Software-Architektur

J. Reichardt
Hochschule Darmstadt

Copyright © J. Reichardt, 2008

Vorwort

- Dieses Vorlesungsskript befasst sich im Wesentlichen mit der Charakterisierung der Aufgaben und Ziele der Software-Architektur. Mit diesem relativ allgemeinen, zum Teil theoretischen, Zugang ist vor allem die Hoffnung verbunden, die derzeit gebräuchlichen Verfahren und Werkzeuge der Software-Architektur besser verstehen und wirkungsvoller einsetzen zu können.

- Es wird dabei die These vertreten, dass Software-Architektur gleichermaßen Struktur-Information als auch textuelle Information in sich vereinen muss, um Syntax und Semantik einer Aussage über den Zweck eines Programms darstellen zu können. Mit anderen Worten, die Software-Architektur muss zwei Verfeinerungen gleichzeitig unterstützen: die Verfeinerung der Struktur und die Verfeinerung der funktionalen Anforderungen (Modulfunktion und – „quer dazu" – Requirements).

- Dies weicht von den gängigen Definitionen der Software-Architektur insofern ab, als hier nicht allein die Struktur eines Programms in den Vordergrund gestellt wird, sondern neben sie gleichberechtigt die textuelle Funktionsbeschreibung.

- Diese These wird in sieben voneinander unabhängigen Betrachtungen entwickelt:

  1. Als Nebenaussage und Tenor gängiger Definitionen
  2. Anhand des Theorems von Rice über Turingmaschinen wird erinnert, dass Struktur allein grundsätzlich nichts über den Zweck eines Programms aussagen kann, sein Verständnis somit zusätzliche textuelle Information erfordert.
  3. Als Nebenaussage und Tenor des Requirements Engineering
  4. Als Konsequenz der Äquivalenz- und Verfeinerungsprobleme beim Übergang zwischen intuitiven und formalen Beschreibungen
  5. Aus der Analogie zwischen Architektur und dem Vererbungsprozess für Klassen.
  6. Aus der Analogie zwischen Architektur und dem Ableitungspro--zess für kontextfreie Sprachen.
  7. Das FracTool-Praktikum soll beispielhaft die Möglichkeiten der Einbettung textueller Design-Information in die Programmstruktur zeigen, und damit die Parallelisierbarkeit der Verfeinerung struktureller und textueller Design-Information.

Gliederung

A. **Einleitung**
Software-Probleme, Problemstellung, Motivation
(Folien 7-11)

B. **Definitionen**
Ziele und Aufgaben der Software-Architektur
(Rolle und Bedeutung der Struktur, der funktionalen Anforderungen, der Kommunikation, des Verstehens)
(Folien 12-34)

C. **Charakterisierungen**

i. Die Doppelfunktion der Software-Architektur als Plan (a priori) und als Erklärung (a posteriori).
(Definition des Zwecks eines Programms, Satz von Rice)
(Folien 35-47)

ii. Software-Architektur als Struktur plus Funktion.
(Gleichzeitige, inkrementelle Verfeinerung der Struktur und der Funktion)
(Folien 48-71)

iii. Software-Architektur als intuitiver Berechenbarkeitsbegriff.
(Church'sche These, Äquivalenz- und Verfeinerungsprobleme beim Übergang zwischen intuitiven und formalen Programm-beschreibungen)
(Folien 72-106)

iv. Software-Architektur als Ergebnis von Interpretationen und Abstraktionen.
(Architektur und Implementierung vs. Sinn und Bedeutung)
(Folien 107-121)

v. Software-Architektur als Syntax einer Aussage.
(Merkmale einer Syntax, attributierte Grammatiken, zweck-gemäße Funktionsverteilung)
(Folien 122-129)

vi. Software-Architektur als Zeichen.
(Zeichenaspekte: Syntax, Semantik, Sigmatik, Pragmatik)
(Folien 130-133)

Folien

1. Software-Architektur, Deckblatt
2. Vorwort
3. Gliederung
4. Inhalt 1
5. Inhalt 2
6. Inhalt 3
7. Literatur
8. „Es geht auch ohne Software-Architektur“
9. Die Software-Krise
10. Erfolgsquote bei IT-Projekten
11. Software architecture is a novel and immature branch …
12. Software-Architektur, Definition 1 (Bass et al.)
13. Software-Architektur, Definition 2 (Bass et al.)
14. Software-Architektur, Definition 3 (Shaw, Garlan)
15. Software-Architektur, Definition 4 (Hohmann)
16. Software-Architektur, Definition 5 (Clements et al.)
17. Software-Architektur, Importance (Clements et al.)
18. Software-Architektur, Definition 6 (Siedersleben)
19. Software-Architektur, Ziele und Aufgaben (GI-Arbeitskreis)
20. SWA-Definitionen aus kognitiver Sicht
21. Software-Architektur, Definition 7 (IEEE 1471)
22. SWA gewährleistet Effizienz des Entwicklungsprozesses
23. Gebäude-Architektur und Effizienz
24. Architektur definiert den Zweck eines Gebäudes
25. Zweck und Zweckmäßigkeit
26. Effizienz der Prozesse
27. Der Effizienzbegriff
28. Effiziente Software-Architektur
29. SWA und Maschine, Übersetzungshierarchie
30. SWA und Maschine, Maschinenschnittstelle 1
31. SWA und Maschine, Maschinenschnittstelle 2
32. SWA und Maschine, Verbreitete Prozessoren
33. SWA und Maschine, Java-Übersetzung
34. SWA und Maschine, Eine Java-Methode
35. Der Mensch, Adressat der Software-Architektur
36. Software-Architektur definiert den Zweck eines Programms
37. „Zweck“ – in diesem Zusammenhang etwa gleichbedeutend mit ...
38. SWA als Abstraktion der Implementierung
39. Essentielle und akzidentelle Designinformation
40. Zweckbestimmung – Hierarchische Verfeinerung
41. Grundsätzliches Problem der Software-Architektur - Verstehen
42. Menschlicher Verstand
43. Verstand und seine Komplexität
44. Prinzipielle Grenzen des Verstehens von Programmen
45. Satz von Rice – Konsequenzen 1
46. Satz von Rice – Konsequenzen 2

47. Satz von Rice – Konsequenzen 3
48. Zusammenhang zwischen Anforderung, Funktion und Struktur
49. Anforderung, Funktion, Struktur
50. Software Architecture Process
51. Das Twin-Peaks-Modell
52. Twin-Peaks, Component-Bus-System-Property
53. Twin-Peaks, Challenges
54. Twin-Peaks, Unterschiede zwischen Requirements und Architektur
55. Twin-Peaks, CBSP-Taxonomy
56. Twin-Peaks, CBSP-Metamodel
57. Architecture and Requirements
58. The Requirements Engineering Process
59. Requirements Engineering – Documents
60. Requirements Analysis
61. Problems with the Requirements Analysis
62. The Requirements Analysis Process
63. Method-based Analysis
64. Requirements Specification – Problems with natural language
65. Requirements Specification – Alternatives to natural language
66. System Models
67. Structured Language Specifications
68. The Software Requirements Document – Six properties
69. The Structure of the Requirements Document
70. Requirements Definition and Specification – Key points
71. Software-Architektur und Requirements-Definition – Zusammenfassung
72. Requirements Verification
73. Requirements Validation
74. Exkurs Turing-Maschinen 1 – Definition, Beispiel
75. Exkurs Turing-Maschinen 2 – berechenbar, entscheidbar
76. Exkurs Turing-Maschinen 3 – Kodierung von TMs
77. Exkurs Turing-Maschinen 4 – Church'sche These
78. Konsequenz aus der Church'schen These
79. Übergänge – Äquivalenzen, Verfeinerungen
80. Intuitive und formale Beschreibungen – Definitionen
81. Intuitive und formale Beschreibungen – Beispiel
82. Äquivalenz-Probleme – Satz 2, Korollar 2.1
83. Äquivalenz-Probleme – Korollar 2.2
84. Hilfssätze – Lemmata 2.1 und 2.2
85. Verfeinerungsprobleme – Korollare 2.3 und 2.4
86. Validierung und Verifikation: Nicht-entscheidbare Probleme
87. Heuristische Lösung nicht-entscheidbarer Probleme
88. Optimistische Sichtweisen
89. Intuitive Beschreibungen und ihre Kommunikation
90. Heuristische Vorgehensweisen, Ein Effizienzvergleich
91. Heuristisches Vorgehen
92. Unzusammenhängende Architektur, Geltendes Paradigma
93. Entwicklung von Anforderung, Funktion, Struktur: Batch-Modell
94. Batch-Vorgehensmodell, Effizienzbetrachtung
95. Zusammenhängende Architektur, Gedankenexperiment
96. Entwicklung von Anforderung, Funktion, Struktur: Inkrementelles Modell
97. Inkrementelles Modell, Abbildung zwischen Anforderung und Funktion 1

98. Inkrementelles Modell, Abbildung zwischen Anforderung und Funktion 2
99. Zusammenhängende Architektur und inkrementelles Modell, Effizienz
100. Rational Unified Process als inkrementeller Prozess 1
101. Rational Unified Process als inkrementeller Prozess 2
102. Konsequenzen der Äquivalenz- und Verfeinerungsprobleme
103. Konsequenz für den Architekturprozess
104. Aggregation und Einbettung intuitiver Beschreibungen
105. Architektur und Schnittstellen
106. Design-Information – Sortier- und Verteilungsaspekte
107. Aggregation und Vererbung – Nachbetrachtung
108. Class Hierarchy
109. Abstraktion als Konstruktionsprinzip
110. Abstraktion – Definition
111. Abstraktion – Wesentliches und Unwesentliches
112. Abstraktion – verschiedene Arten
113. Abstraktion in der Software-Architektur
114. Abstraktionsprinzip
115. Abstraktionsstufen
116. Abstraktion – Visualisierung einer Menge
117. Ohne Interpretation keine Abstraktion
118. Abstraktionen und Interpretationen
119. Interpretation und Bedeutung
120. Sinn und Bedeutung
121. Interpretation – Freiheitsgrade
122. Syntax und Semantik
123. Syntax – Formale Definition
124. Architektur-Syntax
125. Architektur als „Syntax“ – Merkmale
126. Architektur als „Syntax“ – Verfeinerung
127. Architektur als „Syntax“ – Vergleich mit natürlicher Sprache
128. Architektur als „zweckmäßige Funktionsverteilung“
129. Architektur als „Syntax“ – Architekturbeschreibung
130. Architektur-Elemente – Informationsgehalt
131. Architektur-Elemente – Wie implementiert?
132. Architektur-Element – Seine Zeichenaspekte
133. Semiotic Model of Computing

Literatur

1. **Bass et al.:** Software Architecture in Practice, *Addison-Wesley 2003*
2. **Booch, G.:** Object-oriented Analysis and Design, with Applications, *Benjamin/Cummings Publishing Company 1994*
3. **Brooks, F.:** The Mythical Man-Month – Essays on Software Engineering, *Addison-Wesley 1995, 20th Anniversary Edition 1998, 9th printing*
4. **Buschmann et al.:** A System of Patterns – Pattern-oriented Software Architecture, *John Wiley and Sons 1996*
5. **Clements et al.:** Evaluating Software Architectures – Methods and Case Studies, *Addison-Wesley 2002*
6. **Clements, P. et al.:** Documenting Software Architectures – Views and Beyond, *Addison-Wesley 2003*
7. **Gamma, E. et al.:** Design Patterns – Elements of Reusable Object-Oriented Software, *Addison-Wesley 1995*
8. **Garlan, D., Shaw, M.:** An Introduction to Software Architecture, *Advances in Software Engineering and Knowledge Engineering, 1, World Science Publishing 1993*
9. **Hofmeister et al.:** Applied Software Architecture, *Addison-Wesley 2000*
10. **Hohmann, L.:** Beyond Software Architecture – Creating and Sustaining Winning Solutions, *Addison-Wesley 2004*
11. **Hopcroft, J., Ullman, J.:** Einführung in die Automatentheorie, Formalen Sprachen und Komplexitätstheorie, *Addison-Wesley 1992*
12. **Klaus,G., Buhr, M.:** Philosophisches Wörterbuch, *Das Europäische Buch, Berlin 1972*
13. **Kondakow, N.I.** (Hrsg.)**:** Wörterbuch der Logik, *Bibliographisches Institut, Leipzig 1978*
14. **Larman, C.:** Applying UML and Patterns – An Introduction to O-O Analysis and Design and the Unified Process, *Prentice Hall 2002*
15. **Rechenberg, P., Pomberger, G.** (Hrsg.)**:** Informatik-Handbuch, *Carl Hanser Verlag, München – Wien 1999*
16. **Reichardt, J.:** Two-dimensional C++, *ACM Symposium on Software Visualization, Brighton/UK, Sept. 2006*
17. **Reichardt, J.:** Equivalence and refinement problems in the software development process, *Technical Report, Nov. 2005, FH Darmstadt*
18. **Reichardt, J.:** Semiotical aspects of the grid calculus – terminology and models, *Technical Report, May 2003, FH Darmstadt*
19. **Reichardt, J.:** Reduction-of-interconnectivity as a universal design rule, *Technical Report, Sept. 1993, FH Darmstadt*
20. **Reussner, R., Hasselbring, W.:** Handbuch der Software-Architektur, *dpunkt-Verlag, 2006*
21. **Schöning, U.:** Theoretische Informatik – kurzgefasst, *Spektrum Akademischer Verlag 1999*
22. **Schöning, U.:** Logik für Informatiker, *Spektrum Akademischer Verlag 2000*
23. **Shaw, M., Garlan, D.:** Software Architecture – Perspectives on an Emerging Discipline, *Prentice Hall 1996*
24. **Sommerville, I.:** Software Engineering, *Addison-Wesley 2007*
25. **van Vliet, H.:** Software Engineering – Principles and Practice, *Wiley and Sons 2000*

„Es geht auch ohne Software-Architektur!“

„Wichtig ist die Funktionalität eines Programms!“

Thesen der Vergangenheit(?). Zeit der „Softwarekrise“. Tiefgreifende Probleme mit der Qualität, der Planbarkeit, den Kosten und der Produktivität im Software-Entwicklungsprozess.

Frühe Zeugnisse der „software"

Friedrich L. Bauer

Die ‚software crisis'

Spätestens 1967 gab es dann sogar eine ‚software crisis': Das US Verteidigungsministerium wurde nervös, als bestellte Computersysteme wegen Mängeln in der Software ihre Zweckbestimmung nicht voll erfüllten und die Sicherheit der Vereinigten Staaten in der Zeit des Kalten Krieges gefährdet schien. Tatsächlich war der Wildwuchs, der mit der akademischen Freiheit verbunden ist, gelegentlich überbordend und führte zu unnützen Arbeiten von Bastlern und „Bit-Fummlern". In einer ‚Study Group on Computer Science', die auf Anregung des US-Delegierten im NATO Science Commitee, Dr. Isidor Rabi, 1967 eingesetzt wurde und zu der ich delegiert wurde, platzte mir bei einer Debatte über die Gründe für die software crisis der Kragen, und ich sagte: ‚The whole trouble comes from the fact that there is so much tinkering with software', und als ich merkte, dass ich einige meiner akademischen Kollegen schockiert hatte, setzte ich nach: ‚What we need is software engineering'. Zur Strafe dafür durfte ich dann die vom NATO Science Committee geförderte erste Konferenz über Software Enineering, 7.–11. Oktober 1968 in Garmisch ausrichten. ‚The conference marked the end of the age of innocence' (Paul E. Ceruzzi, 1998). Software war nach 16 Jahren erwachsen geworden, hatte sich zu einem Wirtschaftsfaktor gemausert, war zum Gegenstand von Sicherheitsbedenken aufgerückt und ist schließlich heute ein Alltagsobjekt.

to **tinker**	basteln *[bastelte, gebastelt]*
to **tinker**	flicken *[flickte, geflickt]*
to **tinker** with sth.	an etw. herumflicken
to **tinker** with sth.	an etw. herumpfuschen *[ugs.]* *[pfuschte herum, herumgepfuscht]*

Erfolgsquote bei IT-Projekten steigt

West Yarmouth (sk) – Die Rate der fehlgeschlagenen IT-Projekte sinkt: Nach ersten Zahlen des Chaos Reports der Standish Group wurden 2006 immerhin 35 Prozent innerhalb des Zeitplans und des Budgets abgeschlossen – und trafen dabei die Anforderungen der User. 1994 lag der Anteil bei nur 16 Prozent. Die Zahl der Totalausfälle sank von 31 auf 19 Prozent.

cz, 19.2.2007

SOFTWARE ARCHITECTURE

What is software architecture?

There is no standard, universally-accepted definition of the term, "software architecture," although there is no shortage of definitions, either. The following sections attempt to capture an appropriate cross section of what is meant by software architecture.

http://www.sei.cmu.edu/architecture/definitions.html
2005

Software architecture is a novel and still very immature branch of software engineering.

H. van Vliet:
Software Engineering –
Principles and Practice,
Wiley & Sons 2000

SW-Architektur

Definition

What Is Software Architecture?

We now define what *does* constitute a software architecture:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.[1]

"Externally visible" properties are those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on.

Let's look at some of the implications of this definition in more detail.

First, *architecture defines software elements.* The architecture embodies information about how the elements relate to each other. This means that it specifically *omits* certain information about elements that does not pertain to their interaction. Thus, an architecture is foremost an *abstraction* of a system that suppresses details of elements that do not affect how they use, are used by, relate to, or interact with other elements.

Second, the definition makes clear that *systems can and do comprise more than one structure* and that no one structure can irrefutably claim to be *the* architecture. By intention, the definition does not specify what the architectural elements and relationships are. Is a software element an object? A process? A library? A database? A commercial product? It can be any of these things and more.

Third, the definition implies that *every computing system with software has a software architecture* because every system can be shown to comprise elements and the relations among them.

Fourth, *the behavior of each element is part of the architecture* insofar as that behavior can be observed or discerned from the point of view of another element. Such behavior is what allows elements to interact with each other, which is clearly part of the architecture. This is another reason that the box-and-line drawings that are passed off as architectures are not architectures at all. They are simply box-and-line drawings—or, to be more charitable, they serve as cues to provide more information that explains what the elements shown actually do. When looking at the names of the boxes (database, graphical user interface, executive, etc.), a reader may well imagine the functionality and behavior of the corresponding elements. This mental image approaches an architecture, but it springs from the observer's mind and relies on information that is not present.

[1. Bass et al.]

SW-Architektur

Definition

What Software Architecture Is and What It Isn't

Figure 2.1, taken from a system description for an underwater acoustic simulation, purports to describe that system's "top-level architecture" and is precisely the kind of diagram most often displayed to help explain an architecture.

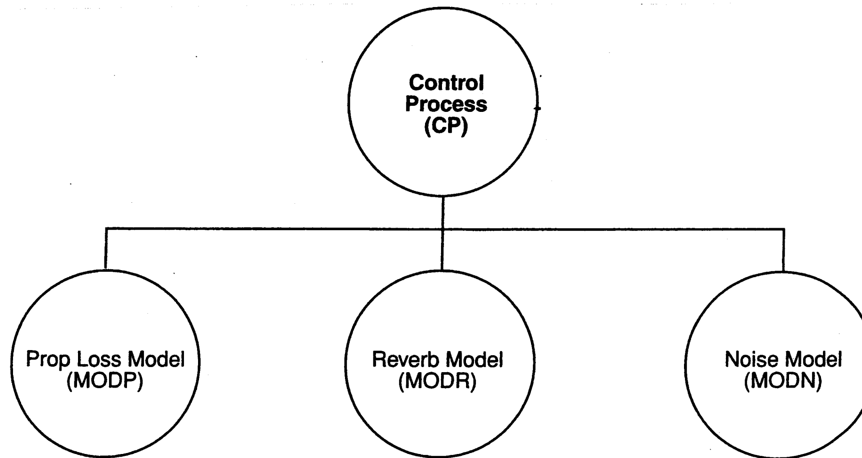


FIGURE 2.1 Typical, but uninformative, presentation of a software architecture

Is this an architecture? Assuming (as many definitions do) that architecture is a set of components (of which we have four) and connections among them (also present), this diagram seems to fill the bill. However, even if we accept the most primitive definition, what can we *not* tell from the diagram?

- *What is the nature of the elements?* What is the significance of their separation? Do they run on separate processors? Do they run at separate times? Do the elements consist of processes, programs, or both? Do they represent ways in which the project labor will be divided, or do they convey a sense of runtime separation? Are they objects, tasks, functions, processes, distributed programs, or something else?
- *What are the responsibilities of the elements?* What is it they do? What is their function in the system?
- *What is the significance of the connections?* Do the connections mean that the elements communicate with each other, control each other, send data to each other, use each other, invoke each other, synchronize with each other, share some information-hiding secret with each other, or some combination of these or other relations? What are the mechanisms for the communication? What information flows across the mechanisms, whatever they may be?
- *What is the significance of the layout?* Why is CP on a separate level? Does it call the other three elements, and are the others not allowed to call it? Does it contain the other three in an implementation unit sense? Or is there simply no room to put all four elements on the same row in the diagram?

We *must* raise these questions because unless we know precisely what the elements are and how they cooperate to accomplish the purpose of the system, diagrams such as these are not much help and should be regarded skeptically.

This diagram does not show a software architecture, at least not in any useful way.

[1. Bass et al.]

Software-Architektur

Definition

What is SW-Architecture?

Design and specification of overall system structure (beyond the choice of algorithms and data structures) including the issues of

- the organization of a system as a composition of components;
- global control structures;
- protocols for communication, synchronization and data access;
- assignment of functionality to design elements;
- the composition of design elements;
- physical distribution;
- scaling and performance;
- selection among design alternatives.

[10. Shaw, Garlan]

Software Architektur

Definition

Defining Software Architecture

Software architecture is a complex topic. Because of its complexity, our profession has produced a variety of definitions, each more or less useful depending on your point of view. Here is a definition from my first book, *Journey of the Software Professional:*

> A system architecture defines the basic "structure" of the system (e.g., the high-level modules comprising the major functions of the system, the management and distribution of data, the kind and style of its user interface, what platform(s) will it run on, and so forth).

This definition is pretty consistent with many others for example, [Bass], [Larman], and [POSA]. However, it lacks some important elements, such as specific technology choices and the required capabilities of the desired system. A colleague of mine, Myron Ahn, created the following definition of software architecture. It is a bit more expansive and covers a bit more ground than my original (2002, personal communication).

> Software architecture is the sum of the nontrivial modules, processes, and data of the system, their structure and exact relationships to each other, how they can be and are expected to be extended and modified, and on which technologies they depend, from which one can deduce the exact capabilities and flexibilities of the system, and from which one can form a plan for the implementation or modification of the system.

We could extend these definitions from the technical point of view, but this wouldn't provide a lot of value. More than any other aspect of the system, *architecture* deals with the "big picture." The real key to understanding it is to adopt this big picture point of view.

[7. Hohmann]

Software-Architektur

Definition

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. [Bass 98]

There are some key implications of this definition.

- Architecture is an abstraction of a system or systems. It represents systems in terms of abstract components that have externally visible properties and relationships (sometimes called *connectors,* although the notion of relationships is more general than connectors and can include temporal relationships, dependencies, uses relationships, and so forth).
- Because architecture involves abstraction it suppresses purely local information; private component details are not architectural.
- Systems are composed of many structures (commonly called *views*). No single view*) can appropriately represent anything but a trivial architecture. Furthermore, the set of views is not fixed or prescribed. An architecture should be described by a set of views that supports its analysis and communication needs.

*) Festlegung einer „essentiellen“ View; alle anderen akzidentell.

Software-Architektur

Motivation

Why is Software Architecture important?

1. *It is a vehicle for communication among stakeholders.**) Software architecture represents a common abstraction of a system that most if not all of the system's stakeholders can use as a basis for creating mutual understanding, forming consensus, and communicating with each other.
2. *It is the manifestation of the earliest design decisions.* The software architecture of a system is the earliest artifact that enables the priorities among competing concerns to be analyzed, and it is the artifact that has the most significant influence on system qualities. The tradeoffs between performance and security, between maintainability and reliability, and between the cost of the current development effort and the cost of future developments are all manifested in the architecture.
3. *It is a reusable, transferable abstraction of a system.* Software architecture constitutes a relatively small, intellectually graspable model for how a system is structured and how its components work together. This model is transferable across systems; in particular, it can be applied to other systems exhibiting similar requirements and can promote large-scale reuse and software product lines.

[5. Clements et al.]

*) customer, user, project manager, coder, tester, and so on

Software-Architektur

Was ist Softwarearchitektur?

Softwarearchitektur ist offenbar wichtig, aber was kann man sich darunter vorstellen? Die Architektur eines Softwaresystems hat mit der Architektur eines Gebäudes wenig gemeinsam: Die Architektur eines Gebäudes ist für jeden sichtbar, auch der Laie kann sich ein Urteil bilden. Wir würden aber Stromversorgung, Abwasserleitungen und Fahrstuhlschächte nicht als Teil der Architektur betrachten.

Die Architektur eines Softwaresystems ist umfassend; dazu gehört einfach alles: die Dialoge, Stapelverarbeitung, alle Softwarekomponenten und deren Zusammenspiel im kleinsten Detail. Die folgende Definition von Softwarearchitektur ist dank ihrer Allgemeinheit weitgehend akzeptiert. Sie stammt von [Bass et al. 1998] und wird oft zitiert (z.B. in [Bosch 2000] und [Starke 2002]):

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them.*

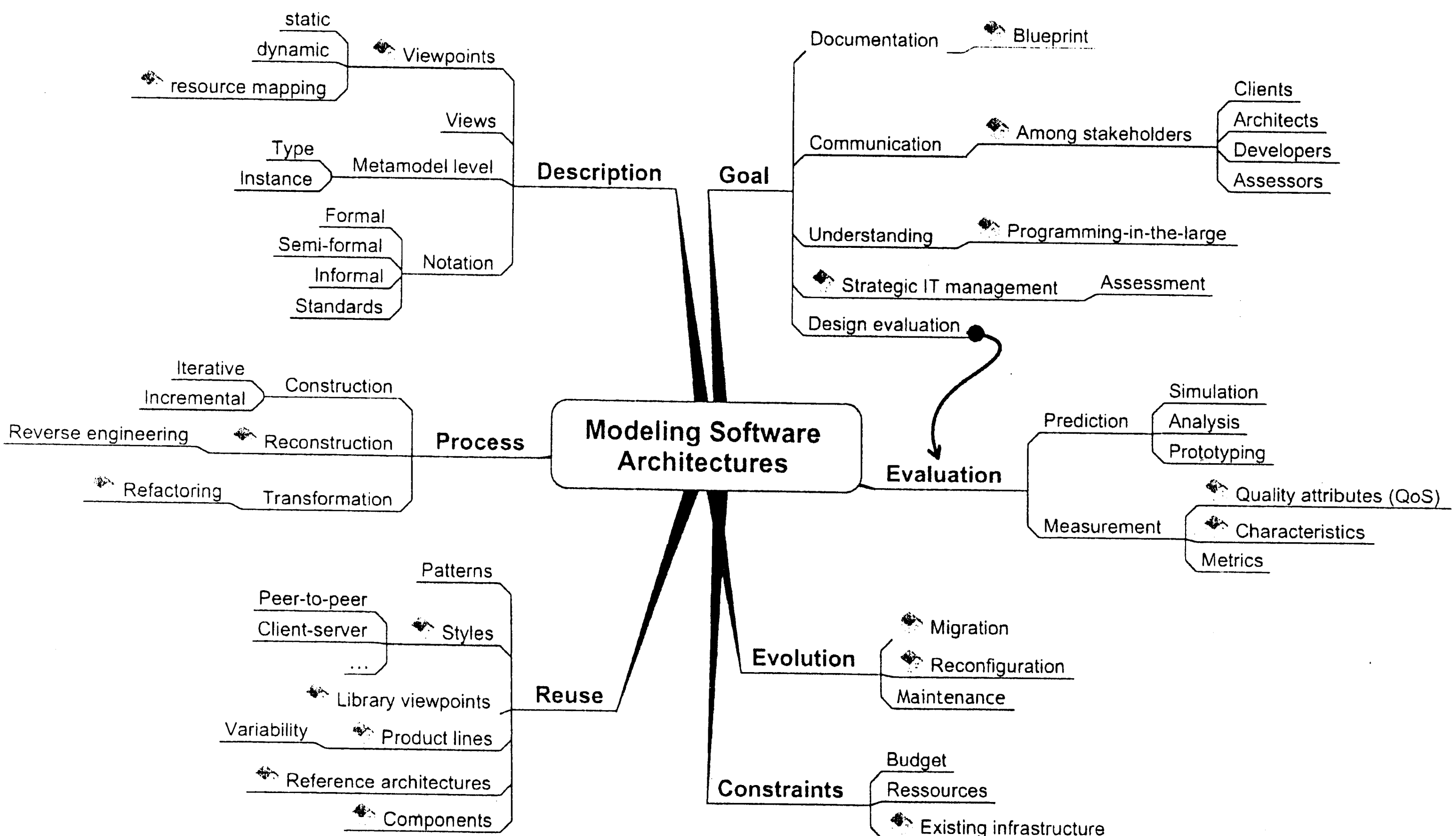
Das ist in dieser Allgemeinheit natürlich richtig. Allerdings wäre der Begriff »Softwarestruktur« treffender und etwas weniger hochtrabend. Trotzdem bleiben wir – der Mehrheit folgend – bei dem nicht ganz glücklichen Begriff der »Softwarearchitektur«.

