Kapitel 12
Software Wiederverwendung

Harald Gall
Software Engineering Group
www.ifi.unizh.ch/swe/

Universität Zürich
Institut für Informatik

Übersicht

- Ziele und Aspekte von Software Wiederverwendung
  - Begriffsklärung und Definition
- Software Component Libraries
  - Software Component Storage and Retrieval
- Reuse Economics
- Zusammenfassung

Ziele und Aspekte von Software Wiederverwendung

Universität Zürich
Institut für Informatik

Inhalt

- Ziele der Software Wiederverwendung
- Probleme des Software Reuse
- Geschichtliche Entwicklung
- Reuse Programme
- Moderne Reuse Ansätze
- Kategorisierung von Software Reuse
- Aspekte von Software Reuse
- Resümee

Einführung und Ziele

- Reuse - ein neues Konzept?
  - andere Ingenieurdisziplinen
  - Software - Hardware
- Software?
  - Konzept verspricht auch im Bereich der Software viele Vorteile
  - Bestehende Entwicklungen nutzen
  - Reuse ist mehr als Software Design
    - Komponenten + Design + Maßsystem + Prozesse
  - Industrielle Erfolgsberichte

Warum Software-Reuse ?

- Verkürzung der Entwicklungszeit
- Erhöhung der Produktivität
- Verbesserung der Vorhersagbarkeit (Kosten)
- Verbesserung der Qualität
- Management zunehmender Komplexität
- Vereinfachung der Wartung

Reuse-based Software Engineering

- Application system reuse
  - The whole of an application system may be reused either by incorporating it without change into other systems (COTS reuse) or by developing application families.
- Component reuse
  - Components of an application from sub-systems to single objects may be reused. Covered in Chapter 19.
- Object and function reuse
  - Software components that implement a single well-defined object or function may be reused.

Reuse benefits 1

Increased dependability	Reused software, that has been tried and tested in working systems, should be more dependable than new software. The initial use of the software reveals any design and implementation faults. These are then fixed, thus reducing the number of failures when the software is reused.
Reduced process risk	If software exists, there is less uncertainty in the costs of reusing that software than in the costs of development. This is an important factor for project management as it reduces the margin of error in project cost estimation. This is particularly true when relatively large software components such as sub-systems are reused.
Effective use of specialists	Instead of application specialists doing the same work on different projects, these specialists can develop reusable software that encapsulate their knowledge.

Reuse benefits 2

Standards compliance	Some standards, such as user interface standards, can be implemented as a set of standard reusable components. For example, if menus in a user interfaces are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability as users are less likely to make mistakes when presented with a familiar interface.
Accelerated development	Bringing a system to market as early as possible is o ften more important than overall development costs. Reusing software can speed up system production because both development and validation time should be reduced.

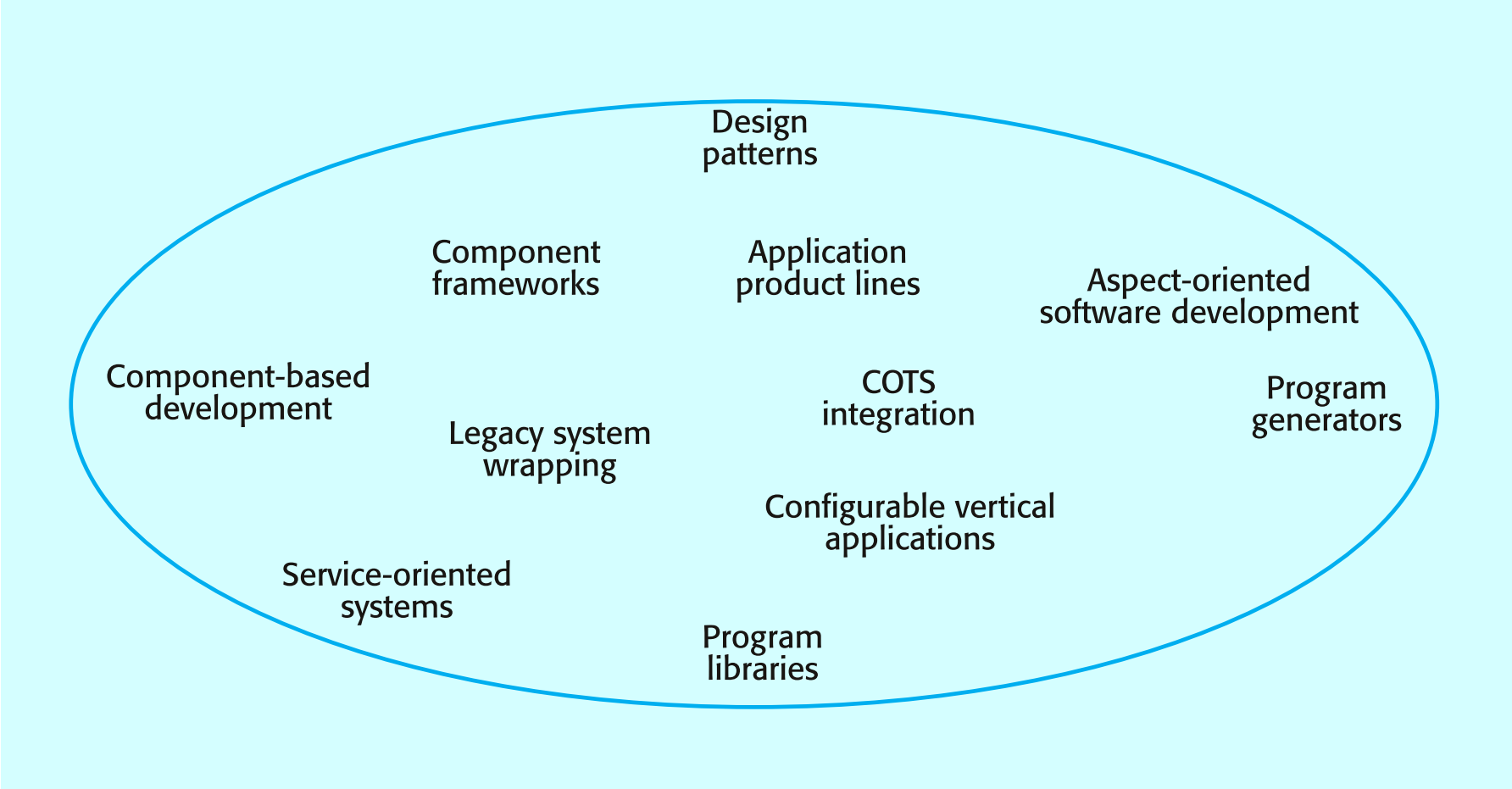
Reuse problems 1

Increased maintenance costs	If the source code of a reused software system or component is not available then maintenance costs may be increased as the reused elements of the system may become increasingly incompatible with system changes.
Lack of tool support	CASE toolsets may not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account.
Not-invented-here syndrome	Some software engineers sometimes prefer to re-write components as they believe that they can improve on the reusable component. This is partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other people's software.

Reuse problems 2

Creating and maintaining a component library	Populating a reusable component library and ensuring the software developers can use this library can be expensive. Our current techniques for classifying, cataloguing and retrieving software components are immature.
Finding, understanding and adapting reusable components	Software components have to be discovered in a library, understood and, sometimes, adapted to work in a n ew environment. Engineers must be reasonably confident of finding a component in the library before they will make routinely include a component search as part of their normal development process.

The reuse landscape



Reuse approaches 1

Design patterns	Generic abstractions that occur across applications are represented as design patterns that show abstract and concrete objects and interactions.
Component-based development	Systems are developed by integrating components (collections of objects) that conform to component-model standards. This is covered in Chapter 19.
Application frameworks	Collections of abstract and concrete classes that can be adapted and extended to create application systems.
Legacy system wrapping	Legacy systems (see Chapter 2) that can be 'wrapped' by defining a set of interfaces and providing access to these legacy systems through these interfaces.
Service-oriented systems	Systems are developed by linking shared services that may be externally provided.

Reuse approaches 2

Application product lines	An application type is generalised around a common architecture so that it can be adapted in different ways for different customers.
COTS integration	Systems are developed by integrating existing application systems.
Configurable vertical applications	A generic system is designed so that it can be configured to the needs of specific system customers.
Program libraries	Class and function libraries implementing commonly-used abstractions are available for reuse.
Program generators	A generator system embeds knowledge of a particular types of application and can generate systems or system fragments in that domain.
Aspect-oriented software development	Shared components are woven into an application at different places when the program is compiled.

Geschichtliche Entwicklung

- “Mass produced software components”
  - McIlroy, Bell Labs, 1968
- zuvor Programmsammlungen
  - International Mathematics Scientific Library (IMSL)
  - Software Package for the Social Sciences (SPSS)
- erste Software Factory, 1974
  - Systems Development Corp. (SDC)
- Standardisierung, Kompontenbibliothek
  - Raytheon Missile Division Project
- japanische Software Factories:
  - Hitachi, 1975
  - Toshiba, 1980 (Matsumoto)

Universitäre Forschungsprojekte

- P. Freeman, 1976 an der University of California, Irvine
  - DRACO von J. Neighbors, 1981
  - “A Software Classification Scheme” von R. Prieto-Diaz, 1985
  - “Domain Engineering for Software Reuse” von G. Arango, 1988
- V. Basili, 1987 an der University of Maryland
  - The TAME project
- Biggerstaff, Perlis
  - First Conference on Software Reuse, 1983; seitdem jährlich
- verschiedene internationale Konferenzen & Workshops
  - International Conference on Software Reuse (ICSR), IEEE
  - Symposium on Software Reusability (SSR), ACM

Software Reuse

- Reusability (IEEE Std. 610.12-1990)

  Reusability ist der Grad zu dem ein Software-Modul oder ein anderes Entwicklungsdokument in mehr als einem Computer Programm oder Software System verwendet werden kann.

Software Reuse /2

- Software Reuse (Prieto-Diaz, 1993)

  Unter Software Reuse versteht man die Verwendung existierender Software-Komponenten für die Erstellung neuer Software-Systeme.

Software Reuse /3

- Software Reuse [Mili et al., 2002]

  “Software reuse is the process whereby an organization defines a set of systematic operating procedures to specify, produce, classify, retrieve, and adapt software artifacts for the purpose of using them in its development activities.”

Software Reuse /4

- Reusable Software Engineering (Freeman, 1983)

  Reusable Software Engineering beschäftigt sich mit der Entwicklung von Software unter Wiederverwendung aller während der Software-Entwicklung generierten Informationen.

