Universität Zürich[UZH]

Institut für Informatik

Martin Glinz Harald Gall

Software Engineering

Kapitel 5

Entwurf von Software

5.1 Grundlagen und Prinzipien

5.2 Architekturentwurf

5.3 Architekturstile

5.4 Entwurfsmuster (design patterns)

5.5 Modulkonzepte und Schnittstellen

5.6 Vertragsorientierter Entwurf

5.7 Zusammenarbeit

Motivation

- Kleinsoftware – kein systematischer Entwurf notwendig
- «Richtige» Software – braucht systematischen, strukturierten Aufbau
- ➪ Lösungskonzept zwingend
  - Lösung verstehen
  - Entwicklungsaufwand verteilen auf mehrere Personen
  - Lösung einbetten
  - Lösung geographisch verteilen
- Lösungskonzept legt Grundstein für leicht pflegbares System
- Konzeptfehler sind teuer

Definitionen und Begriffe

Konzipieren einer Lösung (architectural design) – Erstellung und Dokumentation des Architekturentwurfs oder Grobentwurfs eines Systems.

Dabei werden die wesentlichen Komponenten der Lösung und die Interaktionen zwischen diesen Komponenten festgelegt.

Architektur (architecture) – Die Organisationsstruktur eines Systems oder einer Komponente.

Entwurf (design) – 1. Der Prozess des Definierens von Architektur, Komponenten, Schnittstellen und anderen Charakteristika eines Systems oder einer Komponente. 2. Das Ergebnis des Prozesses gemäß 1.

Lösungskonzept (Software-Architektur, Systemarchitektur, software architecture, system architecture) – das Dokument, welches das Konzept der Lösung, d.h. die Architektur der zu erstellenden Software dokumentiert.

Entwurfsprinzipien – 1: Strukturen und Abstraktionen

Struktur: Gliedern der Lösung in Komponenten und Interaktionen

Abstraktion: Verstehen durch systematisches Vergröbern/Verfeinern

- Gewinnung von Übersicht; Weglassung der Details
- die Darstellung eines Details; Weglassung/Vergröberung des Rests
- Herstellung eines systematischen Zusammenhangs zwischen Übersichten und Detailsichten.
- Hauptsächlich vier Arten:
  - Komposition
  - Benutzung
  - Spezialisierung

  } vgl. Vorlesung Informatik IIa: Modellierung

  - Aspektbildung — vgl. Prinzip 7

Kapselnde Dekomposition

Ein System so in Teile zerlegen, dass

- jeder Teil mit möglichst wenig Kenntnissen des Ganzen und der übrigen Teile verstanden werden kann
- das Ganze ohne Detailkenntnisse über die Teile verstanden werden kann

DAS Abstraktionsmittel für das Verstehen komplexer Systeme

Entwurfsprinzipien – 2: Modularität

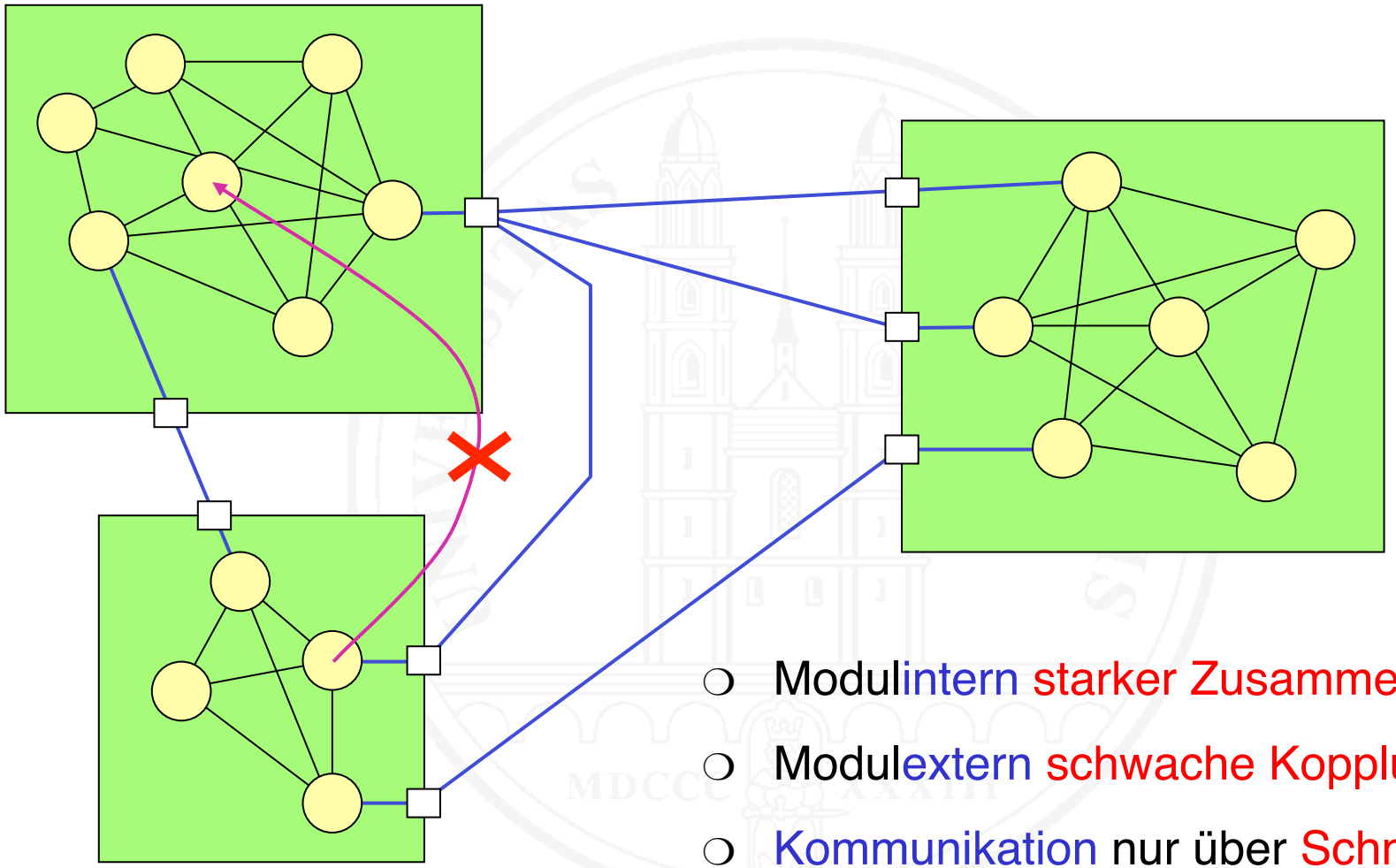
Modularisierung ist eine Hauptaufgabe der Konzipierung von Software

Modul (module) – Ein benannter, klar abgegrenzter Teil eines Systems.

Gute Module haben folgende Eigenschaften:

- In sich geschlossene Einheit
- Ohne Kenntnisse über inneren Aufbau verwendbar
- Kommunikation mit Umgebung ausschließlich über Schnittstellen
- Im Inneren rückwirkungsfrei änderbar
- Korrektheit ohne Kenntnis der Einbettung ins Gesamtsystem prüfbar
- Erlaubt Komposition und Dekomposition

Das Prinzip einer modularen Struktur



- Modulintern starker Zusammenhang
- Modulextern schwache Kopplung
- Kommunikation nur über Schnittstellen
- Modulinneres von außen nicht sichtbar

Messung der Güte einer Modularisierung

Zwei charakteristische Maße: Kohäsion und Kopplung

Kohäsion (cohesion) – Ein Maß für die Stärke des inneren Zusammenhangs eines Moduls.

- Je höher die Kohäsion, desto besser die Modularisierung
  - schlecht: zufällig, zeitlich
  - gut: funktional, objektbezogen

Kopplung (coupling) – Ein Maß für die Abhängigkeit zwischen zwei Modulen.

- Je geringer die wechselseitige Kopplung zwischen den Modulen, desto besser die Modularisierung
  - schlecht: Inhaltskopplung, globale Kopplung
  - akzeptabel: Datenbereichskopplung
  - gut: Datenkopplung

Mini-Übung 5.1

Eine Anlage füllt eine Flüssigkeit in Flaschen ab. Sie besteht aus einem Tank, zwei Förderbändern für das Zuführen und Wegführen der Flaschen und einer Abfüllstation mit Waage.

Die Software für die Steuerung dieser Anlage sei wie folgt modularisiert:

- Tank (Steuerung des Tank-Einlassventils, Feststellen des Füllstands)
- Abfüllung (Steuerung des Tank-Abfüllventils, Ablesen der Waage, Zuführen/Wegführen von Flaschen zur Abfüllstation)
- Band (Steuerung der Förderbänder)
- Init (Initialisierung der gesamten Steuerung)

Beurteilen Sie die Qualität dieser Modularisierung. Wo sehen Sie Probleme?

Entwurfsprinzipien – 3: Geheimnisprinzip

Geheimnisprinzip (information hiding) – Kriterium zur Gliederung eines Gebildes in Komponenten, so dass

- jede Komponente eine Leistung (oder eine Gruppe logisch eng zusammenhängender Leistungen) vollständig erbringt,
- außerhalb der Komponente nur bekannt ist, was die Komponente leistet,
- nach außen verborgen wird, wie sie ihre Leistungen erbringt.

[Parnas 1972]

- ➪ Fundamentales Prinzip zur Beherrschung komplexer Systeme
- ➪ Auch im täglichen Leben fortwährend benutzt
- ➪ Liefert gute Modularisierungen

Modularität und Geheimnisprinzip im Alltag – 1

Modulare Bauweise ermöglicht Produktvielfalt bei geringen Kosten

Modularität und Geheimnisprinzip im Alltag – 2

Die Bahn benutzen können,

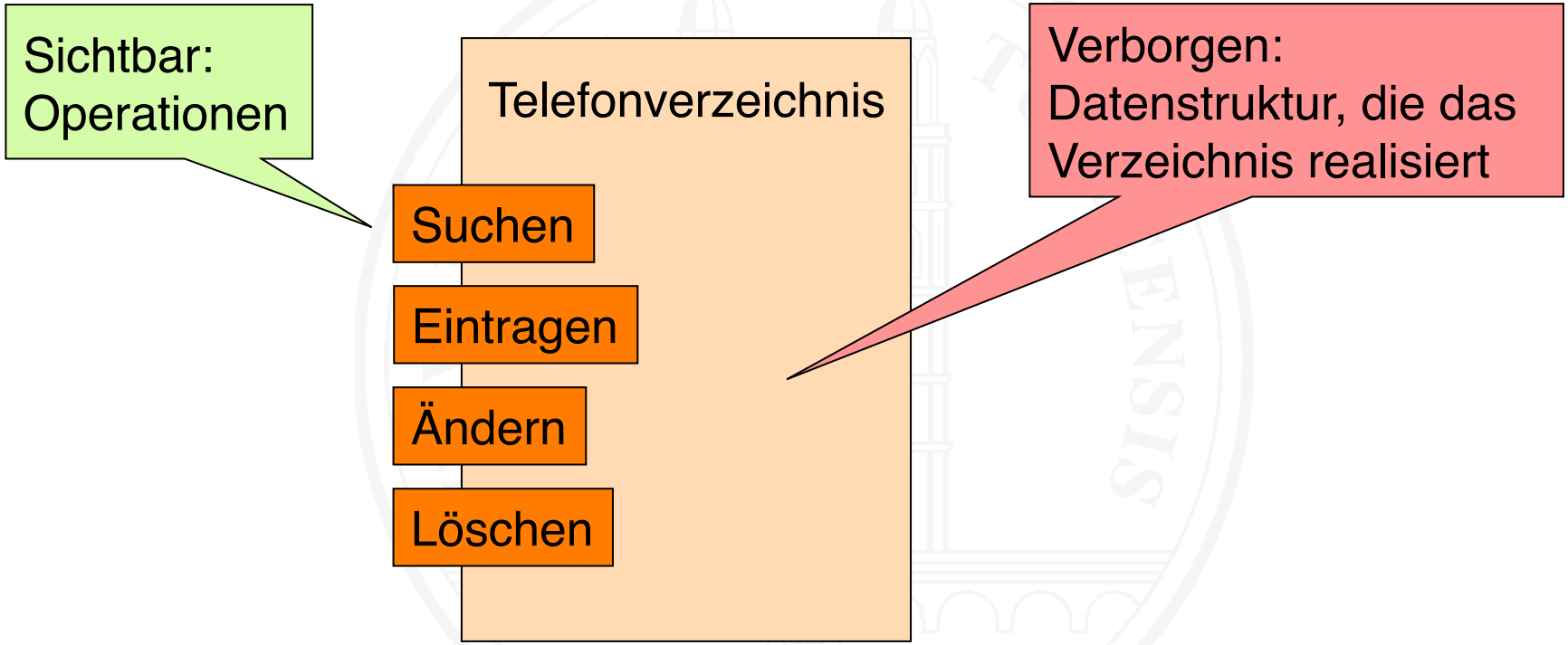
ohne wissen zu müssen, wie eine Bahn betrieben wird

Das Geheimnisprinzip im Software-Entwurf

- Komponente – Modul
- Leistung – Funktionalität des Moduls
- WAS – Modulschnittstelle
- WIE – Entwurfsentscheidungen und deren Realisierung

- Vier typische Arten von Entwurfsentscheidungen bei Software:
  - Wie eine Funktion realisiert ist
  - Wie ein Objekt aus dem Anwendungsbereich repräsentiert/realisiert ist
  - Wie eine Datenstruktur aufgebaut ist / bearbeitet werden kann
  - Wie Leistungen Dritter realisiert sind

Beispiel


Sichtbar:
Operationen
Telefonverzeichnis
Suchen
Eintragen
Ändern
Löschen
Verborgen:
Datenstruktur, die das
Verzeichnis realisiert

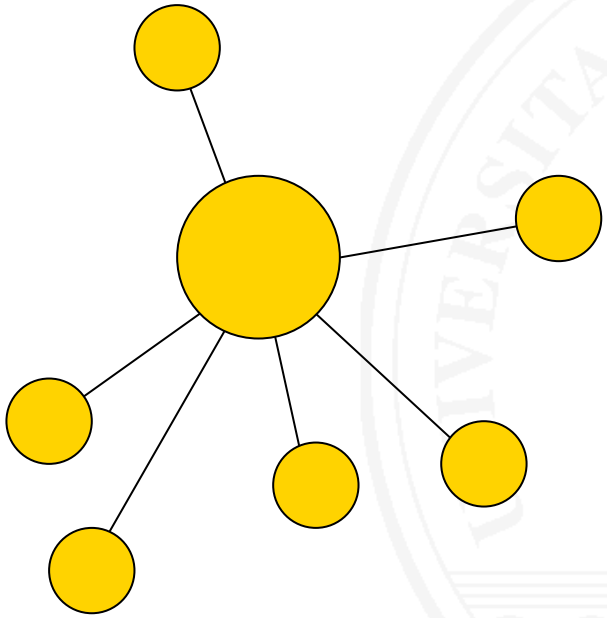
Entwurfsprinzipien – 4: Schnittstellen und Verträge

Schnittstelle (interface) – Verbindungsglied zwischen einem Modul und der Außenwelt zwecks Austausch von Information

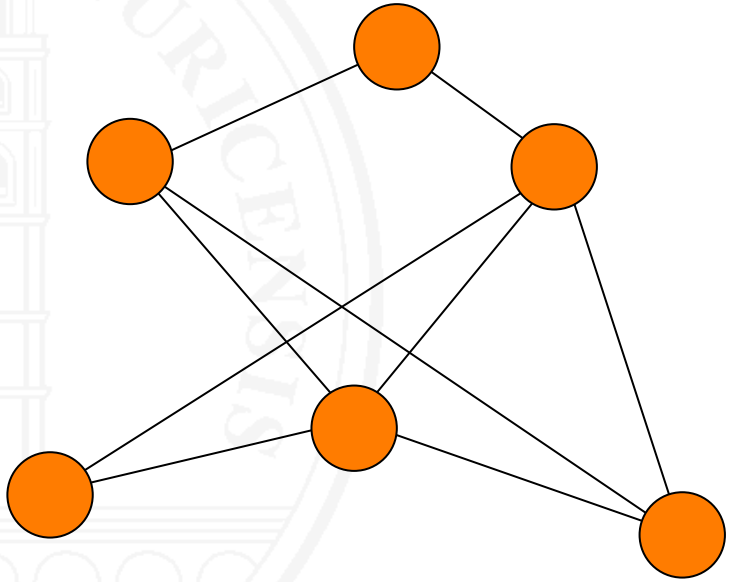
- Leistungen, die ein Modul zur Nutzung anbietet
- Bedarf eines Moduls an Leistungen Dritter
- Beschreibung in Form eines Vertrags (contract) zwischen Anbieter und Verwender: Rechte und Pflichten
- Details siehe Abschnitt 5.6: Vertragsorientierter Entwurf

Entwurfsprinzipien – 5: Nebenläufigkeit (1)

Problem: Gleichzeitige, koordinierte Bearbeitung mehrerer Aufgaben



Mehrere Verwender nutzen parallel oder zeitlich verzahnt gemeinsame Dienstleistungen



Unabhängig arbeitende Agenten kooperieren zwecks Erbringung von Leistungen

Entwurfsprinzipien – 5: Nebenläufigkeit (2)

- Nebenläufigkeit wird durch Prozesse realisiert

Prozess (process) – Eine durch ein Programm gegebene Folge von Aktionen, die sich in Bearbeitung befindet.

