Softwarekomponentensysteme

Teil 1

Softwarekomponentensysteme

Lehrveranstaltung:	Softwarekomponentensysteme
Studiengang:	BIF
Semester:	5
Studiensemester:	WS2021
Organisationsform:	DUA
Lehrbeauftragte(r):	Dipl.-Ing. Michael Vodep
Sprache:	Deutsch
ECTS:	4.50
Incomingplätze:	0
Organisationseinheit:	Kompetenzfeld Software Engineering & DevOps (*Leitung*: DI Walter Wölfel)

Warum hauen wir nicht gleich in die Tasten?
Ich will programmieren …

Live aus der Cafe-Küche

FH University of Applied Sciences TECHNIKUM WIEN

Module

Software-Architektur

NoSQL
Geh bitte schleich die mit deiner RDBMS – RDBMS ist tot! Facebook und Amazon haben auch NoSQL ...

Microservices

Domain-Driven-Design
Damit kann nix mehr schief gehen! Hat Kollege bei einer Bank auch angewandt

Komponenten
Redest du von Modulen?!

Security
Ich hab TLS aufgedreht – wir sind sicher!

DevOps
Hast du des scho installiert?

Solution-Architektur

Docker

Kafka
LinkedIn nutzt das – wir brauchen das!

REST
Löst alle unsere Probleme ....

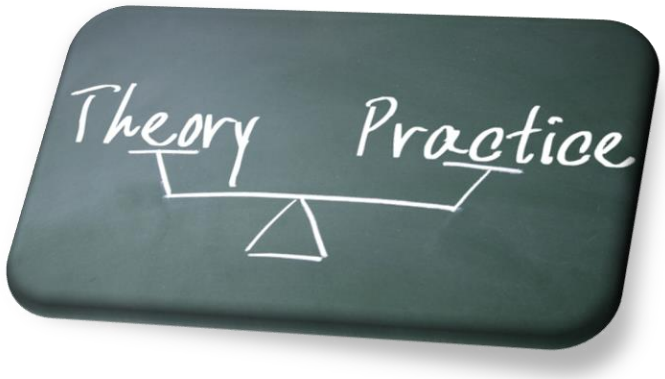
Kubernetes
Das ist doch das mit dem DevOps oder?

Interface

Fault Tolerance

Die Sache mit der Theorie …

- In theory there is no difference between theory and practice.
- Theory without practice is empty; practice without theory is blind
- Trotzdem brauchen wir ein wenig Theorie, um **vom Gleichen zu reden** …
- Und um zu wissen, **wann wir welches** Werkzeug verwenden sollen



Aufbau der Vorlesung

- 5 Minuten Wiederholung am Anfang der Vorlesung
  - Möglichkeit Fragen zu stellen
- 10 Minuten Präsentation der letzten Lösung der Übung in der nächsten Übungseinheit

Aufbau der Übung

- C# / .NET
- https://azureforeducation.microsoft.com/devtools
- Benötigt wird Visual Studio
- Übungen sind kurz und unabhängig
  - Kein Stress, wenn eine Übung mal nicht so klappt wie gewollt
  - Ziel: Konzepte verstehen
- Abgabe: bis Mitternacht des Übungstages in Moodle
- Abgabeformat
  - ZIP
  - Binaries entfernen (Debug / Release Ordner)
  - Zielplattform Windows (Mac OS X User: Virtuelle Maschine)

