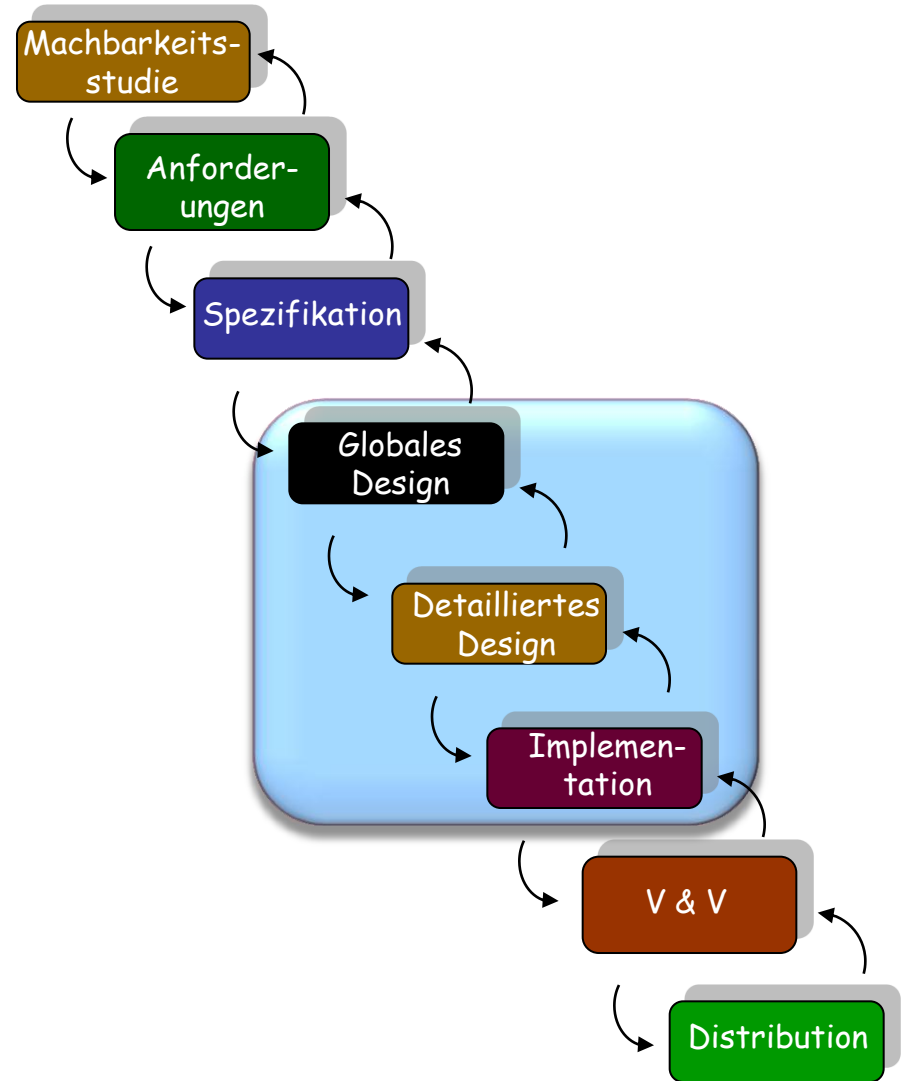
# V-Form



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

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

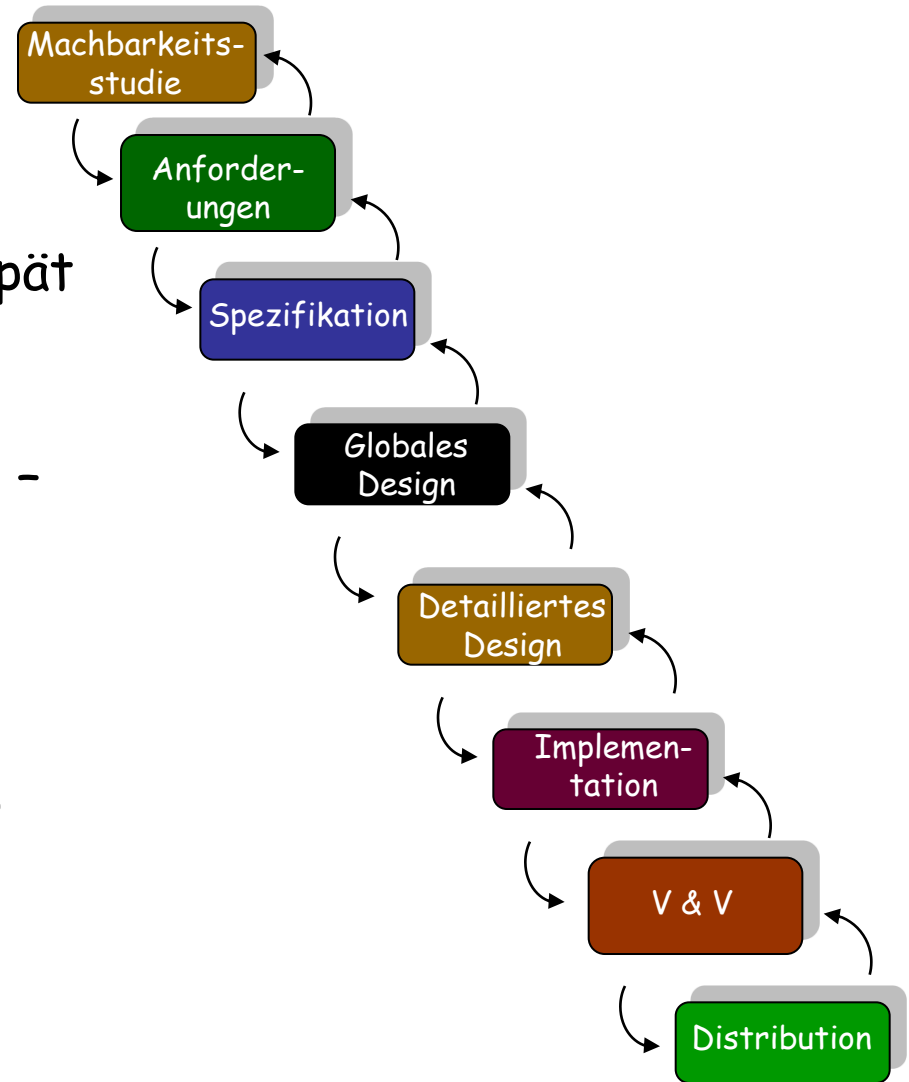
# Verschmelzen von mittleren Aktivitäten