Nebenläufigkeit (concurrency) – Die parallele oder zeitlich verzahnte Bearbeitung mehrerer Aufgaben.

- Entwurfsprobleme
  - Welcher Prozess bearbeitet welche Aufgaben?
  - Wann und wie tauschen Prozesse welche Information aus?
  - Wann und wie synchronisieren Prozesse ihren Arbeitsfortschritt?

Entwurfsprinzipien – 6: Berücksichtigung der Ressourcen

- Zuordnung der Komponenten zu Ressourcen ist notwendig:
  - Abschätzung der technischen Machbarkeit
  - Erfüllbarkeit der gestellten Anforderungen (vor allem Leistungen)

- Zuzuordnen sind
  - Module zu Prozessen
  - Prozesse zu Prozessoren
  - Daten zu Speichern bzw. Medien
  - Prozesskommunikation zu Kommunikationstechnologien und -medien.

Entwurfsprinzipien – 7: Aspektbildung

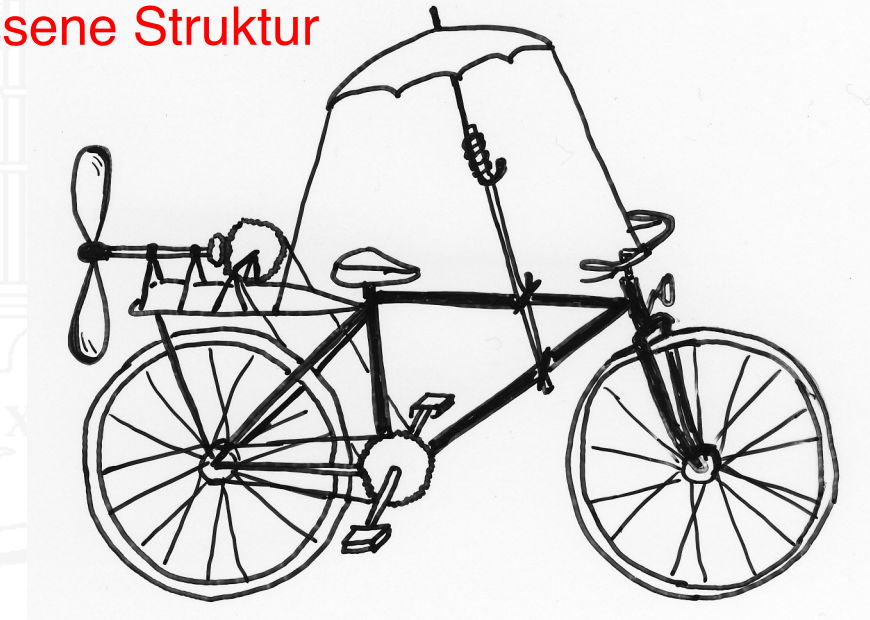
- Beschreibung von Querschnittsaufgaben
- Muss systemweit, dafür aspektspezifisch geschehen
- Typische Aspekte:
  - Datenhaltungskonzept, insbesondere das konzeptionelle Datenbankschema bei Verwendung einer Datenbank
  - Mensch-Maschine-Kommunikationskonzept für die Gestaltung der Benutzerschnittstelle
  - Fehlerbehandlungs-, Fehlertoleranz-, Sicherheitskonzepte

Entwurfsprinzipien – 8: Nutzung von Vorhandenem

- Wo möglich: nicht neu entwickeln
  - ➪ Wiederverwenden, Beschaffen
- Zu untersuchen:
  - Vollständige Beschaffung (Standardsoftware, konfigurierbare Bausteine)
  - Beschaffung abgeschlossener Teilsysteme (zum Beispiel Datenbanksystem)
  - Realisierung durch Einbettung in einen existierenden Software-Rahmen (framework)
  - Nutzung einzelner Komponenten (Programm- /Klassenbibliotheken)
  - Wiederverwendung von Architektur- und Entwurfsideen: Architekturmetaphern, Entwurfsmuster (design patterns)
  - Modifikation des Lösungskonzepts zwecks Verwendung von Standardkomponenten

Entwurfsprinzipien – 9: Ästhetik

- Wahl und konsequente Verwendung eines Architekturstils
- Klar erkennbaren, gestalteten Strukturen
- Wenig Gewordenes
- Kein Gewursteltes
- Der Struktur des Problems angemessene Struktur der Architektur
- Wahl der einfachsten und klarsten Lösung aus der Menge der möglichen Lösungen.



Entwurfsprinzipien – 10: Qualität

Merkmale guter Entwürfe:

- Effektivität: Erfüllt die Vorgaben und löst das Problem des Auftraggebers
- Wirtschaftlichkeit: Gebrauchstauglich, kostengünstig und mehrfachverwendbar bzw. mehrfachverwendet
- Softwaretechnische Güte: Leicht verständlich, robust, zuverlässig änderungsfreundlich

⇨ Erfordert kontinuierliche Prüfmaßnahmen im Entwurfsprozess

Der Entwurfsprozess

- Erster Schritt der Lösung
- Anforderungsspezifikation als Vorgabe notwendig
- Zeitliche und hierarchische Verzahnung von Anforderungsspezifikation und Architekturentwurf möglich
- Ergebnisse immer in separaten Dokumenten

- Architekturentwurf: Komponenten, Schnittstellen, Interaktionen

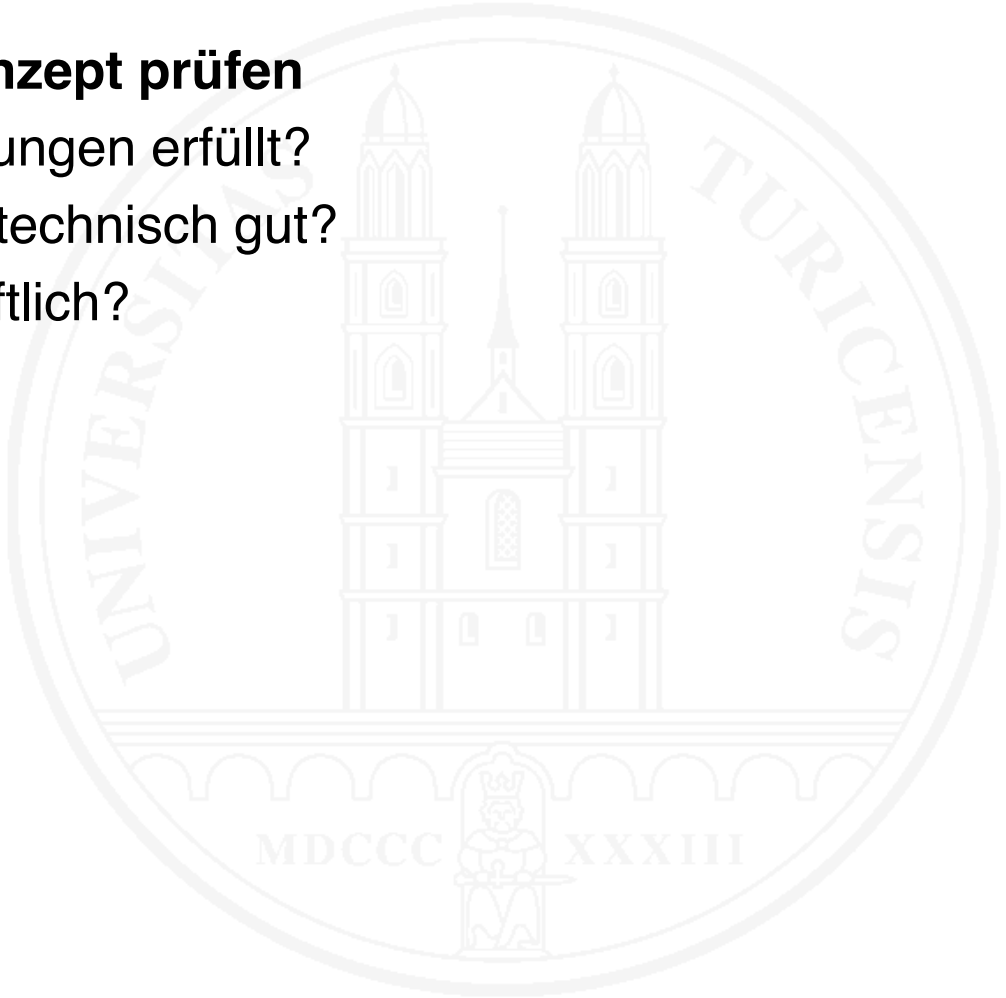
- Detailentwurf: Algorithmen und interne Datenstrukturen

Aufgaben des Architekturentwurfs – 1

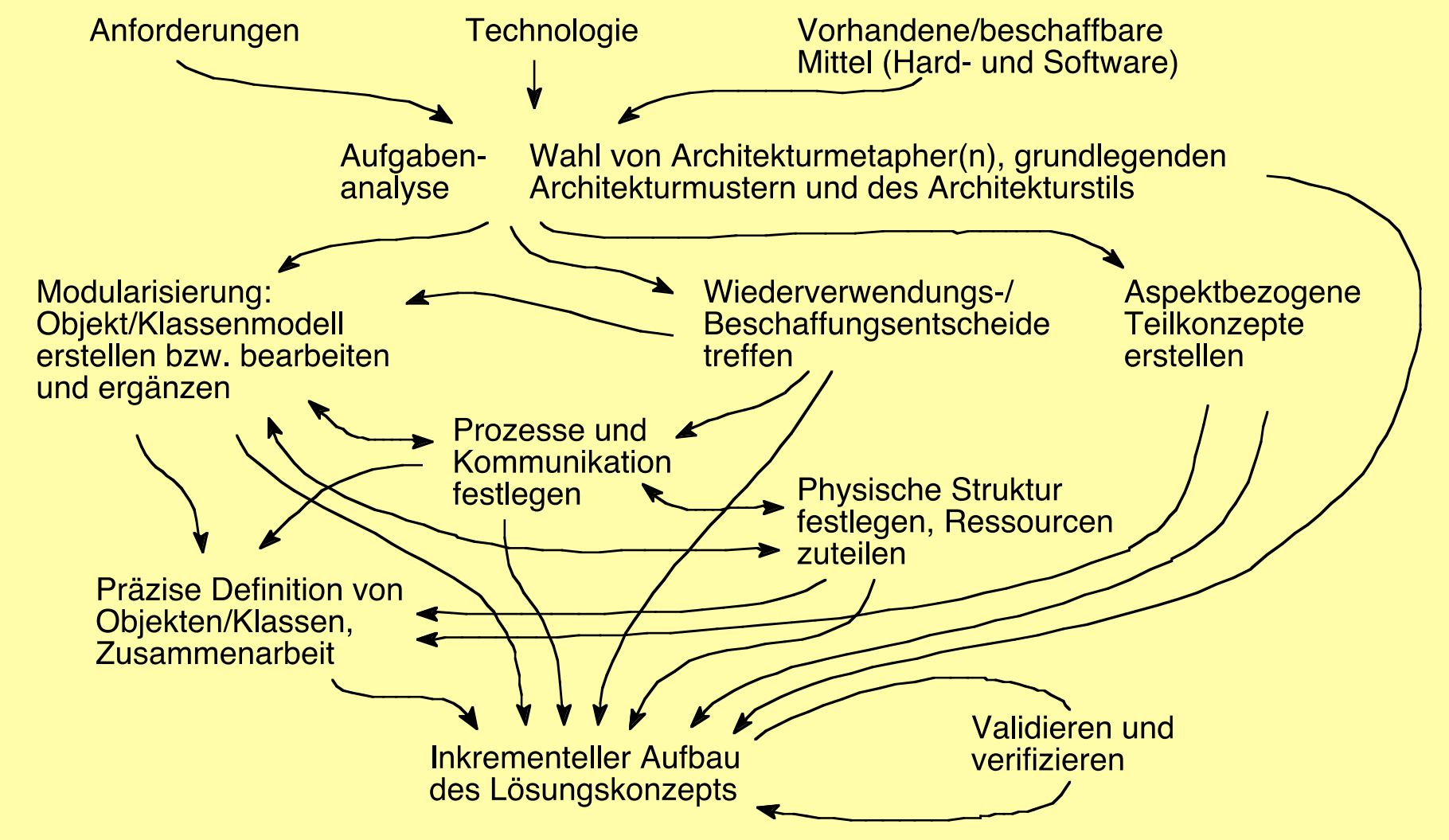
- **Aufgabe analysieren**
  - Anforderungen verstehen
  - Vorhandene bzw. beschaffbare Technologien und Mittel analysieren
- **Architektur modellieren und dokumentieren**
  - Grundlegende Systemarchitektur festlegen: Muster, Metaphern ➪ Stil
  - Modularisieren
  - Nebenläufige Lösungen in Prozesse gliedern
  - Wiederverwendungs- und Beschaffungsentscheide treffen
  - Ressourcen zuordnen
  - Aspektbezogene Teilkonzepte für Querschnittsaufgaben erstellen
  - Lösungskonzept (als Dokument) erstellen

Aufgaben des Architekturentwurfs – 2

- **Lösungskonzept prüfen**
  - Anforderungen erfüllt?
  - Softwaretechnisch gut?
  - Wirtschaftlich?



Vorgehen und Zusammenhänge



Variantenbehandlung

- Ganzen Lösungsraum betrachten
- Lösungsvarianten
  - erkennen
  - verfolgen
  - abwägen
    - was kostet es, das Optimum zu verfehlen?
    - was kostet die Untersuchung?
  - entscheiden
  - dokumentieren
- Kosten
  - Nicht nur Entwicklungskosten der Variante!
  - auch Betriebskosten, Pflegekosten, Folgekosten anderswo
- Beschaffungsvariante immer betrachten

Das Lösungskonzept – 1

Dokumentiert das Ergebnis des Architekturentwurfs

Mögliche Gliederung:

1. **Einleitung**
   - 1.1 Überblick
   - 1.2 Ziele und Vorgaben
   - 1.3 Einbettung und Abgrenzung
   - 1.4 Lösungsalternativen
2. **Struktur der Lösung**
   - 2.1 Übersicht
   - 2.2 Prozessstruktur
   - 2.3 Modulare Struktur
   - 2.4 Entwurf der Module
   - 2.5 Physische Struktur

Das Lösungskonzept – 2

3. Aspektbezogene Teilkonzepte

Ein Unterkapitel je interessierendem Aspekt, zum Beispiel Datenhaltungskonzept, Mensch-Maschine-Kommunikationskonzept, Fehlerbehandlungskonzept, Fehlertoleranzkonzept, Sicherheitskonzept, etc.