Arten von Software Reuse

Substanz	Bereich	Modus	Technik	Absicht	Produkt
Ideen, Konzepte	vertikal	geplant, systematisch	compositional	black-box, as-is	Source Code
Artefakten, Komponenten	horizontal	ad-hoc, opportunistisch	generativ	white-box, modifiziert	Design
Prozeduren, Skills					Spezifikation
					Objekte, Text, Architekturen

aus [Prieto-Diaz 93]

Reuse - Produkt

- Code
  - functional collections
  - macro libraries
  - reusable components
  - code fragments
- Design
  - design patterns
  - code templates
  - Ada generics
- Spezifikationen
  - Prototypen
  - application generators
  - transformational systems
- Architekturen
  - architectural patterns
  - architectural styles

Reuse - Technik

- compositional (building blocks)
  - code fragments (macros)
  - functional collections (mathem. Pakete)
  - generic code components (Ada packages, Smalltalk classes)
  - abstractions (programming plans)
  - design templates (logische Strukturen)
  - Modelle (generische Architekturen, Domain-Modelle)
- generativ
  - Generative Programming (e.g. GenVoca, Parser Generatoren, Code Generatoren in CASE-Tools)
  - Very high level languages (domain languages, dataflow languages)

Reuse - Bereich

- Vertikaler Reuse
  - innerhalb desselben Anwendungsbereichs
  - domänen-spezifische Komponenten
  - System-Familien

- Horizontaler Reuse
  - allgemeine, generische Komponenten
  - anwendbar über verschiedene Anwendungsbereiche hinweg

Reuse - Substanz

- Reuse von Ideen, Konzepten, Wissen
  - wie in anderen Ingenieurdisziplinen
  - teilweise informal

Generische Algorithmen

- Reuse von Artefakten, Komponenten
  - Hardware: ICs, Transistoren, etc.
  - Software: Prozeduren, Module, etc.

Komponenten-Kataloge
domänenspez. Kollektionen

- Reuse von Prozeduren

Prozesse &
Vorgangsweisen für
Reuse Programm

Reuse Kategorisierung

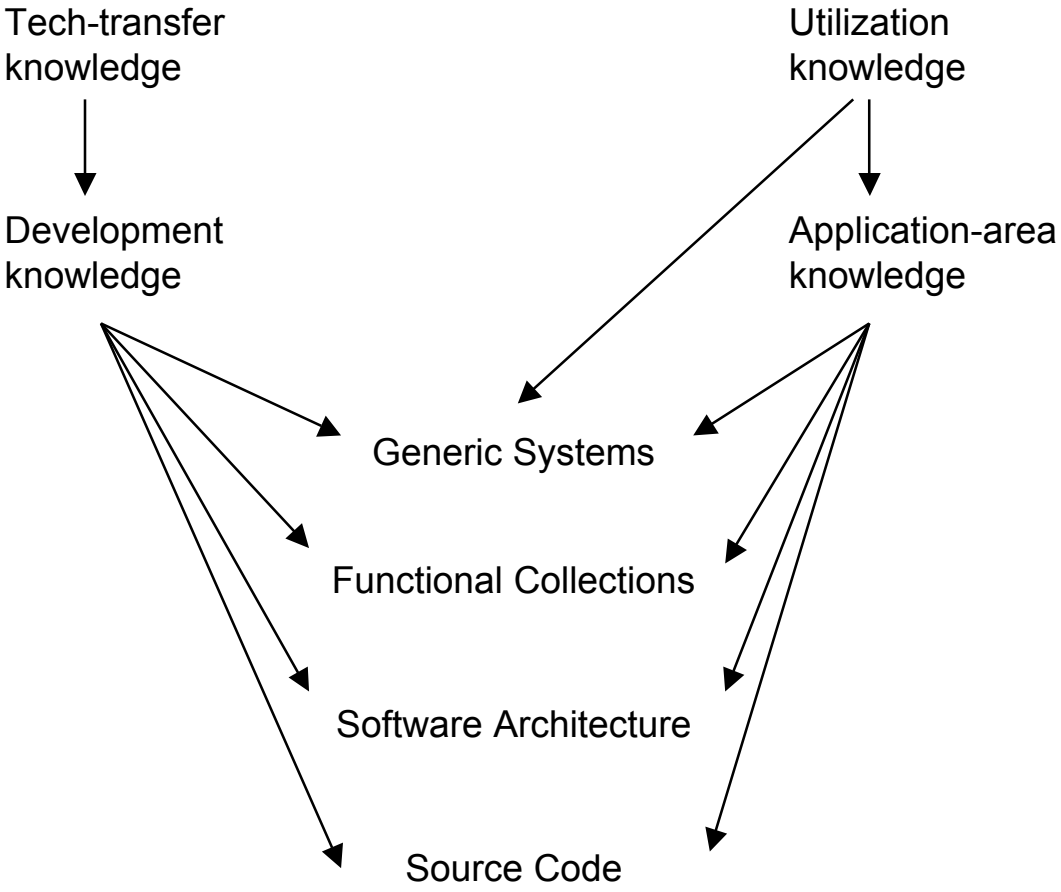
Features	Approaches to Reusability				
Component Reused	Building Blocks		Patterns		
Nature of Component	Atomic and Immutable Passive		Diffuse and Malleable Active		
Principle of Reuse	Composition		Generation		
Emphasis	Application Component Libraries	Organization & Composition Principles	Language Based Generators	Application Generators	Transformation Systems
Typical Systems	- Libraries of subroutines	- object-oriented - Pipe Archs	- VHLLs - POLs	- CRT FTRS - File Mgmt	- Language Transformers

Konzepte der Wiederverwendbarkeit

- Was soll wiederverwendet werden?
  - nicht nur source code
  - wiederverwendbare Information (Freeman, 1983)
  - alle Informationen, die in der Software Entwicklung anfallen
    - = Reusable Software Engineering
  - Hierarchie von Software Entwicklungsinformation

Wiederverwendbare Information

[Freeman83]



Environmental

External

Functional Architectures

Logical Structures

Code Fragments

→ B cannot be realized without A

Aspekte von Software Reuse

- Organisatorische Aspekte
  - Operative Infrastruktur
  - Technologische Infrastruktur
  - Reuse Einführung
- Technische Aspekte
  - Domain Engineering Aspects
  - Application Engineering Aspects
  - Component Engineering Aspects
- Wirtschaftliche Aspekte
  - Software Reuse Metrics
  - Software Reuse Cost Estimation
  - Software Reuse Return on Investment (ROI)
- Rechtliche Aspekte

Technische Aspekte

- Generierung von Komponenten (Design for Reuse, Reverse Engineering, Neu-Entwicklung)
- Methodologie (welcher Prozeß ist erforderlich?)
- Domain Analysis (Standards)
- Environment (Tools, Process Support, CASE)
- Library (Komponenten-Speicherung, Retrieval, etc.)
- Sprache (Abstraktion, Encapsulation, etc.)

Organisatorische Aspekte

- Ausbildung des Personals in Hinblick auf Reuse
- Einführung eines Reuse-Programms
- Motivation
- Installierung einer speziellen Reuse-Gruppe in der Organisation
- Reuse ist nicht gratis - Unterstützung durch das Management muß sichergestellt sein

Wirtschaftliche Aspekte

- Kosten/Nutzen Relation abschätzen (Nutzen erst später)
- Installierung einer Bewertungsmetrik um Pricing zu gewährleisten

Rechtliche Aspekte

- Urheberrechtliche Aspekte von wiederverwendeten Artefakten (Komponenten, Design, etc.)
- Gewährleistung für verteilte/übernommene Artefakte
- Probleme bei Reuse über Unternehmen hinweg
- z.B. Open Source Software
  - http://www.opensource.org/

Open Source Licenses

- The GNU General Public License (GPL)
- The GNU Library or `Lesser' Public License (LGPL)
- The Apache Software License
- The BSD license
- The Mozilla Public License (MPL)
- The IBM Public License
- The MITRE Collaborative Virtual Workspace License (CVW License)
- The Python license
- The Sun Internet Standards Source License (SISSL)
- The MIT license
- The Intel Open Source License
- ...

GNU General Public License (GPL)

- „GNU General Public License is intended to guarantee your freedom to share and change free software - to make sure the software is free for all its users.“
- protect rights to (1) copyright the software, and (2) give legal permission to copy, distribute and/or modify the software.
- „For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.“
- „Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.“

http://www.opensource.org/licenses/gpl-license.html

GNU Lesser General Public License LGPL

- „Applies to some specially designated software packages - typically libraries“
- „You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
  - a) The modified work must itself be a software library.
  - c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.“ [...]
- „A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.“

http://www.opensource.org/licenses/lgpl-license.html

Software Component Libraries

Software Component Storage and Retrieval

Universität Zürich
Institut für Informatik

Inhalt

- Einführung und Problemkreise
- Exemplarische Organisation einer SCL
- Der Begriff der “Software Component”
- Die Gewinnung von Software-Komponenten
- Der Reuse-Prozess
- Die Klassifikation von Komponenten
  - hierarchische/enumerative Klassifikation
  - flache/Facetten-Klassifikation
- Die Attributierung von Komponenten

Software Components Library (SCL)

- ...eine Menge von Komponenten (Software Assets), die von einer Organisation für Browsing und/oder Retrieval gepflegt werden.
  - Assets ... source code units, aber auch Spezifikationen, Designs, Dokumentation, Test Daten
  - Zweck = Reuse (vorwiegend)
  - Browsing ... Inspektion von assets ohne vordefinierte Query
  - Retrieval ... Identifikation und Extraktion von assets, die eine Suchbedingung erfüllen
- Fokus auf Retrieval → Repräsentation der Query und des Assets entscheidend!