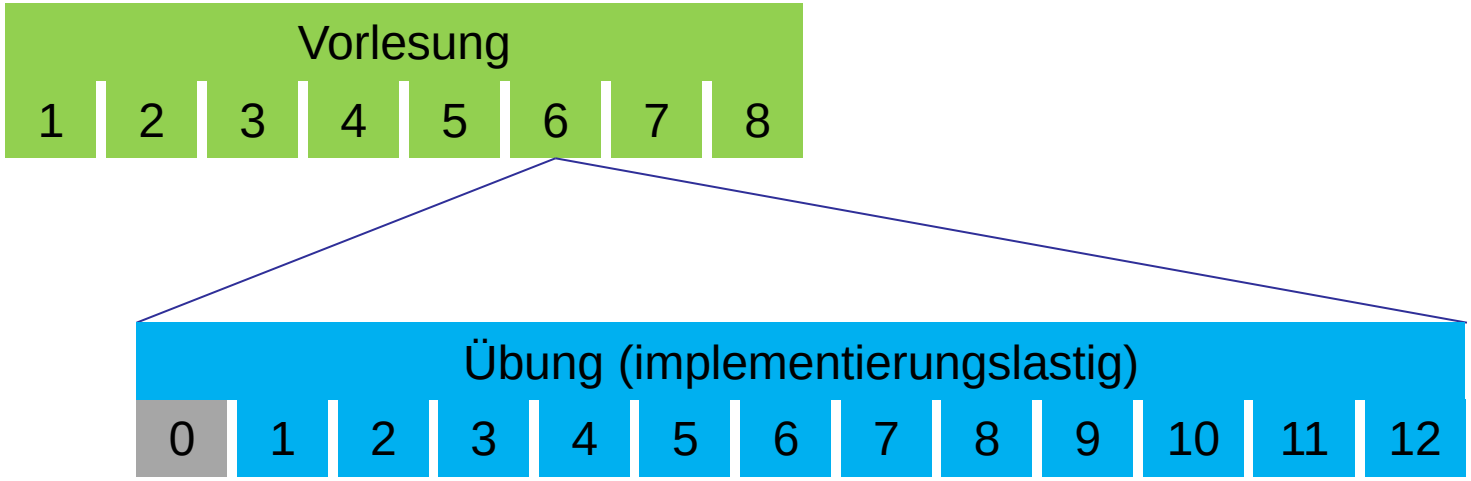
Inhalt

- Sinne schärfen – wann und warum sollte ich planen bevor ich in die Tasten greife?
- Wie teile ich meine Komponenten ein? Wie lade ich Komponenten dynamisch in .NET? Wie kommunizieren Komponenten untereinander? Wie überwache ich Komponenten? Wie deploye ich Komponenten? Wie teste ich Komponenten? …
- Und natürlich: Warum gibt es dieses „Komponenten-Zeug“ überhaupt?
- Out-Of-Scope:
  - Tiefgang in Kubernetes & Co → DevOps und Cloud Computing (CLCOM)
  - Web-Applikationen (UI, …) oder komplette Backends entwickeln → diverse andere Vorlesungen / Labore / Berufserfahrung

Wie komme ich zu meiner Note?

- 60% Übungen – Beurteilung durch
  - Code-Review
- 40% Abschlussprüfung
- Beide Teile müssen positiv absolviert werden

Vorlesung, Übungen …



- Aber keine Angst: Vorlesung untermauert nur Theorien zur Übung – zeitlicher Versatz also kein Problem …
- 12 Übungen
- Übung 0: Docker Desktop / Docker Setup
  - Voraussetzung: Windows 10 / Mac OS X mit Virtualisierung + WSL2

FH University of Applied Sciences TECHNIKUM WIEN

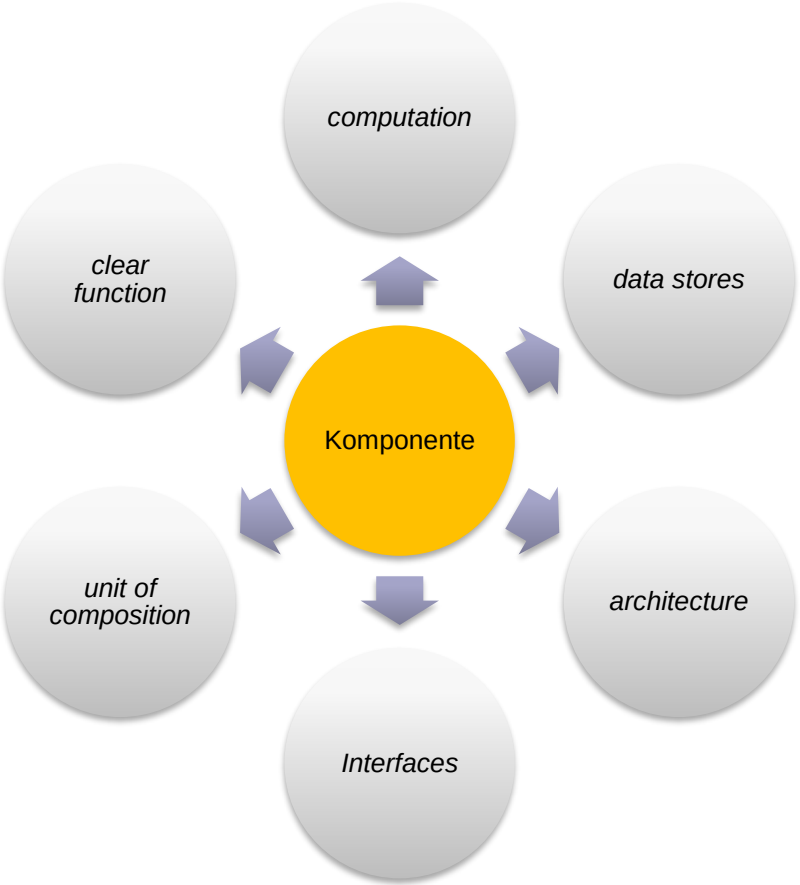
Los geht's …

Was ist eine Komponente?