4. Voraussetzungen und benötigte Hilfsmittel

4.1 Benötigte Software

4.2 Benötigte Hardware

4.3 Benötigtes Umfeld

Quellennachweis

5.1 Grundlagen und Prinzipien

5.2 Architekturentwurf

5.3 Architekturstile

5.4 Entwurfsmuster (design patterns)

5.5 Modulkonzepte und Schnittstellen

5.6 Vertragsorientierter Entwurf

5.7 Zusammenarbeit

Software Architecture

Sears Tower, Chicago

Piazza del Campidoglio, Rome

Torii of Itsukushima, Japan

The Sydney Opera House

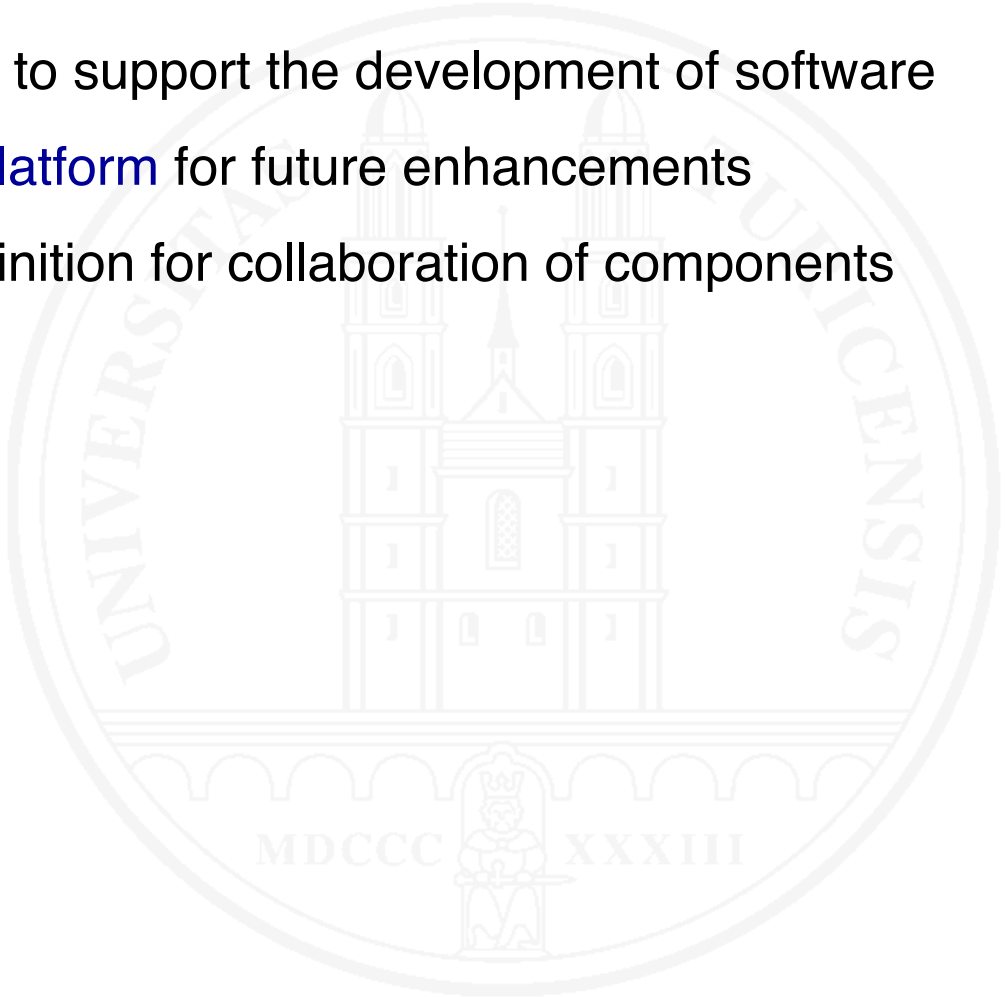
Facts and Figures:

- Was designed by Danish architect Jørn Utzon
- Was opened by Queen Elizabeth II on 20 October 1973
- Presented, as its first performance, The Australian Opera's production of War and Peace by Prokofiev
- Cost $AU 102.000.000 to build
- Conducts 3.000 events each year
- Includes 1.000 rooms

- Is 185 metres long and 120 metres wide
- Has 2.194 pre-cast concrete sections as its roof
- Has roof sections weighing up to 15 tons
- Has roof sections held together by 350 kms of tensioned steel cable
- Has over 1 million tiles on the roof
- Uses 6.225 square metres of glass and 645 kilometres of electric cable

Goals

- A framework to support the development of software
- Integration platform for future enhancements
- Interface definition for collaboration of components



What is Software Architecture?

- **Definition:**
  - A software system's architecture is the set of *principal design decisions* about the system
- Software architecture is the blueprint for a software system's construction and evolution
- Design decisions encompass every facet of the system under development
  - Structure
  - Behavior
  - Interaction
  - Non-functional properties

What is “Principal”?

- “Principal” implies a degree of importance that grants a design decision “architectural status”
  - It implies that not all design decisions are architectural
  - That is, they do not necessarily impact a system’s architecture
- How one defines “principal” will depend on what the stakeholders define as the system goals

Other Definitions of Software Architecture

- Perry and Wolf
  - Software Architecture = { Elements, Form, Rationale }
    what *how* *why*
- Shaw and Garlan
  - Software architecture [is a level of design that] involves
    - the description of elements from which systems are built,
    - interactions among those elements,
    - patterns that guide their composition, and
    - constraints on these patterns.
- Kruchten
  - Software architecture deals with the design and implementation of the high-level structure of software.
  - Architecture deals with abstraction, decomposition, composition, style, and *aesthetics*.

Temporal Aspect

- Design decisions are and unmade over a system's lifetime

  → Architecture has a temporal aspect
- At any given point in time the system has only one architecture
- A system's architecture will change over time

Prescriptive vs. Descriptive Architecture

- A system's *prescriptive architecture* captures the design decisions made prior to the system's construction
  - It is the *as-conceived* or *as-intended* architecture
- A system's *descriptive architecture* describes how the system has been built
  - It is the *as-implemented* or *as-realized* architecture

Architectural Evolution

- When a system evolves, ideally its prescriptive architecture is modified first
- In practice, the system – and thus its descriptive architecture – is often directly modified
- This happens because of
  - Developer sloppiness
  - Perception of short deadlines which prevent thinking through and documenting
  - Lack of documented prescriptive architecture
  - Need or desire for code optimizations
  - Inadequate techniques or tool support

Architectural Degradation

- Two related concepts
  - Architectural drift
  - Architectural erosion
- *Architectural drift* is introduction of principal design decisions into a system's descriptive architecture that
  - are not included in, encompassed by, or implied by the prescriptive architecture
  - but which do not violate any of the prescriptive architecture's design decisions
- *Architectural erosion* is the introduction of architectural design decisions into a system's descriptive architecture that violate its prescriptive architecture

Architectural Recovery

- If architectural degradation is allowed to occur, one will be forced to *recover* the system's architecture sooner or later
- *Architectural recovery* is the process of determining a software system's architecture from its implementation-level artifacts
- Implementation-level artifacts can be
  - Source code
  - Executable files
  - Java .class files

Deployment

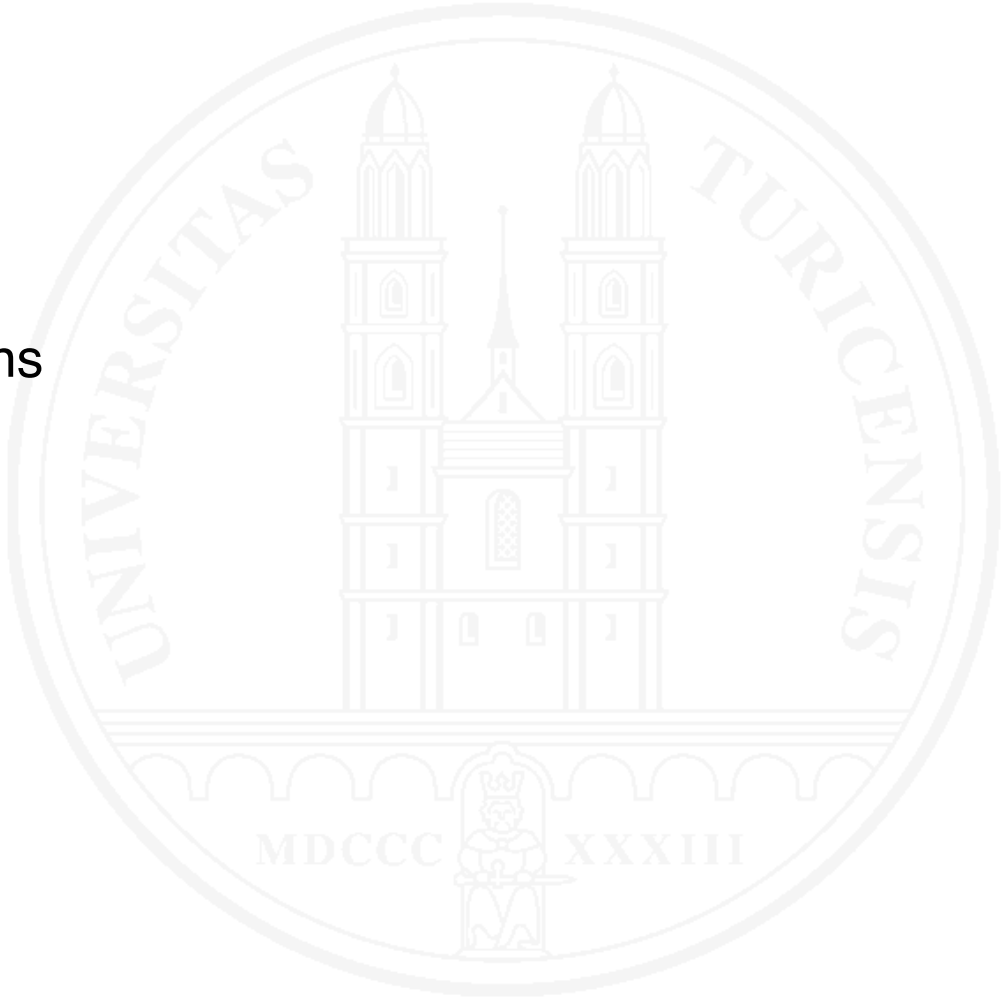
- A software system cannot fulfill its purpose until it is *deployed*
  - Executable modules are physically placed on the hardware devices on which they are supposed to run
- The deployment view of an architecture can be critical in assessing whether the system will be able to satisfy its requirements
- Possible assessment dimensions
  - Available memory
  - Power consumption
  - Required network bandwidth

Software Architecture's Elements

- A software system's architecture typically is not (and should not be) a uniform monolith
- A software system's architecture should be a composition and interplay of different elements
  - Processing
  - Data, also referred as information or state
  - Interaction

Elements of Software Architecture

- Components
- Connectors
- Interfaces
- Configurations



Components

- Elements that encapsulate processing and data in a system's architecture are referred to as *software components*
- **Definition**
  - A *software component* is an architectural entity that
    - encapsulates a subset of the system's functionality and/or data
    - restricts access to that subset via an explicitly defined interface
    - has explicitly defined dependencies on its required execution context
- Components typically provide application-specific services

Connectors

- In complex systems *interaction* may become more important and challenging than the functionality of the individual components
- **Definition**
  - A *software connector* is an architectural building block tasked with effecting and regulating interactions among components
- In many software systems connectors are usually simple procedure calls or shared data accesses
  - Much more sophisticated and complex connectors are possible!
- Connectors typically provide application-independent interaction facilities

Examples of Connectors

- Procedure call connectors
- Shared memory connectors
- Message passing connectors
- Streaming connectors
- Distribution connectors
- Wrapper/adaptor connectors

Configurations

- Components and connectors are composed in a specific way in a given system's architecture to accomplish that system's objective
- **Definition**
  - An *architectural configuration*, or topology, is a set of specific associations between the components and connectors of a software system's architecture

Interfaces

- An *interface* is the external “connection point” on a component or connector that describes how other components/connectors interact with it
- Provided *and* required interfaces are important
- Spectrum of interface specification
  - Loosely specified (events go in, events go out)
  - API style (list of functions)
  - Very highly specified (event protocols across the interface in CSP)
- Interfaces are *the* key to component interoperability (or lack thereof)

Architectural Styles

Architectural Styles

- Certain design choices regularly result in solutions with superior properties
  - Compared to other possible alternatives, solutions such as this are more elegant, effective, efficient, dependable, evolvable, scalable, and so on
- **Definition**
  - An *architectural style* is a named collection of architectural design decisions that
    - are applicable in a given development context
    - constrain architectural design decisions that are specific to a particular system within that context
    - elicit beneficial qualities in each resulting system

Architectural Patterns

- **Definition**
  - An *architectural pattern* is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears
- A widely used pattern in modern distributed systems is the *three-tiered system* pattern
  - Science
  - Banking
  - E-commerce
  - Reservation systems

Three-Tiered Pattern

- Front Tier
  - Contains the user interface functionality to access the system's services
- Middle Tier
  - Contains the application's major functionality
- Back Tier
  - Contains the application's data access and storage functionality

Architectural Models, Views, and Visualizations

- Architecture Model
  - An artifact documenting some or all of the architectural design decisions about a system
- Architecture Visualization
  - A way of depicting some or all of the architectural design decisions about a system to a stakeholder
- Architecture View
  - A subset of related architectural design decisions

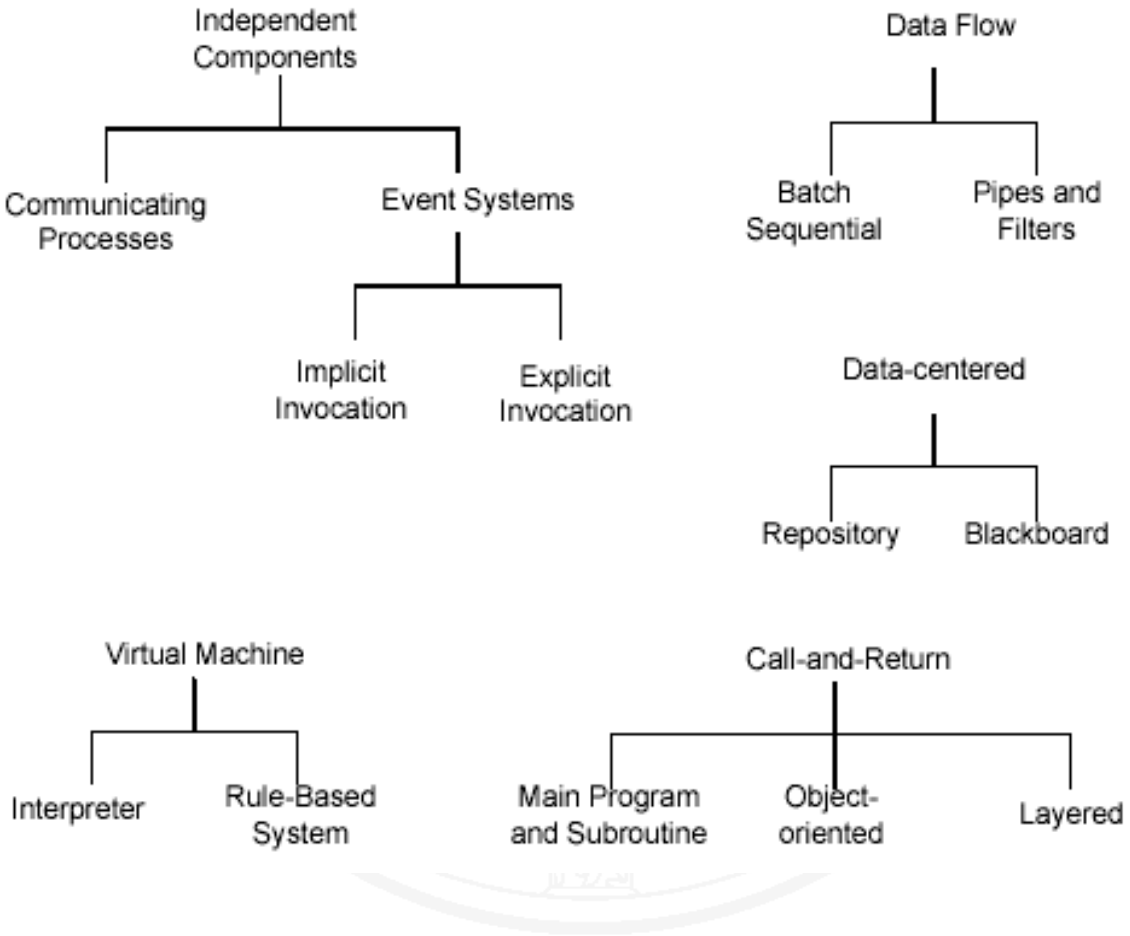
Stakeholders in a System's Architecture

- Architects
- Developers
- Testers
- Managers
- Customers
- Users
- Vendors

Characterizing Architectural Styles

- Component and Connector characteristics
- Allowed configurations
- Underlying computational model
- Stylistic invariants
- Common examples of its use
- Advantages and disadvantages
- Common specializations

Catalogue of Architectural Styles



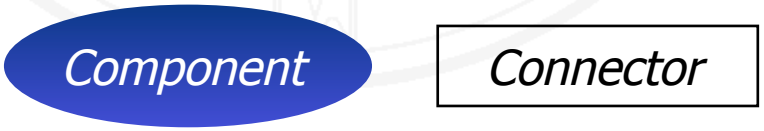
The Pipe-and-Filter Style

- Pipes (= *connectors*)
  - Provide the output of a filter as input to another filter
- Filters (= *components*)
  - Read input data stream and transform it into an output data stream
- Simple example:
  - Unix shell: piping of components (commands) via “|”
    - cat {myfile} | grep “arch” | sort … | more
- Filters are independent components that
  - do not share status with other components
  - do not know the identity of their neighbors (input/output)
- Specializations
  - pipelines
  - bounded pipes (limited memory)
  - typed pipes (for specific type of data)

Pipe-and-Filter Example: UNIX Text Processing


```
% pic mydoc.t | eqn | tbl | troff | lpr
```

Legend:



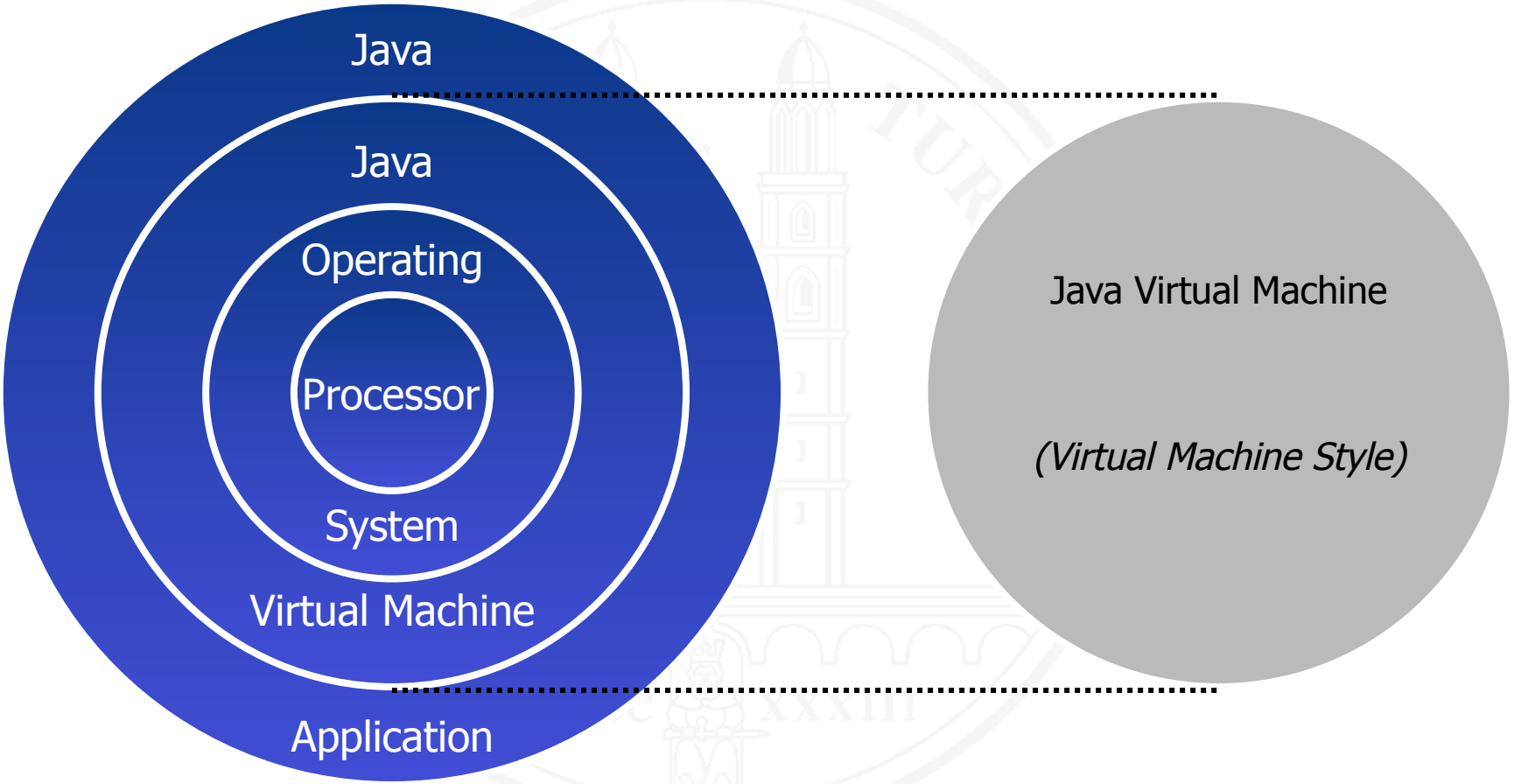
Pipe-and-Filter Advantages and Disadvantages

- Advantages
  - Simplicity: Simple, intuitive, efficient composition of components
  - Reusability: High potential for reuse of components
  - Evolvability: Changing architectures is trivial
  - Efficiency: Limited amount of concurrency (contrast batch-sequential)
  - Consistency: All components have the same interfaces, only one type of connector
  - Distributability: Byte streams can be sent across networks
- Disadvantages
  - Batch-oriented processing
  - Must agree on lowest-common-denominator data format
  - Does not guarantee semantics
  - Limited application domain: stateless data transformation

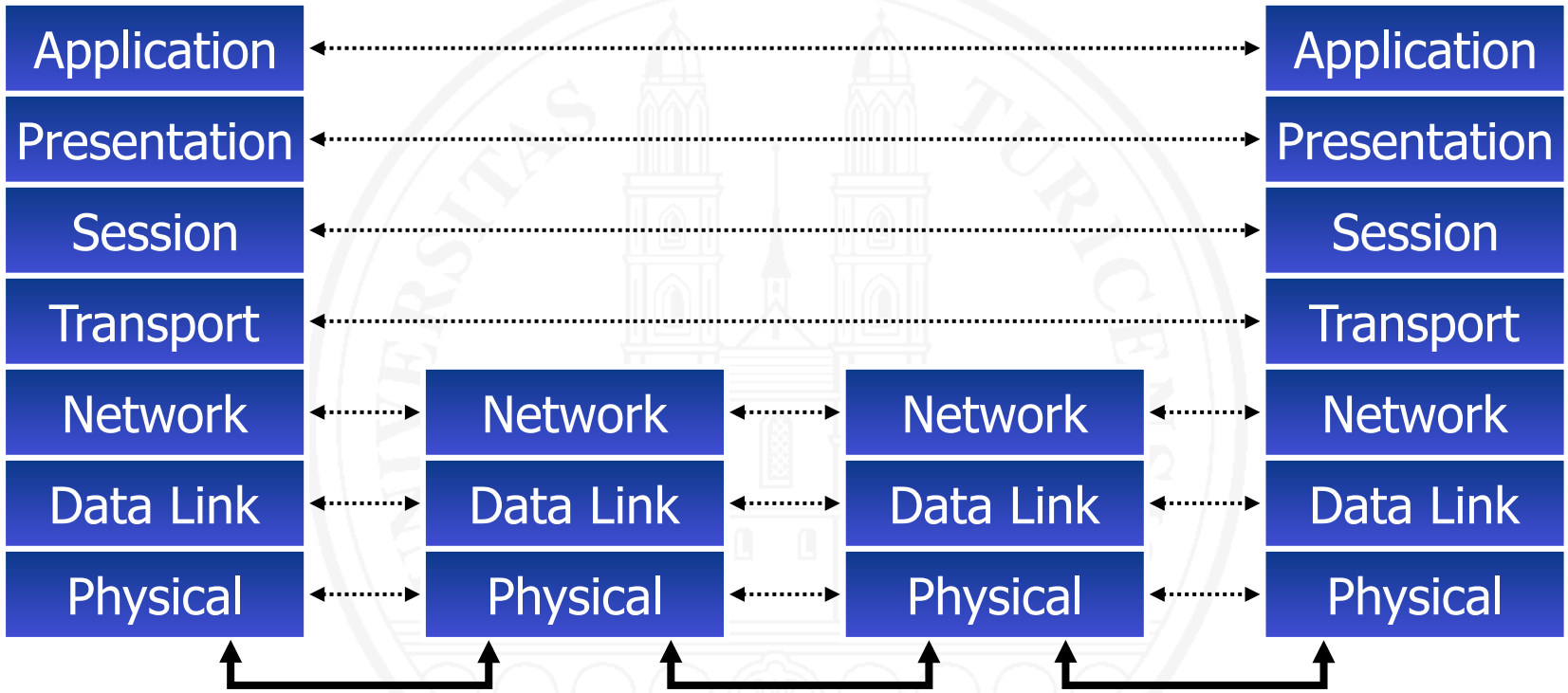
The Layered System Style

- Components
  - Programs or subprograms
- Connectors
  - Procedure calls or system calls
- Configurations
  - “Onion” or “stovepipe” structure, possibly replicated
- Underlying computational model
  - Procedure call/return
- Stylistic invariants
  - Each layer provides a service only to the immediate layer “above” (at the next higher level of abstraction) and uses the service only of the immediate layer “below” (at the next lower level of abstraction)

Layered Virtual Machine Example: Java



Layered System Example: OSI Protocol Stack



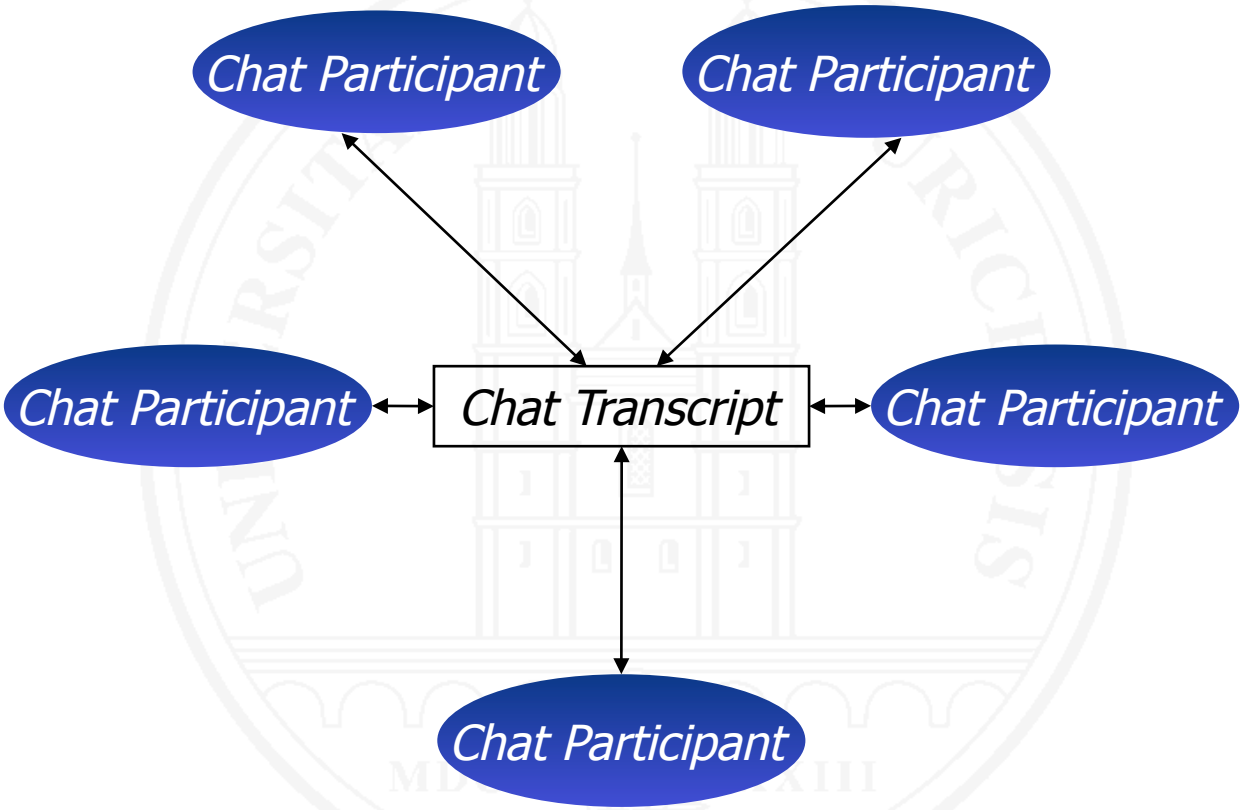
Layered System: Advantages and Disadvantages

- Advantages
  - Decomposability: Effective separation of concerns
  - Maintainability: Changes that do not affect layer interfaces are easy to make
  - Evolvability: Potential for adding layers
  - Adaptability/Portability: Can replace inner layers as long as interfaces remain the same (consider swapping out a Solaris JVM for a Linux one)
  - Understandability: Strict set of dependencies allow you to ignore outer layers
- Disadvantages
  - Performance degrades with too many layers
  - Can be difficult to cleanly assign functionality to the “right” layer

The Blackboard Style

- Components
  - Blackboard client programs
- Connector
  - Blackboard: shared data repository, possibly with finite capacity
- Configurations
  - Multiple clients sharing single blackboard
- Underlying computational model
  - Synchronized, shared data transactions, with control driven entirely by blackboard state
- Stylistic invariants
  - All clients see all transactions in the same order

Blackboard Example: A Chat Room



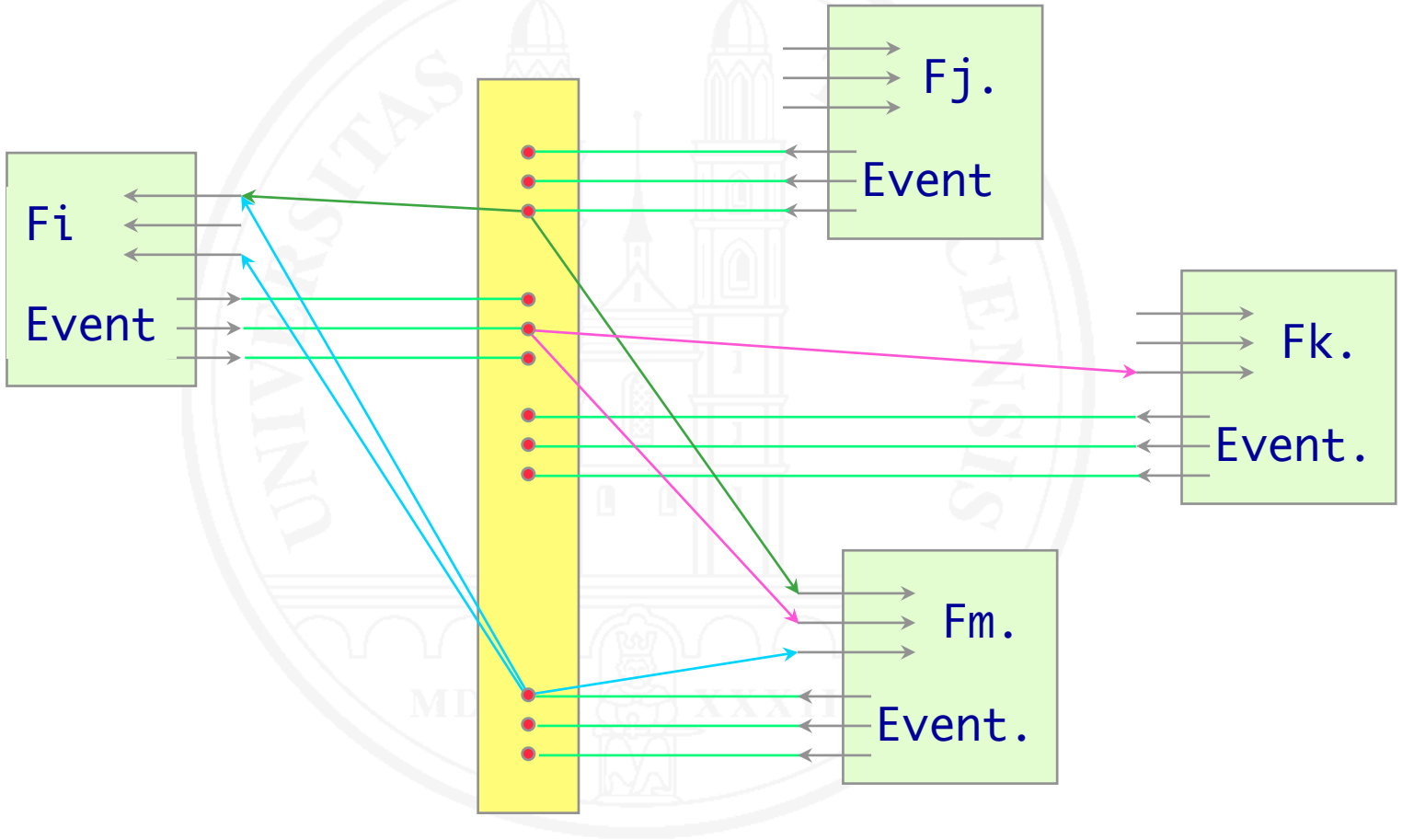
Blackboard Advantages and Disadvantages

- Advantages
  - Simplicity: Only one connector (the blackboard) that everyone uses
  - Evolvability: New types of components can be added easily
  - Reliability(?): Concurrency controls of information, traditionally a tricky problem, can be largely addressed in the blackboard
- Disadvantages
  - Blackboard becomes a bottleneck with too many clients
  - Implicit “partitions” of information on the blackboard may cause confusion, reduce understandability