Aspekte von Software Components Libraries

- Wer sind die Benutzer der SCL?
  - Mensch
  - Maschine
  - --> Schnittstelle
- Wie sehen Komponenten aus?
  - Sprache, Größe, etc.
  - --> Kategorisierung erforderlich
- Welche Anwendungsbereiche deckt die SCL ab?
  - einheitlicher vs. verschiedene Anwendungsbereiche (horizontal vs vertikal)

Aspekte von SCLs /2

- Wie fein ist die Granularität der abgespeicherten Komponenten (= Größe)?
- Wie werden Komponenten aus der SCL entnommen und für SW-Entwicklung verwendet?
- Wie kann man Komponenten in der SCL auffinden? (Navigation durch die SCL)
  - Benutzerdefinition
  - Schnittstellendefinition

Beispiele von Software Libraries

- ComponentSource
  - www.componentsource.com
  - 10.000 components (client-side), keyword search
  - platforms: COM, VCL (Borland Visual Component Library), .NET, Java Beans, EJB, CORBA
- Flashline
  - www.flashline.com
  - server-side components in Java
  - keyword and facet search
- SourceForge
  - sourceforge.net
  - keyword and facet search ('software/Group', 'people', 'Freshmeat')
- Public Ada Library (PAL)
  - http://www.iste.uni-stuttgart.de/ps/AdaBasis/pal_1195/ada/
  - different types of Ada components (source code, documentation, tools)
  - browsing, keyword search
- GoF design patterns
  - www.acm.org/sigada/wg/patterns/patterns/GOF_Toc.html
  - browsing
- Design Patterns for Avionics Control Systems (ACS)
  - g.oswego.edu/dl/acs/acs/acs.html
  - browsing

Representation of Queries

- natural language patterns and templates (*information retrieval methods*)
- lists of keywords (*descriptive retrieval methods*)
- sample input/output data (*operational semantics methods*)
- input/output signature or functional description (*denotational semantics methods*)
- designs or program patterns (*structural methods*)

Representation of Assets

- a wider range of representation; they include reference to actual asset
- is more abstract than asset itself
- 2 aspects:
  - navigation (brute-force or more elaborate depending on storage structure)
  - matching based on testing conditions:
    - relevance criterion (rc): asset is considered to be relevant
    - matching condition (mc): under which an asset is selected
    - mc weaker/stronger than rc
    - mc can be seen as implementation of rc

Asset Retrieval

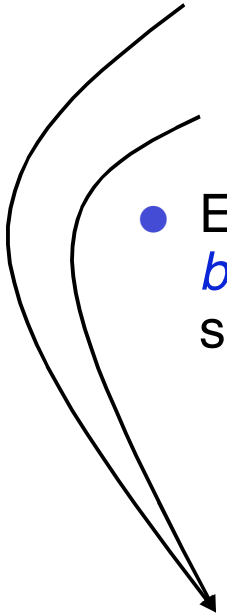
- exact retrieval
  - identify library assets that satisfy the query with respect to every aspect (functional correctness, architectural assumptions, interaction protocols, resource requirements, etc.)
  - fits black-box reuse lifecycle:
    - (1) exact retrieval; (2) assessment; (3) instantiation; (4) integration
- approximate retrieval
  - identify library assets that can be modified with minmal effort to satisfy a query
  - fits white-box reuse lifecycle:
    - (1) approximate retrieval; (2) assessment; (3) modification; (4) integration

Asset Retrieval /2

- 2 patterns of program modification
  - generative modification:
    - reusing the design of an asset while rewriting the lower levels of ist hierarchical structure
  - compositional modification:
    - using the building blocks of the retrieved asset and combining them in a new design structure to satisfy the query at hand

Example

- Retrieval goal, relevance criterion, matching condition:
- Example: method of software storage and *retrieval by signature matching*
  - 2 possible retrieval goals (G1, G2):
    - G1: retrieved components be correct with respect to the query or
    - G2: have the same signature as the query.
  - Example: „*component that computes the depth (height) of a binary tree*“
    signature: <binarytree, naturalnumbers>
    - Component A computes the depth of a binary tree
    - Component B computes the number of nodes of a binary tree
    - under G1: only A relevant; under G2: both are relevant



Example cont‘d

- Storage-retrieval (S-R) method that operates by *behavioral sampling*
  - relevance criterion = component be correct with respect to the reuser‘s intent
  - matching condition = (only) that the component behaves satisfactorily on the data sample provided by the reuser
  - 2 different conditions: an asset may well satisfy the latter and fail to satisfy the former

Technical evaluation of S-R methods

- Precision
  - ratio of relevant retrieved assets over the total number of retrieved assets; [0..1] or [very low, low, medium, high, very high]
    - no irrelevant assets should be returned
    - C … set of components in a library, R … set of relevant components,, c … components retrieved by query, r … set of retrieved relevant components
    - precision = r / c
- Recall
  - ratio of relevant retrieved assets over the total number of relevant assets in the library, no relevant assets should be forgotten
    - recall = r / R
- Coverage ratio
  - average number of assets that are visited over the total size of the library;
  - to preserve recall they must ensure that no excluded portion of the library contains any relevant assets
- Time complexity (constant, linear, polynomial, exponential, unbounded)
- Logical complexity per match (simple/compound boolean, simple/compound predicate, second-order predicate)

Characterizing a S-R Method

- Nature of asset: (source/executable) code, specs, etc.
- Scope of library:
  - single project, organization, larger scale
  - a reuser must share some common knowledge with the maintainer of the library and with other users: the more specialized the smaller the scope
- Query representation:
  - formal functional spec, signature spec, behavioral sample, natural-language query, set of keywords, etc.
- Asset representation:
  - dictates what form user queries take and determines how retrieval is performed
  - formal specs, signature specs, set of keywords, source text, executable code, requirements documentation
  - in a perfectly transparent library the representation is irrelevant to the user

Characterizing a S-R Method /2

- Storage structure
  - most common: no structure at all
  - difficult to define a general key that can be used to order software assets
  - some are based on formal specs and order assets by refinement ordering between their specs
  - AI-based libraries define semantic links between assets
- Navigation scheme
  - correlated to storage structure
  - flat structure: brute-force

Characterizing a S-R Method /3

- Retrieval goal
  - to find one or several programs that are correct with respect to a given query; depending on interest in generative or compositional modification the retrieval goal changes
    - generative: programs whose design can be adapted to solve the query
    - compositional: programs whose components can be combined to solve the query
- Relevance criterion
  - under what conditions a library asset is considered relevant with respect to the predefined retrieval goal
- Matching condition
  - is chosen to check between query and component whether the asset is relevant. Ideally, the matching condition should be equivalent to the relevance criterion, but...
  - if the surrogate (i.e. representation of asset) is too abstract, and/or the relevance criterion is too intractable, both may differ significantly.

Attributes of a Software Library

Attributes	Characterization
Nature of asset	Source code, executable code, requirements specification, design description, test data, documentation, proof
Scope of usage	Within a project, across a program, across a product line, across multiple product lines, worldwide
Query representation	Functional specification, signature specification, keyword list, design pattern, behavioral sample
Asset representation	Functional specification, signature specification, source code, executable code, requirements documentation, keywords
Storage structure	Flat structure, hypertext links, refinement ordering, odering by genericity
Navigation scheme	Exhaustive linear scan, navigation hypertext links, navigating refinement relations
Retrieval goal	Correctness, functional proximity, structural proximity
Relevance criterion	Correctness, signature matching, minimizing functional distance, minimizing structural distance
Matching criterion	Correctness formula, signature identity, signature refinement, equality and subsumption of keywords, natural-language analysis, pattern recognition

Classifying Software Libraries

- Information Retrieval
  - textual analysis of software assets; simply a specialized instance of information storage and retrieval (with all their shortcomings)
- Descriptive Methods
  - textual description of software assets; rely on an abstract representation (surrogate) of the asset; typically a set of keywords or a set of facet definitions
- Operational Semantics Methods
  - operational semantics of software assets; applied to executable code, proceed by matching candidate assets against a query on the basis of the candidates‘ behavior on sample inputs.

Classifying Software Libraries /2

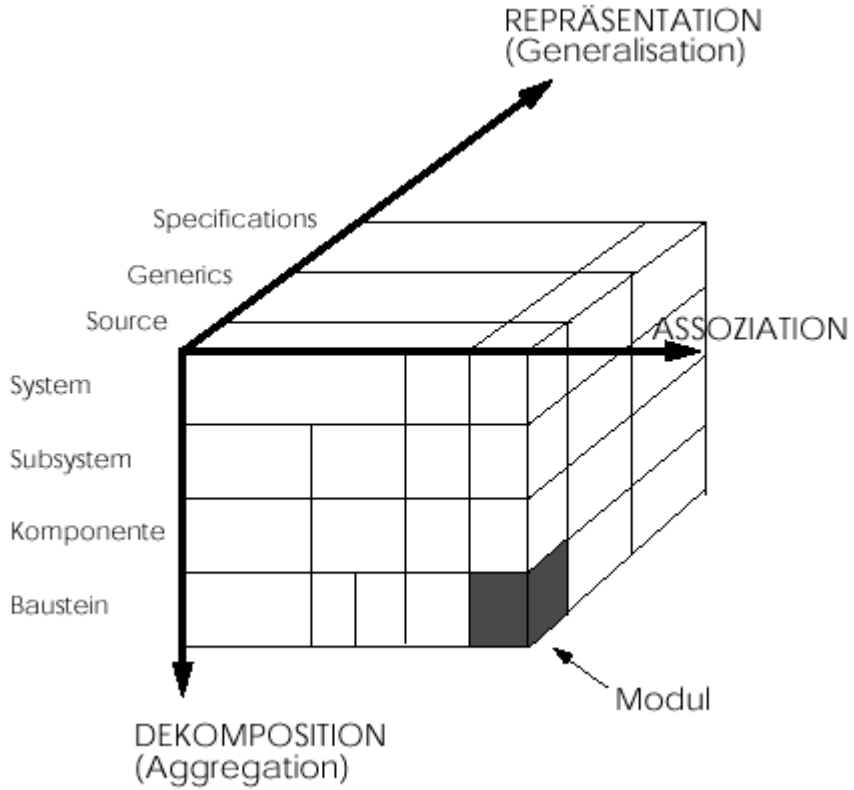
- Denotational Semantics Methods
  - denotational semantics definition of software assets; applied to non-executable assets (such as specifications); proceed by checking a semantic relation between the query and the asset surrogate (e.g. complete/partial functional description, signature)
- Topological Methods
  - goal to identify assets that minimize some measure of distance to the user query; has impact on relevance criterion and matching condition since outcome depends also on comparison with other assets
- Structural Methods
  - nature of software assets: program patterns which are subsequently instantiated to fit the user‘s need; relevance criterion deals with structure of asset rather than their function

Komponenten-Kategorisierung

- Berücksichtigung der Umgebung einer Komponente
  - Schnittstelle
  - notwendige Betriebsmittel
  - Stellung des Moduls zu anderen
- Argumente gegen eine “flache” Organisation
  - Systementwicklung kann nicht eingebracht werden
  - keine natürliche Beziehung zwischen den Modulen darstellbar
  - überspezifizierte Moduldefinitionen schwer lesbar
  - zu hoher Detaillierungsgrad erforderlich