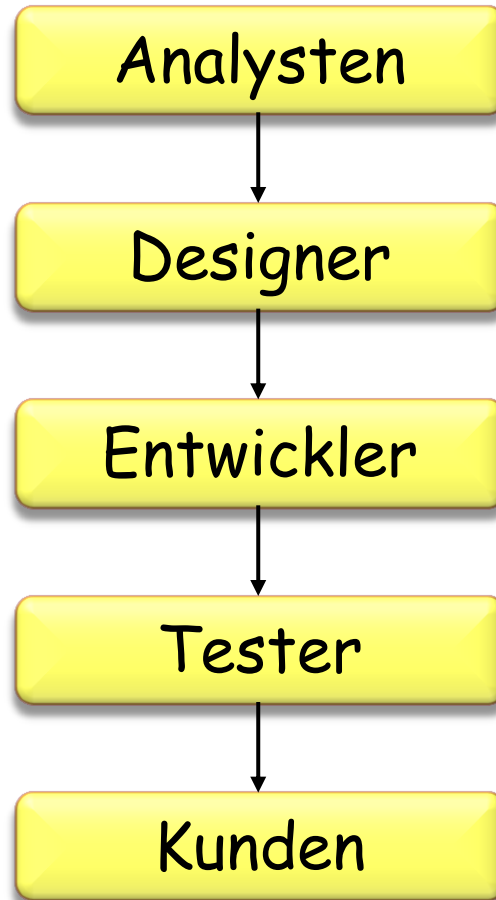


# Probleme mit dem Wasserfall

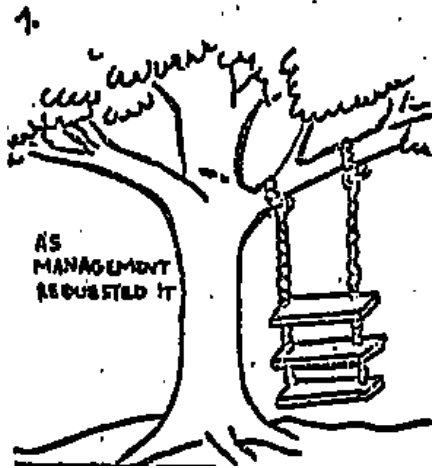


- Eigentlicher Code taucht erst spät auf.
- Keine Unterstützung für Änderungen der Anforderungen - und im Allgemeinen für Erweiterbarkeit und Wiederverwendbarkeit
- Keine Unterstützung für Unterhaltsaktivitäten (70% der Softwarekosten!)
- Arbeitsteilung behindert „Total Quality Management“
- Sehr synchrones Modell

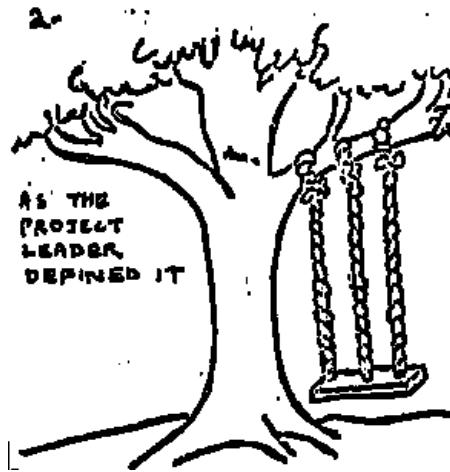




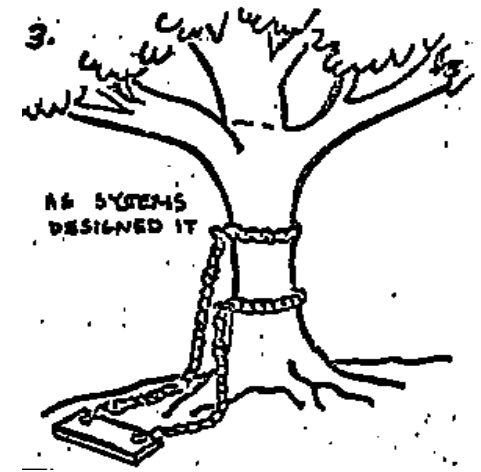
# Lebenszyklus: "Impedance mismatches"