- "*the principal computation elements and data stores that execute in a system*" (Clements et al.)
- *"A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces."* (PhilippeKrutchen, RationalSoftware)
- *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition."* (Clemens Szyperski, ComponentSoftware)
- …

Viele Sichtweisen …

Wir konzentrieren uns auf folgende Eigenschaften:



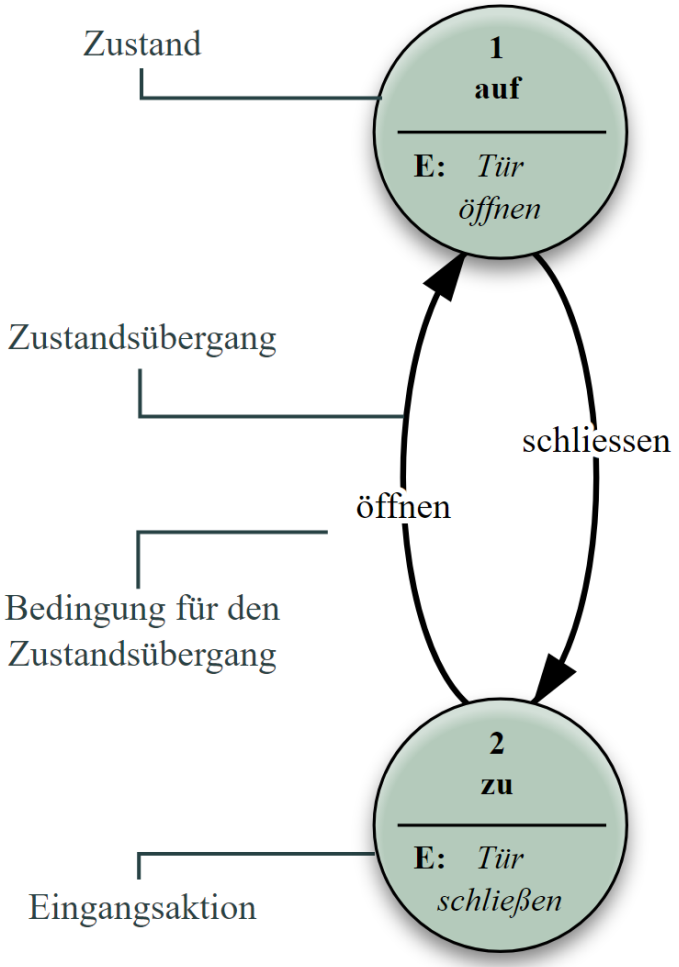
Clear Function

Wichtige Bestandteile einer Komponente

- Bestandteile
  - Port
    - Required
    - Provided
  - Connector
- Konzepte auch in UML (mehr oder weniger)