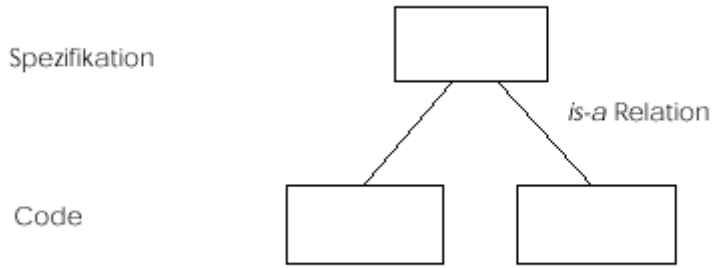
Klassifikationsstruktur

[Rossak, Mittermeir 88]



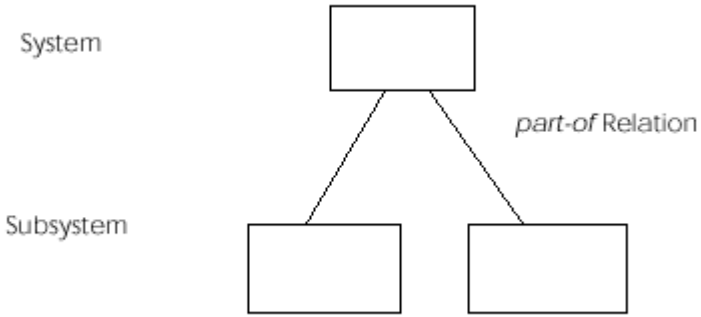
Generalisationshierarchie

- enthält Module unterschiedlichen Abstraktionsniveaus
  - Code → Spezifikation
- Struktur der Generalisationshierarchie <> Programm (Design)-Struktur
- Modellierung der Entwicklungsgeschichte einer Komponente
- Schichtenkonzept (organisatorisches Hilfsmittel)
- Repräsentation durch *is-a Relation*:
  - 1 Vorgänger - mehrere Nachfolger, ie 1 Spezifikation - viele Implementierungen

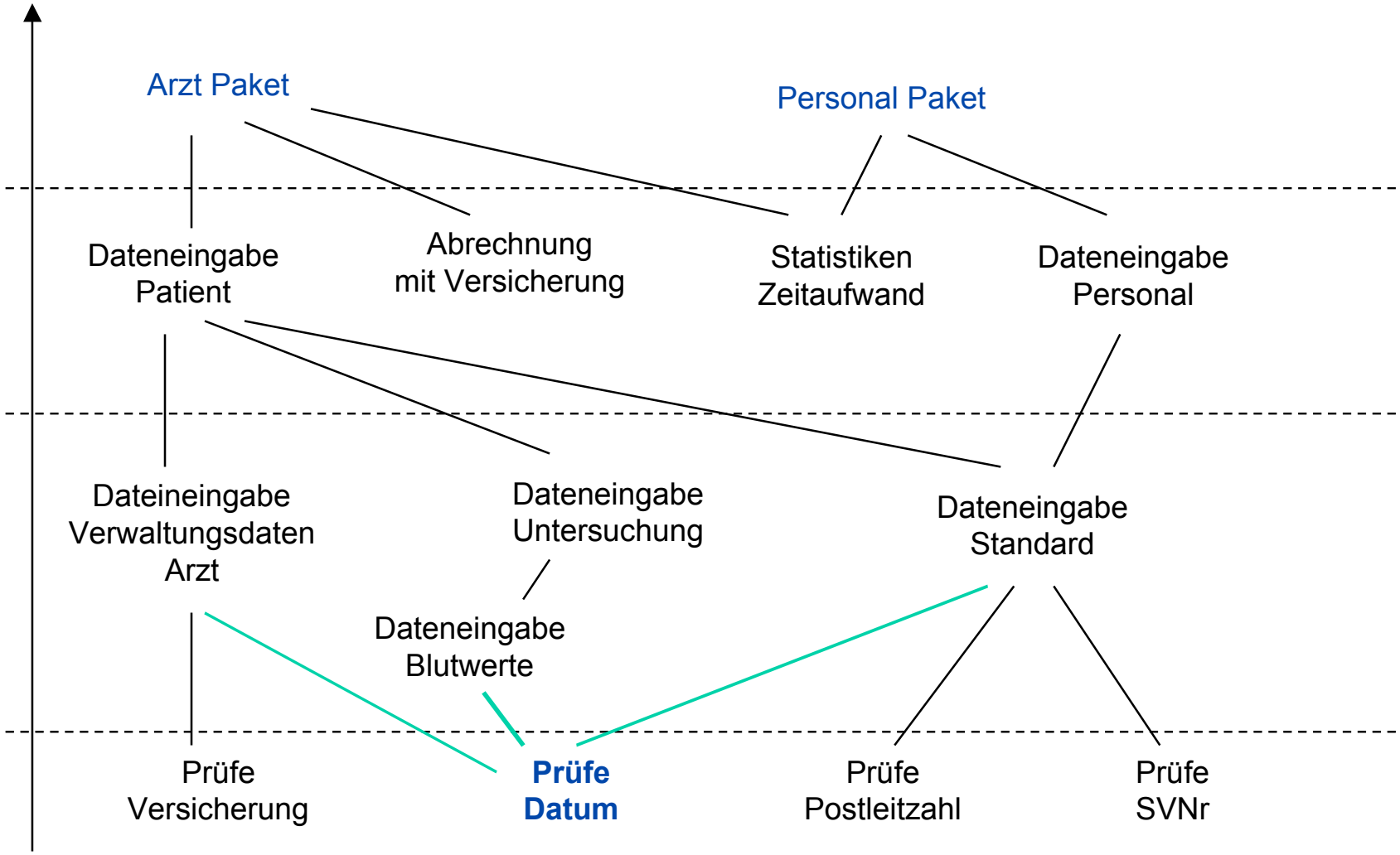


Aggregationshierarchie

- orthogonal zur Generalisationshierarchie
- Einstieg auf verschiedenen Ebenen der Dekomposition
- auch mehrfache Verwendung von Teilmodulen möglich
- Gliederung:
  - System, Subsystem, Komponente, funktionale Einheit
- Repräsentation durch *part-of* Relation:



Beispiel für eine Aggregationshierarchie



Modulbeschreibung

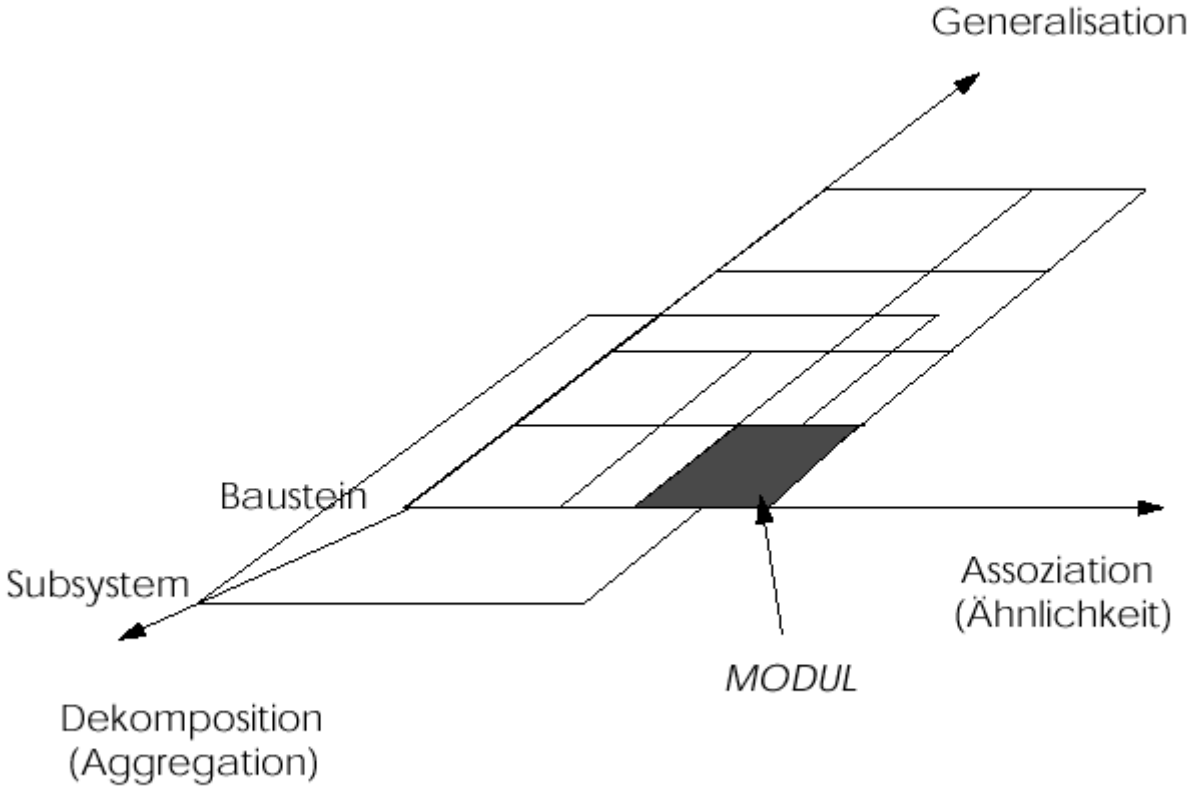
- Typ
  - Funktion, Sammlung von Funktionen, Datenstruktur, u.dgl.
- Schnittstelle
  - Abhängigkeiten von anderen Modulen
- Definition
  - interne Beschreibung, Eigenschaften, globale Einschränkungen, Kommentar
- Angaben zur Unterstützung der Suche
  - Einschränkung des Suchraumes

Suche im Modularchiv

- geg.: Modulbeschreibung (Zielvorgabe)
- Durchschreiten der Archivstruktur (vom Allgemeinen zum Detail)
  - interaktive Einbindung des Benutzers
- Auswahl einer Aggregationsebene
- Auswahl: Startpunkt in Generalisationshierarchie dieser Aggregationsebene
- Modultyp und globale Einschränkungen erlauben Abschätzung über Brauchbarkeit des Moduls

Suche im Modularchiv /2

[Rossak, Mittermeir 88]