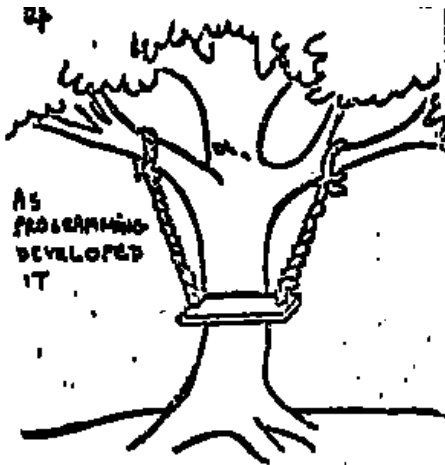
Vom Management gefordert



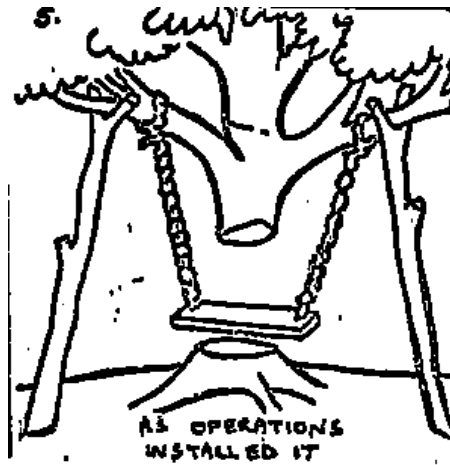
Vom Projektleiter definiert



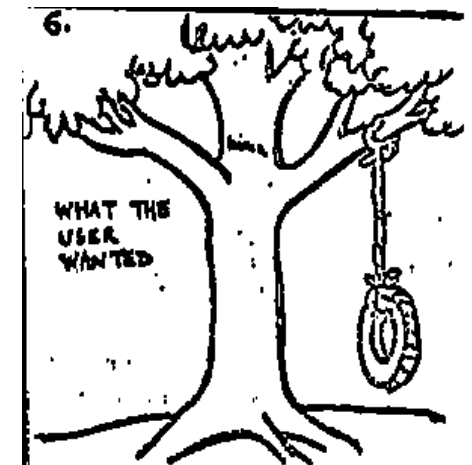
Von den System-Designern gestaltet



Von den Programmierern implementiert



Von den Admins installiert



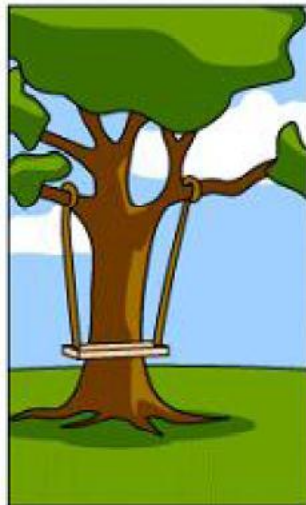
Was der Benutzer wollte



# Eine modernere Variante



How the customer explained it



How the Project Leader understood it



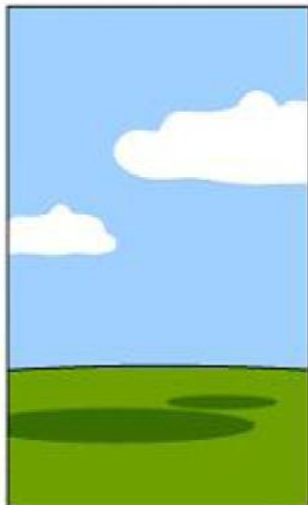
How the Analyst designed it



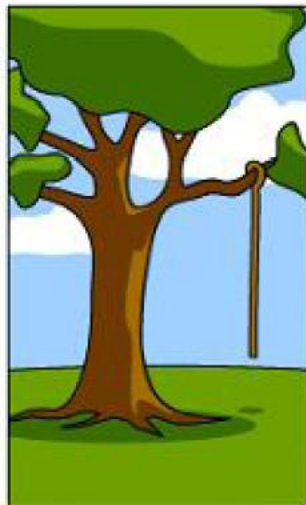
How the Programmer wrote it



How the Business Consultant described it



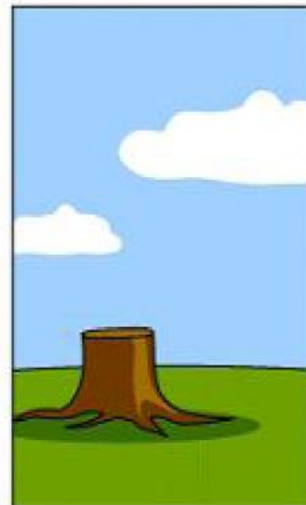