Siedersleben, J.: Moderne Softwarearchitektur,
dpunkt-Verlag 2004

nicht einverstanden

Software-Architektur

Zweck und Aufgaben



Konzeptuelle Karte zur Modellierung von Software-Architekturen

GI-Arbeitskreis *Software-Architekturen.* 2006
http://se.informatik.uni-oldenburg.de/GIAKSoftArch/

Architektur-Definitionen
aus kognitiver Sicht

- Software architecture is a vehicle for communication among customers, users, project managers, coders, testers, and so on. It can be used as a basis for creating mutual understanding, forming consensus, and communicating with each other [5. Clements et al.].

- An architecture is foremost an abstraction of a system. When looking at the names of the boxes, a reader may well imagine the functionality and behaviour of the corresponding elements [1. Bass et al.].

- Considering components (and connectors), what is the significance of their separation? What is the significance of their connection? What is the significance of the layout? We *must* raise these questions because unless we know precisely what the elements are and how they cooperate to accomplish the purpose of the system, diagrams such as these are not much help and should be regarded skeptically [1. Bass et al.].

- This definition lacks some important elements, such as the required capabilities of the desired system. From software architecture one can deduce the exact capabilities and flexibilities of the system and form a plan for its implementation and modification [7. Hohmann].

- An architecture should be described by a set of views that supports its analysis and communication needs [5. Clements et al.].

Software-Architektur

Definition nach IEEE 1471-2000

„Software Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."

„Die Software-Architektur ist die grundlegende Organisation eines Systems, dargestellt durch dessen Komponenten, deren Beziehungen zueinander und zur Umgebung, sowie die Prinzipien, die den Entwurf und die Evolution des Systems bestimmen."

Aus: [Reussner et al.], Handbuch der Software-Architektur

Software-Architektur

Aufgaben

- „Eine Software-Architektur hat großen Einfluss auf die erfolgreiche Durchführung eines Software-Entwicklungsvorhabens. Der Beitrag einer Software-Architektur zum Erfolg ist umso größer, je umfangreicher, langwieriger und komplexer ein Software-Entwicklungsprojekt ist. Diese Beiträge liegen vor allem im Bereich des Projektmanagements.“

- „Eine gute Software-Architektur trägt zur <u>Effizienz des Entwicklungsprozesses</u> bei, indem sie Teilaufgaben voneinander entkoppelt und so Arbeitsteilung und eine flexible Projektorganisation ermöglicht. Sie gibt einen Rahmen vor, in dem <u>iterativ-inkrementell entwickelt</u> werden kann.“

Aus: [Reussner et al.], Handbuch der Software-Architektur

Architektur und Effizienz

Weitere Quellen:

Vitruv: Zehn Bücher über Architektur, Übersetzung von F. Reber 1865, Marix-Verlag 2004

H.-W. Kruft: Geschichte der Architekturtheorie, Verlag C.H. Beck 1986

G.W.F. Hegel: Ästhetik, 2 Bände, 3. Auflage, Aufbau-Verlag Berlin und Weimar 1976

Unterhaltungen mit dem Architekten Egon Jux (Köhlbrand-Brücke, z.B.: Der Spiegel, Nr. 38/1974)

Architektur

Die Architektur definiert den Zweck eines Gebäudes.

„Die erste Frage nun bei einem Bauwerk dieser Art ist die Frage nach seinem Zweck und Bestimmung sowie nach den Umständen, unter denen es zu errichten ist. Diesen angemessen seine Konstruktion zu machen, auf Klima, Lage, landschaftliche Naturumgebung zu achten und in der zweckmäßigen Berücksichtigung aller dieser Punkte ein zugleich zu freier Einheit verbundenes Ganzes hervorzubringen: das ist die allgemeine Aufgabe, in deren vollständiger Erfüllung sich der Sinn und Geist des Baukünstlers zu zeigen hat.“ (G.W.F. Hegel, 1820)

Zweck und Zweckmäßigkeit

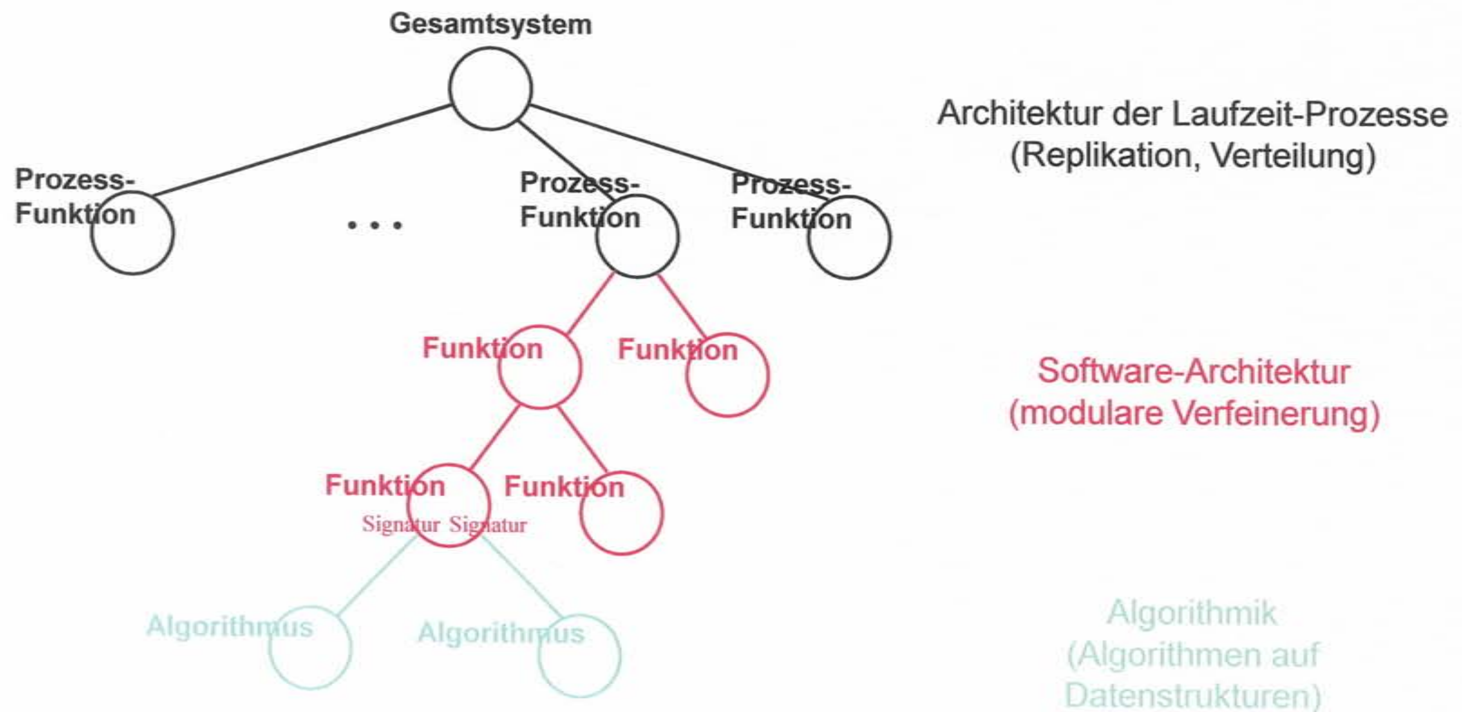
Die Zweckmäßigkeit einer Architektur zeigt sich darin, dass die Effizienz der im Gebäude ablaufenden Prozesse gewährleistet ist.

„Diese Anlagen müssen aber so gebaut werden, dass auf Festigkeit, Zweckmäßigkeit und Anmut Rücksicht genommen wird. Auf ... Zweckmäßigkeit wird Rücksicht genommen sein, wenn die Anordnung der Räume fehlerfrei ist und ohne Behinderung für die Benutzung und die Lage eines jeden Raumes nach seiner Art den Himmelsrichtungen angepasst und zweckmäßig ist.“ (Vitruv, 1. Jhd. n. Chr.)

„Zweckmäßigkeit betrifft die Nutzung von Gebäuden und die Garantie ungehinderter Funktionsabläufe.“
(H.-W. Kruft, Kommentar zu Vitruv)

Effizienz der Prozesse

Welche Software-Prozesse sind zu berücksichtigen?



Der Effizienz-Begriff

Der Begriff „Effizienz“ wird ab hier
im Sinne „ungehinderter Funktionsabläufe“ verwendet.

vgl. [Kruft 1986]

Algorithmen: Hier ist mit Effizienz die Laufzeitkomplexität gemeint, aber auch allgemein die beschleunigte Ausführung von Operationen durch zweckmäßige Anordnung der Operanden und Reihenfolge der Navigationsschritte und Operationen und, gegebenenfalls, deren Parallelisierung in Threads und Rekursionen. Bestimmt die Größenordnung der Laufzeit-Komplexität.

Laufzeit-Prozess: Hier äußert sich die Effizienz für Benutzer und Betreiber durch einen reibungsfreien Betrieb ohne Unterbrechungen, Wartezeiten und Fehler, im Wesentlichen erreicht durch die Replikation von Funktionen und Daten und deren Parallelität und Fehlerunabhängigkeit. Beeinflusst die Laufzeit-Komplexität um einen konstanten Faktor.

Entwicklungsprozess:
- *Modifizieren von Programmen:* Effizientes Ändern eines Programmes setzt voraus, dass dieses modular und in konstruktiver Voraussicht aufgebaut ist. Insbesondere Entwurfsmuster werden eingesetzt, wo Flexibilität für spätere Anpassungen verlangt ist.
- *Verstehen von Programmen:* Das Feststellen der Äquivalenz oder Verfeinerung von Artefakten ist algorithmisch, d.h. systematisch, grundsätzlich nicht lösbar und daher auf eine heuristische Lösung angewiesen. Damit diese effizient ablaufen kann, muss sowohl die Architektur als auch die Entwurfsmethode gewährleisten, dass die Heuristik frei von Unterbrechungen durch Kommunikation, Navigation oder Suchen bleibt. Ebenso ist abstraktes Verstehen einer Programmfunktion nicht systematisch möglich und daher durch aussagefähige Namensgebung vorwegzunehmen. Semantisches Verstehen einer Programmfunktion kostet vermutlich exponentiellen Aufwand, wenn die Programmelemente willkürlich angeordnet sind. Das Einführen einer Ordnung kann auch diesen Aufwand erheblich reduzieren.

Effiziente Software-Architektur

- Bevor wir hier nach der „Effizienz“ fragen, die durch die Software-Architektur gewährleistet werden soll, stellen wir die Frage:

**An wen ist die Software-Architektur gerichtet –
an die ausführende Maschine oder an den Menschen?**

- Im Folgenden überzeugen wir uns, dass die Software-Architektur nach dem Übersetzungsvorgang für die Maschine nicht mehr existent ist. Durch die SWA wird auch nichts eingeführt, was für die Laufzeiteffizienz der Maschine relevant wird, nachdem die SWA im Verlauf des Übersetzungsvorgangs wieder verschwunden ist.

Software-Architektur und Maschine

Übersetzungshierarchie für die Sprache C

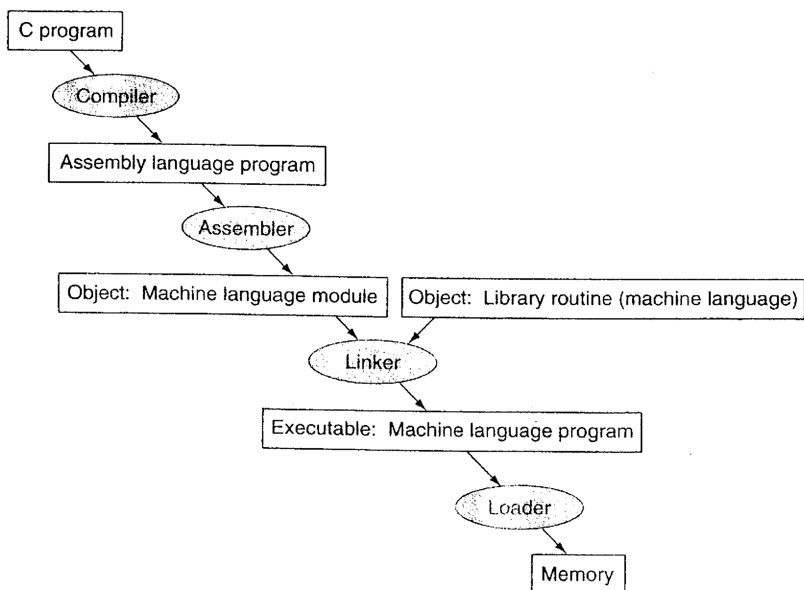


FIGURE 2.28 A translation hierarchy for C.

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

Figure B.12 Categories of instruction operators and examples of each.

Aus: (1) J.L. Hennessy, D.A. Patterson: Computer Architecture – A Quantitative Approach, Fourth Edition, Morgan Kaufmann Publishers, 2007
(2) D.A. Patterson, J.L. Hennessy: Computer Organization and Design – The Hardware/Software Interface, Third Edition, Morgan Kaufmann Publishers, 2007

Software-Architektur und Maschine

Eine konkrete Maschinen-Schnittstelle (1)

Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR*
LB, LBU, SB	Load byte, load byte unsigned, store byte (to/from integer registers)
LH, LHU, SH	Load half word, load half word unsigned, store half word (to/from integer registers)
LW, LWU, SW	Load word, load word unsigned, store word (to/from integer registers)
LD, SD	Load double word, store double word (to/from integer registers)
L.S, L.D, S.S, S.D	Load SP float, load DP float, store SP float, store DP float
MFC0, MTC0	Copy from/to GPR to/from a special register
MOV.S, MOV.D	Copy one SP or DP FP register to another FP register
MFC1, MTC1	Copy 32 bits to/from FP registers from/to integer registers
Arithmetic/logical	*Operations on integer or logical data in GPRs; signed arithmetic trap on overflow*
DADD, DADDI, DADDU, DADDIU	Add, add immediate (all immediates are 16 bits); signed and unsigned
DSUB, DSUBU	Subtract; signed and unsigned
DMUL, DMULU, DDIV, DDIVU, MADD	Multiply and divide, signed and unsigned; multiply-add; all operations take and yield 64-bit values
AND, ANDI	And, and immediate
OR, ORI, XOR, XORI	Or, or immediate, exclusive or, exclusive or immediate
LUI	Load upper immediate; loads bits 32 to 47 of register with immediate, then sign-extends
DSLL, DSRL, DSRA, DSLLV, DSRLV, DSRAV	Shifts: both immediate (DS__) and variable form (DS__V); shifts are shift left logical, right logical, right arithmetic
SLT, SLTI, SLTU, SLTIU	Set less than, set less than immediate; signed and unsigned

Figure 1.5 Subset of the instructions in MIPS64. SP = single precision; DP = double precision.

Aus: [J.L. Hennessy, D.A. Patterson, 2007]

Software-Architektur und Maschine

Eine konkrete Maschinen-Schnittstelle (2)

Control	*Conditional branches and jumps; PC-relative or through register*
BEQZ, BNEZ	Branch GPRs equal/not equal to zero; 16-bit offset from PC + 4
BEQ, BNE	Branch GPR equal/not equal; 16-bit offset from PC + 4
BC1T, BC1F	Test comparison bit in the FP status register and branch; 16-bit offset from PC + 4
MOVN, MOVZ	Copy GPR to another GPR if third GPR is negative, zero
J, JR	Jumps: 26-bit offset from PC + 4 (J) or target in register (JR)
JAL, JALR	Jump and link: save PC + 4 in R31, target is PC-relative (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address
ERET	Return to user code from an exception; restore user mode
Floating point	*FP operations on DP and SP formats*
ADD.D, ADD.S, ADD.PS	Add DP, SP numbers, and pairs of SP numbers
SUB.D, SUB.S, SUB.PS	Subtract DP, SP numbers, and pairs of SP numbers
MUL.D, MUL.S, MUL.PS	Multiply DP, SP floating point, and pairs of SP numbers
MADD.D, MADD.S, MADD.PS	Multiply-add DP, SP numbers, and pairs of SP numbers
DIV.D, DIV.S, DIV.PS	Divide DP, SP floating point, and pairs of SP numbers
CVT._._	Convert instructions: CVT.x.y converts from type x to type y, where x and y are L (64-bit integer), W (32-bit integer), D (DP), or S (SP). Both operands are FPRs.
C.__.D, C.__.S	DP and SP compares: "__" = LT,GT,LE,GE,EQ,NE; sets bit in FP status register

Figure 1.5 Subset of the instructions in MIPS64.

Aus: [J.L. Hennessy, D.A. Patterson, 2007]

Software-Architektur und Maschine

Verbreitete Prozessoren

	ARM	Thumb	SuperH	M32R	MIPS16
Date announced	1985	1995	1992	1997	1996
Instruction size (bits)	32	16	16	16/32	16/32
Address space (size, model)	32 bits, flat	32 bits, flat	32 bits, flat	32 bits, flat	32/64 bits, flat
Data alignment	Aligned	Aligned	Aligned	Aligned	Aligned
Data addressing modes	6	6	4	3	2
Integer registers (number, model, size)	15 GPR x 32 bits	8 GPR + SP, LR x 32 bits	16 GPR x 32 bits	16 GPR x 32 bits	8 GPR + SP, RA x 32/64 bits
I/O	Memory mapped	Memory mapped	Memory mapped	Memory mapped	Memory mapped

Figure J.2 Summary of five recent architectures for embedded applications.

Alle gängigen Prozessoren arbeiten im Wesentlichen nach demselben Prinzip: auf der Basis einer Von-Neumann-Architektur mit einem Satz von Registern und Befehlen. (vgl. M. Berekovic et al.: Rekonfigurierbare Architekturen, Informatik-Spektrum 31 4, Springer 2008)

Aus: [J.L. Hennessy, D.A. Patterson, 2007]

Software-Architektur und Maschine

Übersetzung der Sprache Java

Category	Operation	Java bytecode	Size (bits)	MIPS instr.	Meaning
Arithmetic	add	iadd	8	add	NOS=TOS+NOS; pop
	subtract	isub	8	sub	NOS=TOS-NOS; pop
	increment	iinc I8a I8b	8	addi	Frame[I8a]= Frame[I8a] + I8b
Data transfer	load local integer/address	iload I8/aload I8	16	lw	TOS=Frame[I8]
	load local integer/address	iload_/aload_{0,1,2,3}	8	lw	TOS=Frame[{0,1,2,3}]
	store local integer/address	istore I8/astore I8	16	sw	Frame[I8]=TOS; pop
	load integer/address from array	iaload/ aaload	8	lw	NOS=*NOS[TOS]; pop
	store integer/address into array	iastore/aastore	8	sw	*NNOS[NOS]=TOS; pop2
	load half from array	saload	8	lh	NOS=*NOS[TOS]; pop
	store half into array	sastore	8	sh	*NNOS[NOS]=TOS; pop2
	load byte from array	baload	8	lb	NOS=*NOS[TOS]; pop
	store byte into array	bastore	8	sb	*NNOS[NOS]=TOS; pop2
	load immediate	bipush I8, sipush I16	16, 24	addi	push; TOS=I8 or I16
	load immediate	iconst_{-1,0,1,2,3,4,5}	8	addi	push; TOS={-1,0,1,2,3,4,5}
Logical	and	iand	8	and	NOS=TOS&NOS; pop
	or	ior	8	or	NOS=TOS|NOS; pop
	shift left	ishl	8	sll	NOS=NOS << TOS; pop
	shift right	iushr	8	srl	NOS=NOS >> TOS; pop
Conditional branch	branch on equal	if_icompeq I16	24	beq	if TOS == NOS, go to I16; pop2
	branch on not equal	if_icompne I16	24	bne	if TOS != NOS, go to I16; pop2
	compare	if_icomp{lt,le,gt,ge} I16	24	slt	if TOS {<,<=,>,>=} NOS, go to I16; pop2
Unconditional jump	jump	goto I16	24	j	go to I16
	return	ret, ireturn	8	jr	
	jump to subroutine	jsr I16	24	jal	go to I16; push; TOS=PC+3
Stack management	remove from stack	pop, pop2	8		pop, pop2
	duplicate on stack	dup	8		push; TOS=NOS
	swap top 2 positions on stack	swap	8		T=NOS; NOS=TOS; TOS=T
Safety check	check for null reference	ifnull I16, ifnotnull I16	24		if TOS {==,!=} null, go to I16
	get length of array	arraylength	8		push; TOS = length of array
	check if object a type	instanceof I16	24		TOS = 1 if TOS matches type of Const[I16]; TOS = 0 otherwise
Invocation	invoke method	invokevirtual I16	24		Invoke method in Const[I16], dispatching on type
Allocation	create new class instance	new I16	24		Allocate object type Const[I16] on heap
	create new array	newarray I16	24		Allocate array type Const[I16] on heap

FIGURE 2.14.1 Java bytecode architecture versus MIPS.

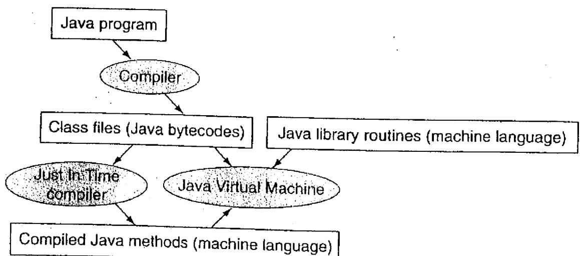


FIGURE 2.30 A translation hierarchy for Java.

Aus: [D.A. Patterson, J.L. Hennessy, 2007]

Software-Architektur und Maschine

Eine Java-Methode

Method body				
Move parameters		move	$s2, $a0	# copy parameter $a0 into $s2 (save $a0)
Test ptr null		beq	$a0,$zero,NullPointer	#if $a0==0, goto Error
Get array length		lw	$s3,4($a0)	# $s3 = length of array v
Outer loop	 for1tst: 	move slt beq	$s0, $zero $t0, $s0, $s3 $t0, $zero, exit1	# i = 0 # reg $t0 = 0 if $s0 ≥ $s3 (i ≥ n) # go to exit1 if $s0 ≥ $s3 (i ≥ n)
Inner loop start	 for2tst: 	addi slti bne	$s1, $s0, -1 $t0, $s1, 0 $t0, $zero, exit2	# j = i - 1 # reg $t0 = 1 if $s1 < 0 (j < 0) # go to exit2 if $s1 < 0 (j < 0)
Test if j too big		slt beq	$t0,$s1,$s3 $t0,$zero,IndexOutOfBounds	# Temp reg $t0 = 0 if j >= length # if j >= length, goto Error
Get v[j]		sll add lw	$t1, $s1, 2 $t2, $s2, $t1 $t3, 0($t2)	# reg $t1 = j * 4 # reg $t2 = v + (j * 4) # reg $t3 = v[j]
Test if j+1 < 0 or if j+1 too big		addi slt bne slt beq	$t1,$s1,1 $t0,$t1,$zero $t0,$zero,IndexOutOfBounds $t0,$t1,$s3 $t0,$zero,IndexOutOfBounds	# Temp reg $t1 = j+1 # Temp reg $t0 = 1 if j+1 < 0 # if j+1 < 0, goto Error # Temp reg $t0 = 0 if j+1 >= length # if j+1 >= length, goto Error
Get v[j+1]		lw	$t4, 4($t2)	# reg $t4 = v[j + 1]
Load method table		lw	$t5,0($a0)	# $t5 = address of method table
Get method addr		lw	$t5,8($t5)	# $t5 = address of first method
Pass parameters		move move	$a0, $t3 $a1, $t4	# 1st parameter of compareTo is v[j] # 2nd param. of compareTo is v[j+1]
Set return addr		la	$ra,L1	# load return address
Call indirectly		jr	$t5	# call code for compareTo
Test if should skip swap	L1:	slt beq	$t0, $zero, $v0 $t0, $zero, exit2	# reg $t0 = 0 if 0 ≥ $v0 # go to exit2 if $t4 ≥ $t3
Pass parameters and call swap		move move jal	$a0, $s2 $a1, $s1 swap	# 1st parameter of swap is v # 2nd parameter of swap is j # swap code shown in Figure 2.34
Inner loop end		addi j	$s1, $s1, -1 for2tst	# j -= 1 # jump to test of inner loop
Outer loop	exit2:	addi j	$s0, $s0, 1 for1tst	# i += 1 # jump to test of outer loop

FIGURE 2.14.5 MIPS assembly version of the method body of the Java version of `sort`.

Aus: [J.L. Hennessy, D.A. Patterson, 2007]

Aufgabe der Software-Architektur

Die Software-Architektur richtet sich ausschließlich an den Menschen – keinesfalls an die ausführende Maschine.

- Jedes Programm wird auf die Maschinenbefehle eines Prozessors abgebildet, der keine anderen Strukturen kennt als Befehle und deren Hintereinanderausführung, Sprungbefehle für die Flusskontrolle und Unterprogrammsprünge. Das heißt, dass jede darüberhinausgehende Architektur beim Übersetzungsvorgang verloren geht und im ausführbaren Programm nicht mehr sichtbar ist. So kann der Prozessor auch keine Programmiersprachen unterscheiden und daher noch nicht einmal, ob das Programm objekt-orientiert ist oder nicht.

- *Weiterer Hinweis:* Auch eine höhere Modularisierungsstruktur in Form des Produktbaums (Archive, DLLs, Quell- und Objekt-Verzeichnisse) geht beim üblichen „flachen Linken“ verloren.