Klassifikation von SW-Komponenten

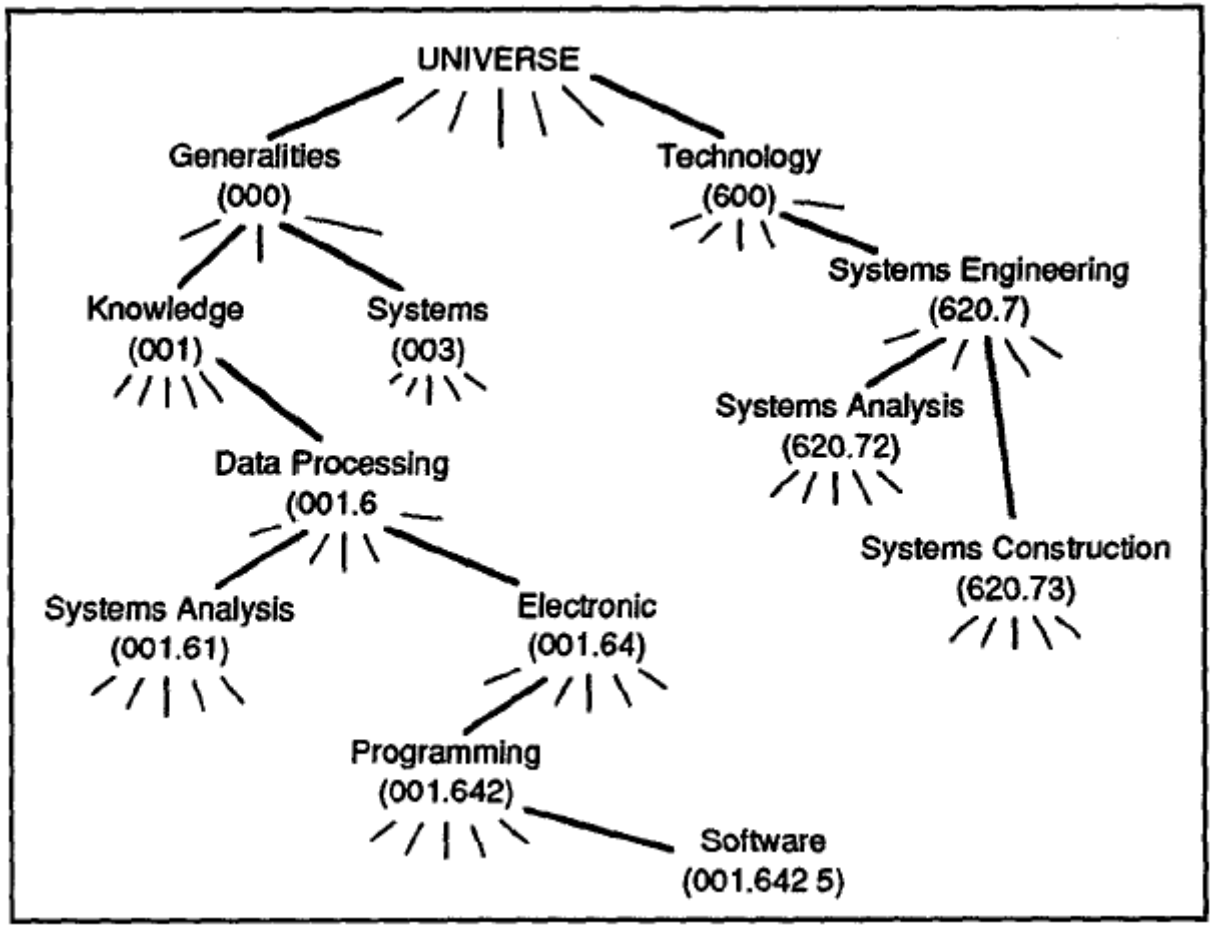
- Klassifikation = Gruppierung einer Menge von Elementen,
  - alle Elemente teilen zumindest eine Eigenschaft
- 2 wichtige Arten von Klassifikation:
  - hierarchische (enumerative) Klassifikation
  - flache (Facetten) Klassifikation

Hierarchische Klassifikation

- Universale Dezimal-Klassifikation
- teilt Gesamtheit der Information in immer feiner werdende Klassen ein
- typischer Suchbaum (Wurzel = Gesamtheit der Information)

7 Arts - includes physical planning and architecture as well as music, fine arts, drama etc.
- 71 Physical planning. Regional, town and country planning. Landscaping
- 72 Architecture
- 73 Plastic arts. Sculpture
- 74 Drawing. Design. Applied arts and crafts
- 75 Painting
- 76 Graphic arts. Graphics
- 77 Photography and similar processes
- 78 Music
- 79 Recreation. Entertainment. Games. Sport

Hierarchische Klassifikation /2



Hierarchische Klassifikation /3

- Vorteile (multidimensional)
  - Ausnützen der Verzweigungen als Relationen (Suchen nach diesen Relationen)
  - Einbau der Komponentenentwicklungsgeschichte in den Suchbaum
  - gute Handhabbarkeit der gegenseitigen Abhängigkeiten von Komponenten
  - Verfolgbarkeit der von der Spezifikation bis zur Implementierung
- Nachteile:
  - Kreuzverweise (Mehrfacheinträge)
  - Navigationswissen erforderlich
  - Hoher Wartungsaufwand und Fehleranfälligkeit bei Löschen/Einfügen (Kreuzverweise)

Facetten-Klassifikation

- Facettenklassifikation:
  - „Nichthierarchische Klassifikation, bei der es nur koordinierte, nicht subkoordinierte Merkmale gibt.
  - Die den Sachinhalt des Dokuments ausmachenden wesentlichen Begriffe werden durch Inhaltsanalyse festgestellt.
  - Jeder dieser Begriffe wird in den Klassifikationstafeln der Facetten aufgesucht.
  - Diese bestehen aus einer Vielzahl von sachlich definierten Gruppen, in denen zugleich die Notation jedes Einzelbegriffes festgelegt ist.
  - Diese Notationen werden in einer festgelegten Reihenfolge nebeneinander gesetzt.“

    aus Lexikon des Bibliothekswesens, H. Kunze (Hrsg.), 1975

Facetten-Klassifikation /2

- keine Teilung der Gesamtheit der Information in Klassen
- Bildung des gesuchten Elementes aus einzelnen Schlüsselwörtern (Synthetisierung)
- gesuchtes Element in elementare Komponentenklassen (= *facet*s) zerlegt
- Facetten werden in einem Schema repräsentiert
- gesuchtes Element wird durch Synthese der einzelnen Schlüsselwörter (= *term*s) definiert

Facetten	F_1	F_2	.....	F_n
	T_{11}	T_{21}	.....	T_{n1}
	T_{12}	T_{22}	.....	T_{n2}
	.....			
	T_{1n}	T_{2n}		T_{nj}

$$d_C = (T_{1A}, T_{2B}, ..., T_{NK})$$

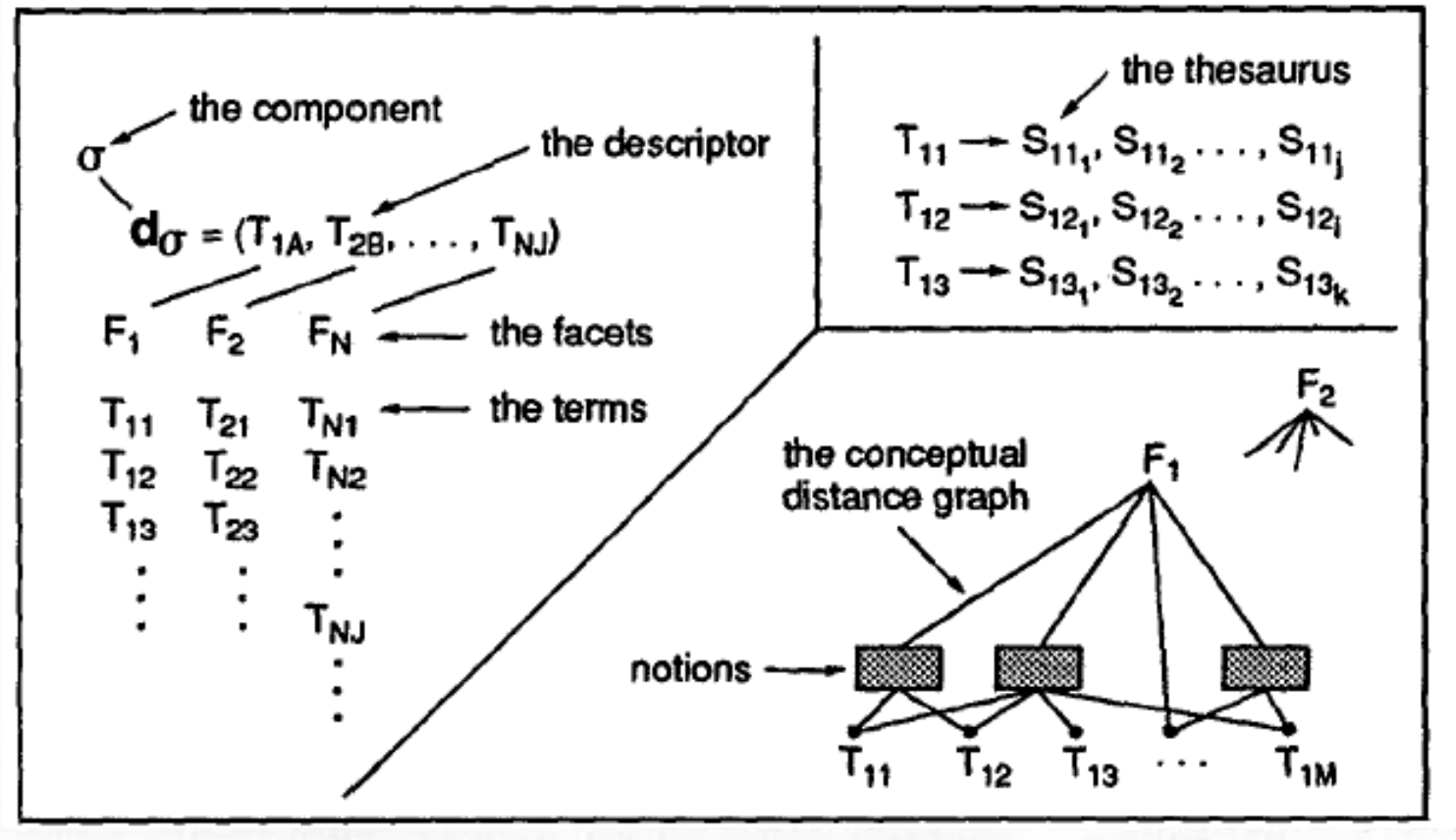
Beispiel

DOMAIN -> UNIX tools

{*by action*}	{*by object*}	{*by data structure*}	{*by system*}
get	file-names	buffer	line-editor
put	identifiers	tree	text-formater
update	line-number	table	
append	character	file	
check	number	archive	
detect	expression		
locate	entry		
search	declaration		
evaluate	line		
compare	pattern		
make			
build			
start			