How the project was documented



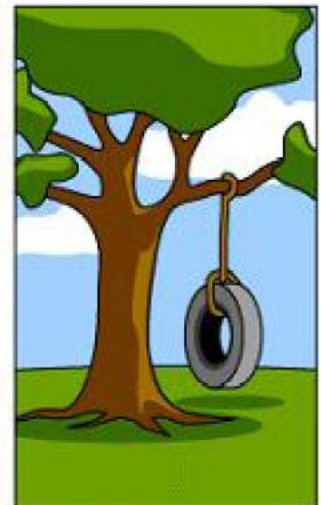
What operations installed



How the customer was billed



How it was supported

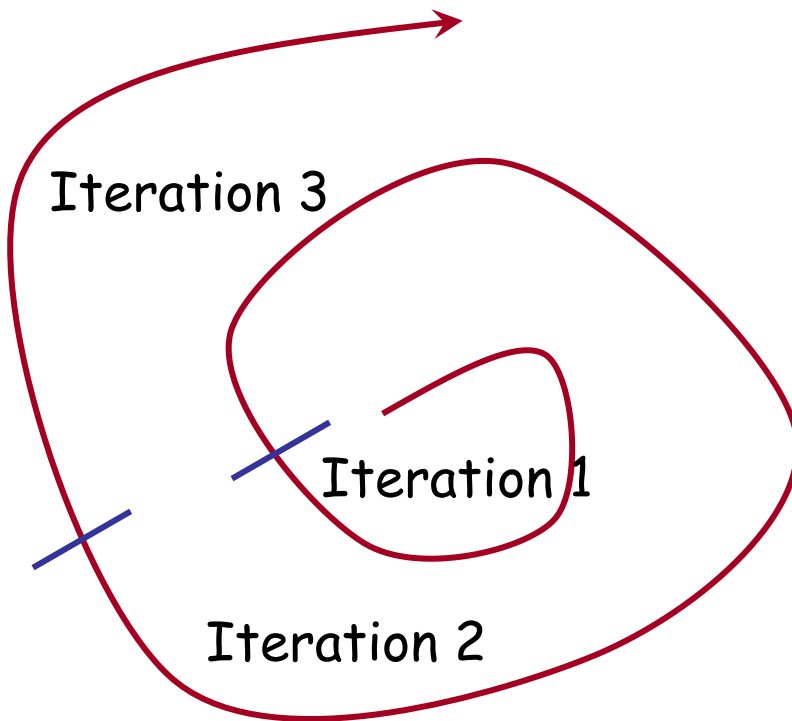


What the customer really needed

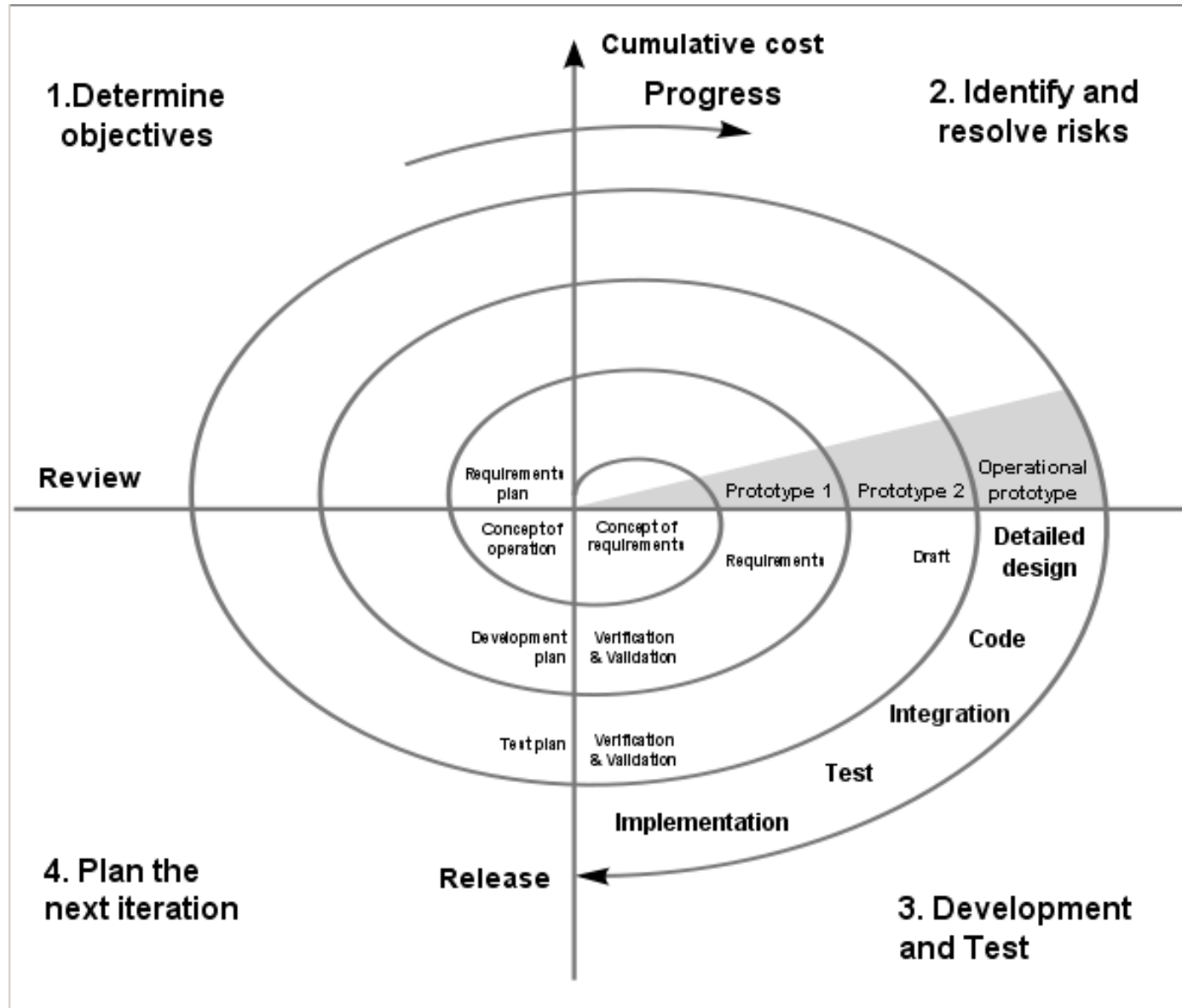
# Das Spiralenmodell (Boehm)



Einen dem Wasserfall-Modell ähnlichen Ansatz auf aufeinanderfolgende Prototypen anwenden



# Das Spiralenmodell

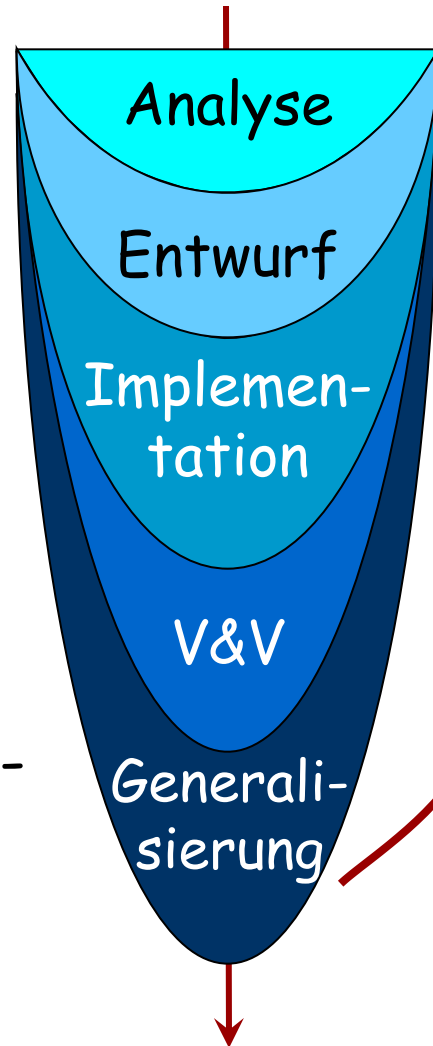


## Die Eiffel-Sicht

- Eine einzige durchgehende und einheitliche Notation, Werkzeuge, Konzepte, und Prinzipien
- Beständige, schrittweise Entwicklung
- Modell, Implementation und Dokumentation konsistent halten