- *Weiterer Hinweis:* Die Gödelnummer einer Turing-Maschine (vergleichbar mit der .exe-Datei eines Programms) besitzt außer der Zeilenstruktur der Zustandsübergangsfunktion, inkl. Sprüngen in Untermaschinen, keine weitere Architektur. Gleichzeitig repräsentiert sie die vollständige Funktionalität.

- **Schlussfolgerung:**

  a) Eine wie auch immer geartete Software-Architektur ist für die ausführende Maschine unsichtbar.

  b) Die Aufgabe der Software-Architektur kann somit nur darin bestehen, den Menschen (stakeholder) beim Verstehen*) eines Programms – als Grundvoraussetzung jeder menschlichen Aktivität im Software-Entwicklungsprozess – zu unterstützen.

---

*) Auch die Modularisierung dient in erster Linie der begrifflichen Verfeinerung und damit dem Verständnis, und erst in zweiter Linie, sozusagen als Nebeneffekt, der Flexibilität beim Einbau und Austausch der Module.

These

Software-Architektur definiert den Zweck eines Programms*).

- **Vor der Implementierung dient diese Definition als <u>Requirements Specification</u> und <u>Plan</u> für die Implementierung.**
- **Nach der Implementierung dient sie als Grundlage für jede weitere Aktivität im Softwarelebenszyklus, die in jedem Fall das <u>Verstehen</u> des Zwecks des Programms oder eines seiner Teile zwingend voraussetzt.**

Die Software-Architektur ist also keine vorübergehende Erscheinung im Soft-ware-Lebenszyklus, sondern ständig präsent. Konsequenterweise sollte die Software-Architektur in einer Notation festgelegt sein, die als Grundlage für den gesamten Lebenszyklus dient.

*) … genau so, wie die Bauwerksarchitektur den Zweck eines Gebäudes definiert.

„Zweck“

in diesem Zusammenhang etwa gleichbedeutend mit

---

(1) Mit „Programmelementen“ sind Operationen und Operanden unterschiedlicher Abstraktionsgrade gemeint.
(2) Da der Begriff „Design“ hier in doppelter Bedeutung zutrifft – einmal als „Entwurf“, zum andern als „Absicht“ – werden wir Information, die den Zweck eines Programms definiert (Name, Kommentar, etc.) im Folgenden als ***Design-Information*** bezeichnen.

Software-Architektur ist eine Abstraktion der Implementierung

- Dieser bekannte Lehrsatz verwendet den anspruchsvollen, aber oft nur umgangssprachlich gebrauchten Begriff der „Abstraktion". Mit diesem Begriff sind weitere Begriffe verbunden, die im Zusammenhang mit Software-Architektur eine Rolle spielen, und die deshalb im Folgenden – zum Teil auf Wörterbuch-Niveau, zum Teil versuchsweise an unseren fachlichen Kontext angepasst – eingeführt werden sollen:

- *Die Architektur ist essentiell (wesentlich), die Implementierung ist akzidentell (zufällig).* Die im Titel aufgestellte Gleichung legt nahe, dass die Architektur das Essentielle (das Wesen) einer berechenbaren Funktion darstellt, die Implementierung das Akzidentelle (das Zufällige). Bei Plato wird das „Wesen" aufgefasst als das *Bleibende*, das Urbild des im Wechsel der Erscheinungen sich zeigende Abbild von etwas (Meiner, WB der phil. Begriffe). Das Akzidentelle umfasst dagegen, nach Aristoteles, die *zufälligen, wechselnden, unwesentlichen* Eigenschaften eines Gegenstandes (siehe ebenfalls: Meiner).

- Das *Bleibende* in unserem Kontext ist sicher die *Anforderung* an die Funktionalität eines Programms (gleichbedeutend mit seinem Nutzen, der dahinterliegenden Absicht, seinem Sinn und Zweck, seiner Funktion). Diese Anforderung, und somit das Wesen eines Programms, soll durch die Architektur widergespiegelt werden.

- Von der Implementierung eines Programms weiß man dagegen, dass sie lediglich *eine* von grundsätzlich *unendlich vielen funktionsgleichen Implementierungen* darstellt, und somit die *zufälligen, wechselnden, unwesentlichen* Eigenschaften eines Programms widerspiegelt.

- Durch Abstraktion (Absehen) von den zufälligen, wechselnden, unwesentlichen Eigenschaften einer Implementierung gelangt man zum Wesen des Programms, dargestellt durch die Architektur. Das ist die Aussage des obigen Lehrsatzes[‡].

- Der Abstraktionsvorgang, der einer Klassifizierung der Programmfunktion entspricht, ist ein nicht-berechenbarer Vorgang (siehe unten: Satz von Rice).

- Aus dem Kontext ist ersichtlich, dass das *Wesen* eines Programms[§] gleich ist der mit ihm verbundenen *Absicht*, seinem *Sinn* und *Zweck*, seinem *Nutzen*, den *Anforderungen* an das Programm und, hier: seiner *Funktion*.

[‡] Die Architektur ist noch in einer weiteren Hinsicht eine Abstraktion, nämlich ein „Modell der realen Welt", ein Abbild einer Anwendungsdomäne. In diesem Zusammenhang sind zu untersuchen: der Zeichenaspekt der Sigmatik und Homomorphismen.

[§] Zur besseren Überprüfbarkeit einiger Aussagen über Programme verallgemeinern wir sie versuchsweise auf beliebige Gegenstände.

Die Designinformation zur Erklärung des Zwecks eines Programms ist vor und nach der Implementierung dieselbe

- Anforderung = Zweck = Nutzen = Funktion. Diese Gleichung gilt, wenn nach der Implementierung des Programms die Anforderungen erfüllt sind. Das heißt, dass die Spezifikation der Anforderungen geeignet ist, auch nach der Implementierung des Programms seine Funktion zu erklären.

- Mit „Anforderungen" sind hier die Nutzer- oder Kundenanforderungen gemeint. Sie sind für die *Beurteilung (Validation) der Architektur* wesentlich. (Do we construct the *right* program?)

- Während der Implementierung wird in der Regel weitere (akzidentelle) Designinformation produziert. Sie gibt die Absicht des Implementierers – seine Lösungs-Trasse – wieder, der versucht, die durch die Architektur ausgedrückte Absicht zu erfüllen.

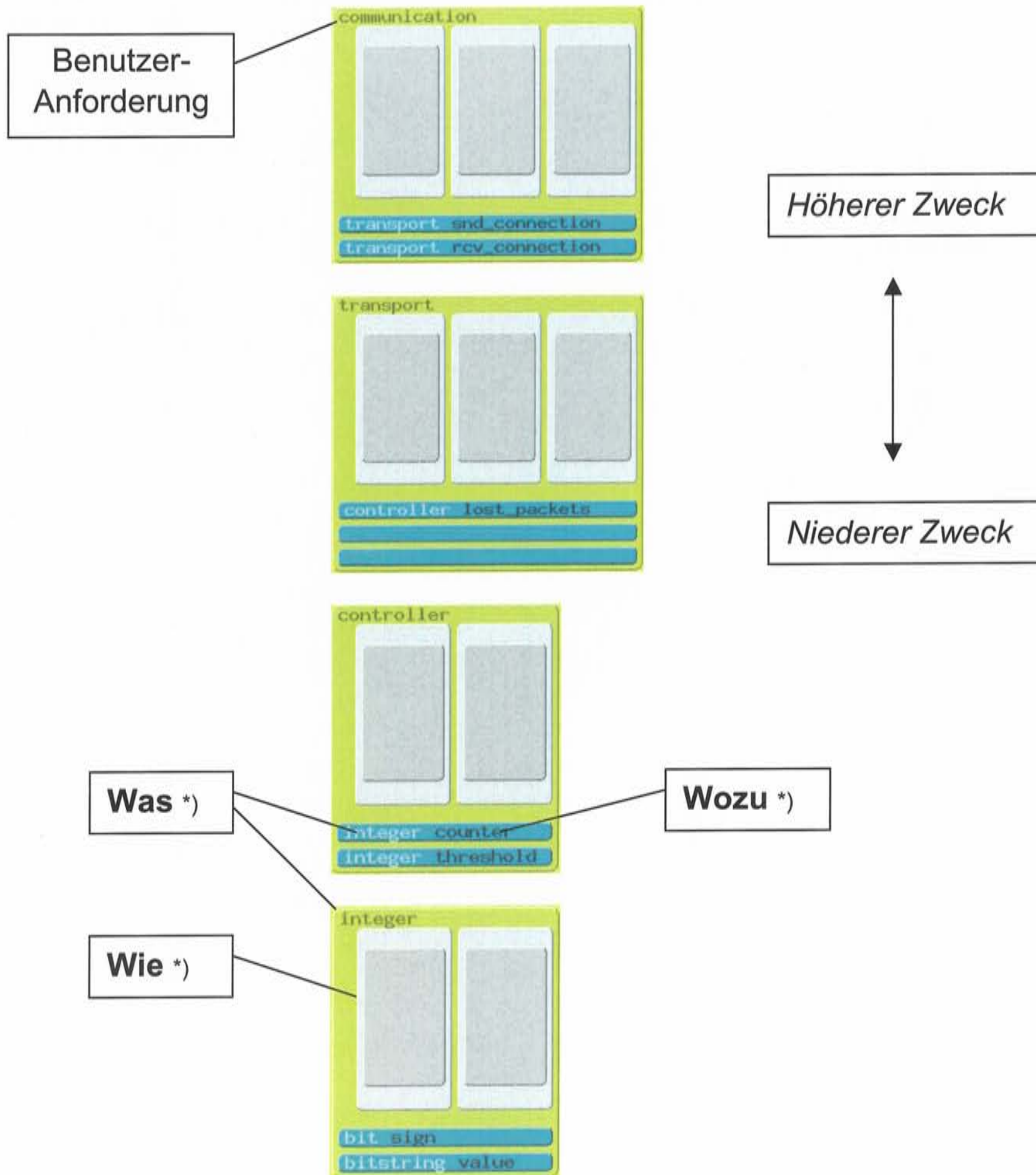
- Die Gesamtdokumentation eines Programms nach seiner Implementierung enthält somit *essentielle und akzidentelle Designinformation* (-> Views), die bewusst unterschieden werden müssen.

- Für eine *Beurteilung (Verifikation) der Implementierung* muss essentielle und akzidentelle Designinformation gegeneinandergehalten werden. (Do we construct the program *right*?)

- Der Zweck, d.h. das Wesen, eines Programms wird somit nach seiner Implementierung durch dieselbe essentielle Designinformation ausgedrückt, die schon vor seiner Implementierung vorhanden war.

Zweckbestimmung

Hierarchische Verfeinerung



*) *'Was'* fragt danach, um welches Objekt es sich handelt; *'wie'* fragt danach, wie das Objekt implementiert ist, dh. wie es funktioniert; *'wozu'* fragt, welchem Zweck das Objekt dient. Das *'was'* stellt wieder den höheren Zweck des *'wie'* dar.

Welches grundsätzliche Problem muss die Software-Architektur lösen?

Sie muss den Zweck, d.h. die Funktion eines Programms verständlich machen:

- vor der Implementierung: als Anforderung
- nach der Implementierung: als Erklärung

Menschlicher Verstand*)

Adressat der Software-Architektur

- Nach der Turing-Church'schen These ist jede im intuitiven Sinne berechenbare Funktion auch Turing-berechenbar.
- Dies impliziert, dass der Verstand, bevor er die berechenbare Funktion als Turingmaschine formalisiert, eine intuitive Vorstellung von ihrer Berechnung haben muss.
- Die Turingmaschine spiegelt dann auf die präzisest mögliche Weise die intuitive Vorstellung von der Berechnung einer Funktion wieder.
- Jede zeit/speicher-effizientere intuitive Vorstellung von der Berechnung einer Funktion erlaubt auch eine effizientere Formalisierung als Turingmaschine.
- Wenn keine effizientere Turingmaschine gefunden werden kann, dann gibt es auch keine effizientere intuitive Vorstellung von der Berechnung der Funktion.
- Wenn überhaupt keine Turingmaschine gefunden werden kann, dann gibt es auch überhaupt keine intuitive Vorstellung von der Berechnung der Funktion.

Schlussfolgerung:

- Jede Komplexitätsaussage über Turingmaschinen ist auch eine Komplexitätsaussage über die intuitive Berechenbarkeit.
- Jede Nichtberechenbarkeitsaussage über Turingmaschinen ist auch eine intuitive Nichtberechenbarkeitsaussage.

---

*) Gemeint ist hier nicht der „vegetative", unwillkürliche Verstand, der z.B. der menschlichen Sprach- und Bildverarbeitung zugrundeliegt und ein weitgehend unbekanntes, der Turingmaschine vermutlich weit überlegenes Berechnungsmodell verwendet. Wir sprechen im Folgenden über den bewussten, willkürlichen Verstand – den Verstand, der sich Algorithmen ausdenkt.

Verstand und seine Komplexität

- Falls ein Problem nicht Turing-berechenbar ist, falls es also dafür keinen Algorithmus gibt, dann kann dieses Problem auch vom Verstand nicht systematisch gelöst werden. Er muss dann versuchen, „heuristisch“ eine Lösung zu finden.

  - ***heuristic.*** *A process, such as trial and error, for solving a problem for which no algorithm exists. A heuristic for a problem is a rule or method for approaching a solution. (S. Blackburn, The Oxford Dictionary of Philosophy, 2005)*

    [vielleicht besser: for which no efficient algorithm exists]

- Es gibt im Laufe der Softwareentwicklung eine Reihe von nicht-entscheidbaren Problemen bei der Handhabung (Äquivalenz, Verfeinerung, Klassifizierung, Verifikation, Validierung) von Softwareartefakten. Wir überlegen, inwiefern die Architektur derartige Probleme umgehen bzw. ihre heuristische Lösung unterstützen kann.

- Wenn sich dasVerstehen (Nachvollziehen) von semantischen Strukturen als Graphenisomorphieproblem darstellen lässt (etwa als Überdeckung von erwarteter zu vorgefundener Struktur), dann ist im Allgemeinen der Aufwand hierfür NP-schwer. Wir überlegen, inwiefern sich dieser Aufwand durch die Architektur reduzieren lässt.

Prinzipielle Grenzen des Verstehens

Satz von Rice

Theorem (H.G. Rice, 1953)
Let R be the set of all turing-computable functions and let S be a subset of R with $S \neq \emptyset$ and $S \neq R$. Then the language $C(S) = \{w \mid f_{M_w} \in S\}$ is undecideable.

By the theorem of Rice on Turing-machines it is generally impossible to read a function from a syntactic structure. In the following formulation of the theorem, w represents the encoding of a Turing-table (the 'program'), M_w represents the corresponding Turing-machine executing w, and f_{M_w} represents the function computed by M_w. Then the theorem states that for any given non-trivial class S of functions it is impossible to decide algorithmically whether a function[1] given by its Turing-table w belongs to S. It is astonishing that this negative statement holds even for the most simple functional classes, as e.g. the class of constant functions.

The problem above, which is shown to be undecideable, is a problem of classification or abstraction. Its concern is the transition from the objects (or extent) of a set to the attributes of the objects (or intent) of a set which, in turn, means assigning a class name to a function. Without knowing the name of a function it is therefore generally impossible to find out its intended purpose. Thus, the assignment of meaning to an unnamed structure[2] forever remains a matter of interpretation ('interpretation' in the colloquial user-related sense).

Es ist somit generell nicht entscheidbar, welchem Zweck eine Turing-Maschine dient.

[1] Strictly speaking, the classification S in the theorem includes far more than ‚functionality'. Any non-trivial property of Turing-machines that can be encoded as a functional class is shown to be undecideable (for restrictions/exceptions, see [17]); for example, the property that the language accepted by TM M is finite. A property is called ‚trivial' if it adheres to none or all of the Turing-machines.
[2] Informally, the statement of the theorem also applies to mechanical structures as shown in Figure 2.8, or even to abstract paintings.

Satz von Rice

Konsequenzen für die Software-Architektur 1

- Wir überzeugen uns, dass die Aussage des obigen Satzes nicht nur für Turing-Tafeln gilt, sondern für beliebige *textuelle und visuelle algorithmische Strukturen.*

  Folgerung 1:
  Einer unbeschrifteten algorithmischen Struktur kann man grundsätzlich nicht ansehen, welche Funktion sie darstellt.

  Plausibilität:
  Sei *P* ein beliebiges textuelles oder visuelles Programm. Ersetze in *P* alle erklärenden Namen von Methoden, Objekten, etc. durch Identifier. Das resultierende Programm P', jetzt reine syntaktische Struktur ohne semantische Hinweise, ist weiterhin funktional äquivalent zu *P*. Könnte man P' ansehen, welche Funktion es darstellt, dann hätten wir damit auch eine solche Möglichkeit für Turing-Tafeln gefunden, denn offensichtlich lässt sich jede Turing-Tafel algorithmisch in ein textuelles oder visuelles Programm gleicher Funktionalität übersetzen und anschließend wie P' auf seine Funktionalität hin überprüfen. Dies steht aber im Widerspruch zu dem obigen Satz. Also muss die Annahme, P' könnte man seine Funktion ansehen, falsch sein □

- Wir überzeugen uns weiter, dass die Aussage des obigen Satzes auch für beliebige *architektonische* Strukturen gilt.

  Folgerung 2:
  Einer unbeschrifteten architektonischen Struktur kann man grundsätzlich nicht ansehen, welche Funktion sie darstellt.

  Plausibilität:
  Wir führen dieses architektonische Problem auf das algorithmische in Folgerung 1 zurück. Angenommen, wir könnten einer unbeschrifteten architektonischen Struktur ansehen, welche Funktion sie darstellt. Dann wäre es auch möglich, eine beliebige unbeschriftete algorithmische Struktur *P* in eine unbeschriftete architektonische Struktur *A* einzubetten – und zwar in stets dieselbe einfache Architektur, die ganz sicher die Funktionalität der Methode *P* unverändert lässt, etwa eine umgebende Klassen-Definition – und anschließend für die so erhaltene unbeschriftete Gesamtarchitektur *A(P)* gemäß unserer Annahme zu bestimmen, welche Funktion sie darstellt. Da die dargestellte Funktion für *A(P)* und *P* stets dieselbe ist, hätten wir so ein Verfahren gefunden, für eine beliebige unbeschriftete algorithmische Struktur die Funktion zu bestimmen. Widerspruch zu Folgerung 1. Also muss unsere Annahme, wir könnten einer unbeschrifteten architektonischen Struktur ansehen, welche Funktion sie darstellt, falsch sein □

Satz von Rice

Konsequenzen für die Software-Architektur 2

- Abstraktes Verstehen, Klassifikation, Erkennen einer Absicht, Verstehen von Sinn und Zweck, sind nicht-berechenbare Vorgänge. Ohne sofortige Dokumentation einer Absicht oder eines Zwecks durch den Autor ist die ursprüngliche Absicht bzw. der Zweck später grundsätzlich nicht mehr rekonstruierbar (selbst durch den Autor nicht).

- Die Dokumentation muss den Zweck eines Programmbausteins durch geeignete Namen und Kommentare wiedergeben. Die Wahl der Namen und Kommentare ist eine eigene Architektur- bzw. Ingenieursleistung.

- Architektur ist mehr als nur „Struktur". Die Aussagekraft einer Architektur ist entscheidend von textueller Information abhängig. Die Struktur dient dabei als Träger der textuellen Information, so wie syntaktische Struktur als Träger von Wörtern eines natürlichen Satzes dient. Die Architektur eines Programms ist eine „Aussage" über den Zweck des Programms. Die linguistische Auffassung eines Programms als zusammengesetzter Begriff (bereits angedeutet durch den Programmbaustein „Klasse") gibt die Probleme und Aufgaben der Software-Architektur recht gut wieder.

- Die Plazierung der Begriffe spielt eine wesentliche Rolle für die Aussagekraft einer Architektur. Unmotivierte Anordnung von Programmbausteinen erschwert das Verständnis.

- Der Satz von Rice zeigt auch die Grenzen der Visualisierung auf. Zum Verständnis einer Visualisierung ist immer Text nötig (sofern es kein freies Erzeugendensystem für die Konstruktion von Icons gibt).

- Anhand des Satzes von Rice lässt sich der Vorgang des Verstehens differenzieren. Daraus ergeben sich auf natürliche Weise Qualifikationen, Aufgaben und Sprachen, die im Software-Entwicklungsprozess eine Rolle spielen.

Satz von Rice

Konsequenzen für die Software-Architektur 3

- **Die Software-Architektur liegt vollständig außerhalb des ausführbaren Programms, und existiert von ihm getrennt bzw. existiert überhaupt nicht.**

- Ein ausführbares Programm (z.B. .exe-Datei) beinhaltet keinerlei Information darüber, welche Funktion es ausführt.

- Ein ausführbares Programm (äquivalent zur Turing-Maschine) kann sich somit nicht selbst erklären. Insbesondere kann keine Dokumentation, die etwas über den Zweck eines Programms aussagt, automatisch generiert werden.

- Nur der Autor des Programms ist – solange er sich intensiv damit beschäftigt – in der Lage, den von ihm beabsichtigten Zweck des Programms durch eine zusätzliche Dokumentation zu erklären. Diese Dokumentation ist identisch mit der Architektur.

- Zu ein und dem selben ausführbaren Programm kann es somit beliebig (genau genommen: unendlich) viele verschiedene Architekturen geben. Alle diese Architekturen können dem ausführbaren Programm zugeordnet, aber auch wieder weggenommen werden.

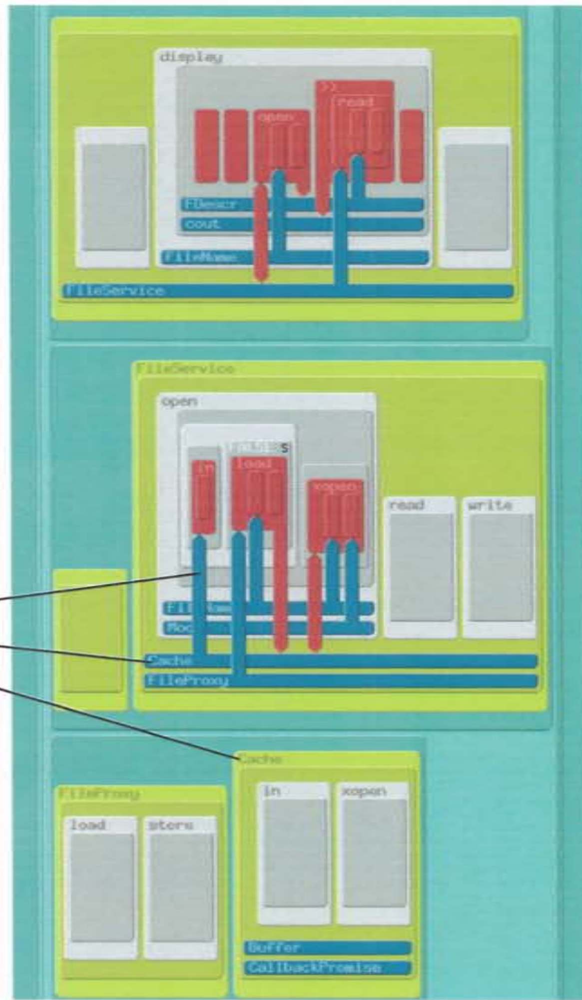
- **Experiment:** Man starte die ausführbare Datei eines Programms, und lösche anschließend seinen Quelltext sowie sämtliche Dokumentation. Wo befindet sich nun seine Architektur? Wie kann jetzt die Aufgabe der Architektur wahrgenommen werden?

- Die Aussage „*Jedes Softwaresystem hat eine Architektur, auch wenn diese nicht explizit modelliert wurde*[‡‡]“ ist daher in Frage zu stellen.

[‡‡] z.B. in W. Hasselbring: Software-Architektur, Informatik-Spektrum 2006, Bd. 29, Heft 1

Software-Architektur

Zusammenhang zwischen Anforderung, Funktion und Struktur



Anforderung:
‚Performance‘

Die einer Anforderung zugeordneten Elemente liegen ‚quer‘ zur Modularisierungsstruktur (die Zuordnung ist akzidentell)

Die Funktionsbeschreibung folgt der Modularisierungsstruktur (die Zuordnung ist essentiell)

Software-Architektur = Funktion + Struktur

Die Software-Architektur muss die Anforderungen erfüllen (hier erfüllt durch einen ‚Cache‘)

Anforderung, Funktion, Struktur

- Anforderungen sind als *Relation* auf den Architekturelementen (= Knoten des Modularisierungsbaums) zu dokumentieren.
- Der Rückschluss (= Traceability) von Architekturelementen auf die ihnen zugeordnete Anforderung ist nicht entscheidbar (Rice).
- Der Rückschluss wird möglich, wenn die Anforderung als beschriftete Relation im Modularisierungsbaum hinterlegt ist.

Ähnlichkeit zur Feature-Relation auf den Implementierungselementen
(ein ‚Feature' umfasst einen Evolutionsschritt von einer lauffähigen Version zur nächsten).

Software Architecture Process

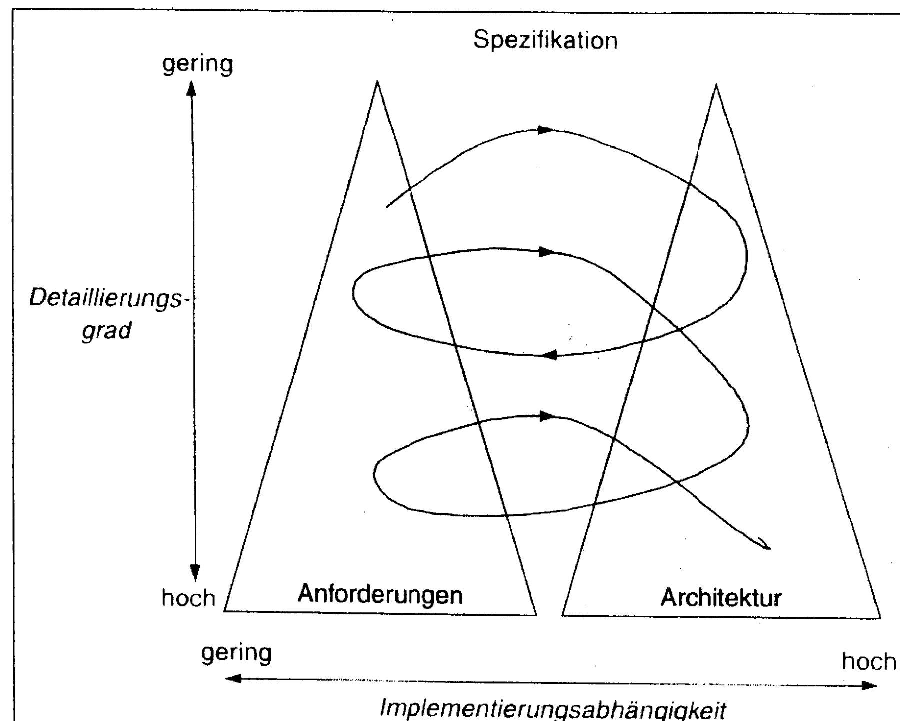
- There is no generally accepted process model for architectural design. The process depends on application knowledge and on the skill and intuition of the system architect.
- As part of the process, the following activities are usually necessary:
  - System structuring. The system is structured into a number of principal sub-systems. Communications between sub-systems are identified.
  - Control modelling. A general model of the control relationships between the parts of the system is established.
  - Modular decomposition. Each identified sub-system is decomposed into modules. The architect must decide on the types of module and their interconnections.
    [12. Sommerville, p. 226]

Das Twin-Peaks-Modell[1]

Iterative Definition von Anforderungen und Architektur

Das Requirements Engineering umfasst neben der Ermittlung der Anforderungen deren Beschreibung und Prüfung. Schlussendlich müssen diese Anforderungen flexibel verwaltet werden, um etwaigen Änderungen Rechnung tragen zu können. Da die Entwicklung einer ambitionierten Web-Applikation stark von der vorhandenen Systemarchitektur beeinflusst wird, wird der Prozess der Anforderungsdefinition iterativ mit der Spezifikation der Architektur verknüpft. Im Twin-Peaks-Modell ist diese Iteration visualisiert.