Figure 1. A Simplified Faceted Scheme for Unix Components

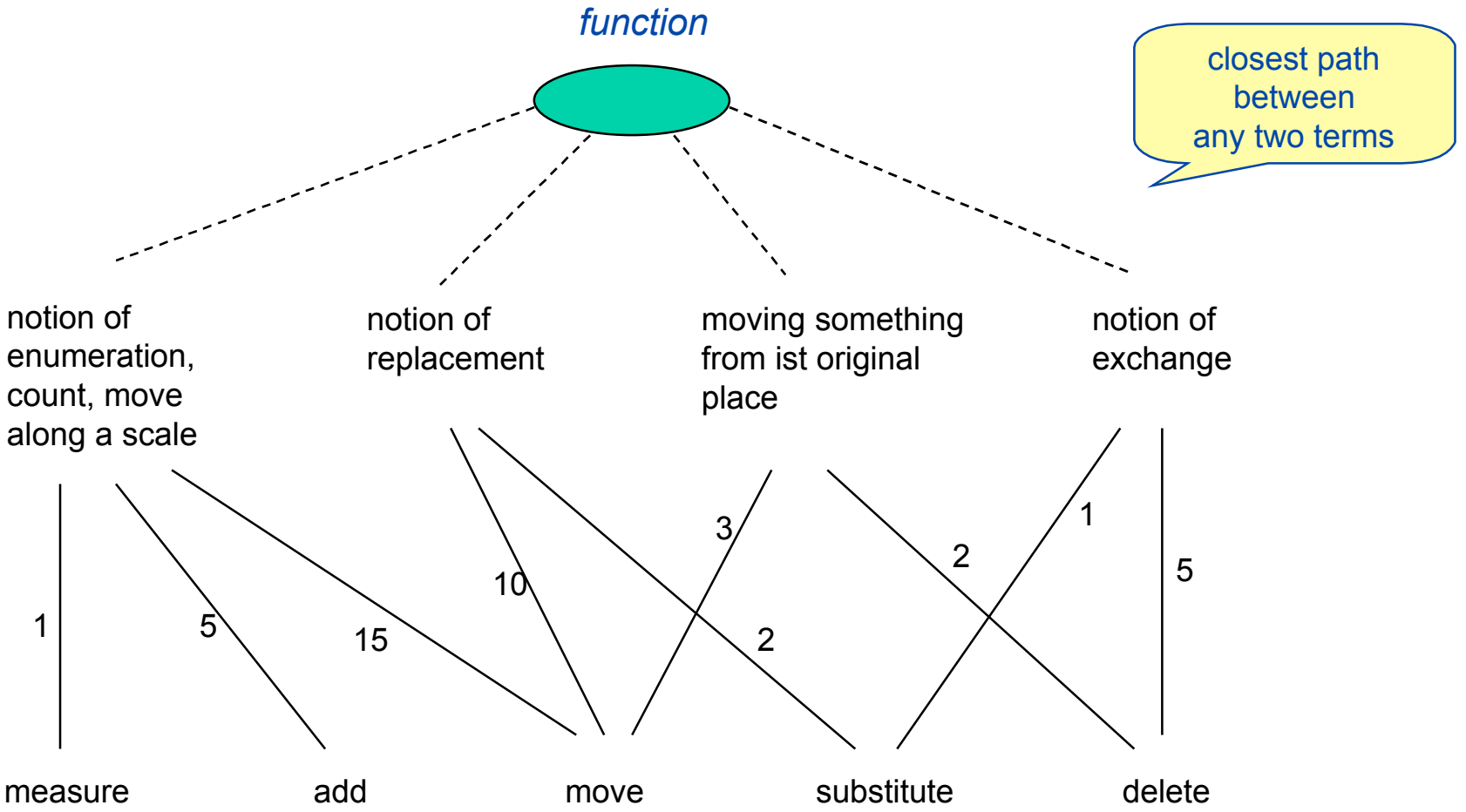
Elemente der FC



Facetten-Klassifikation /4

- Vorteile
  - hohe Flexibilität (Ordnung der *terms* in Facetten ist wählbar)
  - einfache Erweiterbarkeit (Einfügen, Löschen, ...)
  - einfache Adaptierung der Facetten bei geänderten Anforderungen
- Nachteile
  - Navigieren entlang von Relationen ist nicht möglich
  - Notwendigkeit genauer Suchangaben (exaktes Matching erforderlich)
  - Einführung von Relationen zwischen den *terms* kann diesen Nachteil beseitigen [Prieto-Diaz 85]
  - Relation = Maß für den Zusammenhang zwischen zwei *terms (conceptual closeness)*
  - *conceptual distance graph*

Conceptual Distance Graph



Obstacles to software assets classification

- Software assets are very information-rich
  - function, structure, input/output formats, interaction protocols, architectural assumptions, system/hardware/compiler requirements, operational characteristics, resource requirements, design information, and others ...
- Software assets can be arbitrarily similar
  - successive versions of the same product, platform-specific versions of some product, other product variations
  - representations must be sufficiently detailed to reflect differences and retrieval algorithms must be sufficiently precise to recognize these differences

Obstacles to software assets classification /2

- Lack of meaningful relations between assets
  - Equivalence relation (reflex, symm, trans)
    - catalog is structured by an equivalence class
    - e.g. department store catalog, Yellow Pages, hardware catalog
  - Ordering relation (reflex, antisymm, trans)
    - catalog is structured by an ordering relation
    - e.g. bookstore‘s catalog ordered by ISBN-No.
  - Some limited cases such as catalog of common datatypes (stacks, queues, heaps) allow equivalence classes; but assets range over a continuum of functionality (share common properties) thus no transitivity
  - Possible subsumption ordering relation (more general, more functional features, etc.) but this deals with only 1 attribute (functionality)
- As a result: no generally accepted solutions; many sophisticated ideas but few practical solutions. Low-tech solutions in practice

Attributierung von Komponenten

- Beschreibung Komponente durch ihr zugeordnete Attribute
- Arten von Informationen:
  - Information zur Suche *(location)* → Abbildung in Facetten
  - Information zum Verstehen *(understanding)* und zur Modifikation *(modification)*
- Extremstandpunkte bei der Attributierung
  - möglichst wenige Schlüsselwörter (Attribute)
    - rasche Suche möglich (wenige *terms*)
    - ungenaue Beschreibung .viele Komponenten

Attributierung /2

- möglichst viele Schlüsselwörter (Attribute)
  - genaue Spezifikation der Komponente
    - sehr wenige Komponenten als Ergebnis
  - durch eines Generalisationsmechanismus kann eine “Verallgemeinerung” (= Vergrößerung des Suchraumes) erfolgen
  - sehr genaue Spezifikation durch Attribute erforderlich
- Bestehende Ansätze:
  - [Burton et al. 87] verwenden sehr detaillierten Attributierungsansatz (14 Attribute)

Attributierung nach Burton

- Unitname –is the name of the procedure, package, or subroutine
- Category Code – is a predefined code that describes the functionality of the component
- Machine – signifies the computer on which the component was programmed
- Compiler – signifies the compiler used during development of the component
- Keywords – are programmer-defined words that describe the functionality of the component
- Author – is the person who wrote the component
- Date Created – is the date the component was completed
- Last Update – is the date the component was last updated Version – is the version number of the component
- Requirements – any special requirements of the component (e.g. other components that must be available)
- Overview – a brief textual description of the component
- Errors – any error handling or exceptions raised in the component
- Algorithm – algorithm used in the design of the component
- Documentation and Testing – a description of available documentation about the component and a description of test cases

Weitere Attributierungsansätze

- Prieto-Diaz verwendet ein 6-Tupel
  - function,
  - object,
  - medium,
  - type_of_system,
  - functional_area,
  - setting_of_application

- Rossak verwendet 4 Attribute
  - Typ des Moduls
  - Schnittstelle des Moduls
  - Definition des Moduls
  - Angaben zur Unterstützung der Suche

Reuse Process Model: Experience Factory

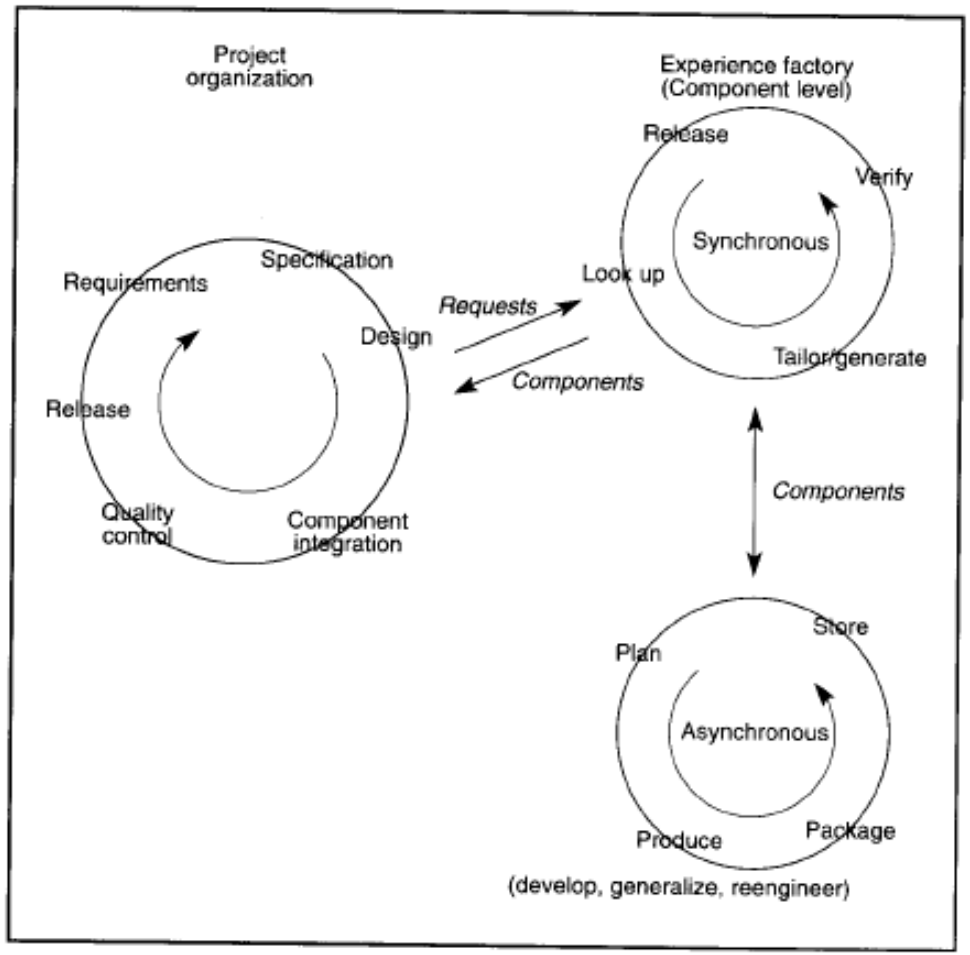


Figure 1. The reuse process model.