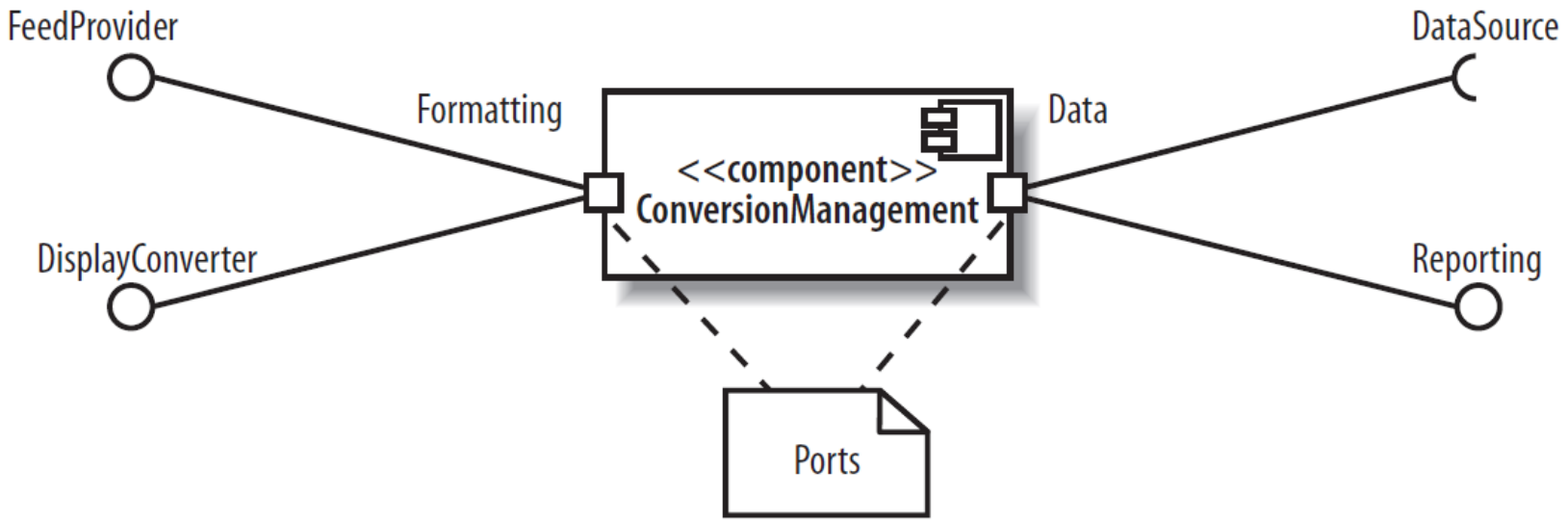
Port

- Alle **öffentliche Methoden** (Public Methods) und **Events** einer Komponente, sind über Ports verfügbar („Interface“)
- Ports können Stateful sein
  - Aufruf der Methoden nur in bestimmter Reihenfolge erlaubt (e.g. durch Finite-State-Machine (FSM))
- Oft modelliert man required (r) und provided (p) Ports



UML 2.0 Ports

Kreis = provided und Halbkreis = required



In UML 2.0 werden Interfaces durch Ports „gruppiert“

Miles, R., & Hamilton, K. (2006). Learning UML 2.0. " O'Reilly Media, Inc.".

Connector

- Komponenten kommunizieren über deren Ports
- Der Port der `Komponente A` ist mit dem Port der `Komponenten B` per Connector verbunden
- Connectors sind der primäre Treiber des Architektur-Styles (Local-Method-Call, RPC, SQL, SOAP over HTTP, …) – mehr dazu im Kapitel „Design“
- Können auch weitere Aufgaben übernehmen: Verschlüsselung, Filtern, Transformieren, Mediate („Vermitteln“) zwischen mehreren Sub-Components …

Exkurs: Connector Spezifikation Beispiel

Connector name	LibrarySystem-PeopleDB Connector
Roles	rPeople, compatible with rPeople port pPeople, compatible with pPeople port
Topology	Binary
Other properties	Protocol: SQL Transport: TCP/IP Throughput: 10,000 person records/sec Synchronous
Functionality	TBD
Type model	A row from the PERSON table in the pPeople role contains ... A Person class in the rPeople role contains ...
Behavior model	The connector starts in the CLOSED state, transitions to the OPEN state after a call to open(), then to the CLOSED state after a call to close().

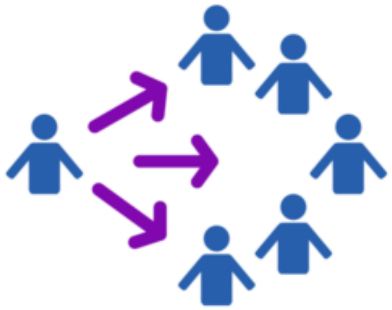
Was ist der „Throughput“?
Nicht funktionale Anforderungen

- *„a non-functional requirement (NFR) is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviors.”* (Wikipedia)
- Nicht-funktionale Anforderungen sollte man nicht „eben Mal“ in der Cafeküche schätzen → können ganzes Projekt zum Kippen bringen
- https://en.wikipedia.org/wiki/Non-functional_requirement#Examples
  - Reusability, Scalability, Portability, …