Diese Vorgehensweise minimiert eventuelle Risiken, die auf Grund von Fehlplanungen mangels Berücksichtigung der Systemarchitektur entstehen können. Aus dieser Iteration heraus entsteht ein Anforderungsdokument, in dem das Problem und die notwendigen Anforderungen ermittelt sind und von allen am Entwicklungsprozess beteiligten Personengruppen die sog. Stakeholder festgehalten sind[2].



[1] Aus: Kappel, Pröll, Reich, Retschitzeger (Hrsg.): Web-Engineering, Dpunkt-Verlag Heidelberg 2004

[2] Aus: P. Mitsopoulou: Konzeption und Realisierung einer Webapplikation unter Verwendung eines CMS, Bachelor-Arbeit, Hochschule DA 2006

Reconciling software requirements and architectures with intermediate models

The Component-Bus-System-Property Approach

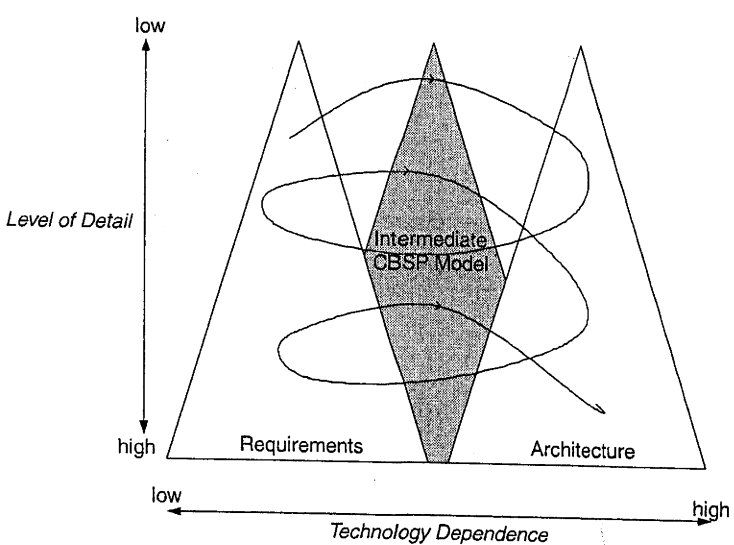


Fig. 1. CBSP model context

Our CBSP (Component-Bus-System-Property) approach helps to refine a set of requirements by applying a taxonomy of architectural dimensions.

CBSP provides an intermediate model between the requirements and the architecture that helps to iteratively evolve the two models [36].

CBSP supports the task of identifying architectural information contained in the requirements and explicating it in an intermediate model.

Figure 1 shows the CBSP model in the context of the Twin Peaks model [36]. The Twin Peaks model suggests that requirements and architectures are evolved iteratively and concurrently. In such a context, the intermediate CBSP model can be used at different levels of detail in the modeling process.

Paul Grünbacher[1], Alexander Egyed[2], Nenad Medvidovic[3]

Softw Syst Model (2003) / Digital Object Identifier (DOI) 10.1007/s10270-003-0038-6

To reconcile = versöhnen

Reconciling software requirements and architectures with intermediate models

Challenges

Software engineers face some critical challenges when trying to reconcile requirements and architectures:

- Requirements are frequently captured informally in a natural language. On the other hand, entities in a software architecture specification are usually specified in a more formal manner causing a semantic gap [31].
- System properties described in non-functional requirements [9] are commonly hard to specify in an architectural model [11, 31].
- The iterative, concurrent evolution of requirements and architectures demands that the development of an architecture be based on incomplete requirements. Also, certain requirements can only be understood after modeling or even partially implementing the system architecture [13, 36].
- Mapping requirements into architectures and maintaining the consistency and traceability between the two is complicated since a single requirement may address multiple architectural concerns and a single architectural element (for example a COTS component) typically has numerous non-trivial relations to various requirements. The increasing importance of component-based software development (CBD) emphasizes the need for agile techniques that allow capture and understanding the complex relationship between requirements and architectural elements.
- Real-world, large-scale systems have to satisfy hundreds, possibly thousands of requirements [6]. It is difficult to identify and refine the architecturally relevant information contained in the requirements due to this scale.
- Requirements and the software architecture emerge in a process involving heterogeneous stakeholders with conflicting goals, expectations, and terminology [3]. Supporting the different stakeholders' interests demands finding the right balance across these often divergent interests.

Differences between requirements and architecture

According to (IEEE-610-1990) [23] a requirement is "a condition or capability needed by a user to solve a problem or achieve an objective." *Requirements* largely describe aspects of the problem to be solved and constraints on the solution, i.e., desired system features and properties (both functional and non-functional [9]).

Requirements may be simple or complex, precise or ambiguous, stated concisely or elaborated carefully. Of particular interest to our work is a large class of requirements that is predominantly stated in a natural language, as opposed to precise formalisms. On the surface, such requirements are easier to understand by humans, but they frequently lead to ambiguity, incompleteness, and inconsistencies in the architecture and, eventually, the software system.

The relationship between a set of requirements and an effective architecture for a desired system is not readily obvious.

Architectures model a solution to a problem described in the requirements and provide high-level abstractions for representing the structure, behavior, and key properties of a software system.

The terminology and concepts used to describe architectures differ from those used for the requirements.

An architecture deals with components, which are the computational and data elements in a software system [40]. The interactions among components are captured within explicit software connectors (or buses) [46]. Components and connectors are composed into specific software system topologies.

Finally, architectures both capture and reflect the key desired properties of the system under construction (e.g., reliability, performance, cost) [46]. These elements of software architectures can be specified formally using architecture description languages (ADLs) [31].

The above-described differences between requirements and architectures make it difficult to build a bridge that spans the two.

Mapping requirements to an intermediate model

CBSP Taxonomy

There are six possible CBSP dimensions discussed below and illustrated with a simple example from a spreadsheet manipulation application. The six dimensions involve the basic architectural constructs [31] and, at the same time, reflect the simplicity of the CBSP approach.

These dimensions can be applied to systematically classify and refine requirements and to capture architectural trade-off issues and options (e.g., impact of a connector's throughput on the scalability of the topology).

1. *C* are model elements that describe or involve an individual Component in an architecture. For example, the requirement:

   R: Allow user to directly manipulate spreadsheet data.

   may be refined into CBSP model elements describing both processing components (C_p) and data components (C_d)

   C_p: *Spreadsheet manipulation UI component.*
   C_d: *Data for spreadsheet.*

2. *B* are model elements that describe or imply a Bus (connector). For example:

   R: Manipulated spreadsheet data must be stored on the file system.

   may be refined into

   B: Connector enabling interaction between UI and persistency components.

3. *S* are model elements that describe System-wide features or features pertinent to a large subset of the system's components and connectors. For example:

   R: The user should be able to select appropriate data filters and visualizations.

   may be refined into

   S: The system should employ a strict separation of data storage, processing, and visualization components.

4. *CP* are model elements that describe or imply data or processing Component Properties. As discussed above, the properties in CBSP are the "ilities" in a software system, such as reliability, portability, incrementality, scalability, adaptability, and evolvability. For example:

   R: The user should be able to visualize the data remotely with minimal perceived latency.

   may be refined into

   CP: The data visualization component should be efficient, supporting incremental updates.

5. *BP* are model elements that describe or imply Bus Properties. For example:

   R: Updates to system functionality should be enabled with minimal downtime.

   may be refined into

   BP: Robust connectors should be provided to facilitate runtime component addition and removal.

6. *SP* are model elements that describe or imply System (or subsystem) Properties. For example:

   R: The spreadsheet data must be encrypted when dispatched across the network.

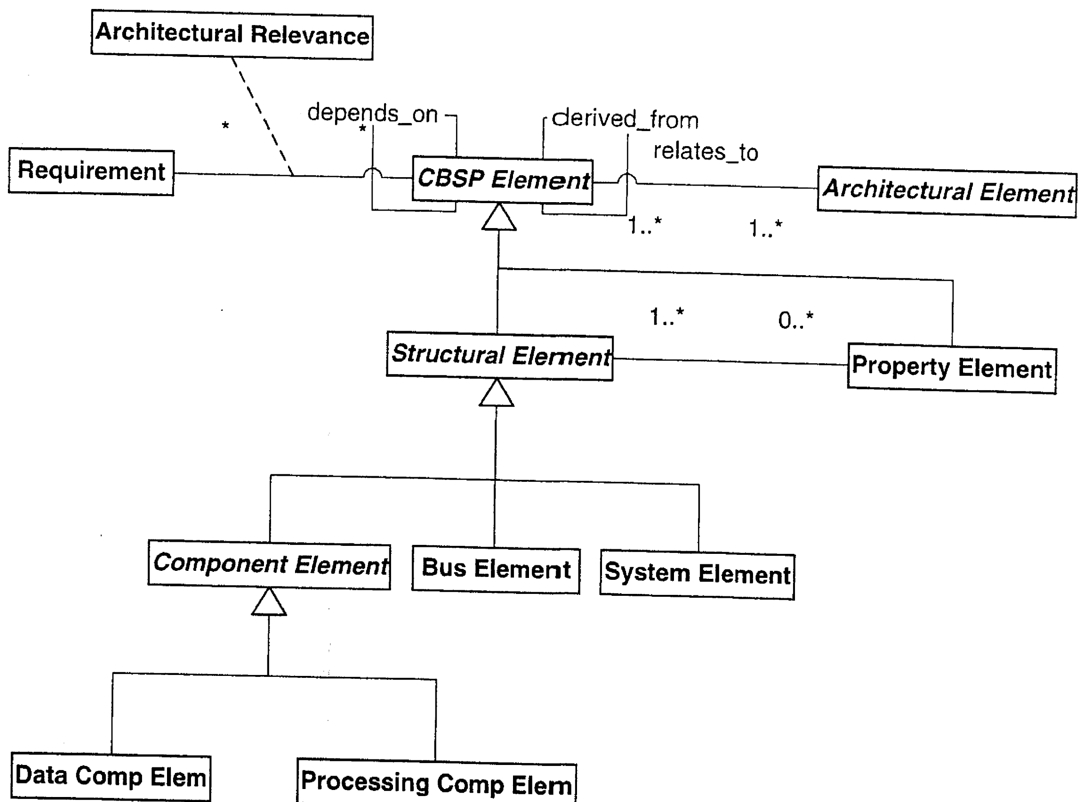
   may be transformed into

   SP: The system should be secure.

Note that, e.g., the BP example (5) involved refining a general requirement into a more specific CBSP element. On the other hand, the SP example (6) involved the generalization of a specific requirement into a CBSP artifact. Refinements and generalizations such as those shown above are a function of the needs of the specific system under construction, the characteristics of the application domain, and the software architect's background and experience.

Mapping requirements to an intermediate model

CBSP Meta Model



Architecture and Requirements

- In addition to specifying the structure and topology of the system, the architecture shows the correspondence between the system requirements and elements of the constructed system, thereby providing some rationale for the design decisions.
  [10. Shaw, Garlan, p. 3]
- Requirements are intended to communicate, in a precise way, the functions which the system must provide. To reduce ambiguity, they may be written in a structured language of some kind. This may be a structured form of natural language, a language based on a high-level programming language or a special language for requirements specification.
  [12. Sommerville, p. 134, key point 4]

The Requirements Engineering Process

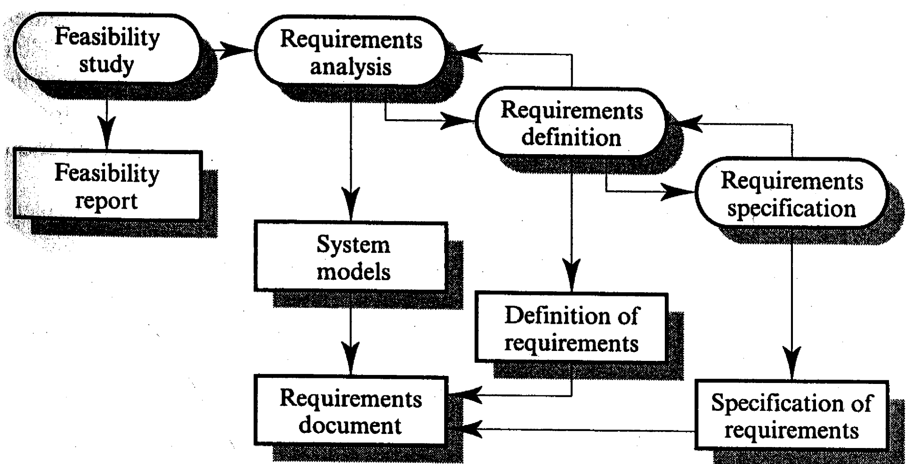
There are four principal stages in this process:

(1) *Feasibility study* An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study will decide if the proposed system will be cost-effective from a business point of view and if it can be developed given existing budgetary constraints. A feasibility study should be relatively cheap and quick. The result should inform the decision of whether to go ahead with a more detailed analysis.

(2) *Requirements analysis* This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis and so on. This may involve the development of one or more different system models. These help the analyst understand the system to be specified. System prototypes may also be developed to help understand the requirements.

(3) *Requirements definition* Requirements definition is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. These should accurately reflect what the customer wants. This document must be written so that it can be understood by the end-user and the system customer.

(4) *Requirements specification* A detailed and precise description of the system requirements is set out to act as a basis for a contract between client and software developer. The creation of this document is usually carried out in parallel with some high-level design. The design and requirements activities influence each other as they develop. During the creation of this document, errors in the requirements definition are inevitably discovered. It must be modified to correct these problems.



Requirements Engineering

Documents

Some of the problems that arise during the requirements engineering process are a result of failing to make a clear separation between the different levels of requirements description [12. Sommerville, p. 64]. We distinguish the following levels:

- A requirements definition is a statement, in a natural language plus diagrams, of what services the system is expected to provide and the constraints under which it must operate. It is generated using customer-supplied information.
- A requirements specification is a structured document which sets out the system services in detail. This document, which is sometimes called a functional specification, should be precise. It may serve as a contract between the system buyer and the system developer.
- A software specification is an abstract description of the software which is a basis for design and implementation. This specification may add further detail to the requirements specification.

Requirements definition

1. The software must provide a means of representing and accessing external files created by other tools.

Requirements specification

1.1 The user should be provided with facilities to define the type of external files.
1.2 Each external file type may have an associated tool which may be applied to the file.
1.3 Each external file type may be represented as a specific icon on the user's display.
1.4 Facilities should be provided for the icon representing an external file type to be defined by the user.
1.5 When a user selects an icon representing an external file, the effect of that selection is to apply the tool associated with the type of the external file to the file represented by the selected icon.

Requirements Analysis

After initial feasibility studies, the first major stage of the requirements engineering process is requirements analysis or elicitation. Technical software development staff work with customers and system end-users to find out about the application domain, what services the system should provide, the required performance of the system, hardware constraints, and so on.

Requirements analysis is an important process. The acceptability of the system after it has been delivered depends on how well it meets the customer's needs and supports the work to be automated. If the analyst does not discover the customer's real requirements, the delivered system is unlikely to meet their expectations.

Requirements analysis may involve a variety of different kinds of people in an organization. These include system end-users who will ultimately interact with the system and their managers. It should involve others in an organization who will be affected by the installation of the system. Engineers who are developing or maintaining other related systems, domain experts, trade union representatives and so on may also be consulted. The term *stakeholder* is used to refer to everyone who may have some direct or indirect influence on the system requirements.

Requirements analysis is a difficult process for a number of reasons:

(1) Stakeholders often don't really know what they want from the computer system except in the most general terms. Even when they have a clear idea of what they would like the system to do, they may find this difficult to articulate. They may make unrealistic demands because they are unaware of the costs of their requests.

(2) Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Engineers, without much experience in the customer's domain, must understand these requirements and translate them to an agreed form.

(3) Different stakeholders have different requirements and they may express these in quite different ways. Engineers must discover all potential sources of requirements and discover commonalities and conflict.

(4) Analysis takes place in an organizational context. Political factors may influence the requirements of the system. These factors may not be obvious to the system end-users. They may come from higher management influencing the system procurement in ways that satisfy their personal agenda.

(5) The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. Hence the importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

Problems with the Requirements Analysis

Large software systems are usually developed to address 'wicked' problems (as discussed in Chapter 2). This makes the formulation of requirements very difficult. Because the problem cannot be fully defined, the software requirements are bound to be incomplete. During the software process, the developer's understanding of the problem is constantly changing. This naturally leads to volatile system requirements.

There are several reasons why it is virtually impossible to define a complete and consistent set of requirements to address a problem:

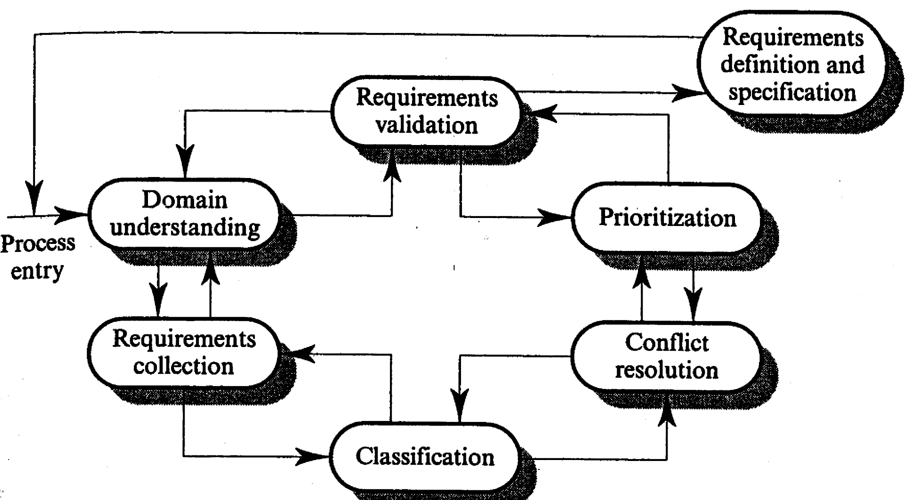
(1) Large software systems are usually required to improve upon the *status quo*. The existing system may be manual or an out-of-date computer system. Although difficulties with the current system may be known, it is hard to anticipate what effects the 'improved' system will have on the organization.

(2) Large systems usually have a diverse user community. Different users have different requirements and priorities. These may be conflicting or contradictory. The final system requirements are inevitably a compromise between them.

(3) The people who pay for a system and the users of a system are rarely the same people. System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements.

Sometimes, formulating outline requirements for a project is difficult as the application domain is poorly understood. In such cases, it is unrealistic to expect a definitive requirements definition before system development begins. A process model based on system prototyping (see Chapter 8) is more appropriate than the classical 'waterfall' model in these cases.

The Requirements Analysis Process



The activities shown in Figure 5.1 are highly iterative with continual feedback from each activity to other activities. The process can be viewed as a cycle, starting with domain understanding and ending with requirements validation. The analyst's understanding of the requirements improves with each round of the cycle.

The process activities are:

(1) *Domain understanding* Analysts must develop their understanding of the application domain. Therefore, if a system for a supermarket is required, the analyst must find out as much as possible about supermarkets.

(2) *Requirements collection* This is the process of interacting with stakeholders in the system to discover their requirements. Obviously, domain understanding will also develop during this activity.

(3) *Classification* This activity takes the unstructured collection of requirements and organizes them into coherent clusters.

(4) *Conflict resolution* Inevitably, where multiple stakeholders are involved, requirements will conflict. This activity is concerned with finding and resolving these conflicts.

(5) *Prioritization* In any set of requirements, some will be more important than others. This stage involves interaction with stakeholders to discover the most important requirements.

(6) *Requirements validation* The identified requirements are checked to discover if they are complete, consistent and in accordance with what stakeholders really want from the system.

Method-based Analysis

Method-based analysis is probably the most widely used approach to requirements analysis. It depends on the application of some structured method to understand the system. The results of the analysis are expressed as a set of system models defined by whatever method is being used. Analysis methods have different emphases. Some are exclusively designed for requirements elicitation and analysis; others are very close to design methods.

Without exception, structured methods collect vast amounts of information and generate a large volume of documentation. The information management problem posed by methods was a significant factor in the development of CASE toolsets

Structured methods usually include some or all of the following:

- *A process model* This defines the activities in the method. Examples of activities are data-flow analysis, control scenario identification, and so on. The process model usually presents activities as a sequence. However, in practice, the analysis iterates between activities.
- *System modelling notations* These notations may be diagrammatic, form-based or linguistic. Examples of diagram types are data-flow diagrams, entity-relation diagrams, object structure diagrams, and such like.
- *Rules applied to the system model* Rules may hold within a single model (for example, every entity in a diagram must have a name) or across models (for example, input and output items in a data-flow diagram must be documented using an entity-relation diagram).
- *Design guidelines* These are not enforceable rules but are intended to avoid poor design. An example of such a guideline might be that an object should normally have no more than five sub-objects.
- *Report templates* These define how the information collected during the analysis should be presented. Information in diagrams is normally supplemented by other textual information. This can be combined with the diagrams into a report for the analyst.

Requirements Specification

Problems with natural language

Natural language is often used to write requirements specifications. However, a natural language specification is not a particularly good basis for either a design or a contract between customer and system developer. There are several reasons for this:

(1) Natural language understanding relies on the specification readers and writers using the same words for the same concept. This leads to misunderstandings because of the inherent ambiguity of natural language words. Jackson (1995) gives an excellent example of this when he discusses signs displayed by an escalator. These said 'Shoes must be worn' and 'Dogs must be carried'. There are many conflicting interpretations of these phrases.

(2) A natural language requirements specification is over-flexible. You can say the same thing in completely different ways. It is up to the reader to find out when requirements are the same and when they are distinct. This is a very error-prone process.

(3) Requirements are not partitioned effectively by the language itself. It is difficult to find all related requirements. To discover the consequence of a change, you may have to look at every requirement rather than just a group of related requirements.

Because of these problems, requirements specifications written in natural language are prone to misunderstandings. These are often not discovered until the design or the implementation phases of the software process. As discussed in Chapter 4, the problems may then be very expensive to resolve. There is, therefore, a good case for using alternative notations that avoid some of the problems of unrestricted natural language.

Requirements Specification

Alternatives to natural language

There are various alternatives to the use of natural language which add structure to the specification and which should reduce ambiguity. These are:

(1) *Structured natural language* This approach depends on defining standard forms or templates to express the requirements specification. This is an extended, more detailed form of the structured definitions discussed in the first part of this chapter. Decision tables (Moret, 1982) are a form of structured language.

(2) *Design description languages* This approach relies on using a language which is like a programming language but with more abstract features to specify the requirements by defining an operational model of the system.

(3) *Requirements specification languages* Various special-purpose languages have been designed to express software requirements, such as PSL/PSA (Teichrow and Hershey, 1977) and RSL (Alford, 1977; Bell *et al.*, 1977; Alford, 1985). The advantage of this approach is that special-purpose tool support can be developed.

(4) *Graphical notations* Perhaps the best known graphical notation for requirements is SADT (Ross, 1977; Schoman and Ross, 1977). SADT has a fairly complex graphical vocabulary so is mostly used by specialists. It has been fairly widely used for analysis and requirements specification.

(5) *Mathematical specifications* These are notations based on a formal mathematical concept such as finite-state machines, Petri nets (Peterson, 1977) or more basic concepts such as sets. These unambiguous specifications reduce the arguments between customer and contractor about system functionality. However, most customers don't understand a formal mathematical specification. They are therefore reluctant to accept it as a system contract. Hall (1990a) suggests that the way to tackle this problem is to develop a formal specification then paraphrase this in a way that is more understandable to the customer.

Davis (1990) summarizes and compares some of these different approaches to requirements specification. In this chapter, I focus on the first two of these approaches, namely structured natural language and the use of design description languages. Specialized requirements languages have never become widely known or used. Graphical notations are similar to the notations used to define system models, discussed in Chapter 6. Formal specifications are discussed in Chapters 9 to 11.

System Models

One of the outputs of the requirements analysis process is a set of system models that present abstract descriptions of the system to be developed.

Examples of the different types of system model which might be produced as part of the analysis process and the notations used to represent these models are:

- A *data-processing model* Data-flow diagrams may be used to show how data is processed at different stages in the system.
- A *composition model* Entity-relation diagrams may be used to show how some entities in the system are composed of other entities.
- A *classification model* Object class/inheritance diagrams may be used to show how entities have common characteristics.
- A *stimulus-response model* State transition diagrams may be used to show how the system reacts to internal and external events.
- A *process model* Process models may be used to show the principal activities and deliverables involved in carrying out some process.

Structured Language Specifications

Structured natural language is a restricted form of natural language for requirements specification. Structured language notations may limit the terminology used and may use templates to specify system requirements. They may incorporate control constructs derived from programming languages and graphical highlighting to partition the specification. The advantage of this approach is that it maintains most of the expressiveness and understandability of natural language. It does, however, ensure that some degree of uniformity is imposed on the specification.

A forms-based approach to requirements specification relies on defining one or more standard forms or templates to express the requirements. The specification may be structured around the objects manipulated by the system, the functions performed by the system or the events processed by the system. The form structure will vary depending on the requirements structuring technique used. Functionally oriented specifications are probably the most common.

ECLIPSE/Workstation/Tools/DE/FS/3.5.1	
Function	Add node
Description	Adds a node to an existing design. The user selects the type of node, and its position. When added to the design, the node becomes the current selection. The user chooses the node position by moving the cursor to the area where the node is added.
Inputs	Node type, Node position, Design identifier.
Source	Node type and Node position are input by the user, Design identifier from the database.
Outputs	Design identifier.
Destination	The design database. The design is committed to the database on completion of the operation.
Requires	Design graph rooted at input design identifier.
Pre-condition	The design is open and displayed on the user's screen.
Post-condition	The design is unchanged apart from the addition of a node of the specified type at the given position.
Side-effects	None
Definition: ECLIPSE/Workstation/Tools/DE/RD/3.5.1	

The Software Requirements Document

Six properties to satisfy:

- It should only specify external system behaviour.
- It should specify constraints on the implementation.
- It should be easy to change.
- It should serve as a reference tool for system maintainers.
- It should record forethought about the life cycle of the system.
- It should characterize acceptable responses to undesired events.

The Structure of the Requirements Document

The requirements document is a combination of requirements definition and requirements specification. The best organization is as a series of chapters with the detailed specification perhaps presented as an appendix to the document. A possible generic structure for a requirements document is shown in Figure 4.4.