Umkehrbarkeit: Man kann vor- und zurückgehen

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



Beispielklassen:

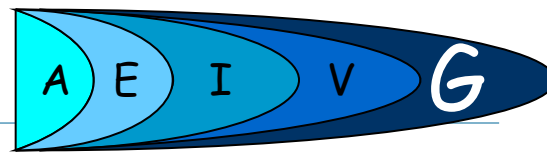
*PLANE, ACCOUNT,  
TRANSACTION...*

*STATE,  
COMMAND...*

*HASH\_TABLE...*

*TEST\_DRIVER...*

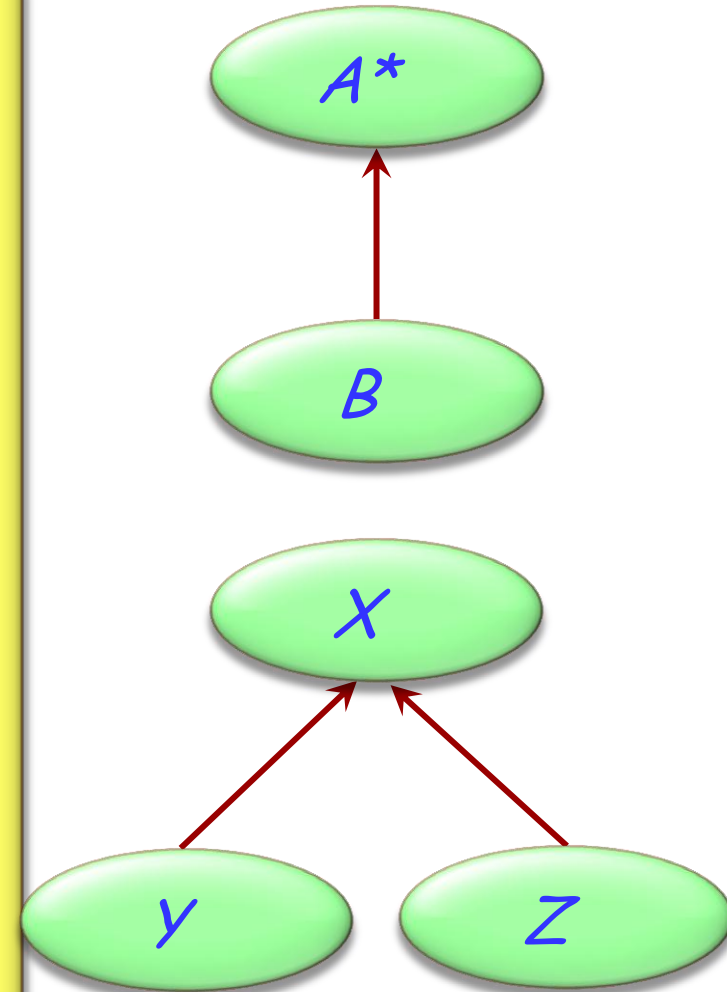
*TABLE...*



Für Wiederverwendung vorbereiten.  
Zum Beispiel:

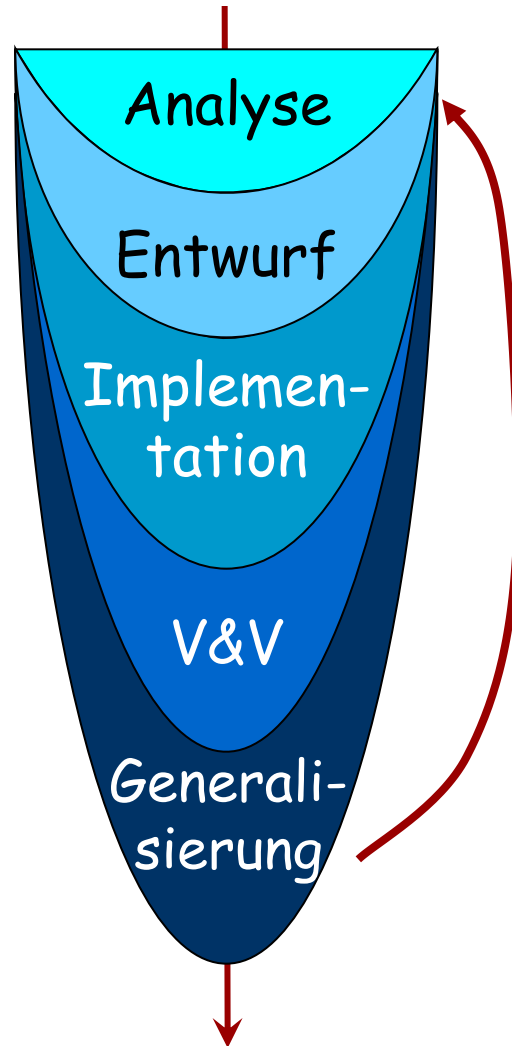
- Eingebaute Beschränkungen entfernen
- Abhängigkeiten von Details des Projektes entfernen
- Dokumentation, Verträge verbessern...
- Abstrahieren
- Ähnlichkeiten extrahieren und Vererbungshierarchie ausbessern