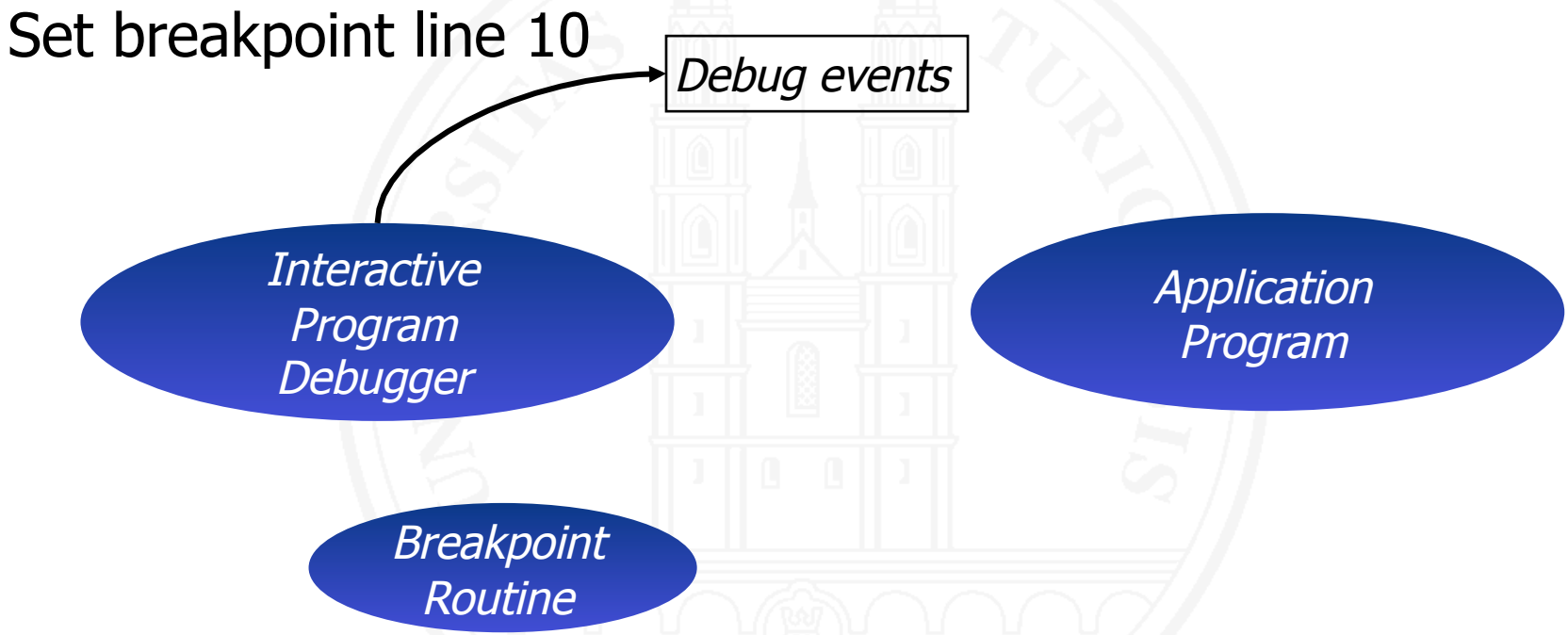
Event-Based Systems and Implicit Invocation Style

- Components
  - Programs or program entities that *announce* and/or *register interest in* events
- Connectors
  - Event broadcast and registration infrastructure
- Configurations
  - Implicit dependencies arising from event announcements and registrations
- Underlying computational model
  1. Event announcement is broadcast
  2. Procedures associated with registrations (if any) are invoked

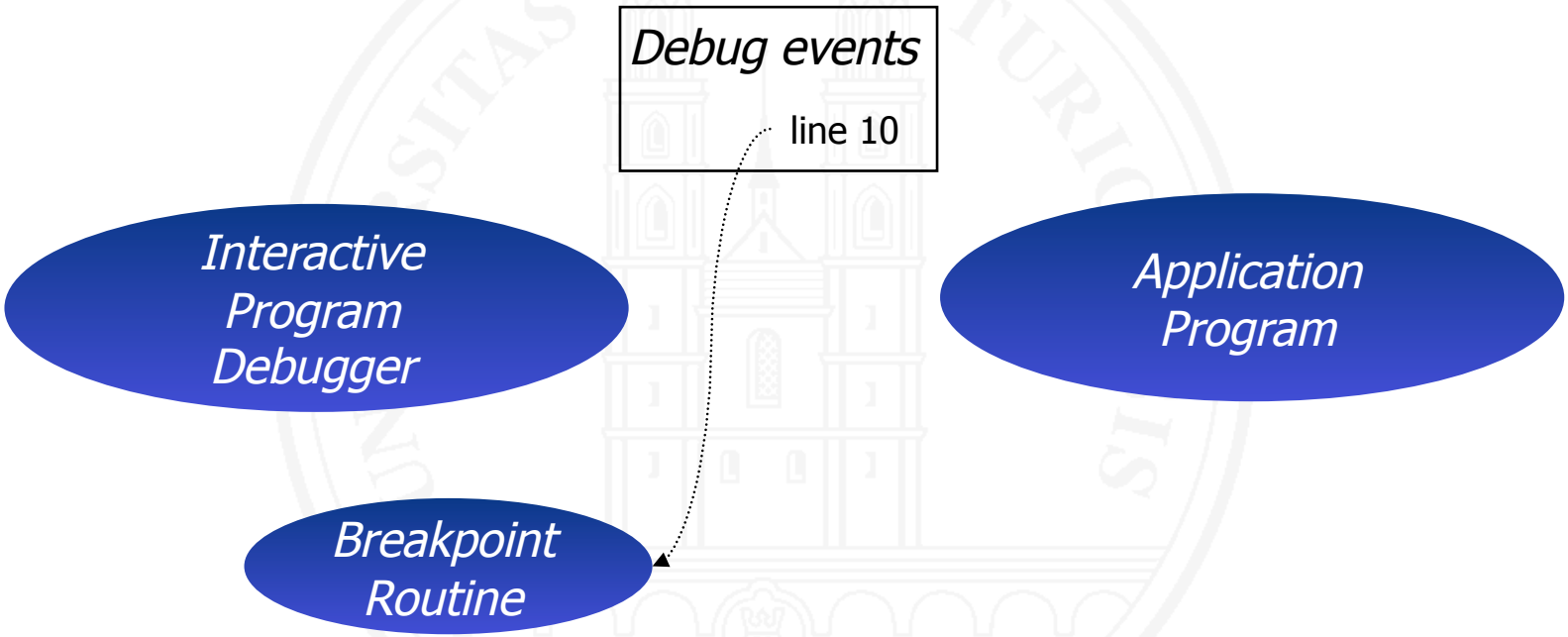
EBS Architecture



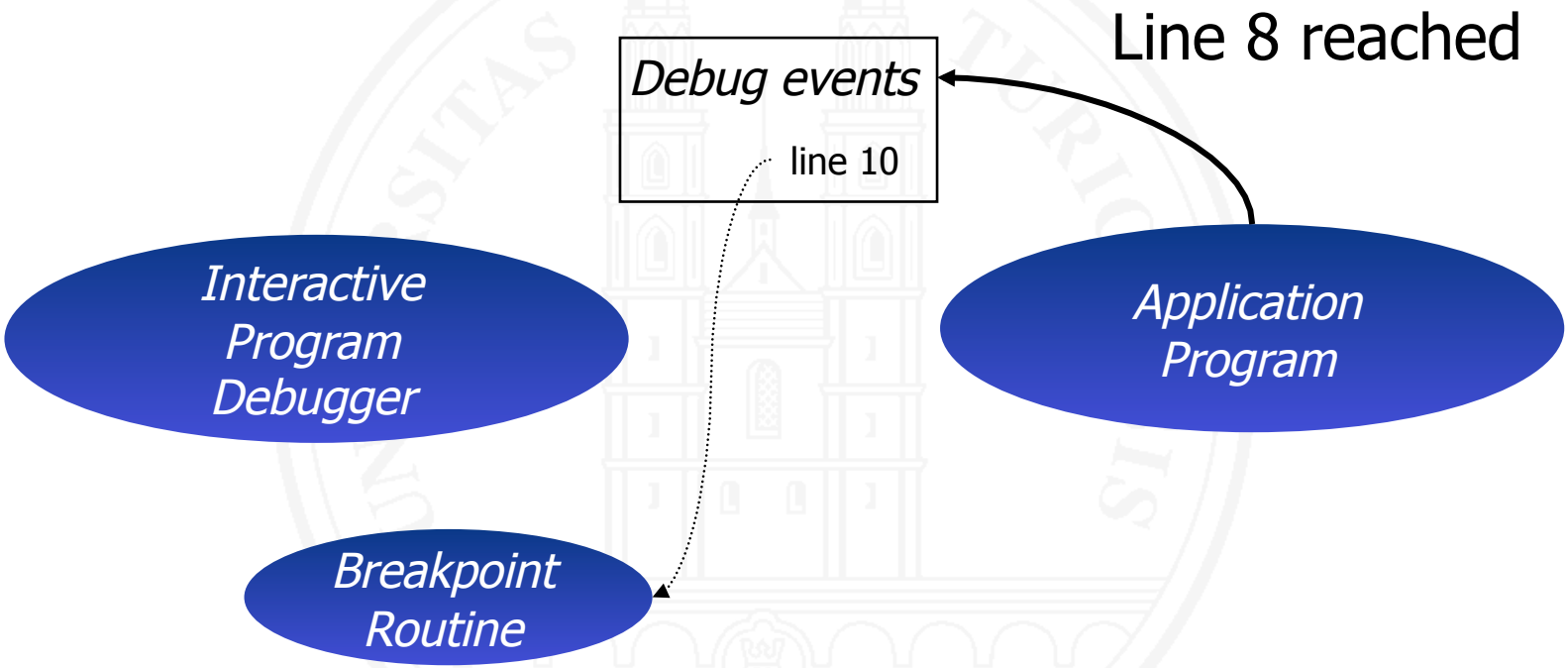
Implicit Invocation Example: Program Debugging (1)



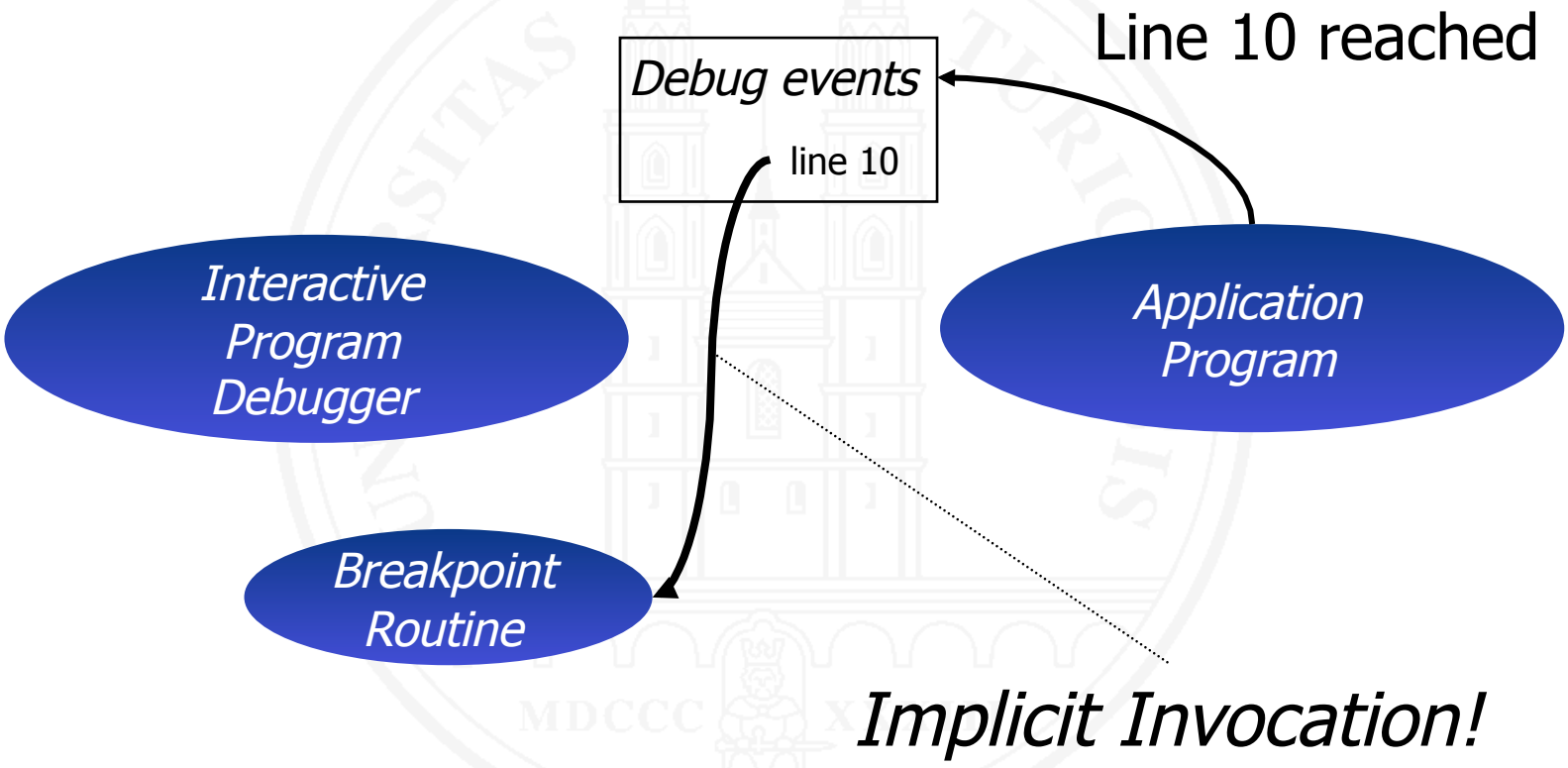
Implicit Invocation Example: Program Debugging (2)



Implicit Invocation Example: Program Debugging (3)



Implicit Invocation Example: Program Debugging (4)



Implicit Invocation: Advantages & Disadvantages

- Advantages
  - Reusability: Components can be put in almost any context
  - Distributability: Events are independent and can travel across the network
  - Interoperability: Components may be very heterogeneous
  - Visibility: Events are a reified form of communication that can be logged and viewed
  - Robustness: Components in this style generally have to be written to tolerate failure or unexpected circumstances well
- Disadvantages
  - Reliability: Components announcing events have no guarantee of getting a response
  - Simplicity: Components announcing events have no control over the order of responses, so they must be robust enough to handle this
  - Understandability: Difficult to reason about the behavior of an announcing component independently of the components that register for its events
  - Event abstraction does not cleanly lend itself to data exchange

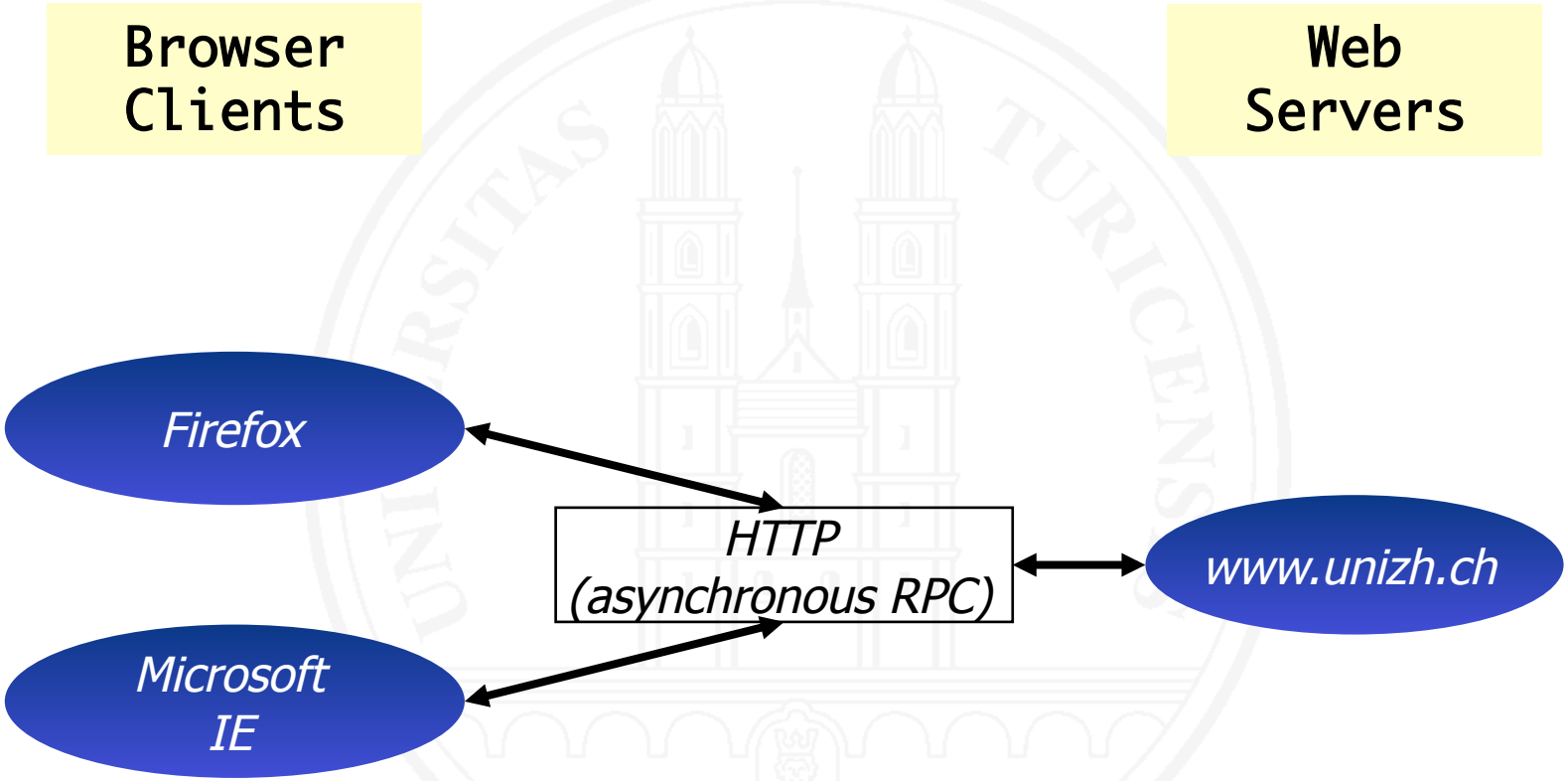
Distributed Peer-to-Peer Systems

- Components
  - Independently developed objects and programs offering public operations or services
- Connectors
  - Remote procedure call (RPC) over computer networks
- Configurations
  - Transient or persistent connections between cooperating components
- Underlying computational model
  - Synchronous or asynchronous invocation of operations or services
- Stylistic invariants
  - Communications are point-to-point