Chapter	Description
Introduction	This should describe the need for the system. It should briefly describe its functions and explain how it will work with other systems. It should describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. No assumptions should be made about the experience or expertise of the reader.
System models	This should set out one or more system models showing the relationships between the system components and the system and its environment. These might include object models, data-flow models and semantic data models.
Functional requirements definition	The services provided for the user should be described in this section. This description may use natural language, diagrams or other notations that are understandable by customers.
Non-functional requirements definition	The constraints imposed on the software and restrictions on the freedom of the designer should be described here and related to the functional requirements. This might include details of specific data representation, response time and memory requirements, and so on. Product and process standards which must be followed should be specified.
System evolution	This should describe the fundamental assumptions on which the system is based and anticipated changes due to hardware evolution, changing user needs, and so on.
Requirements specification	This should describe the functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements, for example interfaces to other systems may be defined.

Requirements Definition and Specification

KEY POINTS

- A requirements definition is intended for use by people involved in using and procuring the system. It should be written using natural language, tables and diagrams so that they can understand it. Requirements definitions should be structured so that they are easier to understand and manage.
- Rationale should always be included in a requirements definition. Without rationale, it is difficult to understand the consequences of changes to the system.
- Requirements should always be written so that they are verifiable. This means that there can be no argument over whether or not the system satisfies the requirement.
- Requirements specifications are intended to communicate, in a precise way, the functions which the system must provide. To reduce ambiguity, they may be written in a structured language of some kind. This may be a structured form of natural language, a language based on a high-level programming language or a special language for requirements specification.
- The three principal classes of non-functional requirement are product requirements which constrain the system being developed, process requirements which apply to the development process and external requirements. External requirements may affect both product and process.
- Non-functional requirements tend to be so varied and complex that natural language must be used for their expression. An exception to this is the specification of interface requirements where an interface definition language may be used.

Software-Architektur und Requirements-Definition

Zusammenfassung

- Twin-Peaks und CBSP zeigen die Nähe von Requirements, Funktionaler Analyse und Architektur.

- Natural language is often used to write requirements specifications. However, a natural language specification is not a particularly good basis for either a design or a contract between customer and system developer because of the inherent ambiguity of natural language words and misunderstandings.

- In addition to specifying the structure and topology of the system, the architecture shows the correspondence between the system requirements and the elements of the constructed system.

- The creation of the requirements specification is usually carried out in parallel with some high-level design. The design and requirements activities influence each other as they develop.

- Requirements are not partitioned effectively by the language itself. It is difficult to find all related requirements. To discover the consequence of a change, you may have a look at every requirement rather than just a group of related requirements.

- There is, therefore, a good case for using alternative notations that avoid some of the problems of unrestricted natural language.

- There are various alternatives to the use of natural language which add structure to the specification and should reduce ambiguity.

- A forms-based approach to requirements specification relies on standard forms or templates. The specification may be structured around the objects manipulated by the system, the functions performed by the system or the events processed by the system.

- The best organization is as a series of chapters with detailed specifications perhaps presented as an appendix to the document.

Requirements Verification

Validation: Are we building the right product?
Verification: Are we building the product right?
[Boehm, 1979]

- Verification involves checking that the program conforms to its specification. Validation involves checking that the program as implemented meets the expectations of the software customer.
  [12. Sommerville, p. 446]
- A requirements specification should be verifiable. This means that there must be a finite process to determine whether or not the requirements have been met.
  [13. van Vliet, p. 225]
- Requirements should always be written so that they are verifiable. This means that there can be no argument over whether or not the system satisfies the requirement.
  [12. Sommerville, p. 134, key point 3]

Requirements Validation

Requirements validation is concerned with showing that the requirements actually define the system that the customer wants. If this validation is inadequate, errors in the requirements will be propagated to the system design and implementation. Expensive system modifications may be required at a later stage to correct problems with the requirements.

The cost of errors in requirements is particularly high if these errors are not discovered until the system is implemented. The cost of making a system change resulting from a requirements problem is much greater than repairing design or coding errors. A requirements change implies that the design and implementation must also be changed. The system testing and validation processes must be repeated. The cost of changing a system after delivery because of a requirements change can therefore be up to 100 times more than the cost of repairing a programming error.

There are several aspects of the requirements which must be checked:

(1) *Validity* A user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that are required. Systems have diverse users with different needs and any set of requirements is inevitably a compromise across the user community.

(2) *Consistency* Any one requirement should not conflict with any other.

(3) *Completeness* The definition should include all functions and constraints intended by the system user.

(4) *Realism* There is no point in specifying requirements that are unrealizable. It may be acceptable to anticipate some hardware developments but developments in software technology are much less predictable.

Demonstrating that a set of requirements meets a user's needs is difficult if an abstract approach is adopted. By reading a definition and specification, users must picture the system in operation. They must imagine how that system would fit into their work. It is hard for skilled computer professionals to perform this type of abstract analysis; it is almost impossible for system users. As a result, many systems are delivered which do not meet the user's needs and which are simply discarded after delivery.

Prototyping, whereby an executable model of the system is demonstrated to users, is an important requirements validation technique. It is useful because it gives users hands-on experience with a system. They can see how it actually supports their work. Prototyping is an important subject in its own right which is discussed in Chapter 8.

Validation should not be seen as a process to be carried out after the requirements document has been completed. Regular requirements reviews involving both users and software engineers are essential while the requirements definition is being formulated.

A requirements review is a manual process which involves multiple readers from both client and contractor staff checking the requirements document for anomalies and omissions.

Exkurs

TURING-MASCHINEN

Definition. Eine *Turingmaschine* (kurz: TM) ist gegeben durch ein 7-Tupel

$$M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$$

Hierbei sind:

- Z die endliche *Zustandsmenge*,
- Σ das *Eingabealphabet*,
- $\Gamma \supset \Sigma$ das *Arbeitsalphabet*,
- $\delta : Z \times \Gamma \longrightarrow Z \times \Gamma \times \{L, R, N\}$ im deterministischen Fall (bzw. $\delta : Z \times \Gamma \longrightarrow \mathcal{P}(Z \times \Gamma \times \{L, R, N\})$ im nichtdeterministischen Fall) die *Überführungsfunktion*,
- $z_0 \in Z$ der *Startzustand*,
- $\Box \in \Gamma - \Sigma$ das *Blank*,
- $E \subseteq Z$ die Menge der *Endzustände*.

---

ormal erklärt bedeutet

$$\delta(z, a) = (z', b, x) \text{ bzw. } \delta(z, a) \ni (z', b, x)$$

gendes:

nn sich M im Zustand z befindet und unter dem Schreib-Lesekopf das chen a steht, so geht M im nächsten Schritt in den Zustand z' über, schreibt f den Platz von a) b auf das Band und führt danach die Kopfbewegung : $\{L, R, N\}$ aus. (Hierbei steht L für *links*, R für *rechts* und N für *neutral*,) Stehenbleiben).

Beispiel: Gegeben sei folgende Turingmaschine, die eine Eingabe $x \in \{0, 1\}^*$ als Binärzahl interpretiert und 1 hinzuaddiert:

$$M = (\{z_0, z_1, z_2, z_e\}, \{0, 1\}, \{0, 1, \Box\}, \delta, z_0, \Box, \{z_e\})$$

wobei

$$\begin{aligned}
\delta(z_0, 0) &= (z_0, 0, R)\\
\delta(z_0, 1) &= (z_0, 1, R)\\
\delta(z_0, \Box) &= (z_1, \Box, L)\\[4pt]
\delta(z_1, 0) &= (z_2, 1, L)\\
\delta(z_1, 1) &= (z_1, 0, L)\\
\delta(z_1, \Box) &= (z_e, 1, N)\\[4pt]
\delta(z_2, 0) &= (z_2, 0, L)\\
\delta(z_2, 1) &= (z_2, 1, L)\\
\delta(z_2, \Box) &= (z_e, \Box, R)
\end{aligned}$$

Wenn diese Maschine mit der Eingabe 101 gestartet wird, so stoppt sie schließlich mit 110 auf dem Band, wobei sich der Schreib-Lesekopf wieder auf das erste nicht-leere Zeichen zurückbewegt hat.

$$z_0 101 \vdash 1 z_0 01 \vdash 10 z_0 1 \vdash 101 z_0 \Box \vdash 10 z_1 1 \Box$$
$$\vdash 1 z_1 00 \Box \vdash z_2 110 \Box \vdash z_2 \Box 110 \Box \vdash \Box z_e 110 \Box$$

Exkurs

TURING-MASCHINEN

Definition. Eine Funktion $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ heißt *Turing-berechenbar*, falls es eine (deterministische) Turingmaschine M gibt, so daß für alle $n_1, \ldots, n_k, m \in \mathbb{N}$ gilt:

$$f(n_1, \ldots, n_k) = m$$

genau dann wenn

$$z_0 bin(n_1)\#bin(n_2)\#\ldots\#bin(n_k) \vdash^* \Box\ldots\Box z_e bin(m)\Box\ldots\Box$$

wobei $z_e \in E$.

Hierbei bezeichnet $bin(n)$ die Binärdarstellung der Zahl $n \in \mathbb{N}$ (ohne führende Nullen).

Definition. Eine Menge $A \subseteq \Sigma^*$ heißt *entscheidbar*, falls die *charakteristische Funktion* von A, nämlich $\chi_A : \Sigma^* \longrightarrow \{0, 1\}$, berechenbar ist. Hierbei ist für alle $w \in \Sigma^*$:

$$\chi_A(w) = \begin{cases} 1, & w \in A \\ 0, & w \notin A \end{cases}$$

Exkurs

TURING-MASCHINEN

Kodierung von Turing-Maschinen

Zunächst kodieren wir Turing-Maschinen mit beschränkten Alphabeten als Zeichenketten über $\{0, 1\}$. Sei

$$M = (Q, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$$

eine Turing-Maschine mit Eingabealphabet $\{0, 1\}$ und dem Blank als einzigem zusätzlichen Bandsymbol.

Es ist praktisch, die Symbole $0, 1, B$ mit den Synonymen X_1, X_2, X_3 zu benennen. Wir geben auch den Richtungen L und R die Namen R_1 und R_2. Dann wird eine generische Bewegung $\delta(q_i, X_j) = (q_k, X_l, R_m)$ durch die binäre Zeichenkette

$$0^i 1 0^j 1 0^k 1 0^l 1 0^m \tag{8.1}$$

kodiert. Eine binäre Kodierung für eine TM M ist

$$111 code_1 11 code_2 11 ... 11 code_r 111, \tag{8.2}$$

wobei jeder $code_i$ eine Zeichenkette der Form aus (8.1) ist, und jede Bewegung von M durch eine der Kodierungen kodiert wird.

Beispiel
Sei

$$M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$$

und habe die Bewegungen

$$\begin{aligned} \delta(q_1, 1) &= (q_3, 0, R) \\ \delta(q_3, 0) &= (q_1, 1, R) \\ \delta(q_3, 1) &= (q_2, 0, R) \\ \delta(q_3, B) &= (q_3, 1, L) \end{aligned}$$

Also ist die folgende eine durch $\langle M, 1011 \rangle$ bezeichnete Zeichenkette:

111010010001010011000101010010011000100100101001100010001000100101111011

Exkurs

TURING-MASCHINEN

Churchsche These.
DIE DURCH DIE FORMALE DEFINITION DER *Turing-Berechenbarkeit* (ÄQUIVALENT: *WHILE-Berechenbarkeit, GOTO-Berechenbarkeit, μ-Rekursivität*) ERFASSTE KLASSE VON FUNKTIONEN STIMMT GENAU MIT DER KLASSE DER IM INTUITIVEN SINNE BERECHENBAREN FUNKTIONEN ÜBEREIN.

---

wenn von einer Funktion nachgewiesen ist, daß sie *nicht* Turingmaschinen-berechenbar ist, dann folgt aus dieser Überzeugung, daß die Funktion *überhaupt nicht* berechenbar ist.

Konsequenz
aus der Church'schen These

- Die Implementierung ist gleich der formalen Beschreibung eines Programms

- Die Architektur ist gleich der intuitiven Beschreibung eines Programms

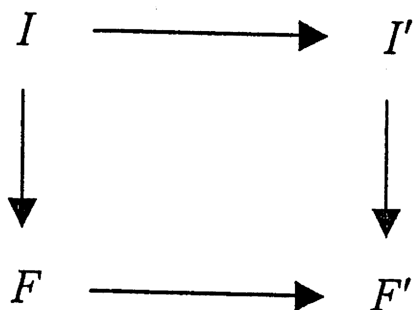
---

Wir betrachten im Folgenden die Übergänge zwischen intuitiven und formalen Beschreibungen eines Programms:

Übergänge

Äquivalenzen und Verfeinerungen

zwischen intuitiven Beschreibungen I , I'
und formalen Beschreibungen F , F'



Intuitive und formale Beschreibungen

- Unter formalen Beschreibungen verstehen wir Turingmaschinen (= Programme = Implementierungen) und unter intuitiven Beschreibungen sämtliche Spezifikationen (Artefakte) *vor* der Implementierung.
- Wir sagen, dass eine intuitive Beschreibung *I* auf eine formale Beschreibung *F* zutrifft, wenn *F* eine Implementierung von *I* ist, und umgekehrt. (Beide Begriffe werden intuitiv verwendet und nicht näher definiert).
- Mit $[F]_{\approx}$ bezeichnen wir die Menge (Äquivalenzklasse) aller formalen Beschreibungen, die dieselbe Funktion wie *F* berechnen.
- Mit $[F]_I$ bezeichnen wir die Menge aller formalen Beschreibungen, auf die die intuitive Beschreibung *I* zutrifft. *F* ist ein Repräsentant dieser Menge.
- Mit *F* ist stets auch $[F]_{\approx}$ in $[F]_I$ enthalten, d.h. wenn auf *F* eine intuitive Beschreibung *I* zutrifft, dann auch auf alle anderen formalen Beschreibungen, die dieselbe Funktion wie *F* berechnen.
- Je genauer eine intuitive Beschreibung *I* ist, desto kleiner ist auch die Menge $[F]_I$, auf die *I* zutrifft.
- Die ungenaueste intuitive Beschreibung, etwa „berechenbare Funktion", trifft auf alle formalen Beschreibungen zu, während die präziseste intuitive Beschreibung nur noch auf eine einzige formale Beschreibung *F* und ihre Äquivalenzklasse $[F]_{\approx}$ zutrifft.

Intuitive und formale Beschreibungen

Zunehmende Präzision einer intuitiven Beschreibung
verkleinert die Menge der zutreffenden Implementierungen

Sortieren

von ganzen Zahlen

in aufsteigender Ordnung

ohne Wiederholung

Darstellung auf Punktskala

Übertragung verschlüsselt

Starten mit Passwort

Rechenzeit O(nlogn)

Äquivalenz-Probleme

Satz 2:
Das Äquivalenzproblem für Turing-Maschinen $E = \{u\#v \mid M_u$ berechnet dieselbe Funktion wie $M_v\}$ ist nicht entscheidbar.

Definition 2.1:
Wir nennen eine intuitive Beschreibung I und eine formale Beschreibung F äquivalent ($I \approx F$) genau dann wenn $[F]_I = [F]_\sim$ gilt.

Korollar 2.1:
Die Äquivalenz $I \approx F$ von intuitiver Beschreibung I und formaler Beschreibung F eines Programms ist nicht entscheidbar.

Beweis:
Betrachte die intuitive Beschreibung I und beliebige formale Beschreibungen F und F'. Es gilt $I \approx F$ und $I \approx F'$ genau dann wenn $[F]_I = [F]_\sim$ und $[F']_I = [F']_\sim$. Da I sowohl auf F als auch auf F' zutrifft, gilt $[F]_I = [F']_I$ und damit $[F]_\sim = [F']_\sim$. Unter der Annahme, dass $I \approx F$ und $I \approx F'$ entscheidbar sei, hätten wir ein Entscheidungsverfahren für die Gleichheit der Äquivalenzklassen und damit für die Äquivalenz von F und F' gewonnen. Widerspruch zu Satz 2

Bemerkung 2.1:
- Die Übereinstimmung von intuitiver und formaler Beschreibung eines Programms kann somit niemals formal, sondern grundsätzlich nur intuitiv nachgeprüft werden. Dies schließt sowohl die algorithmische Durchführung der Überprüfung als auch die formale Sicherheit aus, die ein solcher Algorithmus bieten würde.
- „Intuitive Überprüfung“ bedeutet, dass man sich durch Nutzung des Programms davon überzeugt, dass es die Anforderungen erfüllt. Dieser Vorgang kann niemals ad-hoc erfolgen, sondern beansprucht lange Zeit. Genau genommen kann auch nach unendlicher Zeit keine Gewissheit erreicht werden. Erst nach einer langen „Reifezeit“ des Programms, in der iterativ durch Nutzung, Anpassung der Anforderungen und Anpassung der Implementierung allmählich eine Übereinstimmung von Anforderung und Implementierung entsteht, kann von „Äquivalenz im intuitiven Sinne“ gesprochen werden.
- „Anpassung an sich ändernde Anforderungen“ bedeutet somit nicht nur, im Rahmen des längerfristigen Lebenszyklus durch „konstruktive Voraussicht“ sich auf notwendige Änderungen einzustellen (design for change), sondern auch im Rahmen der Korrektur der grundsätzlich unscharfen Anforderungen und ihrer Implementierung für Flexibilität zu sorgen (für Änderung der Anforderungen und ihrer intuitiven Beschreibung, sowie für Änderung der Implementierung).

Äquivalenz-Probleme

Definition 2.2:
Intuitive Beschreibungen I und I' nennen wir *äquivalent* ($I \approx I'$) genau dann wenn $[F]_I = [F']_{I'}$ gilt, für formale Beschreibungen F und F'.

Bemerkung 2.2:
Wir nennen intuitive Beschreibungen also genau dann äquivalent, wenn sie auf dieselbe Menge von Turingmaschinen zutreffen, d.h. von derselben Menge implementiert werden.

Korollar 2.2:
Für intuitive Beschreibungen I und I' eines Programms ist die Äquivalenz $I \approx I'$ nicht entscheidbar.

Beweis:
Seien I und I' intuitive Beschreibungen eines Programms. Nach der Church'schen These gibt es formale Beschreibungen F und F', welche I bzw. I' implementieren.
Annahme, die Äquivalenz von I und I' sei entscheidbar. Dann ist nach Definition 2.2 auch die Gleichheit $[F]_I = [F']_{I'}$ entscheidbar. Widerspruch zu Lemma 2.1 □

Hilfssätze

Lemma 2.1:
Die Gleichheit $[F]_I = [F']_{I'}$ ist nicht entscheidbar.

Beweis:
Da mit $F \in [F]_I$ stets auch $[F]_{\approx} \subseteq [F]_I$ gilt, lassen sich $[F]_I$ und $[F']_{I'}$ disjunkt in Äquivalenzklassen bzgl. $\approx$ zerlegen. Wir führen den Beweis durch Induktion über die Anzahl der Äquivalenzklassen.
Induktionsanfang n=1:
Für $n=1$ gilt $[F]_I = [F]_{\approx}$ und $[F']_{I'} = [F']_{\approx}$. Annahme, $[F]_I = [F']_{I'}$ sei entscheidbar. Somit ist auch $[F]_{\approx} = [F']_{\approx}$ entscheidbar. Da diese Gleichung genau dann gilt, wenn $F \approx F'$ gilt, hätten wir mit einem Entscheidungsverfahren für $[F]_I = [F']_{I'}$ auch eines für $F \approx F'$ gewonnen. Widerspruch zu Satz 2.
Induktionsannahme:
Für beliebige $[F]_I$ und $[F']_{I'}$ mit jeweils höchstens $n \geq 1$ Äquivalenzklassen sei die Gleichheit $[F]_I = [F']_{I'}$ nicht entscheidbar.
Induktionsschluss von n auf $n+1$:
Seien $[F]_I$ und $[F']_{I'}$ jeweils disjunkt zerlegbar in höchstens $n+1$ Äquivalenzklassen bzgl. $\approx$. Es gilt $[F]_I = [F']_{I'}$ genau dann wenn

1. $[F]_{\approx} = [F']_{\approx}$ und (*falls* $F \approx F'$)
2. $[F]_I - [F]_{\approx} = [F']_{I'} - [F']_{\approx}$
   oder
3. $([F]_I - [F]_{\approx}) - [F']_{\approx} = ([F']_{I'} - [F']_{\approx}) - [F]_{\approx}$ und (*andernfalls*)
4. $([F]_I - [F]_{\approx}) - [F']_{\approx} \neq ([F]_I - [F]_{\approx})$ und
5. $([F']_{I'} - [F']_{\approx}) - [F]_{\approx} \neq ([F']_{I'} - [F']_{\approx})$.

Die Gleichheit $[F]_I = [F']_{I'}$ ist also genau dann entscheidbar bzw. nicht entscheidbar, wenn für die Gleichungen 1. bis 5. die Gleichheit jeweils entscheidbar bzw. nicht entscheidbar ist. Da jede Seite der Gleichungen 1. bis 5. als Ergebnis der dortigen Subtraktionen von jeweils mindestens einer Äquivalenzklasse durch Mengen von höchstens n Äquivalenzklassen repräsentiert ist, gilt nach unserer Induktionsannahme, dass dafür die Gleichheit, und damit auch die Ungleichheit, nicht entscheidbar ist. Somit ist auch $[F]_I = [F']_{I'}$ nicht entscheidbar□

Lemma 2.2:
Die Inklusion $[F]_I \subseteq [F']_{I'}$ ist nicht entscheidbar.

Beweis:
Da $[F]_I = [F']_{I'}$ gilt genau dann wenn $[F]_I \subseteq [F']_{I'}$ und $[F']_{I'} \subseteq [F]_I$, hätten wir mit einem Entscheidungsverfahren für die Inklusion auch eines für die Gleichheit gefunden. Widerspruch zu Lemma 2.1. Somit gilt die Behauptung □

Verfeinerungsprobleme

Definition 2.3:
Für intuitive Beschreibungen I und I' sagen wir, I ist *Verfeinerung von* I' ($I \leq I'$) genau dann wenn $[F]_I \subseteq [F']_{I'}$ gilt, für formale Beschreibungen F und F'

Bemerkung 2.3:

- I ist also genau dann Verfeinerung von I', wenn die Menge der Implementierungen von I enthalten ist in der Menge der Implementierungen von I' oder, mit anderen Worten, wenn I nur auf eine Teilmenge der Implementierungen von I' zutrifft.
- Die *schrittweise Verfeinerung* einer intuitiven Beschreibung schränkt also die Menge der Implementierungen sukzessive ein. Die ultimative Verfeinerung ist dann erreicht, wenn $[F]_I$ nur noch Turingmaschinen enthält, die zu F äquivalent sind, d.h. wenn $I \approx F$ gilt.

Korollar 2.3:
Für intuitive Beschreibungen I und I' eines Programms ist die Verfeinerung $I \leq I'$ nicht entscheidbar.

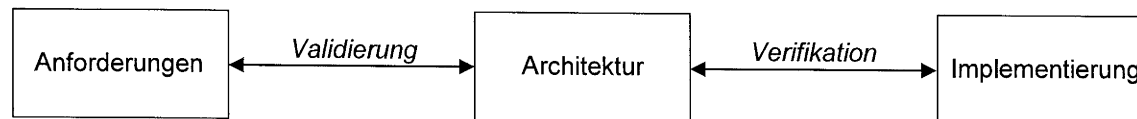
Beweis:
Seien I und I' intuitive Beschreibungen eines Programms. Nach der Church'schen These gibt es formale Beschreibungen F und F', welche I bzw. I' implementieren.
Wir nehmen an, $I \leq I'$ sei entscheidbar. Dann ist auch die Inklusion der Mengen $[F]_I$ und $[F']_{I'}$ entscheidbar. Widerspruch zu Lemma 2.2 □.

Korollar 2.4:
Für eine intuitive Beschreibung I und eine formale Beschreibung F eines Programms ist nicht entscheidbar, ob I durch F implementiert wird, d.h. ob $F \leq I$ gilt.

Beweis:
Nach der Church'schen These existiert eine formale Beschreibung F', welche I implementiert. Annahme, $F \leq I$ sei entscheidbar. Damit ist auch $F \in [F']_I$ entscheidbar. Da mit F stets auch alle zu F äquivalenten Implementierungen in $[F']_I$ liegen, und umgekehrt, ist mit $F \in [F']_I$ auch die Inklusion $[F]_{-} \subseteq [F']_I$ entscheidbar. Widerspruch zu Lemma 2.2 □

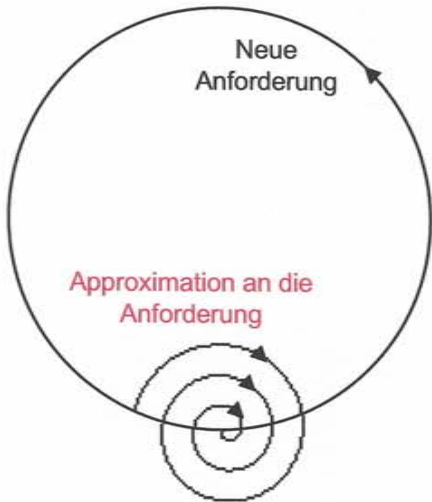
Validierung und Verifikation

Nicht-berechenbare Prozesse



- **Validierung.** Ob eine Architektur die Anforderungen erfüllt, ist nicht entscheidbar.
- **Verifikation.** Ob eine Implementierung die Architektur erfüllt, ist nicht entscheidbar.

Heuristische Lösung nicht-entscheidbarer Probleme



- Es ist grundsätzlich nicht möglich, die Konsistenz zwischen intuitiven und formalen Programmbeschreibungen auf algorithmische Weise, in endlicher Zeit und mit formaler Sicherheit zu überprüfen.

- Eine „Konsistenz im intuitiven Sinne“ kann jedoch annähernd erreicht werden, wenn nach einer Transformation oder Verfeinerung die Ausgangsbeschreibung erneut revidiert wird (und dabei sowohl besser verstanden als auch modifiziert werden kann), die Transformation bzw. Verfeinerung daraufhin angepasst und dieser Vorgang zyklisch wiederholt wird.

- Ein Spezialfall ist die Approximation der Implementierung an die Anforderung durch zyklischen Programmtest mit alternierender Anpassung von Implementierung und bestehender Nutzeranforderung.

- Eine höhere Genauigkeit der heuristischen Lösung wird erreicht, wenn sich die Verifikation und Validierung des Programms als nicht-abbrechender Prozess über den gesamten Lebenszyklus erstreckt. Da jede Aktivität des Entwicklungsprozesses das Verstehen des Programms zwingend voraussetzt, und dieses wiederum als implizite Konsistenzprüfung betrachtet werden kann, erhält man eine zyklische Verifikation und Validierung sozusagen als Nebeneffekt des Entwicklungsprozesses.

Optimistische*) Sichtweisen

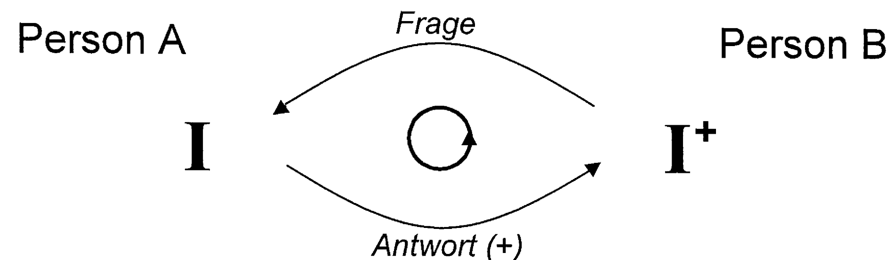
- A requirements specification should be verifiable. This means that there must be a finite process to determine whether or not the requirements have been met [van Vliet, p.225].
- Requirements should always be written so that they are verifiable. This means that there can be no argument over whether or not the system satisfies the requirements [Sommerville, p.134].
- Es dauert erfahrungsgemäß eine Stunde, um vier bis zehn Anforderungen zu reviewen [Siemens Corporate Research, in Broy et al.**), S. 132].