FH University of Applied Sciences TECHNIKUM WIEN

Typ vs. Instanz

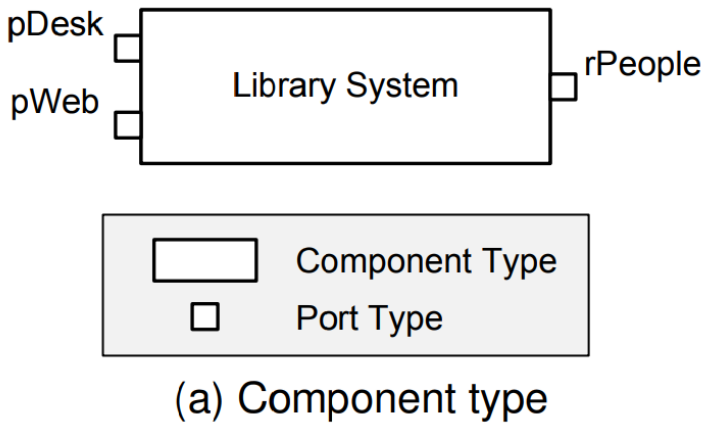
Typ vs. Instanz



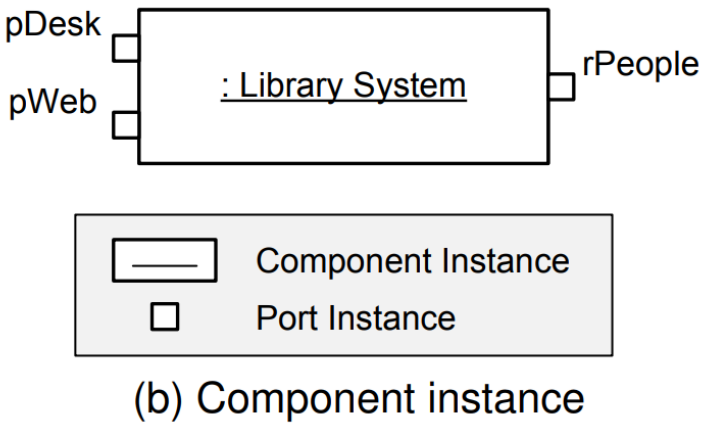
- In der objektorientierten Entwicklung haben wir auch Klassen, welche im System mehrfach zu Objekten „instanziiert“ werden können
- Ähnliches Konzept bei Komponenten:
  - Component Type
  - Component Instance
- In UML prinzipiell möglich – aber selten gesehen

Component Type vs. Instance

Component Type kann mehrfach instanziiert werden (Component Instance)



(a) Component type



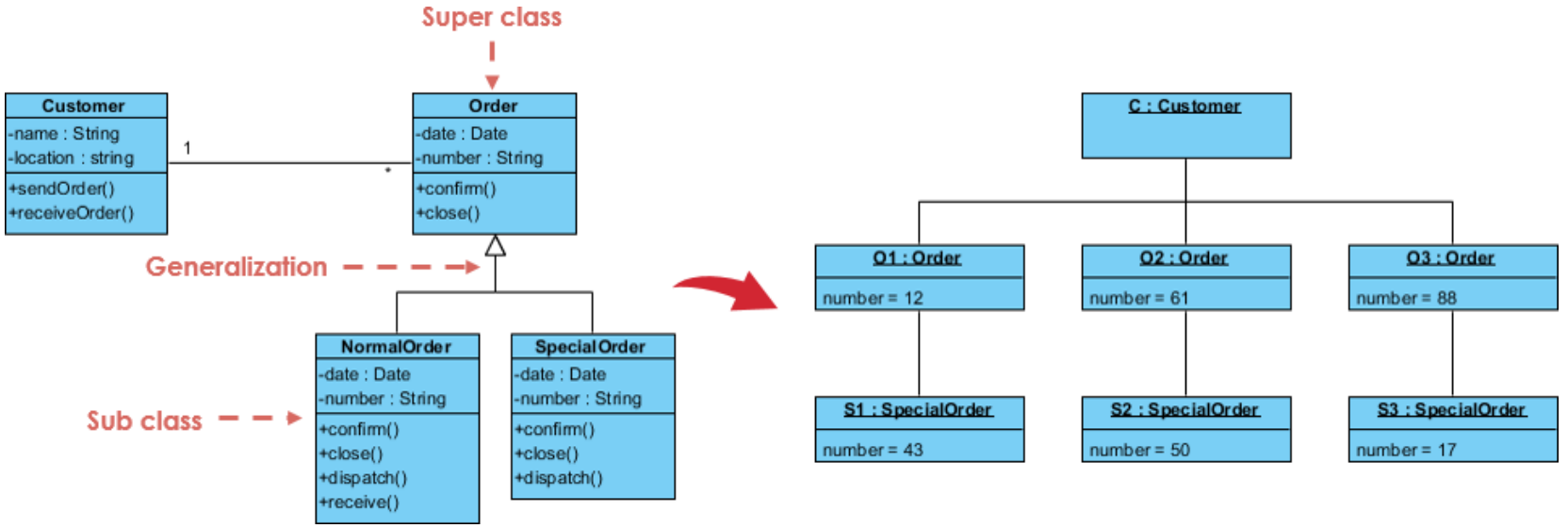
(b) Component instance

Siehe

- https://www.uml-diagrams.org/object.html
- https://www.uml-diagrams.org/class.html

UML 2.0: Gleich wie bei Class vs. Object Diagramm

Klasse kann mehrfach im System instanziiert werden (Objekt)