Component extraction

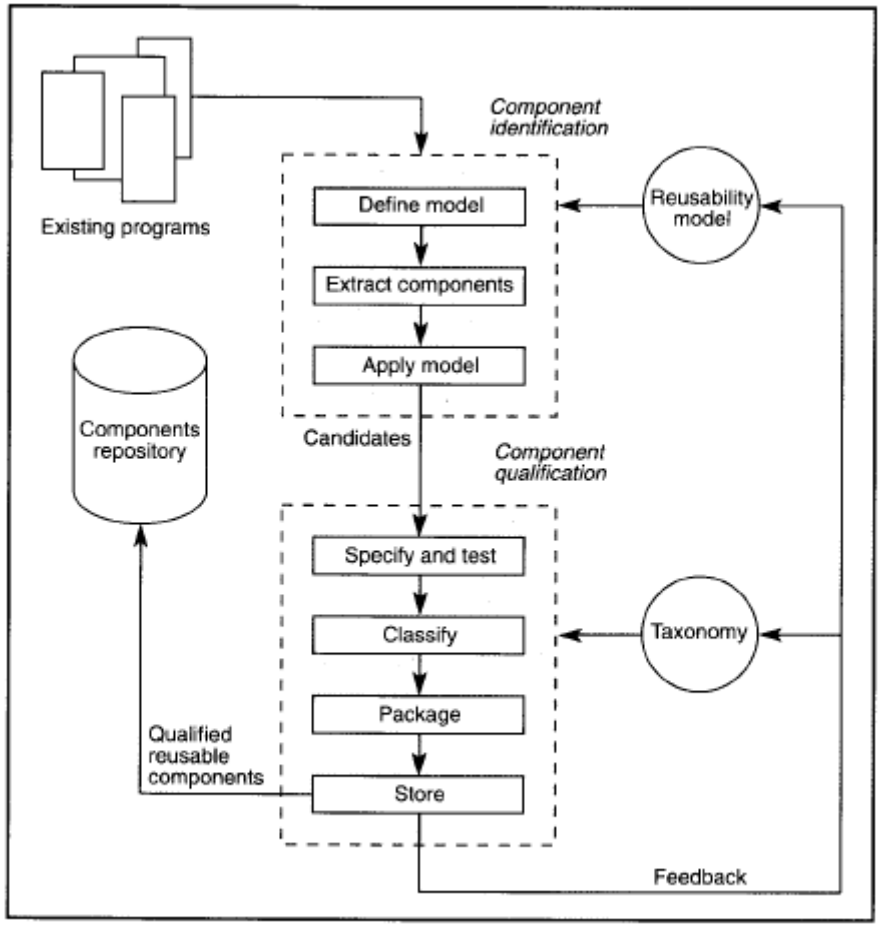
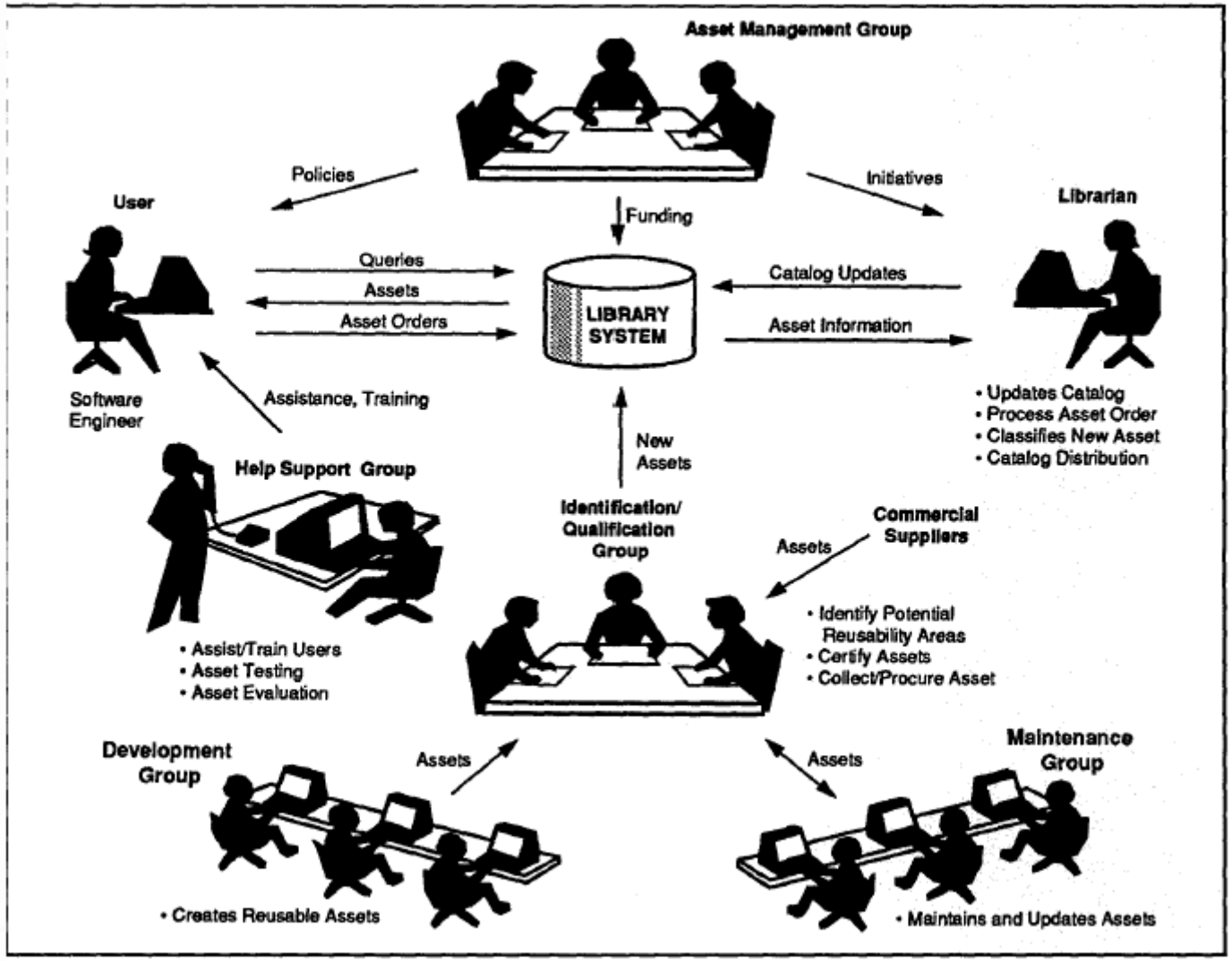


Figure 2. Component extraction.

SCL – Rollen & Artifakte



Resümee

- Allgemeine Struktur / Aufbau einer SCL
  - Librarian
  - User (User-Interface)
  - Query-System
  - Database
- Komponenten in einer SCL
  - Spezifikation-Design-Implementierung-Dokumentation
- Gewinnung von Komponenten
  - from scratch versus extraction (=reverse engineering)
- Die Klassifikation von Komponenten
  - hierarchisch (enumerativ) versus flach (Facetten)
- Attributierung von Komponenten

Reuse Economics

Universität Zürich
Institut für Informatik

Inhalt

- Kosten der Wiederverwendung
- Kostenschätzungen
- Technologien und Kosten
- Fallstudien
- Reuse Programme
- Organisatorische Aspekte

Reuse Kosten

- Reuse Investitionskosten
  - Summe der Kosten des Produzenten, um Teile für den Reuse bereitzustellen
- Komponenten Generizität
  - Anzahl der Variationen einer Komponente in Relation zur Reuse Technologie
- Kosten der Wiederverwendung
  - Kosten des Reusers für Finden, Anpassen, Integrieren und Testen einer wiederverwendbaren Komponente

Reuse Kostenschätzung

- $C_{no\text{-}reuse}$ = Entwicklungskosten ohne Reuse
- Reuse Level, R = $\frac{\text{total size of reused components}}{\text{size of application}}$
- F_{use} = Relative Kosten für den Reuse einer Komponente
  - typischerweise 0.1 bis 0.25 von Entwicklungskosten
- $C_{part\text{-}with\text{-}reuse} = C_{no\text{-}reuse} * (R * F_{use})$
- $C_{part\text{-}with\text{-}no\text{-}reuse} = C_{no\text{-}reuse} * (1 - R)$

- $C_{\mathbf{with\text{-}reuse}} = C_{part\text{-}with\text{-}reuse} + C_{part\text{-}with\text{-}no\text{-}reuse}$
- $C_{\mathbf{with\text{-}reuse}} = C_{no\text{-}reuse} * (R * F_{use} + (1 - R))$

Reuse Kostenschätzung /2

- Beispiel: R = 50%, F_{use} = 0.2
  - Kosten für die Entwicklung mit Reuse = 60% der Entwicklungskosten ohne Reuse

- $C_{saved} = C_{no\text{-}reuse} - C_{with\text{-}reuse}$
  $= C_{no\text{-}reuse} * (1 - (R * F_{use} + (1 - R)))$
  $= C_{no\text{-}reuse} * R * (1 - F_{use})$

- $ROI_{saved} = \frac{C_{saved}}{C_{no\text{-}reuse}}$
  $= R * (1 - F_{use})$

Reuse Kostenschätzung /3

- F_{create} = Relative Kosten zur Erzeugung und Verwaltung eines reusable component systems
- $C_{component\text{-}systems}$ = Kosten für die Entwicklung von genug component systems für R Prozent
- $F_{create} >> F_{use}$ $\quad 1 <= F_{create} <= 2.5$
- $C_{family\text{-}saved} = n * C_{saved} - C_{component\text{-}system}$
  $= C_{no\text{-}reuse} * (n * R * (1 - F_{use}) - R * F_{create})$

Reuse Kostenschätzung /4

$$ROI = \frac{C_{family\text{-}saved}}{C_{component\text{-}systems}} = \frac{n * R * (1 - F_{use}) - R * F_{create}}{R * F_{create}}$$