*) „Optimistisch" ist insofern der passende Ausdruck, als dem Leser auch in einem größeren Zusammenhang nicht bewusst gemacht wird, dass es sich hier um nicht-entscheidbare Probleme handelt, die in der angegebenen Strenge, z.B. „finite process", gar nicht gelöst werden können.

**) M. Broy, E. Geisberger, J. Kazmeier, A. Rudorfer, K. Beetz: Ein Requirements-Engineering-Referenzmodell, Informatik Spektrum 30 (3), Springer 2007

Intuitive Beschreibungen

und ihre Kommunikation



- Die intuitive Beschreibung I ist personen-gebunden. Sie spiegelt genauestmöglich die intuitive Vorstellung der Person A wieder.

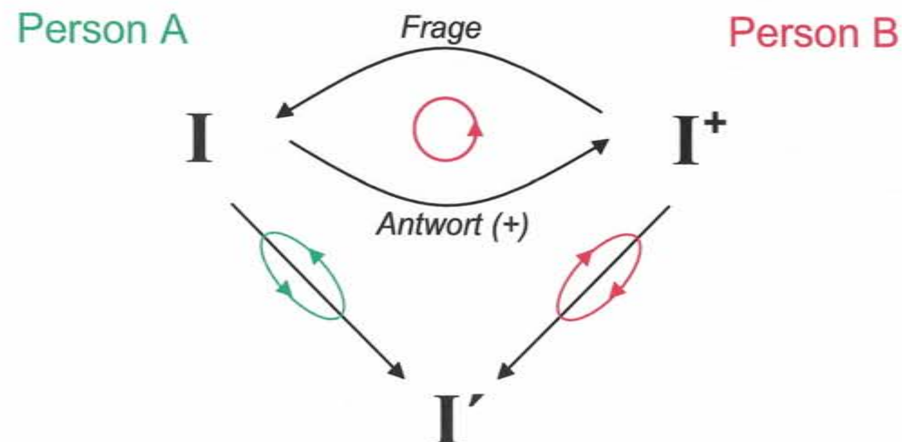
- Damit Person B die intuitive Beschreibung I, d.h. die Absicht der Person A versteht, muss I durch Rückfragen und Antworten ergänzt werden zu einer intuitiven Beschreibung I+.

- Die intuitiven Beschreibungen I und I+ sind nun insofern gleichwertig, als – bei sonst gleichen persönlichen Voraussetzungen – für A und B die gleichen Bedingungen vorliegen für eine Verfeinerung oder Transformation von I.

- Die Kommunikation der intuitiven Beschreibung I entspricht der heuristischen Lösung des Äquivalenzproblems I ~ I+.

Heuristische Vorgehensweisen

Ein Effizienzvergleich



- Person A verfeinert oder transformiert eigene Absicht I.
- Konsistenz durch iterative Anpassung der eigenen Absicht.
- Keine externe Kommunikation nötig.

- Anfängliche Kommunikation der Absicht I von Person A zu Person B.
- Person B verfeinert oder transformiert fremde Absicht I.
- Bei Konsistenzproblemen: Jede iterative Anpassung von I+ muss zwischen Person B und A kommuniziert werden.
- Hoher externer Kommunikationsaufwand.

Heuristisches Vorgehen,

d.h. Nachdenken,
wird am Besten unterstützt,
wenn es nicht unterbrochen wird –
weder durch Kommunikation,
noch durch Navigation/Suche.

Bei Unterbrechungen könnte zum Tragen kommen, was G. Miller 1957 in einem Experiment herausgefunden hat, nämlich dass das menschliche Kurzzeitgedächtnis nicht mehr als sieben Informationsquanten gleichzeitig aufnehmen kann.

Miller, G.A. (1957). The magical number 7 plus or minus two: some limits on our capacity for processing information. *Psychological Review, 63, 81-97*

Unzusammenhängende*) Architektur

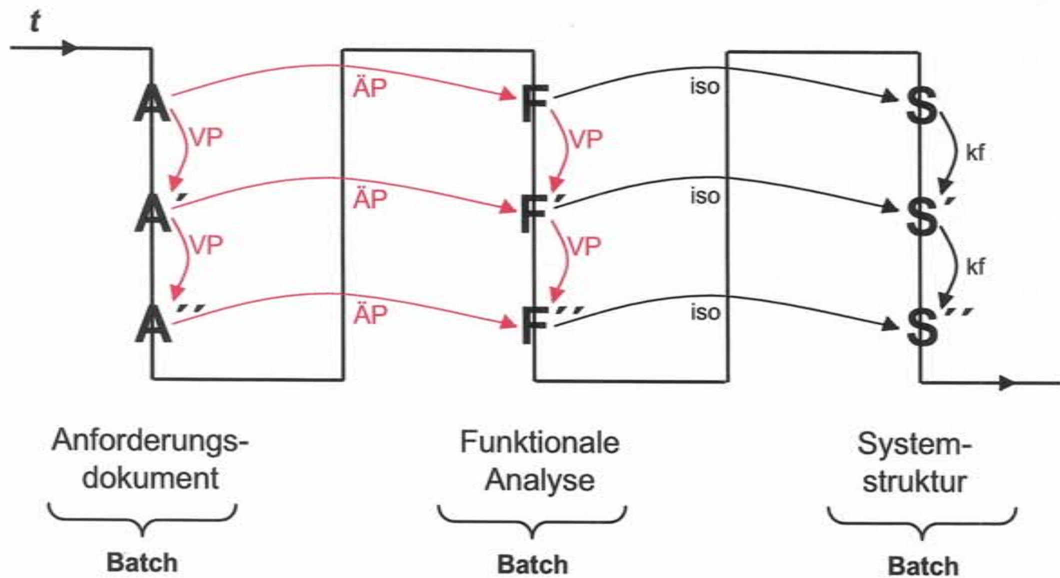
Geltendes Architektur-Paradigma

- Unzusammenhängende Dokumentation der Aspekte „Anforderung“, „Funktion“ und „Struktur“.
- Dokumentation eines jeden Aspekts in Form unzusammenhängender Einzeldiagramme.
- „Software is very difficult to visualize. Whether we diagram control flow, variable scope nesting, variable cross-references, data-flow, hierarchical data structures, or whatever, we feel only one dimension of the intricately interlocked software elephant. If we superimpose all the diagrams generated by the many relevant views, it is difficult to extract any global overview. ... There is no natural single mapping from a conceptual design to a diagram. ... One needs multiple diagrams.“ (F. Brooks, The Mythical Man-Month)
- „The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. ... Systems are composed of many structures. No single view can appropriately represent anything but a trivial architecture“ (Bass et al., Software Architecture in Practice)

*) Möglicherweise bedingt durch Fehlen einer passenden Visualisierung, sowie durch kleine Bildschirme.

Entwicklung von Anforderung, Funktion, Struktur

Batch-Vorgehensmodell



Legende:

ÄP: Äquivalenzproblem, zeitlicher und physischer Abstand, unterschiedliche Personen
VP: Verfeinerungsproblem, physischer Abstand
iso: isomorph, unkritisch
kf: kontextfrei, unkritisch

Unzusammenhängende Architektur und Batch-Vorgehensmodell

Effizienzbetrachtung

- Unzusammenhängende Dokumentation der Aspekte „Anforderung“, „Funktion“ und „Struktur“ bedingt Unterbrechungen der Heuristik durch externe und zeitversetzte Kommunikation. Behindert gleichzeitige Verfeinerung von Anforderung, Funktion und Struktur. Traceability-Problem.

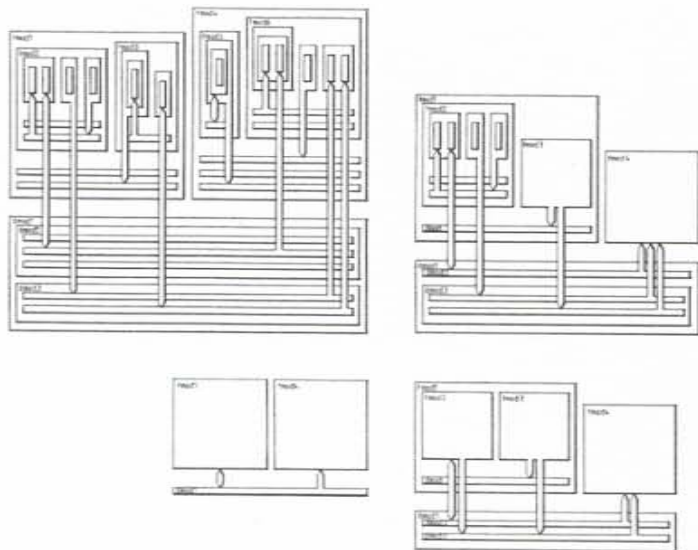
- Unzusammenhängende Dokumentation innerhalb eines Aspekts in Form von Einzeldiagrammen bedingt Unterbrechungen der Heuristik durch Navigation/Suche. Lokalitätsproblem. Weitere Unterbrechungen gegebenenfalls durch externe und zeitversetzte Kommunikation.

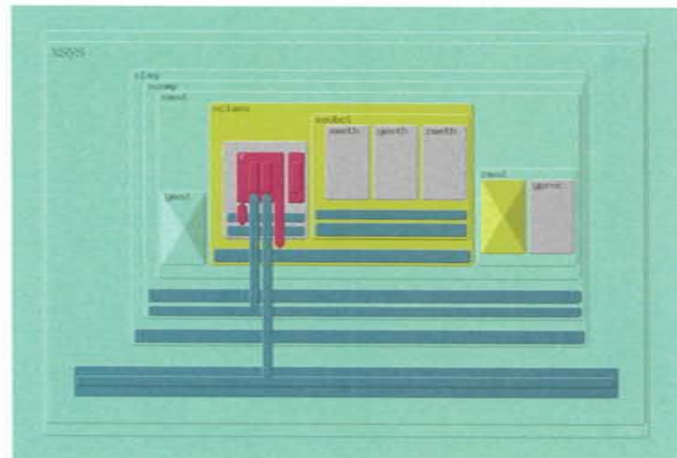
- Eine unzusammenhängende Software-Architektur behindert heuristisches Vorgehen durch Unterbrechungen und gewährleistet somit nicht die Effizienz des Entwicklungsprozesses.

Zusammenhängende *) Architektur

Gedankenexperiment

- Zusammenhängende Modularisierungsstruktur auf der Basis einer attributierten kontext-freien Grammatik (Beispiel: Fractool).
- Jeder Knoten repräsentiert Anforderung, Funktion und Struktur.
- Abstraktionen möglich, ohne den Zusammenhang zu verlieren.

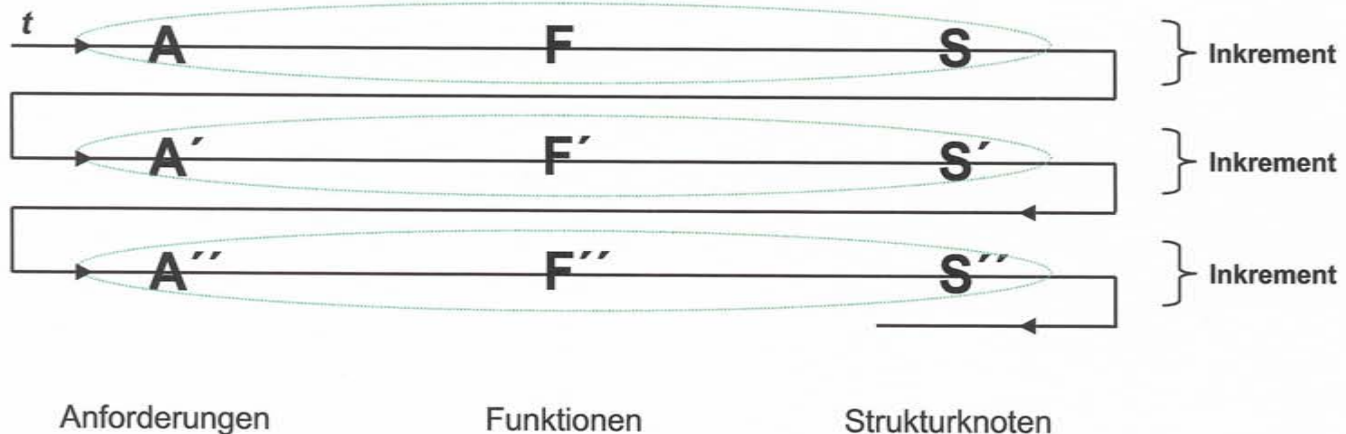




*) Analog zu VLSI-Schaltkreisen

Entwicklung von Anforderung, Funktion, Struktur

Inkrementelles Vorgehensmodell

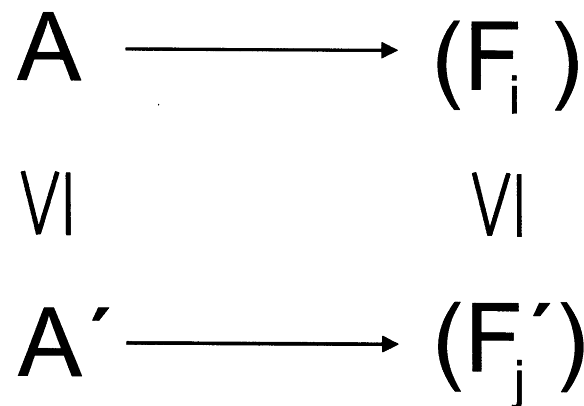


Legende:

Von derselben Person und zeitnah durchgeführt (lange Transaktion);
Heuristische Lösung der Äquivalenz- und Verfeinerungsprobleme wird nicht durch externe Kommunikation erschwert;
Vertikale Arbeitsteilung folgt der Modularisierungsstruktur.

Inkrementelles Vorgehensmodell

Anforderungen und funktionale Elemente



Legende:

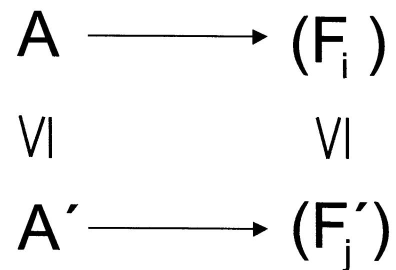
A, A': Anforderungen
(F_i): Funktionale Elemente, die A umsetzen
(F'_j): Funktionale Elemente, die A´ umsetzen
$A' <= A$: A´ Unteranforderung von A
$(F'_j) <= (F_i)$: (F'_j) Unterknoten von (F_i)

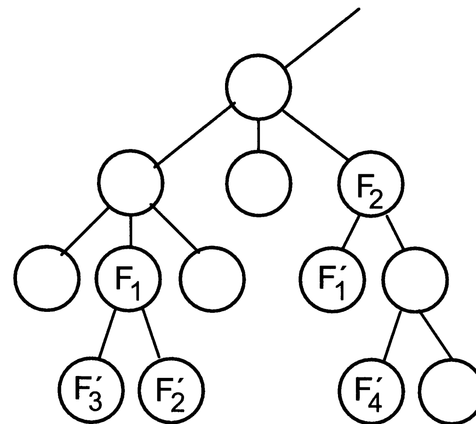
Beobachtung:
Wenn eine Anforderung A durch die funktionalen Elemente (Fi) umgesetzt wird, dann wird auch jede Unteranforderung von A durch dieselben Elemente (Fi) – und deren Unterelemente (F´j) – umgesetzt („schichtenweise“ Realisierung der Anforderungen).

Inkrementelles Vorgehensmodell

Anforderungen und funktionale Elemente

Beispiel für i=1,2 und j=1,2,3,4





Mögliche Nutzung als Architektur-View:
Ein Zoom-In in die Anforderungen kann begleitend visualisiert werden durch ein (verteiltes) Zoom-In in den Strukturbaum.

Zusammenhängende Architektur und Inkrementelles Vorgehensmodell

Effizienzbetrachtung

- Ermöglicht gleichzeitige Verfeinerung von Anforderung, Funktion und Struktur - auch bei unvollständigen Anforderungen. Keine Unterbrechung der Heuristik durch externe Kommunikation oder Navigation/Suche. Kein Traceability-Problem, kein Lokalitätsproblem. Unterstützt Validierung und Verifikation über den gesamten Lebenszyklus.

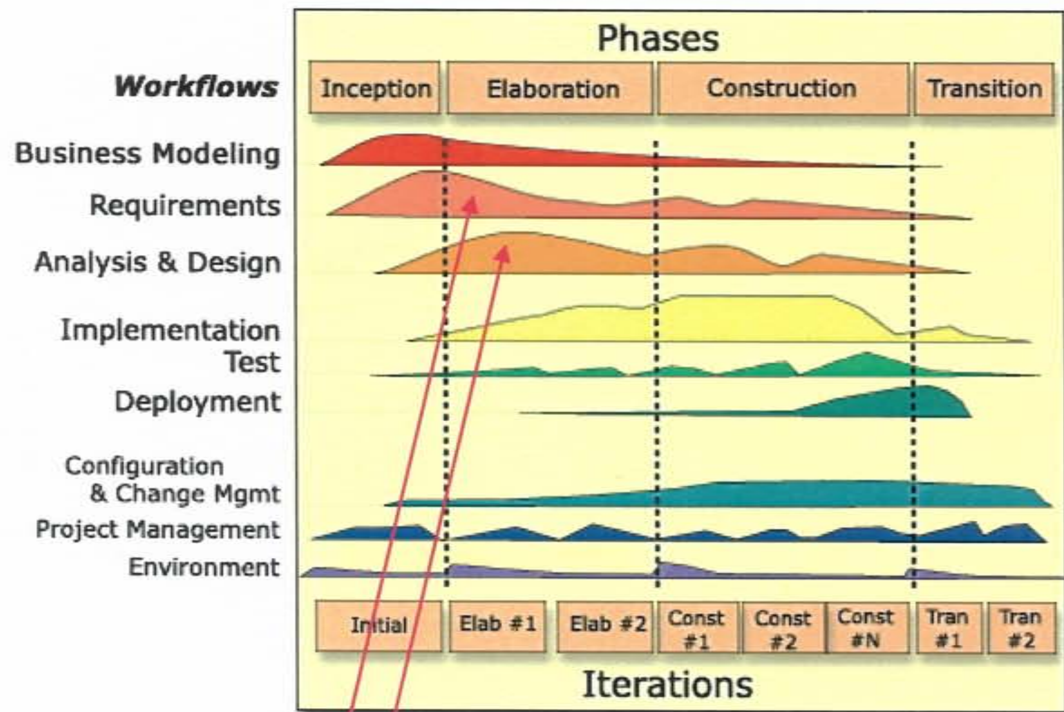
- Bei Arbeitsteilung entlang der Modularisierungsstruktur, geringe externe Kommunikation.

- Eine zusammenhängende Architektur fördert die Kontinuität heuristischen Vorgehens und gewährleistet damit die Effizienz des Entwicklungsprozesses.

Weiter unten betrachtet:
Zusammenhang der Darstellung ist Grundvoraussetzung für homomorphe Abbildungen / Ähnlichkeit / Verständlichkeit.

Rational Unified Process

Als „inkrementeller“ Prozess



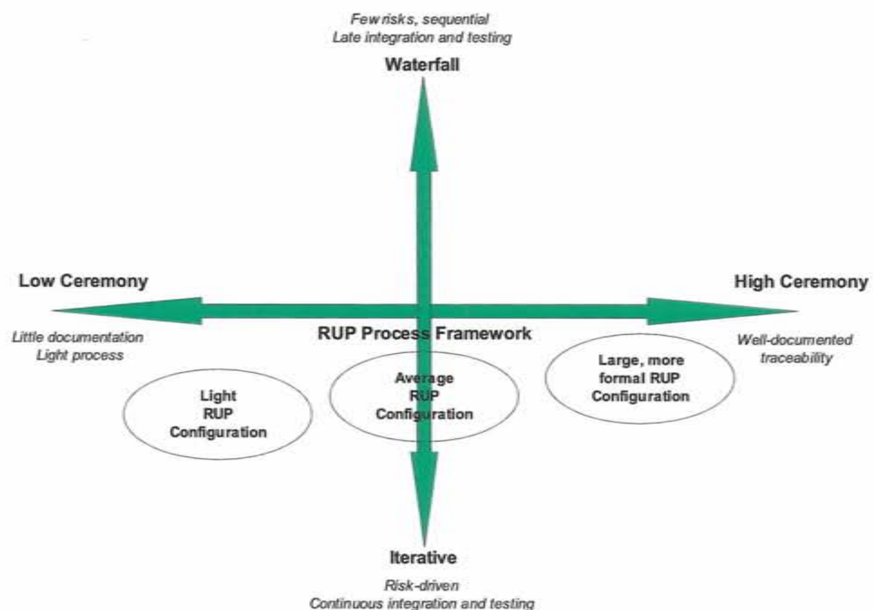
Gleichzeitige Verfeinerung von Anforderungen, Funktion und Struktur

Rational Unified Process

Als „inkrementeller“ Prozess

- The RUP strongly promotes an iterative, risk-driven approach to software development with continuous testing and integration.
- The RUP uses a sequence of incremental steps or iterations.
- In each iteration you do a little requirements, analysis, design, implementation and testing.
- Waterfall approach matched with pipeline organization: analysts send the completed requirements to designers, who send a completed design to programmers, ... integrators ... testers.
  These many handoffs are sources of errors and misunderstandings ... RUP widens the scope of expertise of the team members, allowing them to play many roles ..., while simultaneously removing harmful handoffs.
- Tool technologies to support higher ceremony approach, without bearing the costs of documentation overhead: e.g. visual modeling support providing synchronization between requirements, design, and implementation.

Prozess-Konfigurationen



P. Kroll, Ph. Kruchten: The Rational Unified Process made easy, A practitioner's guide to the RUP, Addison-Wesley 2003

Konsequenzen

der Äquivalenz- und Verfeinerungsprobleme

insbesondere für die Übergänge zwischen den Artefakten
im Laufe des Software-Entwicklungszyklus.

Konsequenz aus Satz 1 (Rice):

Design-Information*) ist unersetzlich. Sie kann nicht durch verfeinerte Design-Information kompensiert, noch bei Verlust rekonstruiert werden.

Konsequenz aus Satz 2:

Es kann grundsätzlich nicht festgestellt werden, ob Design-Information eine Verfeinerung darstellt, noch ob Design-Information verlorengegangen ist.

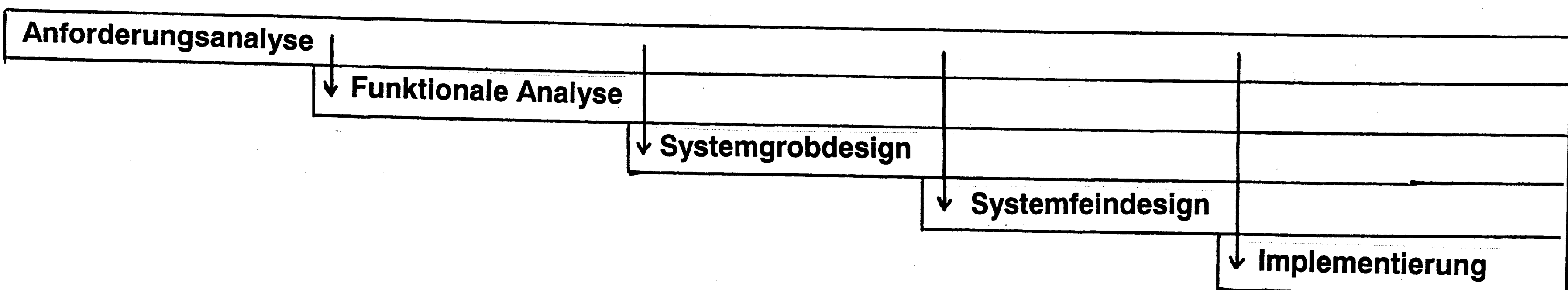
---

*) d.h. Information, die den Zweck eines Programms definiert

Konsequenz

für den Architekturprozess

Präzisierung der Architektur
durch Aggregation[1] – nicht durch Transformation



Vorteile:

- Durch Aggregation geht keine Information verloren, noch wird sie durch neue Information verdrängt. Jederzeit ist die ursprüngliche Absicht erkennbar, und kann mit neu hinzugekommener Information auf Konsistenz geprüft werden. Auch ist so die Rückverfolgbarkeit (*traceability*) eines Implementierungsschritts zu den Requirements gewährleistet.

- Wird die über die Phasen aggregierte Information nicht nach Phasen, sondern gemäß der Architektur angeordnet, so befindet sie sich jeweils in unmittelbarer Nähe der Implementierungsdetails (*locality*), d.h. direkt am kritischen Übergang von der intuitiven zur formalen Beschreibung. Hierdurch wird die Übereinstimmung der Implementierung mit den Requirements gefördert.

[1] Anhäufung, Ansammlung, Vereinigung; lat. *aggregare* „beigesellen“

Aggregation

aller intuitiven Beschreibungen, und ihre

Einbettung

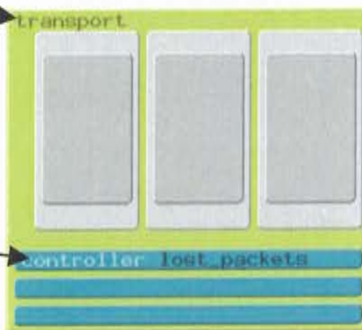
in die Programmstruktur

Clickzonen
für die Eingabe der intuitiven Beschreibung

Höherer Zweck
Information aus den frühen Entwicklungsphasen

Niederer Zweck
Information aus den späten Entwicklungsphasen

Klassenname und weiterer Kommentar in Form natürlicher Sprache, Diagramme etc.



Klassen-/Objektname und weiterer Kommentar in Form natürlicher Sprache, Diagramme etc.

Methodenname und weiterer Kommentar in Form natürlicher Sprache, Diagramme etc.

Kritischer Übergang
von der intuitiven zur formalen Beschreibung (Implementierung).

Für den Implementierungsschritt müssen die intuitiven Beschreibungen aller übergeordneten Moduln komprimiert und lokal zur Verfügung gestellt werden (*locality; vgl. mit Frac-Tool-Funktion FOCUS*)

vgl. mit Syntax eines Aussagesatzes (Aussage über den Zweck des Programms)

Architektur und Schnittstellen

- Als klassische Aufgabe der Software-Architektur wird oft die Definition von Schnittstellen angesehen.

- Dies stimmt mit der bisher entwickelten These insofern überein, als die Software-Architektur die Anforderungen strukturiert herunterzubrechen hat bis auf die kritische Übergangsstelle zur Implementierung, nämlich auf die Signaturen der Schnittstelle.