Source: visual-paradigm.com

FH University of Applied Sciences TECHNIKUM WIEN

Component vs. Module

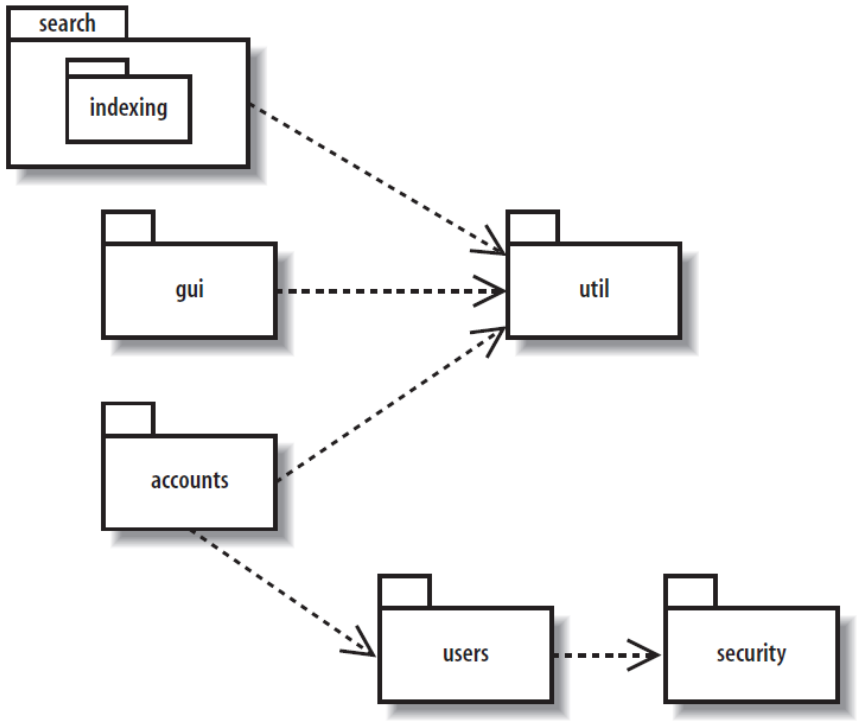
Was ist der Unterschied Component vs. Module?

- Module: Artifakte (Klassen, Interfaces), welche zusammen gruppiert sind
  - Existieren nur zur **Design-Time**
  - Module Strukturen existieren im Dateisystem (*.cs Files in einer Class-Library)
- Component-Instances sind instanzierte Component-Types, welche über einen wohldefinierten Weg kommunizieren
  - **Augenmerk auf Schnittstellen und Kommunikation**
  - Abgrenzung Module - Component-Type nicht eindeutig (*)
  - daher Augenmerk Component(-Instances) existieren nur zur **Runtime**

(*) Je nach Literatur unterschiedlich – wir belassen es daher bei: Module - Design-Time vs. Component - Runtime

UML 2.0 Package (Module)

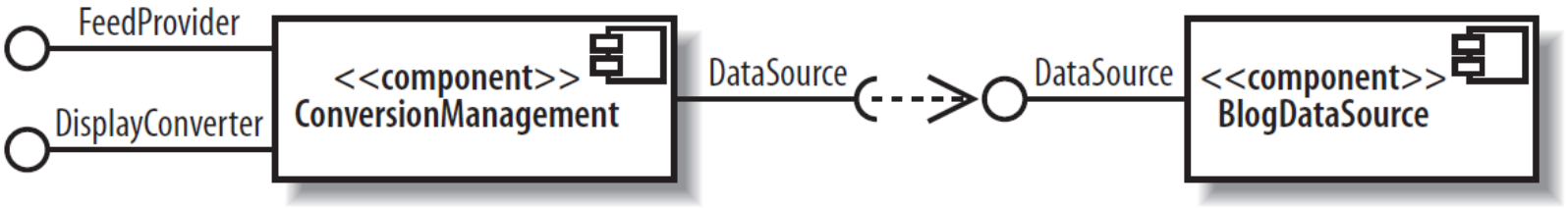
„*Paketdiagramme stellen dort zum Beispiel die Schichtung der Software oder die Unterteilung der Software in Module dar*.“ (Wikipedia)



Miles, R., & Hamilton, K. (2006). Learning UML 2.0. " O'Reilly Media, Inc.".