Nur wenige Firmen haben das Budget für diesen Ansatz



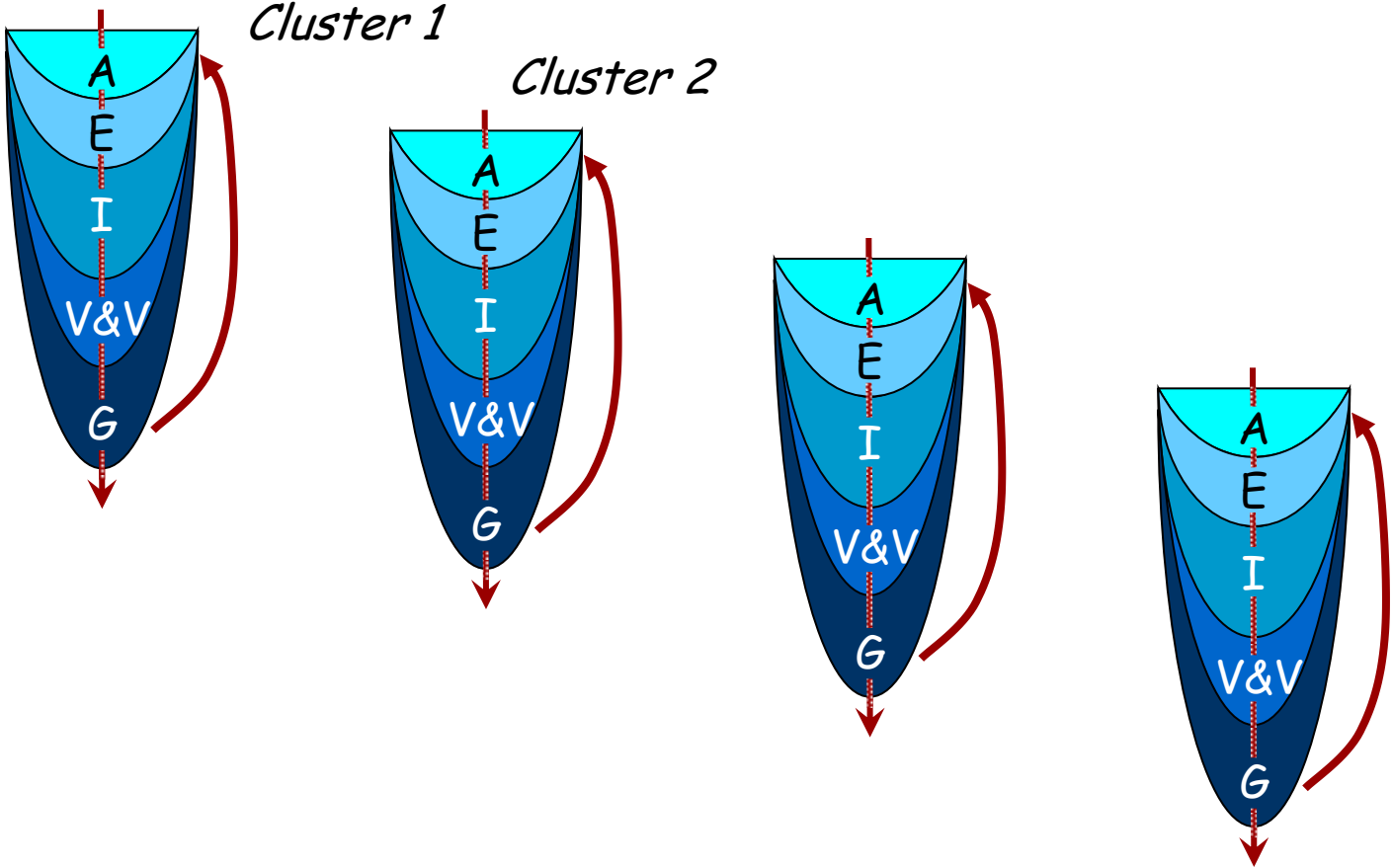
*Es scheint folglich, dass die Arbeit der Ingenieure, der Designer und der Kalkulatoren in den Konstruktionsbüros darin besteht, zu verwischen und zu polieren, jenes Verbindungsstück leichter zu machen, diesen Flügel auszubalancieren bis man ihn nicht mehr wahrnimmt, bis er nicht mehr ein am Rumpf befestigter 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)