Design-Information

Sortier- und Verteilungsaspekte

- **Entwicklungsphasen:**
  Die Information, die während der schrittweisen Systemverfeinerung anfällt, wird üblicherweise nach Phasen sortiert und in Phasen-Dokumenten zusammengefasst. Die Sortierung und Verteilung erfolgt üblicherweise nicht nach der Programmstruktur. Entsprechend sind Probleme bekannt bzgl. *locality* und *traceability*.
- **Programmstruktur:**
  Design-Information lässt sich nach der Programmstruktur invertiert anordnen. Die zeitliche Entstehungsfolge der Information entspricht damit top-down der Verschachtelungstiefe der Programmstruktur. Jedem Programmelement ist damit genau die Information zugeordnet, die es betrifft. Die Programmstruktur eignet sich aus folgenden Gründen als Primärschlüssel: a) Die Programmstruktur ist *eindeutig identifizierend*; b) die Funktion, und damit die Programmstruktur kann als *wesentlicher* Aspekt der Design-Information bezeichnet werden. Alle anderen Aspekte können als *akzidentell* bezeichnet werden und bieten sich somit an, als *View* aus der Programmstruktur abgeleitet zu werden. Dies steht im Gegensatz zu der Auffassung, dass ein Programmsystem notwendigerweise aus verschiedenen separaten, nicht weiter integrierbaren Diagrammen bestehen müsse.
- **Entwurfsmuster:**
  Ein Beispiel für eine View, die sich aus der Programmstruktur ableiten lässt, ist die Menge aller Programmelemente, die ein ausgewähltes Entwurfsmuster bilden. Die Programmelemente können vor dem Hintergrund der Programmstruktur farblich hervorgehoben werden.
- **Features:**
  Ein weiteres Beispiel für eine View, die sich aus der Programmstruktur ableiten lässt, ist die Menge der Programmelemente, die ein ausgewähltes Leistungsmerkmal implementieren. Auch hier können die beteiligten Programmelemente vor dem Hintergrund der Programmstruktur farblich hervorgehoben werden.

Entsprechendes sollte auch für andere Relationen oder Ergebnismengen von Programmelementen gelten, die ein Suchkriterium erfüllen.

Aggregation und Vererbung

Nachbetrachtung

- Wir haben gesehen, dass die Architektur eines Programms im Laufe des Verfeinerungsprozesses nicht durch Transformationen sondern durch Aggregationen von Design-Information entsteht.

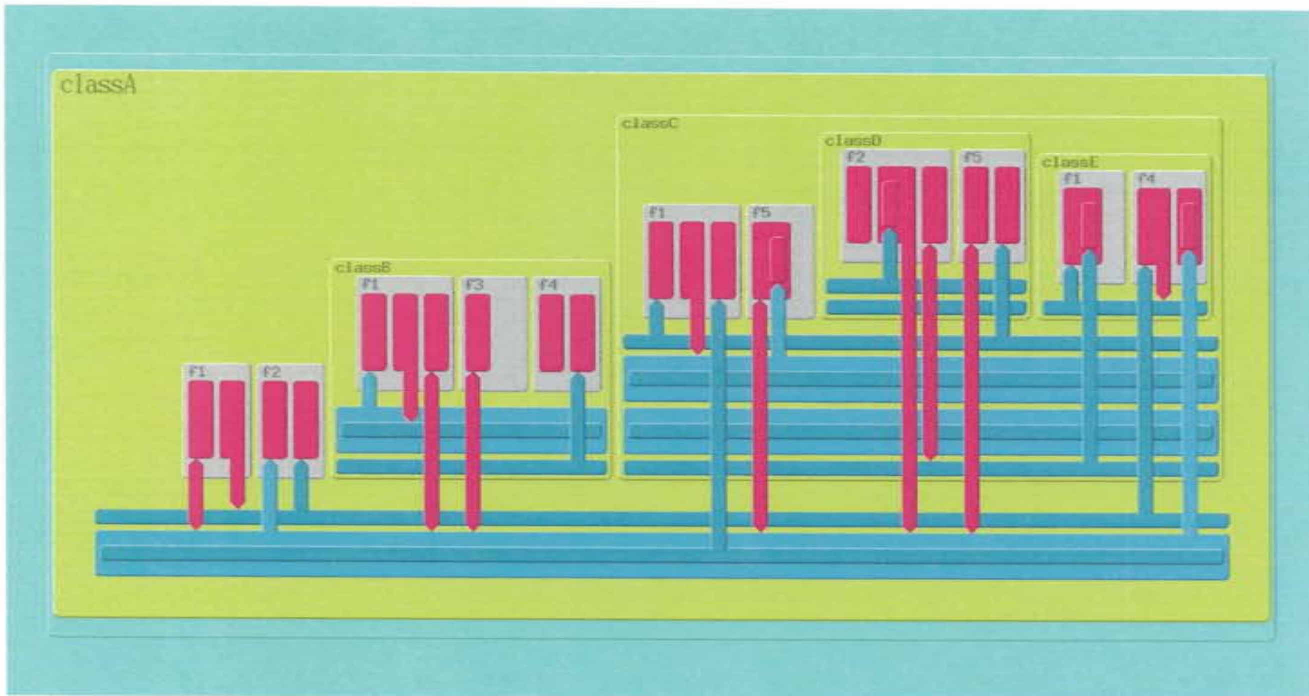
- Dies deckt sich mit dem Verfeinerungsprozess für Klassen. Auch dort wird eine Verfeinerung, nämlich die Unterklasse, gebildet durch Aggregation von Merkmalen der Oberklasse mit neuen zusätzlichen Merkmalen der Unterklasse *(die neuen Merkmale der Unterklasse entstehen nicht etwa durch Transformation der Merkmale der Oberklasse, sondern sind – relativ – frei gewählt).*

- Die Analogie zur Klassenvererbung weist darauf hin, dass auch hier die Aggregationsschritte im Laufe der Verfeinerung den Abstraktionsstufen einer Systemhierarchie entsprechen.

- Wir sind damit beim Hauptthema und wichtigsten Konstruktionsprinzip der Software-Architektur, der Abstraktion angelangt.

- Nachdem wir gesehen haben, dass auch unsere bisherige Untersuchung im Grunde genommen nichts anderem als dem Abstraktionsvorgang gegolten hat, werden wir im Folgenden weitere Facetten der Abstraktion betrachten.

Class Hierarchy



Abstraktion

„Der Abstraktionsprozess ist
das wichtigste *Konstruktionsprinzip*
der klassischen Mathematik.“

[G. Klaus, M. Buhr: PhilWB]

Wir gehen davon aus, dass der
Abstraktionsprozess auch das wichtigste
Konstruktionsprinzip der Informatik ist.

Abstraktion

Abstraktion [*abstrahere*, lat., wegziehen, abziehen, entfernen]: **I.** Resultat des gedanklichen Abgehens, Absehens von bestimmten Eigenschaften aus einer Menge von Eigenschaften des zu untersuchenden konkreten Gegenstands. [N.I. Kondakow: WBdLogik]

Die A. kann ein sinnlich-anschauliches Modell sein, z. B. ein Atommodell, sie kann ein *Urteil*, z. B. »dieser weiße Gegenstand« oder ein *Begriff* sein, wenn z. B. eine Gesamtheit von Merkmalen, Eigenschaften, Seiten und Zusammenhängen eines Gegenstands oder einer Klasse von Gegenständen abstrahiert wurde, deren Kern die Kenntnis der wesentlichen Merkmale, Eigenschaften, Seiten und Zusammenhänge eines Gegenstands oder einer Klasse von Gegenständen ist. Die A. kann auch eine *Kategorie*, d. h. der umfassendste Begriff einer bestimmten Wissenschaft, sein, z. B. eine philosophische Kategorie wie *Materie, Bewegung, Raum, Qualität* oder *Quantität*.

II. Man spricht von *isolierender* oder *analytischer A.*, wenn Eigenschaften von Gegenständen und anderen Eigenschaften, mit denen sie untrennbar zusammenhängen, gedanklich entfernt und mit einem besonderen Namen bezeichnet werden. Durch eine derartige A. werden abstrakte allgemeine Begriffe gebildet, z. B. *Wärmekapazität, Unbeweglichkeit*.

Abstraktion

abstrahieren [*abstrahere* lat., abziehen, wegziehen, entfernen]: **I.** gedanklich heraussondern, herausgliedern einzelner interessierender Merkmale, bzw. Eigenschaften, Zusammenhänge und Beziehungen eines konkreten Gegenstands oder einer konkreten Erscheinung und ihr gedankliches Entfernen von der Menge der anderen Merkmale, bzw. Eigenschaften, Zusammenhänge und Beziehungen dieses Gegenstands.

Im *Abstraktionsprozeß* „reinigt“ der Mensch gleichsam den Gegenstand der Untersuchung von nebensächlichen Merkmalen, Eigenschaften, Zusammenhängen und Beziehungen, deren Kenntnis nicht nur den Untersuchungsgang nicht fördert, sondern oft auch erschwert.

In der Tat, welchen Gegenstand wir auch untersuchen mögen, wir brauchen nicht ausnahmslos alle seine Eigenschaften zu kennen. Die Erfahrung zeigt, daß man zur echten Erkenntnis eines Gegenstands oder einer Erscheinung die wesentlichen Eigenschaften herausstellen und von den zufälligen trennen muß. Stellen wir uns die Aufgabe, aus einer Reihe von Gegenständen den auszuwählen, mit dem man Glas schneiden kann, richten wir unsere Aufmerksamkeit auf eine einzige Eigenschaft des gesuchten Gegenstandes, auf die Härte. Und gerade ein solcher Gegenstand ist der Diamant. Bei der Auswahl des von uns gebrauchten Gegenstands haben wir von allen übrigen Eigenschaften der vor uns liegenden Gegenstände abstrahiert, indem wir sie als unwesentlich betrachteten. Im Denkprozeß verwirft der Mensch das Zufällige, Unwesentliche und kommt zur *Erkenntnis des Notwendigen, des Wesentlichen.*

Das zeitweilige Absehen von einer Reihe von Eigenschaften, Merkmalen und Zusammenhängen des zu untersuchenden Gegenstands ist durchaus notwendig, da nur der in „ reiner Form“ genommene Gegenstand dem Untersuchenden verständlich wird.

II. *Welche Merkmale, Eigenschaften, Zusammenhänge, Beziehungen werden gedanklich abstrahiert?* – Wenn die Aufgabe gestellt ist, das Wesen eines Gegenstands, einer Erscheinung aufzudecken, werden im Abstraktionsprozeß grundlegende allgemeine Merkmale, Eigenschaften, Zusammenhänge und Beziehungen ausgewählt und zufällige, nebensächliche, unwesentliche werden fallengelassen. Durch ein derartiges Abstrahieren werden Begriffe, Kategorien, d. h. die umfassendsten Begriffe, geschaffen, in denen wesentliche Merkmale der Gegenstände und Erscheinungen der Wirklichkeit widergespiegelt werden.

Abstraktion ist der Prozeß des Abgehens vom Konkreten, von einer Reihe von Eigenschaften, Zusammenhängen und Relationen des materiellen Objekts. Im Verlaufe der Abstraktion werden oft Begriffe geschaffen, z. B. der des *absolut schwarzen Körpers,* in denen die Gesamtheit der wesentlichen Merkmale enthalten ist, die nicht genau den Merkmalen der realen Gegenstände entsprechen, da es einen absolut schwarzen Körper, wie ihn die Physiker verstehen, in der objektiven Realität nicht gibt. Dieses abstrakte Wissen ist in dem Sinne konkret, daß es sich von dem im Verlaufe der lebendigen Anschauung gewonnenen konkreten Wissen dadurch unterscheidet, daß es die Synthese aus *Wissen vom Wesentlichen* ist, das nicht sinnlich anschaulich ist, und aus der Kenntnis anderer Eigenschaften des untersuchten Objekts, die im Lichte der Kenntnis des Wesentlichen verstanden werden.

[N.I. Kondakow: WBdLogik]

Abstraktion

Abstraktion [lat] – 1. wichtiges Moment des Erkenntnisprozesses beim Übergang von der sinnlichen zur rationalen Erkenntnis, 2. das Resultat dieses Prozesses. Der Abstraktionsprozeß besteht allgemein darin, daß in einer Reihe von analytischen Denkakten, in denen das konkrete Sinnesmaterial verarbeitet wird, von bestimmten Merkmalen, Eigenschaften und Beziehungen des betreffenden Gegenstandes abgesehen wird, andere dagegen als wesentlich herausgehoben und zugleich variabel gemacht werden. Als Ergebnis des Abstraktionsprozesses, der eng mit der Verallgemeinerung verbunden ist, entstehen Begriffe, die das Wesen der Gegenstände widerspiegeln.

Das Verfahren der Abstraktion enthält eine Reihe unterschiedlicher Aspekte. Die *klassischen Formen der Abstraktion* sind die folgenden:

a) Die *generalisierende Abstraktion* sondert die unwesentlichen Eigenschaften der Dinge, Relationen usw. aus und hebt die wesentlichen hervor. Der Begriff der wesentlichen Eigenschaft ist dabei relativ, ohne subjektiv zu sein. Er bezieht sich immer auf ein bestimmtes wissenschaftliches System (die vergleichende Anatomie wird bei der Untersuchung der Klasse der Säugetiere andere Merkmale für wesentlich ansehen als etwa die Biochemie). Nach moderner Auffassung betrachtet man als Aufgabe der generalisierenden Abstraktion das Auffinden von Invarianzen.

b) Die *isolierende Abstraktion* löst bestimmte Eigenschaften, Beziehungen usw. von Klassen von Gegenständen gedanklich aus ihrem Zusammenhang heraus und verleiht ihnen gewissermaßen selbständige Existenz. Eine solche Verselbständigung erleichtert die Erkenntnisgewinnung dadurch, daß sie den zu untersuchenden Bereich übersichtlicher gestaltet. Wenn man sie jedoch verabsolutiert, verfällt man zumindest in den Fehler einer unzulässigen Simplifizierung, evtl. sogar in objektiven Idealismus.

c) Die *idealisierende Abstraktion* schafft Begriffe, indem sie zuvor ideale Objekte, Klassen usw. konstruiert (→ Idealisierung), denen diese Begriffe dann entsprechen.

Eine *dialektische Auffassung* der Abstraktion wurde zum erstenmal von HEGEL entwickelt. Er hielt die Abstraktion, das abstrahierende Denken für ein wesentliches Moment des Erkenntnisprozesses und wandte sich gegen die Vorstellung, daß beim Abstrahieren nur zu unserem subjektiven Behuf das eine oder andere Merkmal weggelassen werde. «Das abstrahierende Denken ist daher nicht als bloßes Auf-die-Seite-Stellen des sinnlichen Stoffes zu betrachten, welcher dadurch in seiner Realität keinen Eintrag leide, sondern es ist vielmehr das Aufheben und die Reduktion desselben als bloßer Erscheinung auf das Wesentliche, welches nur im Begriff sich manifestiert» (Logik II, 226).

Nach *dialektisch-materialistischer Auffassung* ist die Abstraktion ein wichtiges Erkenntnisinstrument, das es gestattet, aus dem Material der Sinneserfahrung die wesentlichen, notwendigen, allgemeinen Beziehungen und Eigenschaften der Gegenstände herauszuheben, um von der Erscheinung zum Wesen gelangen zu können. Die Möglichkeit der Abstraktion ist objektiv bedingt, denn die materielle Welt ist keine Anhäufung isolierter Einzeldinge, sondern eine zusammenhängende Mannigfaltigkeit, in der objektiv Klassen, allgemeine Beziehungen existieren. Der Abstraktionsprozeß ist darauf gerichtet, die (im gegebenen Fall) unwichtigen Eigenschaften, Beziehungen, Umstände usw. abzusondern, die wesentlichen, für das Verhalten des Gegenstandes bestimmenden, herauszuheben, eine Reihe gemeinsamer Eigenschaften und Merkmale variabel zu machen, um auf diese Weise im Begriff das Wesentliche einer Klasse von Gegenständen in reiner, von allen störenden Einflüssen befreiter Form oder in idealisierter Form zu erfassen.

[G. Klaus, M. Buhr: PhilWB]

Abstraktion

- **abstrahieren** :
  - das *Wesentliche* aus dem Zufälligen[1] herausheben; zum Begriff erheben; verallgemeinern. [Wahrig]

- **Ergebnis der Abstraktion** :
  - Erkennen und Darstellen des *Wesentlichen*.

- **Abstraktion, Wesen, Funktion** :
  - Das *Wesentliche* an einem Programm ist seine Funktion[2].
  - Die Abstraktion eines Programms soll dieses *Wesentliche* wiedergeben.

- **Abstraktion, Software-Architektur**
  - Die Software-Architektur ist eine Abstraktion der Implementierung.
  - Die Software-Architektur muss die Funktion und somit das *Wesen* eines Programms vollständig erklären.
  - Damit die Software-Architektur das *Wesentliche* eines Programms ausdrücken kann, müssen ihre textuellen und strukturellen Anteile mit seiner Funktion intuitiv übereinstimmen[3].

---

[1] „Zufällig" ist etwa die Implementierung einer Funktion,die ja grundsätzlich immer nur eine von unendlich vielen möglichen Implementierungen darstellt. Die Abstraktion einer Funktion ist etwa ihr Name oder ihre intuitive Beschreibung. Dass auch der Name bzw. die Beschreibung einer Funktion wieder nur eine von vielen Möglichkeiten darstellt, weist auf den relativen Charakter der Abstraktion hin.

[2] Wir haben oben festgestellt, dass die Funktion eines Programms mit der Absicht übereinstimmen sollte, die durch die Architektur festgelegt wird. Somit könnte man in diesem Zusammenhang schließen, dass das *Wesen* eines Programms mit der damit verbundenen *Absicht* übereinstimmt. Abstraktion wäre demnach gleichbedeutend mit dem „Aufdecken eines Plans". Die Software-Architektur stellt in ihrer schrittweisen Verfeinerung einen solchen Plan dar. Eine Absicht setzen wir bei technisch konstruierten Systemen stets voraus. Für manche Menschen erscheint es aber auch selbstverständlich, den Erscheinungen der Natur eine Absicht zuzusprechen.

[3] Als Grundlage der Übereinstimmung wird später der Begriff des „Homomorphismus" eingeführt. Für die Software-Architektur werden entsprechend Homomorphieeigenschaften gefordert.

Abstraktionsprinzip

Prinzip, aufgrund dessen jede Aussageform $P(x)$ eine bestimmte Menge M definiert, deren Elemente genau die Gegenstände a sind, für die $P(a)$ eine wahre Aussage ist.

Anwendungen:

- I ist Abstraktion der Äquivalenzklasse $[F]_I$, welche alle Turingmaschinen umfasst, auf die das intuitive „Prädikat" I zutrifft. Diese Turingmaschinen sind hinsichtlich I äquivalent, auch wenn sie unter Umständen verschiedene Funktionen berechnen.

- Eine Turingmaschine T ist Abstraktion ihrer Input-Output-Relation, d.h. der Menge $\{(i,o) \mid o = T(i)\}$, die alle Ein- und Ausgaben (i,o) umfasst, die durch T ineinander überführt werden. Die Paare (i,o) sind damit hinsichtlich T äquivalent.

P definiert dabei eine Äquivalenzrelation auf Gegenständen *a, b, c ...*, die dadurch hinsichtlich der Eigenschaft P äquivalent werden.

Abstraktionsstufen

am Beispiel einer Menge, die durch das Prädikat *P* definiert ist

$$M = \{x \mid P(x)\}$$

Intuitive Beschreibung (= Name der Turingmaschine, falls P berechenbar ist)

Formale Beschreibung (= Turingmaschine, falls P berechenbar ist)

- Das formale Prädikat *P* (bzw. die ihm zugeordnete Turingmaschine) ist eine Abstraktion[1] der Menge $\{x_1, x_2, \ldots\}$
- Der intuitive Name *M* ist eine Abstraktion[2] des Prädikats *P* (bzw. der ihm zugeordneten Turingmaschine)
- Die obige Mengendefinition kann demnach folgendermaßen aufgefasst werden:
    - *M* als die intuitive Funktionsbeschreibung eines Programms
    - *P* als die formale Implementierung des Programms, und
    - $\{x_1, x_2, \ldots\}$ als die Ausführung[3] des Programms.
- Nach dem Satz von Rice ist die Abstraktion von $\{x_1, x_2, \ldots\}$ nach *P* nicht entscheidbar[4].
- Nach Korollar 2.1 und 2.4 ist die Abstraktion von *P* nach *M* nicht entscheidbar[5].

---

[1] Dabei nennt man $\{x_1, x_2, \ldots\}$ die *extensionale* Beschreibung der Menge (Umfang) und *P* die *intensionale* Beschreibung (Absicht).

[2] Hier ist *M* die intensionale Beschreibung von *P*. *P* ist in diesem Fall extensionale Beschreibung von *M*, denn genaugenommen ist *P* wieder Repräsentant einer Extension, nämlich der Menge aller zu *P* äquivalenten Prädikate – mehr noch: der Menge aller Prädikate, auf die die intuitive Beschreibung *M* zutrifft.

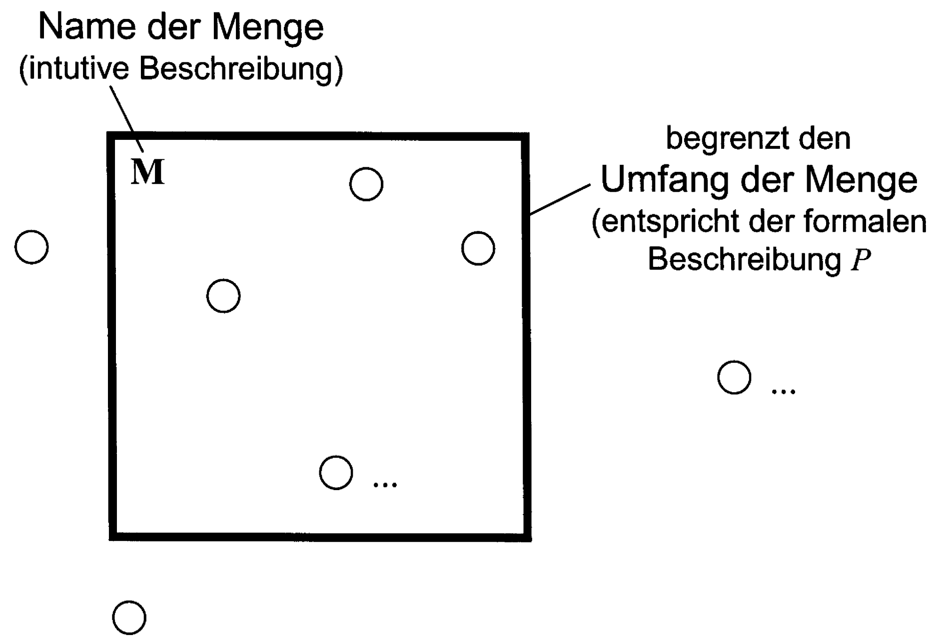
[3] Als „Ausführung“ des Programms kann etwa die rekursive Aufzählung der Elemente gelten.

[4] Wenn auch grundsätzlich unentscheidbar, so ist diese Abstraktion in abgeschwächter Form doch technisch handhabbar und von Interesse. So kann etwa durch Methoden der grammatikalischen Inferenz aus endlich vielen Beispielen und Gegenbeispielen von Wörtern einer regulären Sprache eine Grammatik „erlernt“ werden, die als Hypothese die bis dahin bekannte Extension der Sprache formal erklärt (Biermann, 1972).

[5] Dass es nicht entscheidbar ist, ob eine TM eine intuitive Spezifikation erfüllt, ist weniger erstaunlich als dass es nicht entscheidbar ist, ob eine TM eine formale Spezifikation (Satz von Rice) erfüllt.

Abstraktion

Visualisierung einer Menge $M = \{x \mid P(x)\}$



Diese Visualisierung wurde bei der Programmiersprache FracTool verwendet.

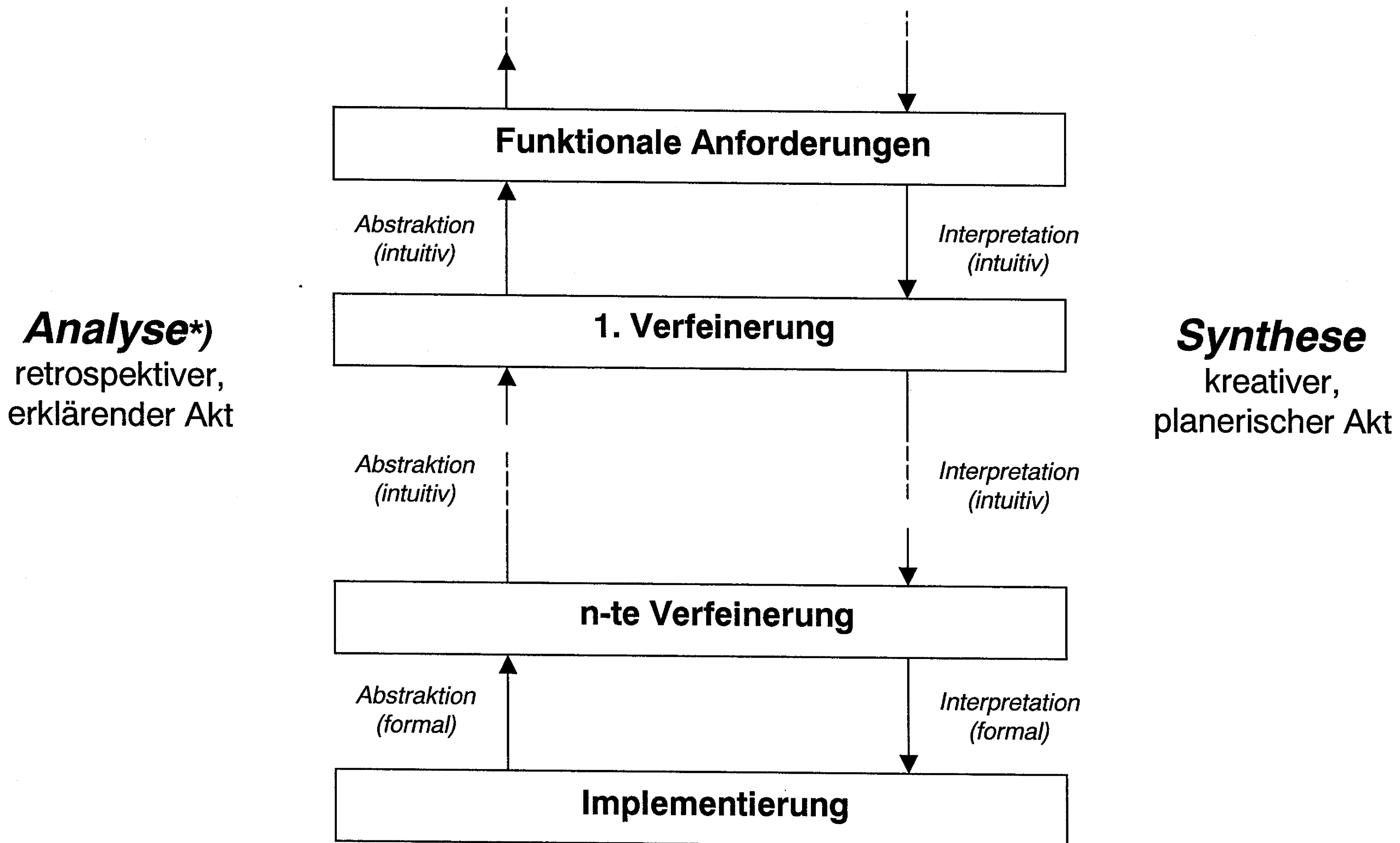
Ohne Interpretation keine Abstraktion

Ohne Plan
kein „Aufdecken“ des Plans

Erläuterung:

- Interpretation ist die Umkehrabbildung der Abstraktion.
- Während die Interpretation einen Verfeinerungsschritt darstellt, macht die Abstraktion den Verfeinerungsschritt wieder rückgängig.
- Jede abstrakte Darstellung kann nur die Information wiedergeben, die zuvor durch Interpretation erzeugt wurde.
- Beispiel 1: Die *Implementierung* einer Methode ist die „Interpretation“ der *Signatur* der Methode.
- Beispiel 2: Das *Fein-Design* eines Systems ist die „Interpretation“ des *Grob-Designs* des Systems.
- Beispiel 3: Die Menge $[F]_I$ ist die „Interpretation“ der intuitiven Beschreibung I.
- Beispiel 4: Die Booleschen Funktionen *und*, *oder*, etc. sind „Interpretationen“ der Symbole $\wedge$, $\vee$, etc., während die *Aussagenvariablen*, und damit der gesamte *Boolesche Ausdruck*, durch die Belegung mit *Wahrheitswerten* „interpretiert“ werden.