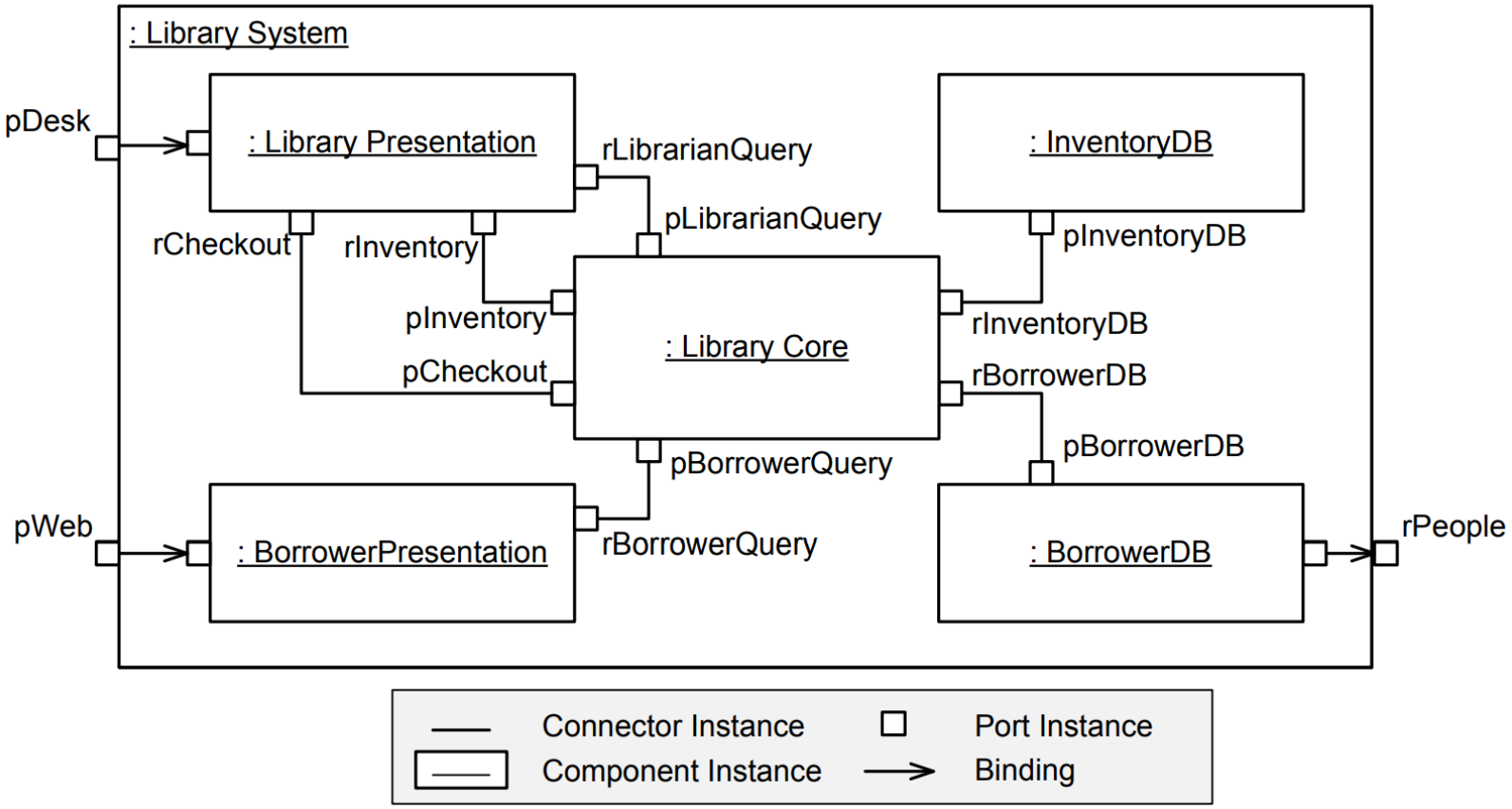
UML 2.0 Component

Man sieht: provided (Kreis) und required (Halbkreis) Interfaces bzw. den Connector (die ganze Verbindung bzw. die strichlierte Linie – je nach Darstellung)



Miles, R., & Hamilton, K. (2006). Learning UML 2.0. " O'Reilly Media, Inc.".

Verschachteln von Komponenten: Das Refinement



Ports, Connector … Graue Theorie?

- Connectors:
  - Wird man im Regelfall öfters wiederverwenden → die Separation macht daher durchaus Sinn
  - Ein Software Projekt wächst – oft leider mit technischen Schulden bzw. kommen ständig neue Anforderungen. Steht ein Refactoring an, so ist es nicht ungewöhnlich, dass der Kommunikations-Style wechselt. E.g. WCF → Broker-Style RabbitMQ → die Separation macht es daher einfach
  - Der eigentliche Code, wird vom „Noise“ der Kommunikation getrennt → einfacher zu verstehen
  - … ?
- Ports:
  - Hat man das Ganze gekapselt, wird man sich beim Testen relativ einfach tun, weil man schnell einen Stub basteln kann
  - U.U. mehrere kundenabhängige Implementierungen
  - … ?

Warum brauchen wir Komponenten?

- Sind ein essentieller Bestandteil einer **Software-Architektur**
- Um eine Software-Architektur beschreiben zu können braucht man u.a.
  - Components (beschrieben durch Interfaces / Ports, Constraints, Nonfunctional Properties, …)
  - Connectors
  - Modeling Configurations (vereinen der Elemente wie Interfaces, Components, … zu einer Topologie → das Verdrahten)

Medvidovic, N., & Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. IEEE Transactions on software engineering, 26(1), 70-93.