Client/Server Systems

- Client/Server systems are the most common specialization (restriction) of the peer-to-peer style
- One component is a server offering a service
- The other components are clients using the service
- Server implementation is transparent but can be centralized or distributed, single-threaded or multi-threaded
  - Single interface point with physically distributed implementation
  - Dynamic, transparent selection from among multiple interface points

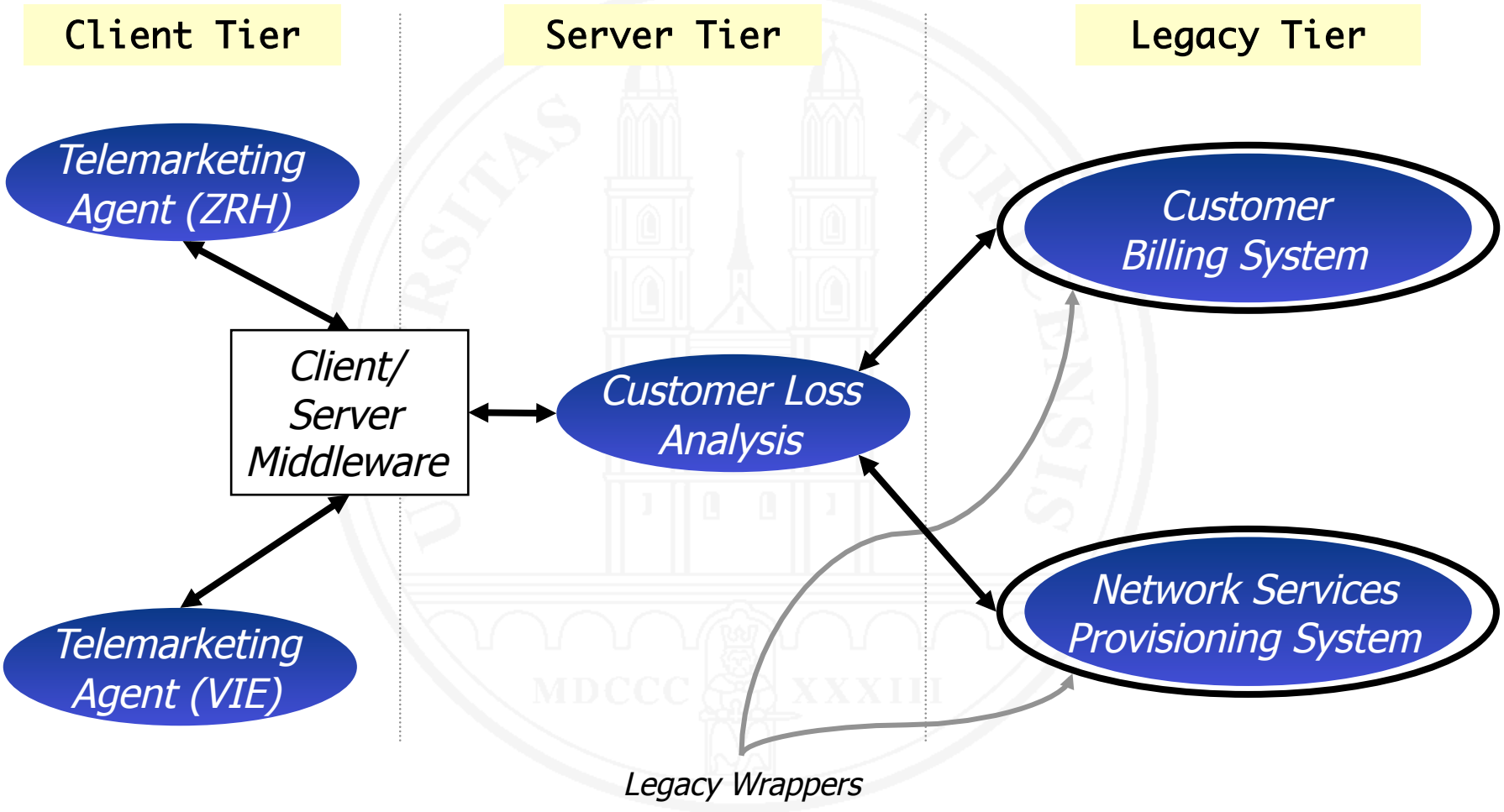
Client/Server Example: The World Wide Web



3-Tier Client/Server Systems

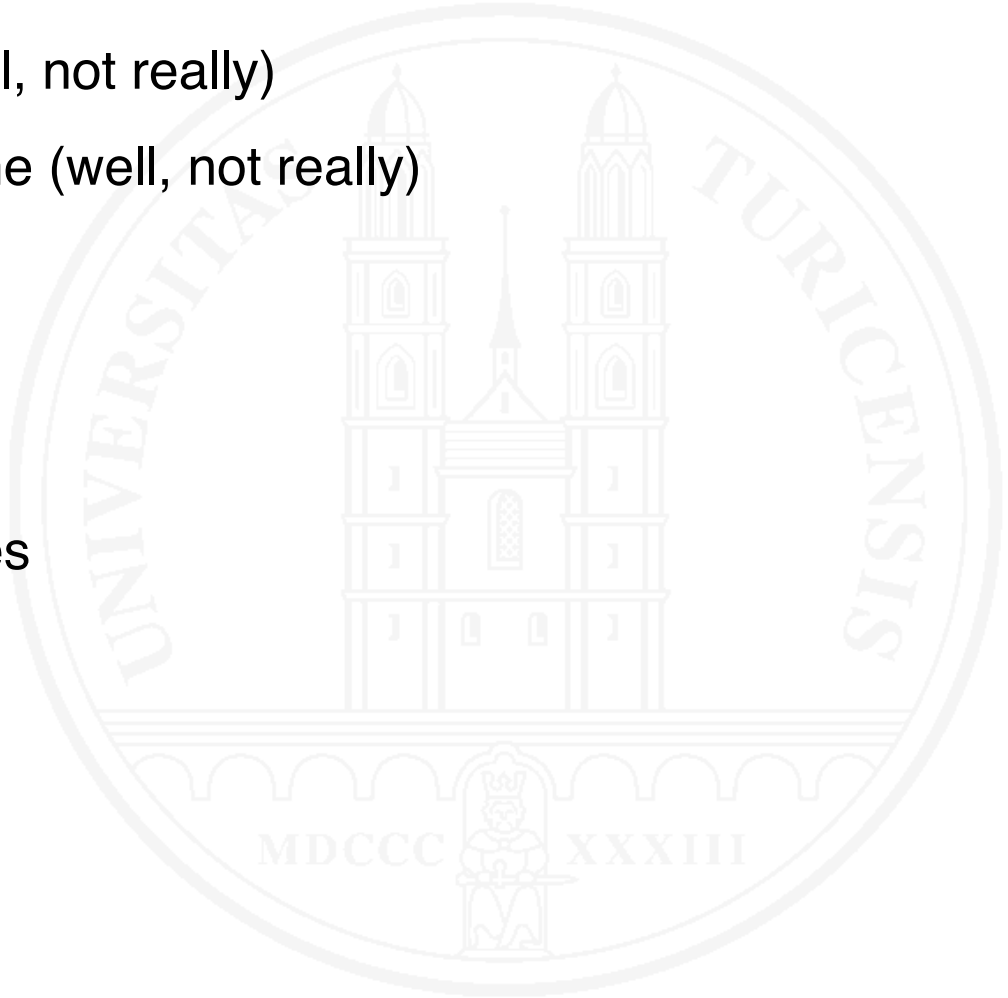
- 3-Tier Client/Server systems are a common class of distributed business systems
  - Increasing competition requires business to find new ways of exploiting legacy information assets
- *First tier:* Client (user interface) tier
- *Second (middle, "business logic") tier:* Servers acting as "business objects", encapsulating abstract, integrated models of multiple, disparate data sources
- *Third (back-end, database) tier:* Legacy business applications providing data services

3-Tier System Example: Marketing at a TelCo



The Real Peer-to-Peer Apps

- Napster (well, not really)
- SETI-at-home (well, not really)
- Gnutella
- Groove
- Magi
- Web Services



Peer-to-Peer: Advantages and Disadvantages

- Advantages
  - Interoperability: A natural high-level architectural style for heterogeneous distributed systems
  - Understandability: Small number of tiers, similar to layered-system properties
  - Reusability: Especially with regard to legacy applications
  - Scalability: Powerful enough server tiers can accommodate many clients
  - Distributability: Components communicate over a network, generally
- Disadvantages
  - Visibility, Maintainability: Difficult to analyze and debug
    - Distributed state
    - Potential for deadlock, starvation, race conditions, service outages
  - Simplicity: Require sophisticated interoperability mechanisms
    - Data marshalling and unmarshalling
    - Proxies and stubs for RPC
    - Legacy wrappers

5.1 Grundlagen und Prinzipien

5.2 Architekturentwurf

5.3 Architekturstile

5.4 Entwurfsmuster (design patterns)

5.5 Modulkonzepte und Schnittstellen

5.6 Vertragsorientierter Entwurf

5.7 Zusammenarbeit

Wiederverwendung von Entwurfswissen

Entwurfsmuster (design pattern) – Eine spezielle Komponente, die eine allgemeine, parametrierbare Lösung für ein typisches Entwurfsproblem bereitstellt.

In der Architektur von Softwaresystemen:

Architekturmuster – Eine vorgefertigte, parametrierbare Schablone für die Gestaltung der Architektur eines Systems oder einer Komponente.

Framework. Eine Menge kooperierender Programm-Module, die das Grundgerüst für die Lösung einer bestimmten Klasse von Problemen bilden.

Entwurfsmuster / Design Patterns

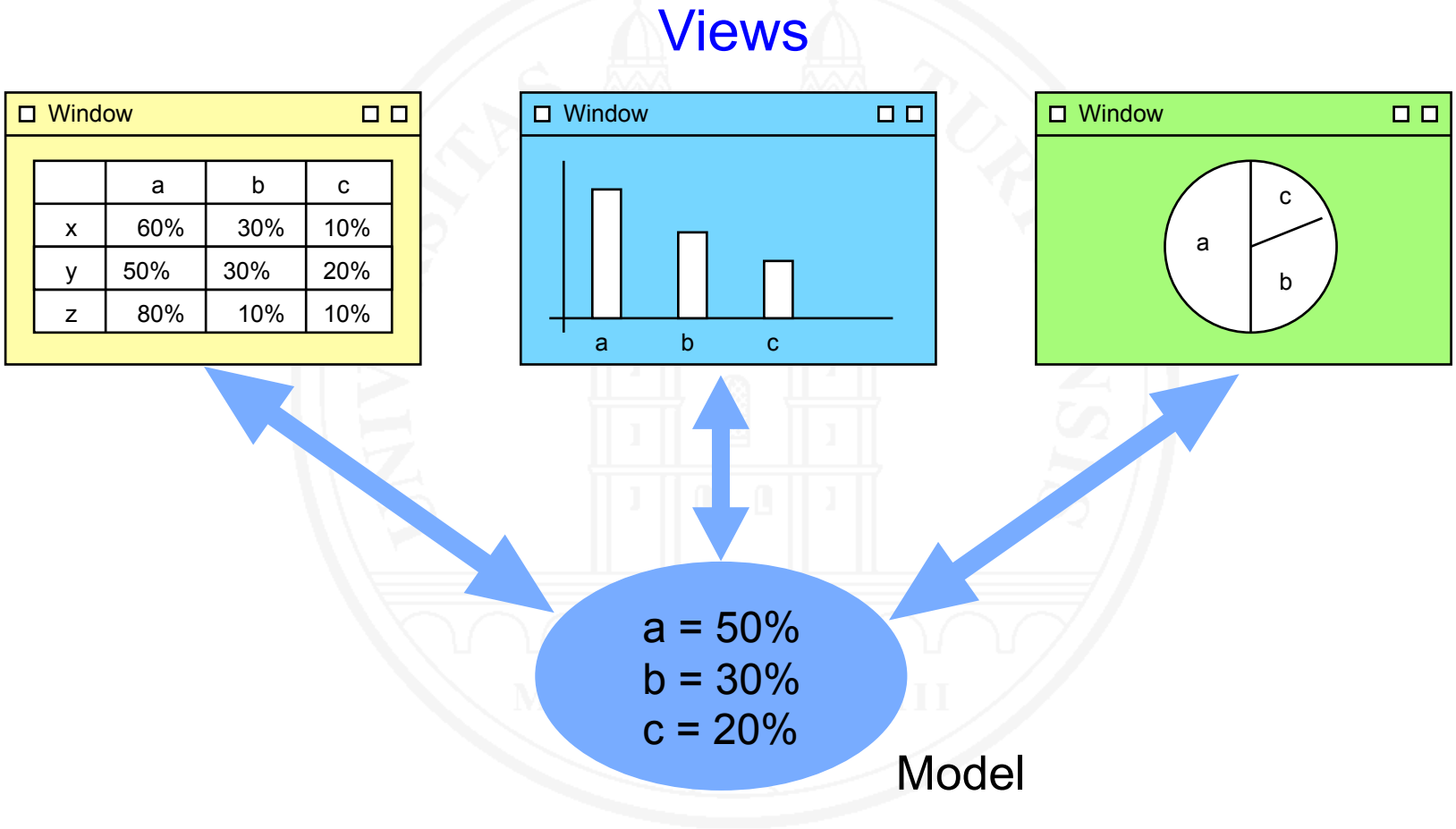
- Manche Entwurfsprobleme treten in sehr ähnlicher Form immer wieder auf ➪ Idee:
  - Problem nicht jedes Mal aufs Neue lösen,
  - ...sondern einmal eine vorgefertigte, parametrierbare Lösungs-schablone bereitstellen,
  - ...von der konkrete Lösungen rasch abgeleitet werden können

- Wiederverwendung von Entwurfswissen
- Begriffliche Basis für die Kommunikation unter den Beteiligten

Was ist ein Design Pattern (Entwurfsmuster)?

- Christopher Alexander (Architekt):
  - “Jedes Pattern beschreibt ein Problem, das immer wieder vorkommt und zeigt weiters den wesentlichen Teil einer Lösung für dieses Problem auf, in einer Weise, sodaß man die Lösung sehr oft wiederverwenden kann.”
- Im objektorientierten Design gibt es ebensolche Patterns für immer wiederkehrende Probleme
- Design Patterns nach Gamma, Helm, Johnsson, Vlissides [GHJV94]

Beispiel: Model/View/Controller (MVC)



Elemente eines Design Patterns

- Pattern Name
  - Design Vokabular
- Problem
  - Wann soll das Pattern verwendet werden?
  - Liste von Bedingungen
- Lösung
  - kein spezifisches Design oder Implementierung
  - Abstrakte Beschreibung mit einem vorgeschlagenen Verwendung von Objekten und Klassen
- Anwendung und Trade-Offs
  - Einfluss auf System-Flexibilität, Erweiterbarkeit, Portabilität, etc.