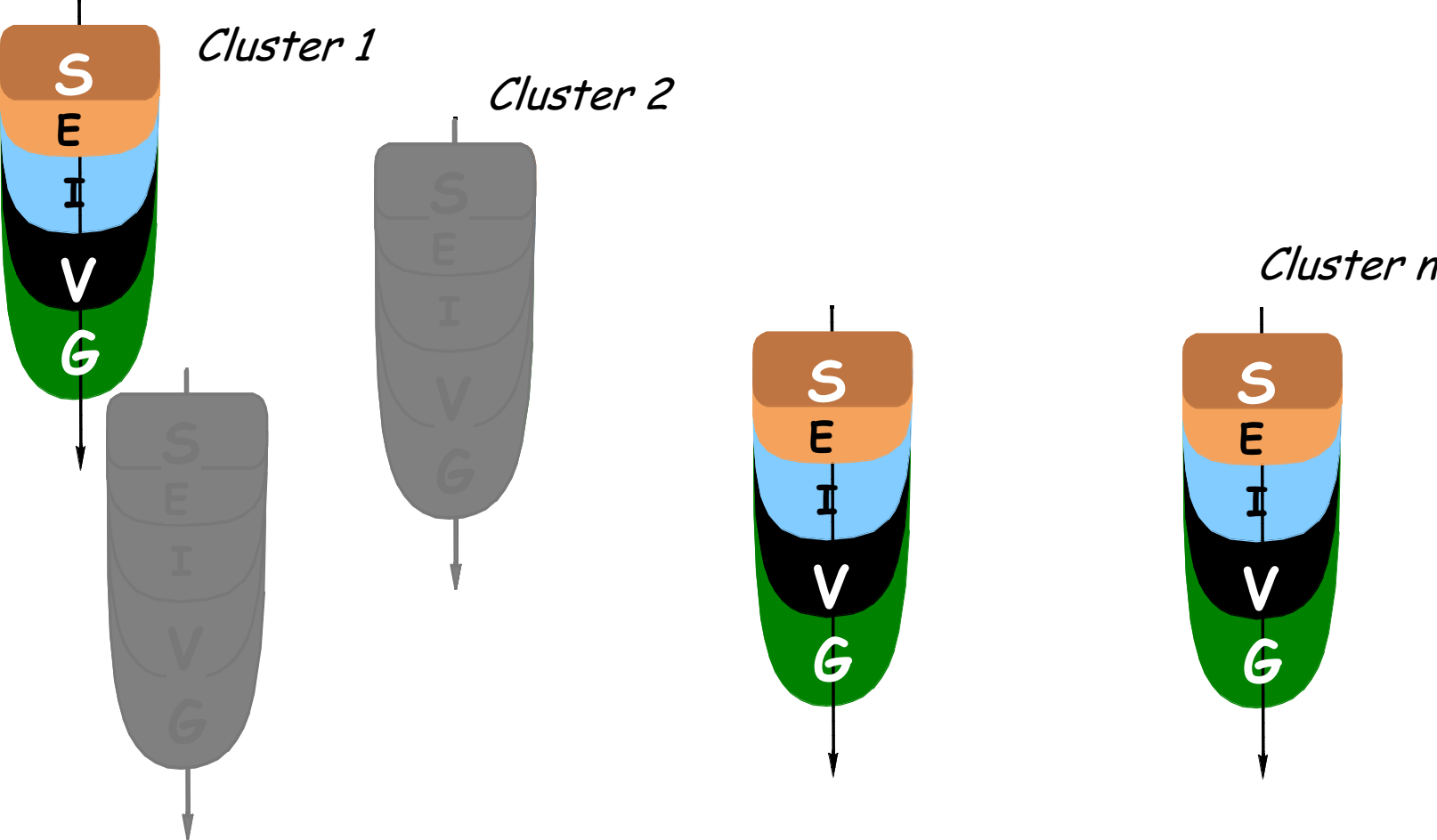




# Das Cluster-Modell



# Das Cluster-Modell



## 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 an separaten UML-Werkzeugen

Metrik-Werkzeug

Profiler-Werkzeug

Werkzeug zur Generierung von Dokumentation

...

# Agile/schlanke Methoden und „extreme programming“

---

Weniger Wert auf **formale** Prozesse legen

Betonung von kurz-zyklischer, zeitlich begrenzter, **iterativer** Entwicklung

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

Nutzen eines zweiten Paar Augen: **Paarprogrammierung**

Betonung der Rolle des **Refactoring**

**Selbst-organisierte** Teams

Mehr Wert auf die **Miteinbeziehung der Kunden**

Kollaborative, verteilte Entwicklung

Konzentrische Vertrauenskreise

Erfolg mit starkem Projektleiter (z.B. Linux)

“Mit genügend Augen sind alle Bugs oberflächlich”

Nicht nur Testen:

- **Statische Analyse**-Werkzeuge durchsuchen den Code nach möglichen Schwächen, z.B. uninitialisierte Variablen
- **Korrektheitsbeweise** werden immer realistischer
- **Model checking** erkundet den Zustandsraum einer abstrahierten Version des Programmes

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

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

Dokumentationswerkzeuge

Werkzeuge, um Anforderungen zu sammeln

Analyse- und Entwurfswerkzeuge

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

Formale Entwicklungs- und Beweiswerkzeuge

Integrierte CASE (Computer-Aided Software Engineering)-Umgebungen

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

Zwei prinzipielle Varianten:

- Build-Management
- Versionsmanagement



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

Beispiel

```
make program
```

Mit dem makefile

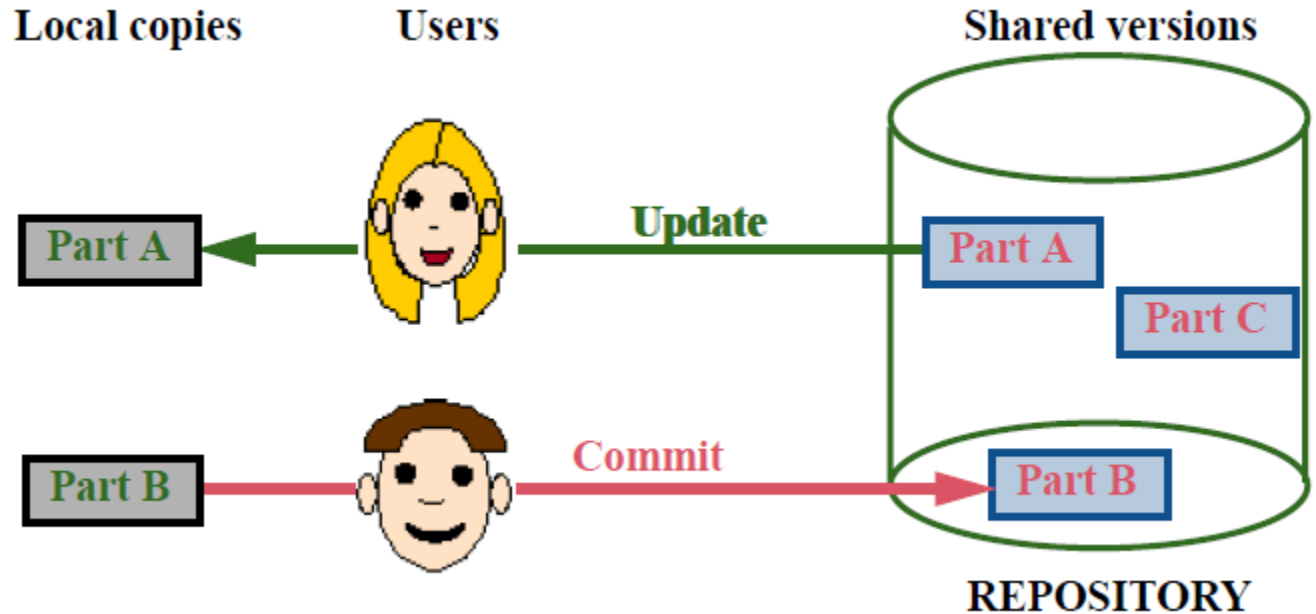
```
program: main.o module1.o module2.o  
cc main.o module1.o module2.o  
%.c: %.o  
cc $<
```