Abstraktionen und Interpretationen



*) Hier kann nur die abstrakte Information zur Erlärung herangezogen werden, die zuvor (a priori) während des Synthesevorgangs bereitgestellt wurde. Generiert werden kann abstrakte Information aus den bekannten Gründen nicht.

Interpretation und Bedeutung[1]

Interpretation

- Eine Interpretation ordnet Variablen und Symbolen eine Bedeutung zu, indem sie
    - Subjekt-Variablen auf eine Menge von Individuen abbildet,
    - Funktor-Variablen auf Funktionen, und
    - Prädikator-Variable auf Prädikate
    - Beispielsweise wird im Kontext der Formalen Sprachen das Symbol ‚o' durch die Konkatenation interpretiert

- Durch Interpretation wird
    - der *Signatur* eines Programms Bedeutung in Form der *Implementierung* zugeordnet (intensionale Sicht), und
    - dem *Aufruf* eines Programms Bedeutung in Form seiner *Ausführung* (extensionale Sicht).

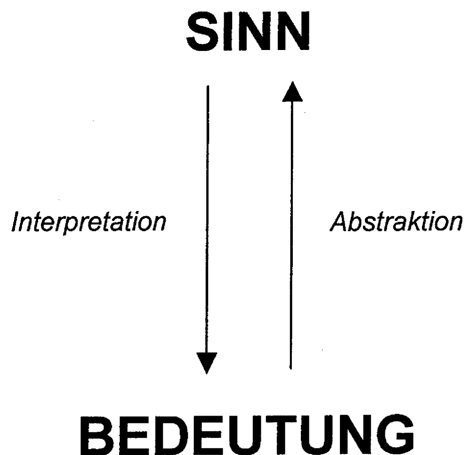
Abstraktion

- Im Gegensatz zur Interpretation wird durch die Abstraktion Bedeutung entfernt, beispielsweise
    - durch die Reduktion einer *Implementierung* auf ihre *Signatur* (intensionale Sicht), und
    - durch die Reduktion einer *Ausführung* auf ihren *Aufruf* (extensionale Sicht).

[1] Bedeutung = Semantik

Sinn und Bedeutung[1]

Ihre Zuordnung durch Abstraktion und Interpretation



- Bei der Interpretation ist der Sinn vorgegeben, z.B. in Form einer Formel. Für diese Formel ist nun eine Belegung (= Bedeutung) zu finden, welche die Formel in eine wahre Aussage verwandelt. In unserem Fall ist der Sinn in Form der Requirements vorgegeben. Dafür ist nun eine Implementierung zu bestimmen, welche die Requirements erfüllt.

- Bei der Abstraktion ist die Bedeutung vorgegeben, z.B. in Form einer Situation, einer Handlung oder eines Gegenstands. Dafür ist nun eine Formel (= Sinn) zu finden, welche die Absicht hinter dieser Situation etc. erklärt, die Formel damit in eine wahre Aussage verwandelt. In unserem Fall ist die Bedeutung in Form der Implementierung vorgegeben. Dafür sind nun die Requirements zu bestimmen, die durch die Implementierung erfüllt werden.

---

[1] Siehe auch: G. Frege: *Sinn und Bedeutung*. ‚Bedeutung' steht bei Frege für das durch ein Zeichen ‚Bezeichnete'. Weiter unten werden wir das ‚Bezeichnete' von der ‚Wirkung' bzw. der ‚Ausführung' des Zeichens unterscheiden. Das letztere ist das Ergebnis der Interpretation, die eigentliche Semantik des Zeichens. Das erstere unterstützt das Verständnis des Zeichens lediglich durch eine Assoziation zur realen Welt.

Interpretation

Freiheitsgrade

- „Die *Interpretation* eines abstrakten Axiomensystems besteht darin, den zunächst bedeutungslosen Zeichen bzw. Zeichenreihen des Systems *Bedeutungen* so zuzuordnen, dass sich die Axiome des Systems in *wahre Aussagen* über das Gebiet verwandeln, in dem das System interpretiert werden soll, und die Umformungsregeln des formalen Systems zu Regeln über die Gewinnung bzw. Umwandlung von Ausdrücken bzw. Aussagen über das betreffende Sachgebiet werden."“
  [G. Klaus, M. Buhr, PhilWB]

- Übertragung auf die Software-Architektur: Die Menge $[F]_I$ ordnet der intuitiven Beschreibung I nur die Turingmaschinen zu,
  - welche I implementieren, d.h.
  - auf welche die Beschreibung I zutrifft, d.h.
  - welche die Beschreibung I erfüllen, und damit I zu einer „wahren Aussage“ über die Implementierung machen.

- Damit muss die zuvor gemachte Bemerkung, dass „die zusätzlichen Merkmale einer Unterklasse nicht durch Transformation der Merkmale der Oberklasse entstehen, sondern *frei* gewählt sind“, relativiert werden:
  - Die Merkmale der Unterklasse – die ja eine Interpretation der Oberklasse darstellt – müssen die Oberklasse zu einer „wahren Aussage“ machen, indem sie die Merkmale der Oberklasse erfüllen.
  - Die Merkmale der Oberklasse sind sozusagen die „Requirements“ für die Unterklasse und ihre Merkmale.
  - Damit sind die Merkmale der Unterklasse nicht mehr völlig frei wählbar, sondern sind auf einen Interpretationsspielraum[2] beschränkt.
  - Beispielsweise können einer Unterklasse der Klasse „Tier“ keine „Räder“ als Merkmale zugeordnet werden.
  - Analog dazu ist auch die Interpretationsfreiheit während der Verfeinerung einer Software-Architektur eingeschränkt

[2] Auch bei der *Interpretation* eines Musikstücks kann man nicht willkürlich vom Notenbild abweichen, sondern muss sich darauf beschränken, die *Absicht* des Komponisten lediglich um Nuancen zu *verfeinern*.

Syntax und Semantik

- **Syntax:**
  „Zur *Syntax* gehören Formeln und Herleitungen; sie lassen sich beide durch endliche Zeichenreihen niederschreiben. Im engeren Sinn versteht man unter Syntax die Gesamtheit der Vorschriften, die den strukturellen Aufbau von Formeln bzw. Herleitungen festlegen." [A. Jung, IHB]

- **Semantik:**
  "Zur *Semantik* gehört, was hier [Belegung] genannt wurde, sowie der Übergang von einer Formel zu ihrer Interpretation bei einer bestimmten [Belegung]." [A. Jung, IHB]

- **Übertragung auf die Software-Architektur:**

  - Wir hatten festgestellt, dass die *Semantik* einer Formel durch Interpretation festgelegt wird. Beispielsweise wird die Signatur einer Methode durch die *Implementierung* der Methode interpretiert. Als *Syntax* der Methode verbleibt die *Signatur*, die in diesem Fall identisch ist mit der *Architektur*.

  - Wir haben somit herausgefunden, dass die *Implementierung* die *Semantik* eines Programms[1] darstellt, die *Architektur* aber die *Syntax*.

  - Damit kommen wir zurück zu unserer Feststellung, dass die Architektur eines Programms gleichzusetzen ist mit der <u>Syntax einer Aussage über die Funktion des Programms</u>, d.h. über seine Implementierung.

---

[1] Selbstverständlich wiederholt sich das Muster Syntax-und-Semantik auf jeder Abstraktionsstufe des Programms, so dass wir beispielsweise auch innerhalb einer Implementierung (hier: Semantik) selbst wieder eine Aufteilung in Syntax und Semantik vorfinden. Ebenso kann auch jede andere Verfeinerung innerhalb der Architektur als „Semantik“ einer höheren Abstraktionsstufe betrachtet werden.

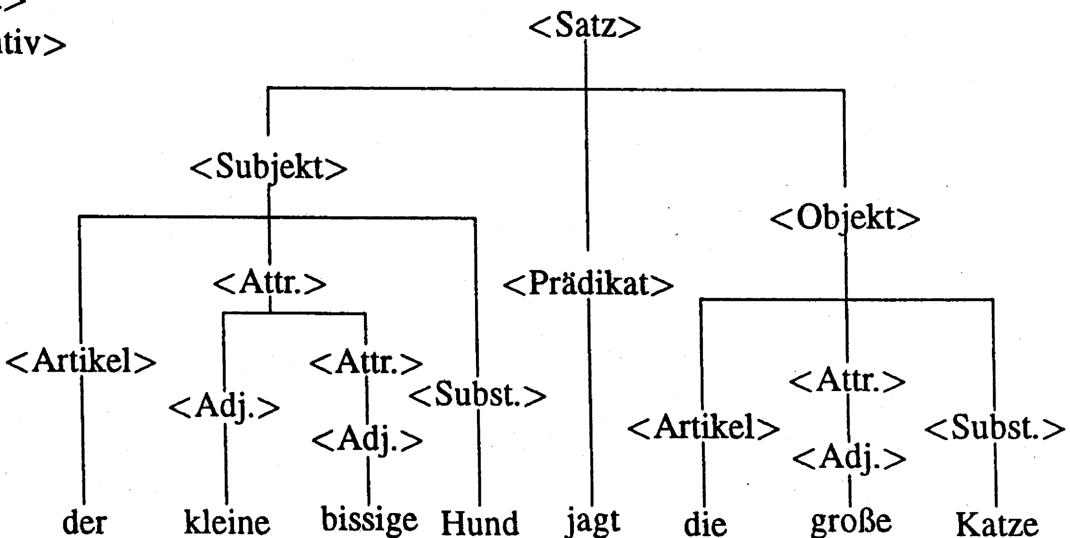
Syntax

Formale Definition

Grammatik

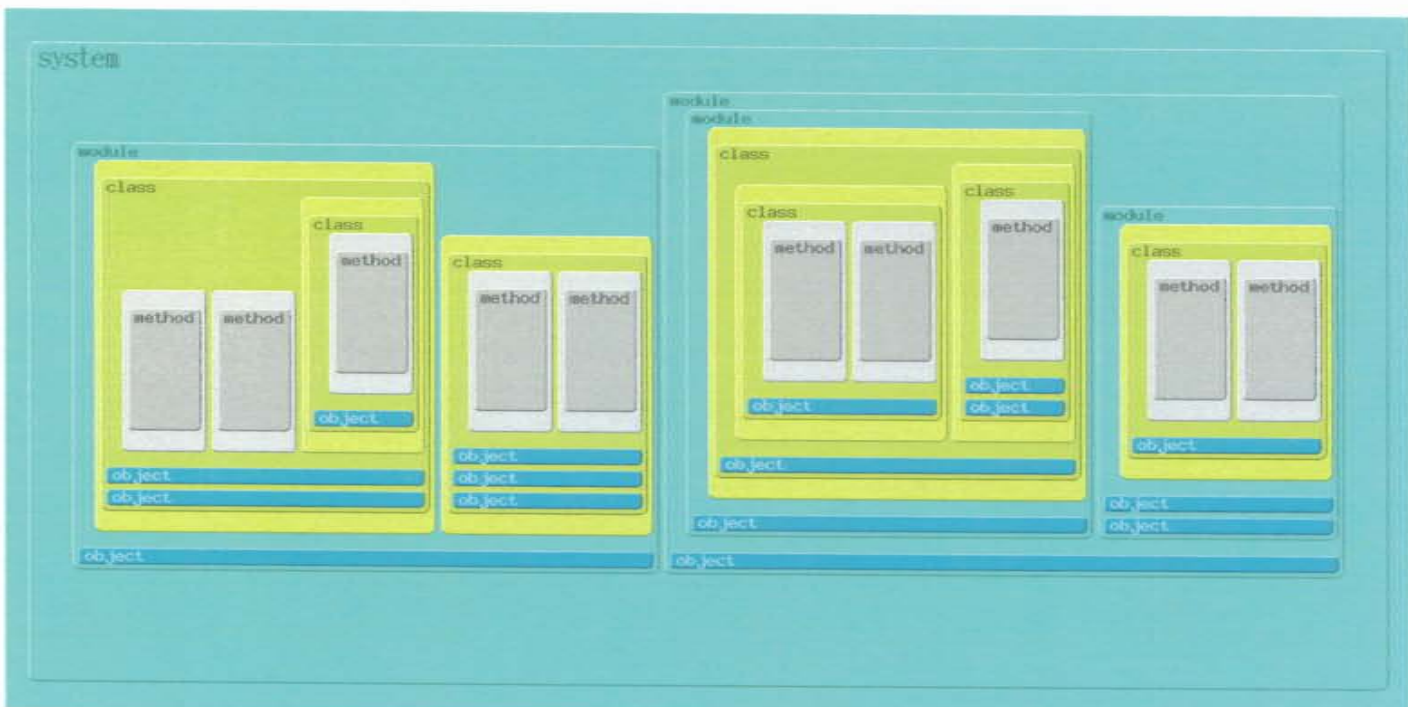
<Satz>	→	<Subjekt> <Prädikat> <Objekt>
<Subjekt>	→	<Artikel> <Attribut> <Substantiv>
<Artikel>	→	ε
<Artikel>	→	der
<Artikel>	→	die
<Artikel>	→	das
<Attribut>	→	ε
<Attribut>	→	<Adjektiv>
<Attribut>	→	<Adjektiv> <Attribut>
<Adjektiv>	→	kleine
<Adjektiv>	→	bissige
<Adjektiv>	→	große
<Substantiv>	→	Hund
<Substantiv>	→	Katze
<Prädikat>	→	jagt
<Objekt>	→	<Artikel> <Attribut> <Substantiv>

Syntaxbaum



Hierbei sind die sog. *Variablen*, die Platzhalter für syntaktische Einheiten sind, durch spitze Klammern kenntlich gemacht.

Architektur-Syntax



Architektur als „Syntax“

Merkmale

Am Beispiel der formalen Grammatik lassen sich zwei grundlegende Merkmale einer Syntax erkennen:

1. Es gibt darin unterschiedliche <Platzhalter>, und
2. es gibt unterschiedliche Ersetzungsregeln für jeden <Platzhalter>

Die Eigenschaften 1. und 2. bewirken, dass ein <Platzhalter> nicht an jeder beliebigen Position des Syntaxbaumes vorkommen kann. Hierdurch wird das syntaktische Element der *Position* eingeführt. Als Folge der Unterscheidung von Positionen ergibt sich die syntaktische Struktur.

zu 1.: Weiterhin können <Platzhalter>, die zum Zweck der Unterscheidung lediglich Identifier sein müssen, mit *Namen* belegt werden und somit beliebige *Gegenstände* der Realität modellieren.

zu 2.: Die kontextfreie Form der Ersetzungsregeln resultiert in einer Baumstruktur mit über- und untergeordneten Knoten und erlaubt es, diese Gegenstände *hierarchisch* zu verfeinern.

* * *

Bis hierhin haben wir die Mittel zur Verfügung gestellt, die *Struktur* einer Architektur zu beschreiben und zu verfeinern. Die Architektur hat jedoch die Aufgabe, *zwei* Verfeinerungen parallel zu beschreiben:

1. Die Verfeinerung der Struktur, und
2. die Verfeinerung der funktionalen Anforderungen.

Zu diesem Zweck werden wir zu *attributierten Grammatiken*[1] übergehen, die einem <Platzhalter> durch ein zusätzliches Attribut auch eine funktionale Anforderung zuordnen.

---

[1] Dieselbe Strategie wird im Compiler verfolgt, wo durch zusätzliche Attribute der Knoten des Syntaxbaumes die Codeerzeugung (ebenfalls eine spezielle Form der Semantik oder Implementierung) gesteuert wird.

Architektur als „Syntax“

Verfeinerung der Struktur und „Stellenwert“

Für die Verfeinerung der Struktur gilt das gleiche wie für die Verfeinerung der Anforderungen:

1. Die feinere Information kann die gröbere Information nicht ersetzen, sondern nur *ergänzen*.
2. Der Verfeinerungsschritt entsteht nicht durch Transformation, sondern durch *Aggregation*.
3. Um einen Satz zu verstehen, muss der *gesamte* Syntaxbaum rekonstruiert werden, bis zur Wurzel.

Beispiel:
In unserem Beispiel wird <Subjekt> auf die gleiche Weise verfeinert wie <Objekt>, nämlich als <Artikel><Attribut><Substantiv>, und doch ist es für das Verständnis des Satzes entscheidend zu wissen, dass ‚Hund’ nicht nur <Substantiv>, sondern auch <Subjekt> ist.

Dadurch erhält das Wort, unabhängig von seiner individuellen Bedeutung, einen syntaktischen „Stellenwert“. Ebenso erhalten Architekturelemente, durch ihre relative Position zueinander, zusätzlich zu ihrer individuellen Bedeutung einen „Stellenwert".

Beispiele:

- Baustein-Definitionen in den unteren Architekturschichten werden intuitiv als maschinen-nah eingestuft, Definitionen in den höheren Schichten dagegen als anwendungs-nah.
- Objekte im Parameter-Bereich einer Methode sind anders zu verstehen als solche im Attribut-Bereich eine Klasse oder an einer globalen Position.
- Ziffern einer Binärzahl repräsentieren einen Zahlenwert, der ausschließlich durch ihre Position bestimmt ist.

Architektur als „Syntax“

Vergleich mit natürlicher Sprache

Frage:
Warum sind im Syntaxbaum des obigen Beispielsatzes keine zusätzlichen Attribute vorgesehen, die etwas über den Sinn und Zweck der Terminalwörter aussagen. Mit anderen Worten, wo sind die „Requirements“ für die Implementierung des Satzes „Der kleine bissige Hund jagt die große Katze“?

Antwort:
Die Requirements verbergen sich im *Kontext* der Sprechhandlung. Nur in einem bestimmten Kontext erhält ein Satz einen „Sinn“. *Implizit* wird im Laufe der Strukturverfeinerung des Satzes auch dieser Kontext verfeinert, in dem dann jeweils die einzelnen Wörter einen weiter präzisierten Sinn erfüllen müssen. Die wohlbekannte alltägliche „Suche nach dem passenden Wort“ ist etwa durch diesen Kontext eingeschränkt. Ebenso verhält es sich mit der Suche nach der passenden Implementierung im „Kontext der Requirements“.

Wir müssen also in der Software-Architektur, so wie wir es von unserer Alltagssprache her gewohnt sind, unterscheiden zwischen den grammatikalisch korrekten „Sätzen“ und den „Sätzen“, die zusätzlich noch einen Sinn ergeben. Nur die letzteren repräsentieren eine „Architektur“.

Architektur als „zweckmäßige Funktionsverteilung“

Eine kontextfreie <Variable> repräsentiert einen bestimmten *Gegenstand* an einer bestimmten *Position*. Durch ein ihr angeheftetes, zusätzliches Attribut, den Kontext[1], gibt sie wieder, welche *Absicht* mit dem Gegenstand an dieser Position verbunden ist:

1. **<Gegenstand>**
   Die Variable in ihrer Eigenschaft als *Name* bezeichnet einen Gegenstand (z.B. <Modul>, <Objekt>, <Methode>), der durch Anwendung der Ersetzungsregeln weiter verfeinert werden kann (z.B. in <Klasse>, <Attribut>, <Parameter>).

2. **<Position>**
   Die Variable in ihrer Eigenschaft als *Identifier* (z.B. <X1>, <X2>, <X3>) schränkt die Menge der Ersetzungsregeln zu ihrer Verfeinerung ein. Damit können Variable (Gegenstände) nicht mehr an beliebige sondern nur noch an ausgewählte Positionen des Syntaxbaumes gelangen.

3. **<Verwendungszweck>**
   Die Variable in ihrer Eigenschaft als *Anforderung* erhält als drittes Attribut Design-Information, welche über den Sinn und Zweck des Gegenstandes an dieser Position Auskunft gibt.

---

[1] Der Ausdruck ‚Kontext' sowie alle folgenden Ausdrücke werden hier, wie schon gesagt, als gleichbedeutend betrachtet: ‚Absicht', ‚Verwendungszweck', ‚Anforderung', ‚Design-Information', ‚Sinn und Zweck' (weitere, siehe oben).

Architektur als „Syntax“

Architektur-Beschreibung

- Die (attributierte) kontextfreie Grammatik ist die wohl naheliegendste Art, eine Software-Architektur zu beschreiben. Sie stellt eine natürliche Fortsetzung der formalen, algorithmischen Beschreibung dar und kann sehr gut die in der Praxis vorherrschenden hierarchischen Strukturen wiedergeben.

- Nicht kontextfrei sind allerdings die Referenzen zwischen den Knoten einer hierarchischen Struktur, die gewöhnlich für die verschiedenen Arten der Kommunikation definiert werden müssen. Hierfür werden oft (nicht-visuelle) Namensreferenzen oder (visuelle) graphische Querverbindungen verwendet.

- Unterschied zu Bauwerksarchitektur und Maschinenbau: Durch die dort vorherrschenden homomorphen Beziehungen zwischen den funktionalen Abhängigkeiten und der physikalischen Struktur gibt es in der Regel keinen Bedarf für „Querverbindungen“. Man könnte hier von einem „Einklang“ von Funktion und Struktur sprechen. Bauelemente, die sich stützen, sind stets physikalisch benachbart, ebenso wie Zahnräder, die sich treiben. Für die Software-Architektur ist dieser Zusammenhang per se nicht gegeben und bislang noch unerforscht.

Architektur-Elemente

Informationsgehalt

Eine <Variable> im Architekturbaum repräsentiert ein bestimmtes *Architekturelement* an einer bestimmten *Position*. Das ihr angeheftete Attribut gibt Auskunft über ihren *Verwendungszweck*. Die nachgeordneten <Variablen> geben den inneren Aufbau dieses Architekturelements wieder. Damit verfügen die Knoten des Architekturbaums über ausreichend Information, um folgende Fragen zu jedem Architekturelement zu beantworten:

1. **<was>?**
   Um welches Architekturelement handelt es sich? Dies ist eine Frage nach dem *Typ*. Ist das Element einem *Operanden* zuzuordnen, ist dies der *<Klassen-Name>*. Ist das Element einer *Operation* zuzuordnen, so ist dies der <Eigen-Name> der *Methoden-Definition, des Methoden-Aufrufs, der Klassen-Definition, der Komponenten-Definition, der Modul-Definition* etc.

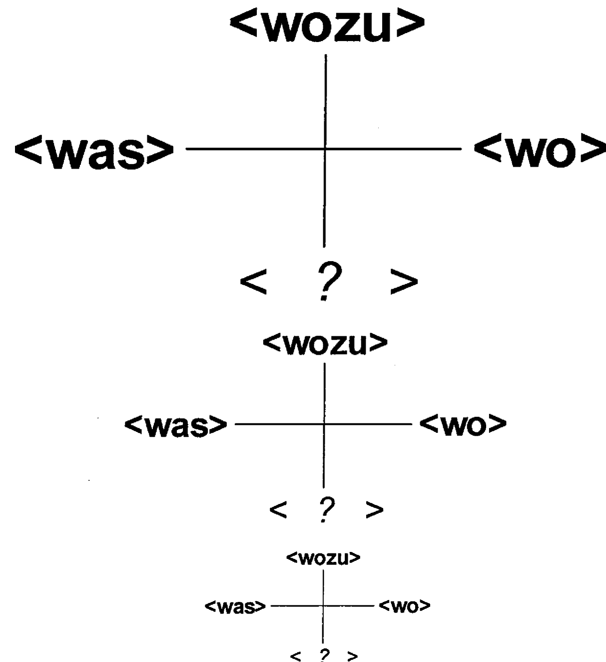
2. **<wo>?**
   Wo befindet sich das Architekturelement, relativ zu den anderen Elementen? Hierdurch werden *Benutzt-Beziehungen, Entity-Relations* und ähnliches ausgedrückt.

3. **<wozu>?**
   Welchem Zweck dient das Architekturelement an dieser Position? Ist das Element einem *Operanden* zuzuordnen, ist dies der *<Objekt-Name>*. Ist das Element einer *Operation* zuzuordnen, so ist dies der *<Kommentar> zum Methoden-Aufruf, zur Methoden-Definition, zur Klassen-Definition, zur Komponenten-Definition, zur Modul-Definition* etc.

4. **<wie>?**
   Diese Frage gilt der Implementierung: Wie funktioniert das Element, wie ist seine innere Struktur? Die Frage wird *rekursiv beantwortet* durch die Beantwortung der Fragen nach dem <was>, <wo> und <wozu> für die nachgeordneten Elemente.

Architektur-Elemente

<Wie> implementiert?



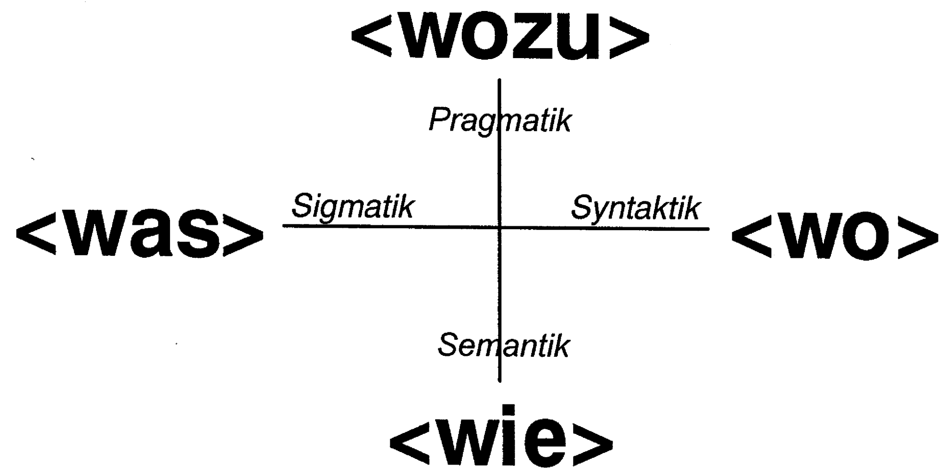
- Die Frage nach der Implementierung eines Architektur-Elements wird *rekursiv beantwortet*, indem die Fragen nach dem <was>, <wo> und <wozu> für die nachgeordneten Elemente beantwortet werden.

- Daraus kann man ersehen, dass es *keine eigentlich semantische Information* gibt, sondern nur abstrakte, wenn auch auf verschieden hohen Abstraktionsstufen. Man erkennt hier einen Vorrang der Abstraktion gegenüber der Interpretation.

- Dies ist umso erstaunlicher, als wir festgestellt haben, dass es keine Abstraktion ohne vorherige Interpretation gibt; dass sozusagen die Interpretation einen Vorrang gegenüber der Abstraktion hat, da die *Abstraktion keinen eigenen Informationsgehalt* hat.

Architektur-Element

Seine Zeichenaspekte



Semiotic Model of Computing