Was ist eine Software-Architektur?

- "*The software architecture of a system is the set of structures **needed to reason** about the system, which comprise software elements, relations among them, and properties of both." (Software Architecture in Practise)*
- *"The question, "Which U.S. cities can you travel to by boat?" **does not require a complete model** of the country"*
- Nicht vergessen: *"All models are wrong, but some are useful" (George Box)*
  - Ein Model ist immer eine Abstraktion – detailliert genug, ob die gestellten Fragen zu beantworten
- Details folgen in den nächsten Kapiteln …

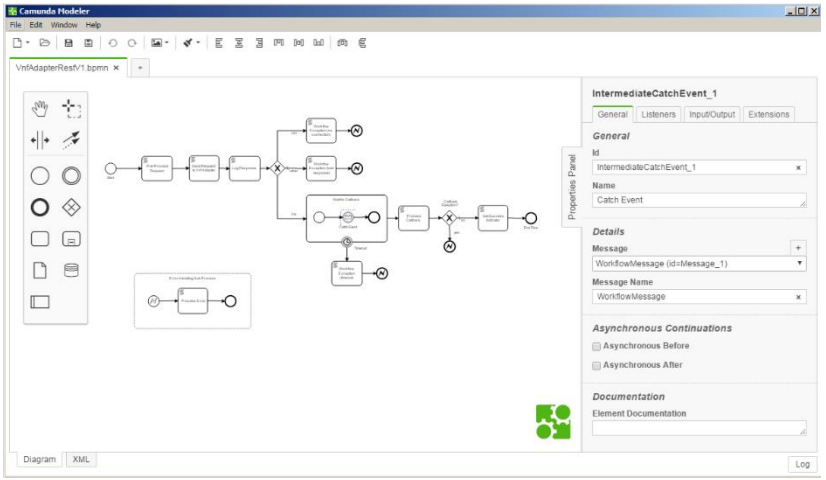
Exkurs: Warum brauchen wir Architektur jetzt?

- One of the best ways to **combat complexity** of software development is through the use of **abstraction**, **problem decomposition**, and **separation of concerns**. [1]

[1] Sendall, S., & Kozaczynski, W. (2003). Model transformation: The heart and soul of model-driven software development. IEEE software, 20(5), 42-45.

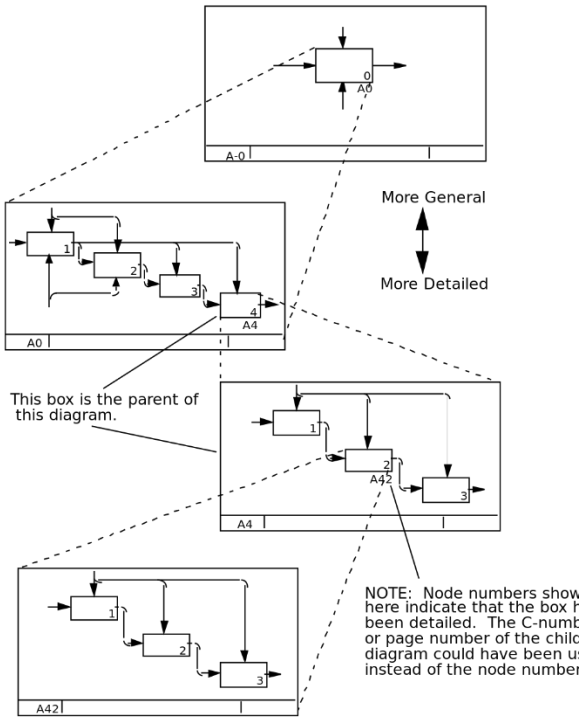
Abstraction

- Wikipedia: **Weglassens von Einzelheiten** und des Überführens auf etwas Allgemeineres oder Einfacheres
- In der Software allgegenwertig:
  - Methoden / Funktionen
  - Programmiersprachen (wie C#) selber
- Modellgetriebene Softwareentwicklung
  - E.g. Abbilden von Geschäftsprozessen (e.g. BPMN → Camunda)



Problem decomposition

- Wikipedia: **breaking a complex problem or system into parts** that are easier to conceive, understand, program, and maintain.
- Allgegenwärtig: Anforderungsanalyse, Umsetzung, …



Separation of concerns (SoC)

- Wikipedia: is a design principle for **separating** a computer program into **distinct sections**, so that each section addresses a **separate concern**
- Beispiele:
  - IP-Stack: Jede Schicht hat ihre Aufgabe → der Entwickler kann in der erforderlichen Schicht arbeiten
  - HTML, CSS, JavaScript: Jeweils unterschiedliche Aufgaben
  - Allgemein: Modulare Software