Design Pattern Beschreibung

- Pattern Name und Klassifikation
- Ziel
- weiterer Name
- Motivation
- Anwendbarkeit
- Struktur
- Beteiligte
- Zusammenwirken
- Konsequenzen
- Implementation
- Beispiel Code
- Bekannte Verwendungen
- Verwandte Patterns

Design Patterns Klassifikation (1)

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Design Patterns Klassifikation (2)

- Zweck
  - Creational: Objekt Erzeugung
  - Structural: Komposition von Klassen oder Objekten
  - Behavioral: Interaktion von Klassen oder Objekten
- Bereich
  - Klasse:
    - Relation zwischen Klassen und Subklassen
    - durch Vererbung, statisch, festgelegt zur Compile-Zeit
  - Objekt:
    - Objekt Relationen
    - dynamischer

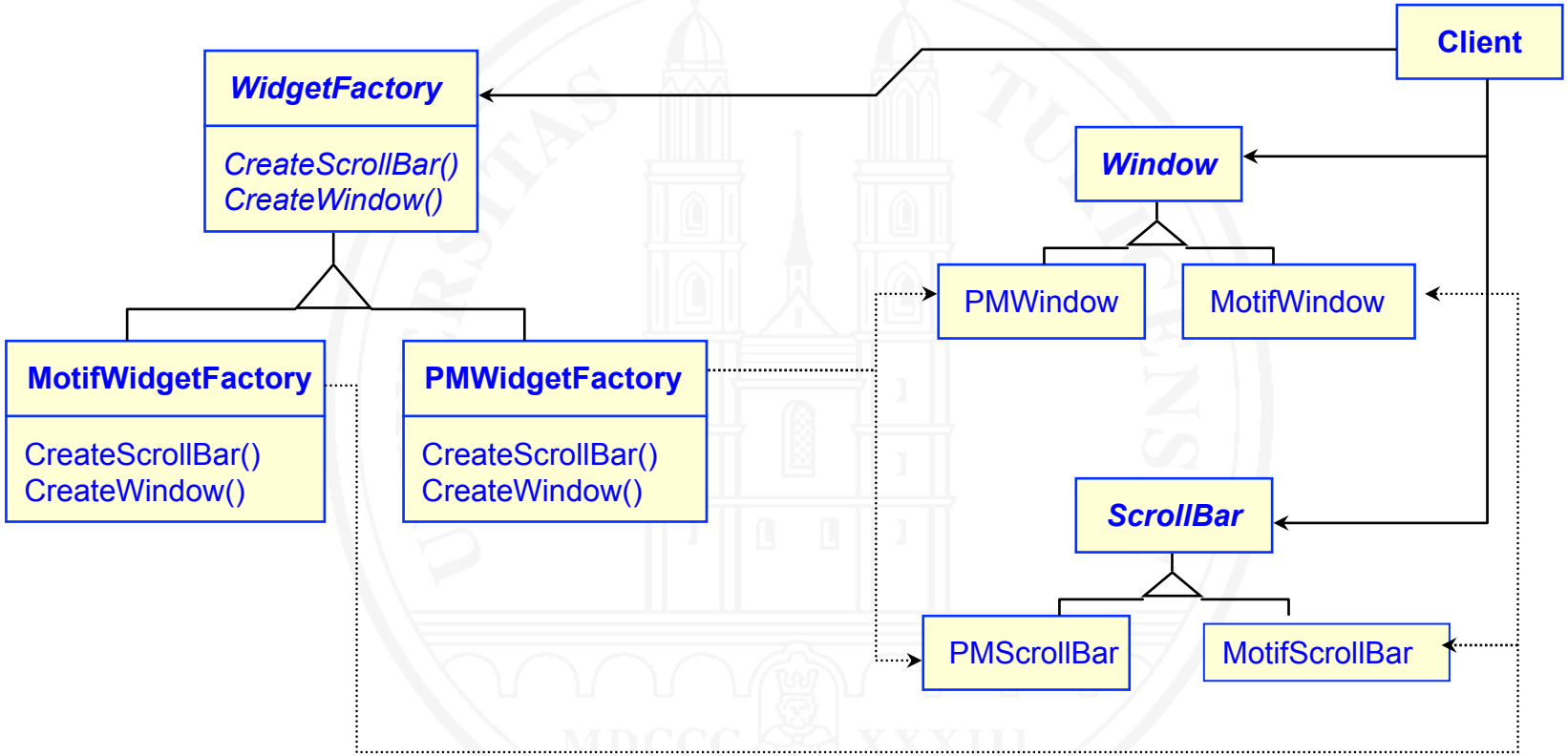
Design Patterns Klassifikation (3)

- Creational patterns
  - Klasse: aufschieben von Teilen der Objekt Erzeugung zu Subklassen
  - Objekt: aufschieben zu anderen Objekten
- Structural patterns
  - Klasse: verwenden von Vererbung zur Klassen-Komposition
  - Objekt: beschreiben Objekt-Komposition
- Behavioral patterns
  - Klasse: verwenden von Vererbung um Algorithmen und Kontrollfluss zu beschreiben
  - Objekt: beschreiben, wie eine Gruppen von Objekten kooperiert, um einen Task auszuführen, den kein einzelnes Objekt erbringen kann

Abstract Factory

- Intent
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- Applicability
  - a system should be independent of how its products are created, composed, and represented
  - a system should be configured with one of multiple families of products
  - provide a class library of products and reveal just their interfaces, not their implementations

Abstract Factory (2)



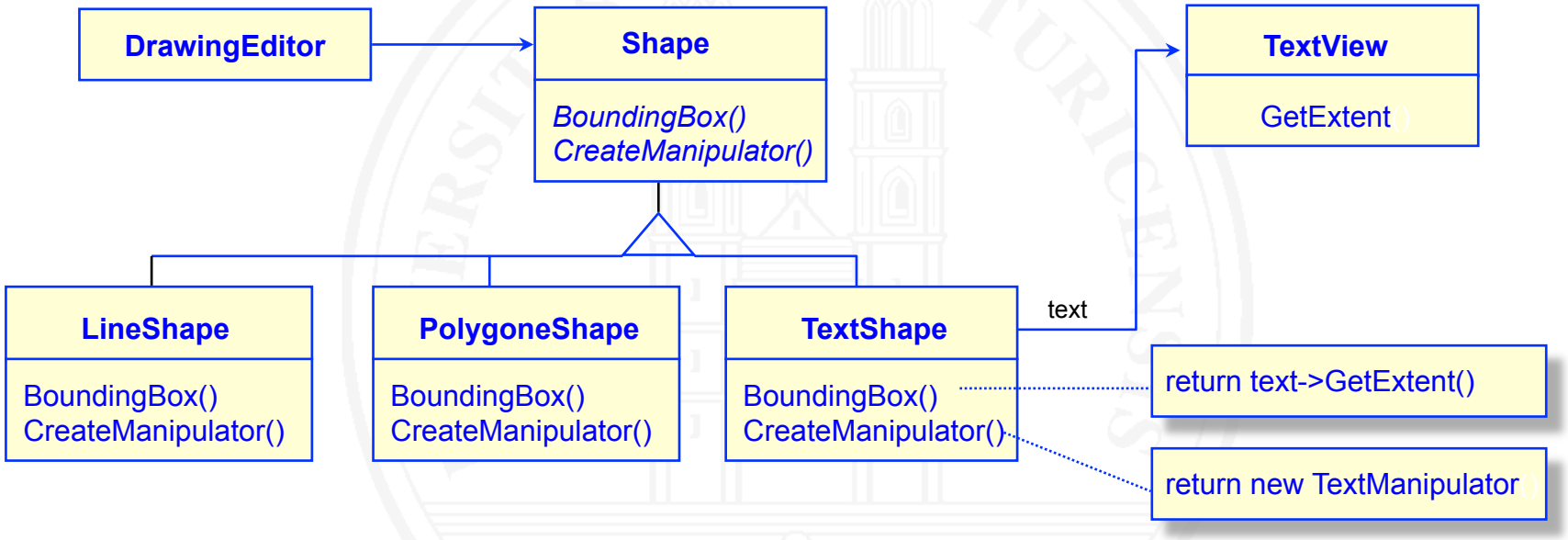
Abstract Factory (3)

- Consequences
  - It isolates concrete classes
  - It makes exchanging product families easy (different product configurations)
  - It promotes consistency among products
  - Supporting new kinds of products is difficult

Adapter

- Intent
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. (Wrapper)
- Applicability
  - use an existing class and its interface does not match
  - create a reusable class that cooperates with incompatible classes

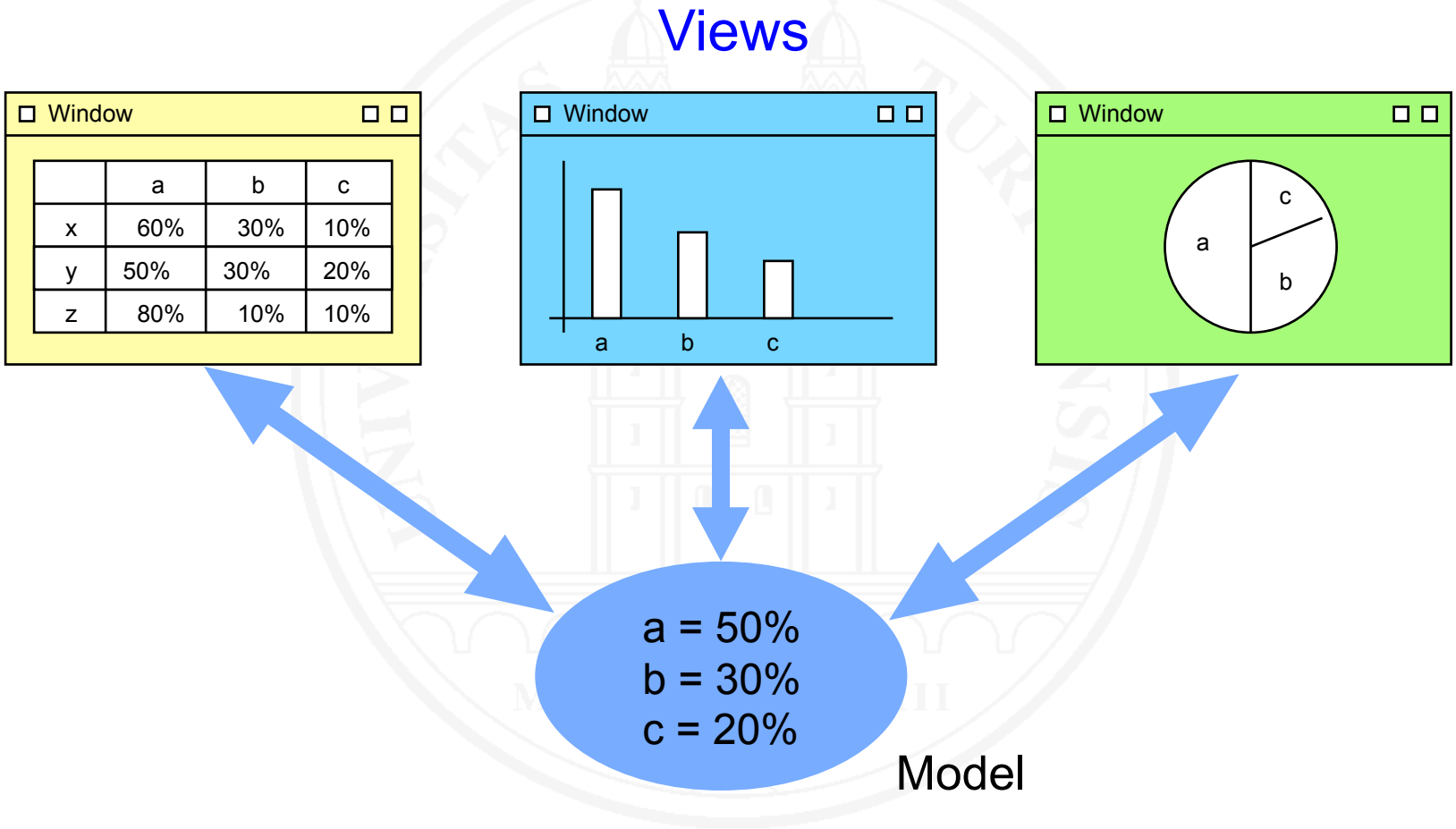
Adapter (2)


DrawingEditor
Shape
BoundingBox()
CreateManipulator()
TextView
GetExtent()
LineShape
BoundingBox()
CreateManipulator()
PolygoneShape
BoundingBox()
CreateManipulator()
TextShape
BoundingBox()
CreateManipulator()
text
return text->GetExtent()
return new TextManipulator()

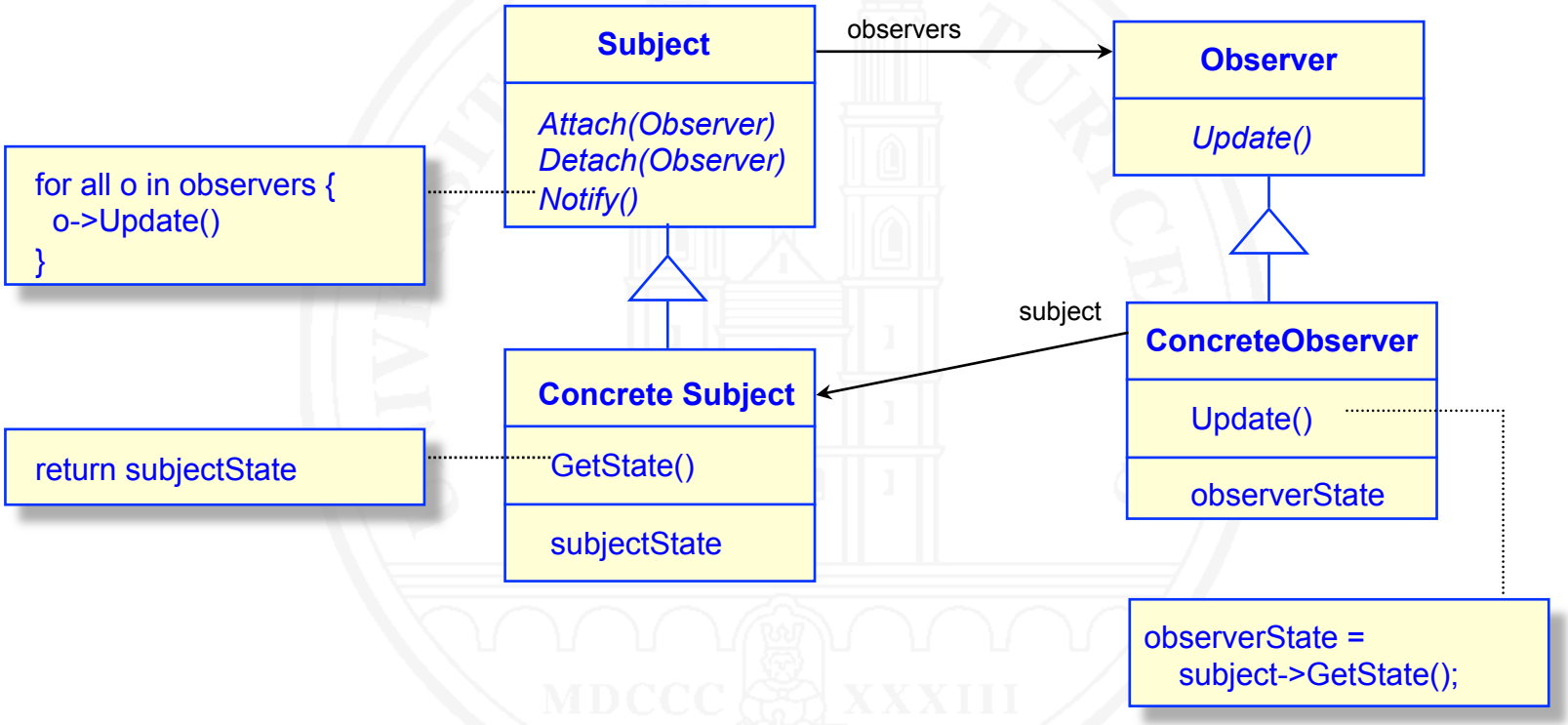
Observer

- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Applicability
  - when an abstraction has two aspects, one dependent on the other. Encapsulating these in separate objects lets you vary and reuse them independently
  - when a change to one object requires changing some others
  - when an object should be able to notify other objects without tighly coupling

Beispiel: Model/View/Controller (MVC)



Observer (2)


Subject
observers
Observer
Attach(Observer)
Detach(Observer)
Notify()
Update()
for all o in observers {
o->Update()
}
subject
ConcreteObserver
Concrete Subject
Update()
GetState()
observerState
return subjectState
subjectState
observerState =
subject->GetState();

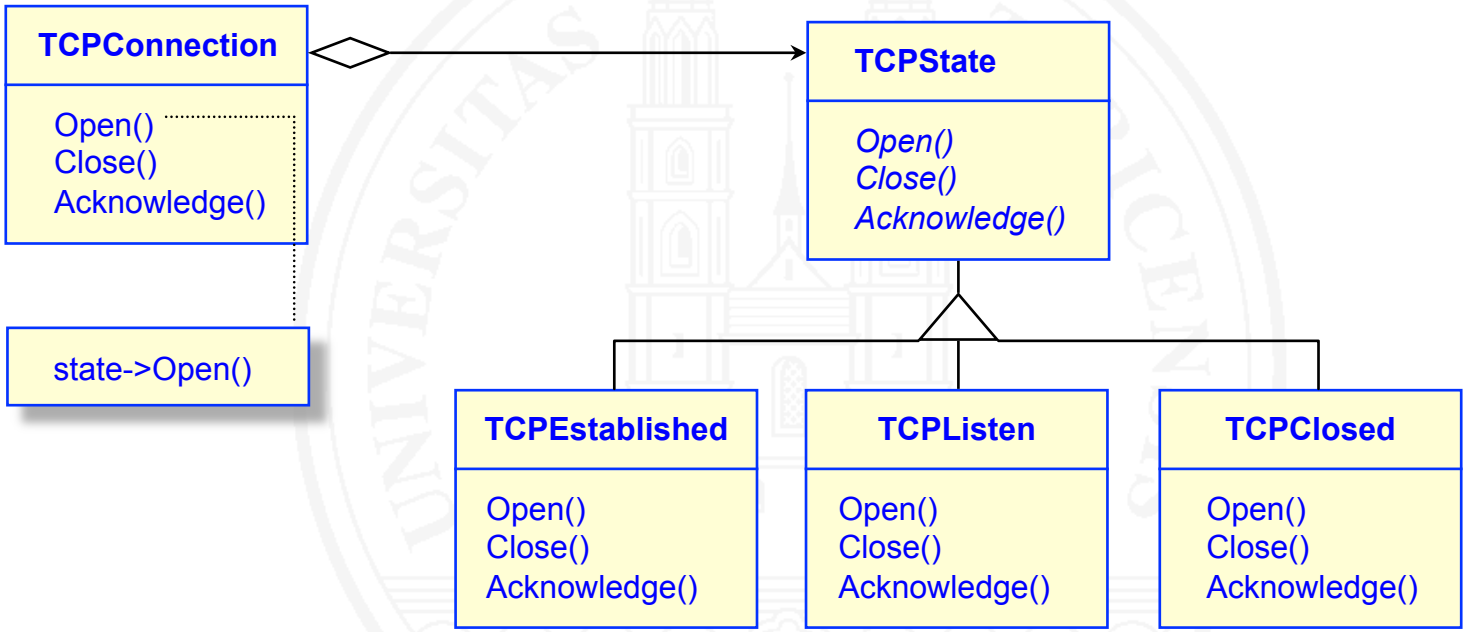
Observer (3)