Grösste Limitierung: Die Abhängigkeiten müssen manuell erfasst werden

# Versionsmanagement



Beispiele:  
RCS, CVS  
Subversion



Haupt-Operationen:

- Commit
- Update

Speichert die "diffs" zwischen Versionen

Tipp: Verzweigungen («branches») vermeiden; früh und oft abgleichen

Diese Werkzeuge sind erhältlich und einfach zu gebrauchen

Kein Projekt kann es sich leisten, diese nicht zu gebrauchen

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*, Sicherheitssystem der neusten Metrolinie in Paris

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)

| <i>Programmtyp</i> | <i>Aufwand (pm)</i> | <i>Zeit</i>       |
|--------------------|---------------------|-------------------|
| Applikation        | $2.4 * L * 1.05$    | $2.5 * pm * 0.38$ |
| Utility            | $3.0 * L * 1.12$    | $2.5 * pm * 0.35$ |
| System             | $3.6 * L * 1.20$    | $2.5 * pm * 0.32$ |

Anzahl der Bugs abschätzen durch:

- Charakteristiken eines Programmes
- Anzahl bisher gefundener Bugs

Variante: "Fault injection"

Teamspezialisierungen: Kundendienstleister, 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

Gute Technologien machen ein Projekt erfolgreich

Ein schlechtes Management tötet Projekte

Nicht nur, um mit ihrem Computer zu sprechen!

Eine Programmiersprache ist ein Denkwerkzeug

"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
```

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

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

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

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

Algol W führte den «record»-Datentyp ein

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

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

---

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

Zu Beginn eng mit Unix verbunden

Betonung des Maschinenzugriffs auf tiefer Ebene: Zeiger, Adressarithmetik, Umwandlungen

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

**LIS**t 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



Eine Liste hat die Form  $(x_1 x_2 \dots)$ , wobei jedes  $x_i$  entweder ein Atom (Zahl, Bezeichner, etc.), oder (rekursiv) wieder eine Liste ist.

Beispiele:

$()$

$(x_1 x_2)$

$(x_1 (x_2 x_3) x_4 (x_5 (x_6 () x_7)))$

$((x_1 x_2))$  ist nicht dasselbe wie  $(x_1 (x_2))$

# 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):

$(\text{define (factorial } n)$

$\text{ (if (eq? } n\ 0)$

$\text{ 1}$

$\text{ (* } n\ (\text{factorial } (-\ n\ 1))))$

Um eine solche Anwendung zu vermeiden, kann man eine Quote benutzen:

$(f\ (a\ b\ c))$   $\rightarrow$  wendet  $f$  auf das Resultat der Anwendung  $(a\ b\ c)$  an

$(f'\ (a\ b\ c))$   $\rightarrow$  wendet  $f$  auf die Liste  $(a\ b\ c)$  an

# Grundfunktionen

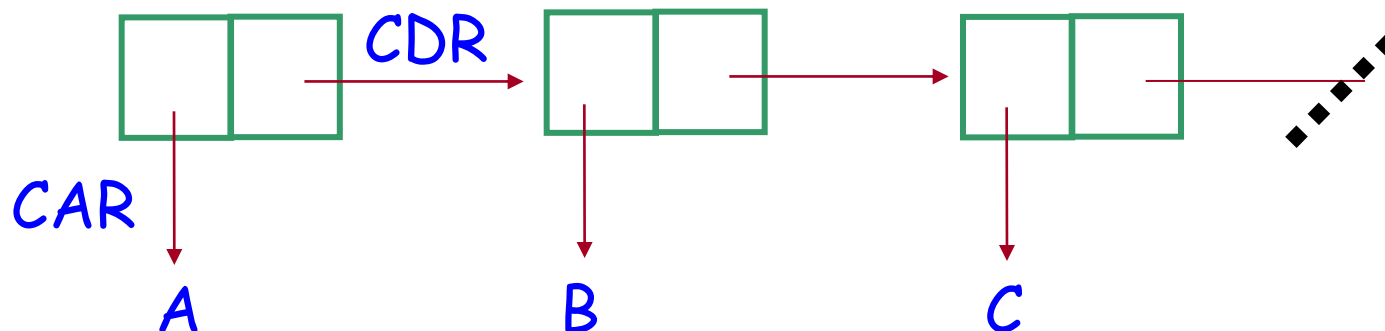


Sei  $my\_list = (A B C)$

$(CAR\ my\_list) = A$

$(CDR\ my\_list) = (B C)$

$(CONS\ A\ (B\ C)) = (A\ B\ C)$



# 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))) ) )
```

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

Wurde zu einer vollwertigen Programmiersprache

Smalltalk (Xerox PARC) fügte Ideen von Lisp und  
innovative Ideen für Benutzerschnittstellen hinzu.  
*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 (Conference on Object-Oriented Programming Systems, Languages, and Applications), 1986

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

Java wurde zuerst als Programmiersprache für „Applets“ im Zusammenhang mit der Explosion des Internets vermarktet, 1995

C# führte „delegates“ ein (Agenten-ähnlicher Mechanismus)

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



Bachelor/Master:

Distributed and Outsourced Software Engineering (DOSE)

Languages in depth: Java and C#

Languages in depth: Eiffel

Concepts of Concurrent Computation

Software Verification

Software Engineering Seminar

+ Gelegentliche Gast-Vorlesungen