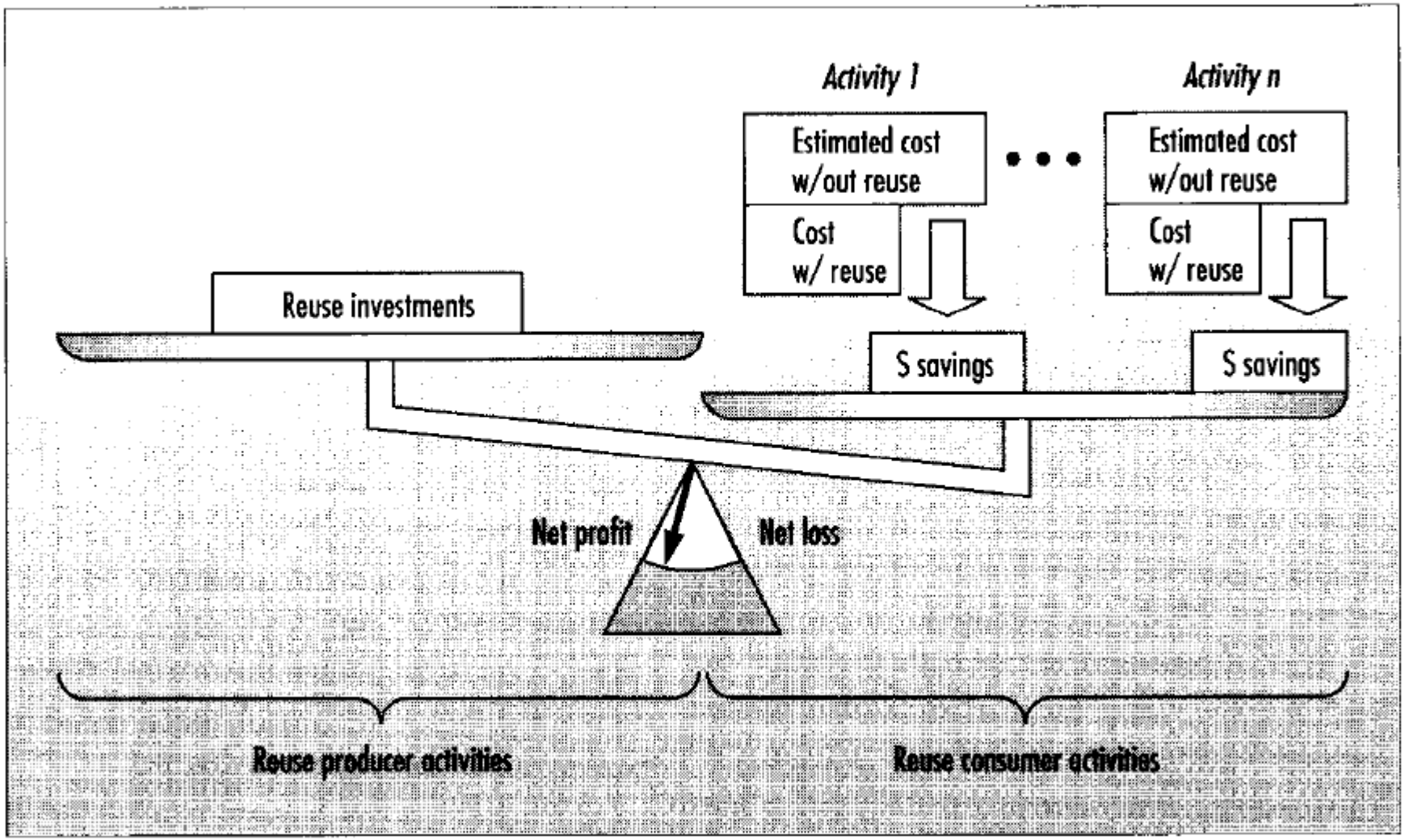
$$= \frac{n * (1 - F_{use}) - F_{create}}{F_{create}}$$

Beispiel: $F_{use} = 0.2$ und $F_{create} = 1.5$

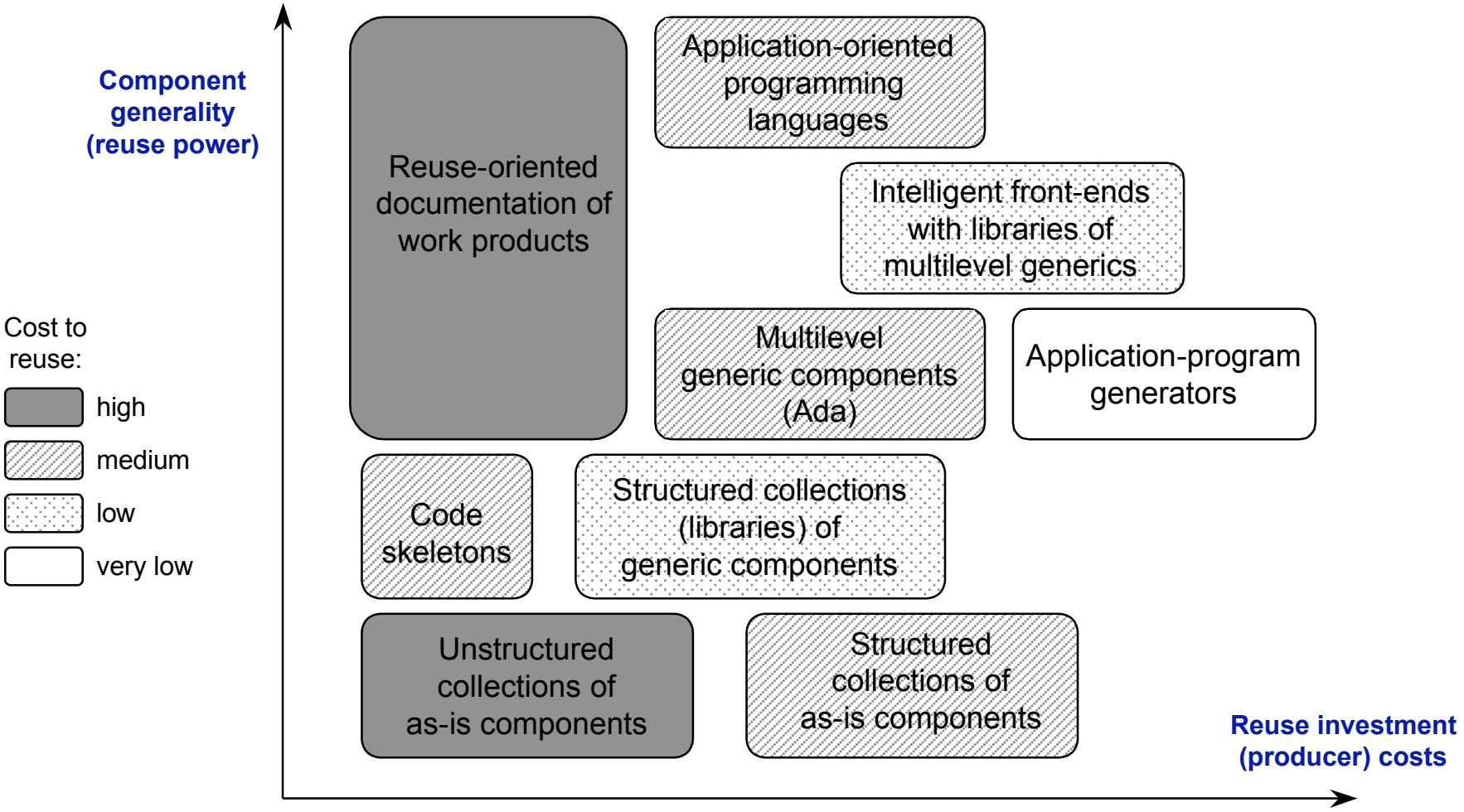
$$ROI = \frac{n * 0.8 - 1.5}{1.5}$$

Break-even mit $n > 2$

Reuse Investment Relation



Reuse Technologien & Kosten



Fallstudien

Fallstudie	Zeitraum	non-comment source statements	programming language	Entwicklungs OS	Ziel OS
HP's Manufacturing Productivity Section	1983-1994+	55 KNCSS (685 reusable workproducts)	Pascal, SPL	MPEXL für HP3000	MPEXL
HP's San Diego Graphics Division	1987 - 1994+	20 KNCSS	C	HPUX	PSOS

aus W.C. Lim, IEEE Software, Sept. 1994

Ökonomische Profile von Reuse Programmen

Organisation	Manufacturing	Technical Graphics
Zeithorizont	1983-1992 (10 Jahre)	1987-1994 (8 Jahre)
Erforderliche Start-up Ressourcen	26 Engineering Monate (für 6 Produkte) USD 0.3 Millionen	107 Engineering Monate (3 Engineers für 3 Jahre) USD 0.3 Millionen
Laufende Ressourcen	54 Engineering Monate (1 halber Engineer für 9 Jahre) USD 0.3 Millionen	99 Engineering Monate (1-3 Engineers für 5 Jahre) USD 0.7 Millionen
Gesamtkosten	80 Engineering Monate USD 1 Million	206 Engineering Monate USD 2.6 Millionen
Gesamtersparnis	328 Engineering Monate USD 4.1 Mio	446 Engineering Monate USD 5.6 Mio
Return on Investment (Ersparnis/Kosten)	410%	216%
Net present value	125 Engineering Monate USD 1.6 Mio	75 Engineering Monate USD 0.9 Mio
Break-even Jahr	2. Jahr	6. Jahr

aus W.C. Lim, IEEE Software, Sept. 1994

Qualität, Produktivität, Time-to-Market

Organisation	Manufacturing	Technical Graphics
Qualität	51% Fehlerreduktion	24% Fehlerreduktion
Produktivität	57% Erhöhung	40% Erhöhung
Time-to-Market	n.v.	42% Reduktion

aus W.C. Lim, IEEE Software, Sept. 1994

Kosten

Domain	Air-traffic-control System	Menu- und Forms-Mgmt System	Graphics Firmware
Relative Kosten zur Erzeugung von reusable Code	200 %	120 - 480%	111%
Relative Kosten der Wiederverwendung	10 - 20%	10 - 63%	19%

aus W.C. Lim, IEEE Software, Sept. 1994

Zusammenfassung

- Reuse
  - von ad-hoc auf systematisch
  - ist ein Level von “maturity” (cf. CMM)
  - ROI ist mittelfristig
  - braucht Investment: technologisch wie organisatorisch
  - benötigt Infrastruktur und Komponenten (!)
  - benötigt commitment (Management bis Developer)

Software Reuse

if you reuse code
you'll save a lot

if you reuse design
you'll save a legacy