- Consequences
  - Abstract coupling between Subject and Observer
  - Support for broadcast communication
  - Unexpected updates (observers have no knowledge about each others presence)

State

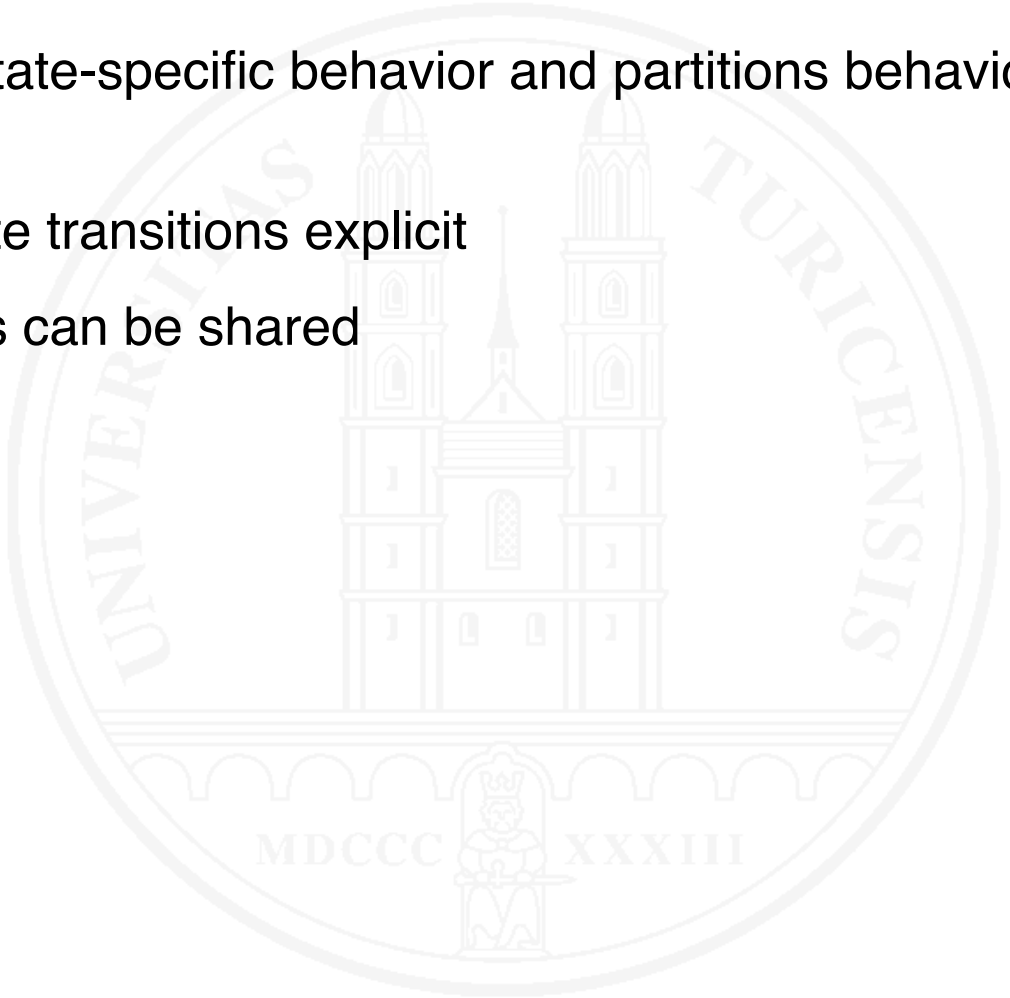
- Intent
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Applicability
  - An object's behavior depends on its state, and it must change its behavior at run-time depending on that state
  - Operations have large, multipart conditional statements that depend on the object's state. Often, several operations will contain this same conditional structure. The state pattern puts each in a separate class.

State (2)


TCPConnection
Open()
Close()
Acknowledge()
state->Open()
TCPState
Open()
Close()
Acknowledge()
TCPEstablished
Open()
Close()
Acknowledge()
TCPListen
Open()
Close()
Acknowledge()
TCPClosed
Open()
Close()
Acknowledge()

State (3)

- It localizes state-specific behavior and partitions behavior for different states
- It makes state transitions explicit
- State objects can be shared



Factory Method Pattern

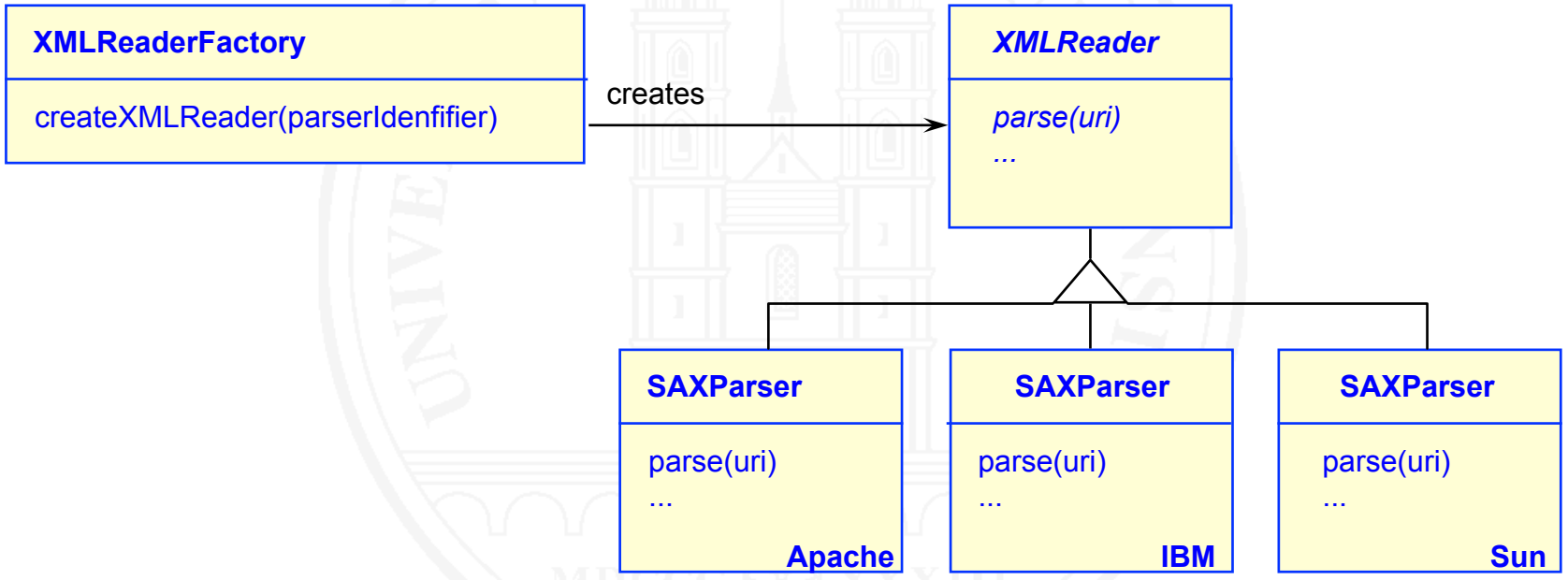
- Beispiel Programm: Parsen eines XML Dokuments

```
import org.xml.sax.XMLReader;
import org.apache.xerces.parsers.SAXParser;

public class Parser {
    public static void main (String[] args) {
            try {
       XMLReader parser = new SAXParser();
       parser.parse("test.xml");
            } catch (Exception e)
{e.printStackTrace();}
    }
}
```

- Problem:
  - Verwendeter XML Parser hard-coded
  - Für Framework ungeeignet, da Code meist nicht zugänglich

Factory Method Pattern (2)



Factory Method Pattern (3)

```
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

public class ParserFact {
   public static void main (String[] args) {
     try {
        XMLReader parser =
        XMLReaderFactory.createXMLReader(
           "org.apache.xerces.parsers.SAXParser");
        parser.parse("test.xml");
     } catch (Exception e) {e.printStackTrace();}
   }
}
```

Design Patterns versus Frameworks

- Framework = eine Menge kooperierender Klassen, die ein wiederverwendbares Design für eine spezifische Klasse von Software darstellen
- z.B. Graphische Editoren, Compiler, DB-Access etc.
- Design Patterns sind
  - abstrakter,
  - kleinere architekturelle Elemente, und
  - weniger spezialisiert als Frameworks

Zusammenfassung

- Vorteile von Design Patterns
  - Gemeinsames Design Vokabular
  - Erleichtern Verständnis von existierenden Systemen sowie deren Beschreibung
  - Objektorientiertes Design wird ergänzt
  - Flexibilität und Wiederverwendbarkeit der entwickelten Systeme

Module

Modul (module). Ein benannter, klar abgegrenzter Teil eines Systems.

(vgl. 5.1; Prinzip 2)

Beispiele für Module (auf verschiedenen Stufen)

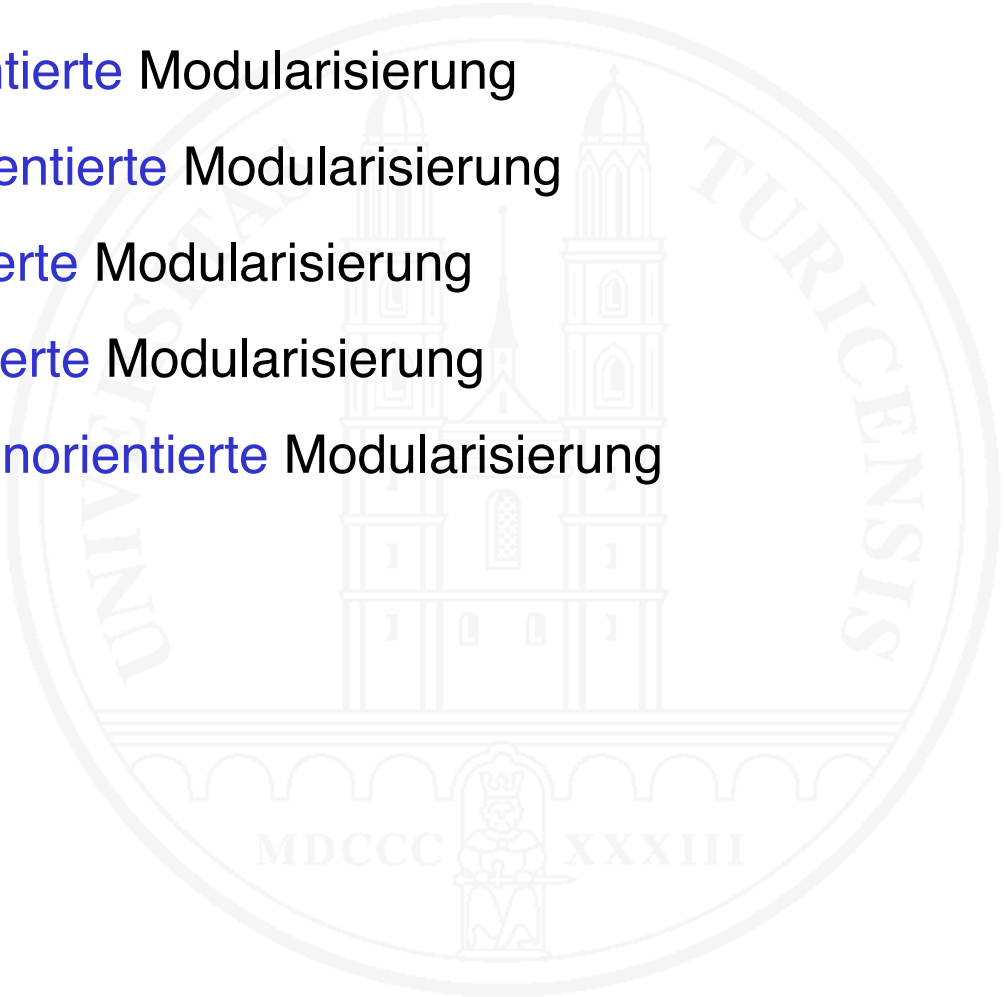
- Prozedur/Methode
- abstrakter Datentyp
- Klasse
- Komponente

Charakteristika

- Ein Modul bildet eine Einheit für
  - Verstehen
  - (Wieder-)Verwendung
  - Komposition / Dekomposition
- Die Verwendung eines Moduls erfordert keine Kenntnisse über seinen inneren Aufbau
- Ein Modul beschreibt sein Leistungsangebot für Dritte in Form einer Schnittstelle
- Ein Modul kann selbst Leistungen Dritter benötigen

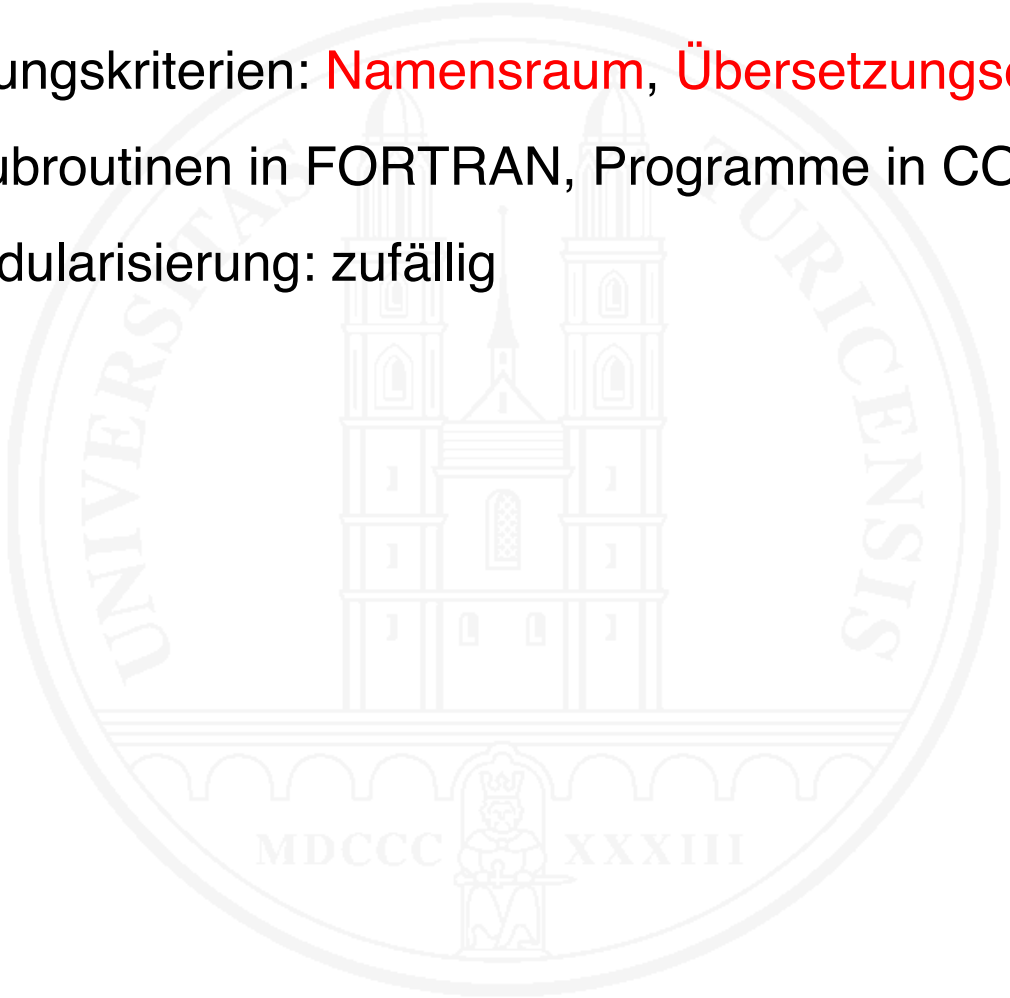
Modularisierungsarten

- Strukturorientierte Modularisierung
- Funktionsorientierte Modularisierung
- Datenorientierte Modularisierung
- Objektorientierte Modularisierung
- Komponentenorientierte Modularisierung



Strukturorientierte Modularisierung

- Modularisierungskriterien: Namensraum, Übersetzungseinheit
- Beispiele: Subroutinen in FORTRAN, Programme in COBOL
- Güte der Modularisierung: zufällig



Funktionsorientierte Modularisierung

- Modularisierungskriterium: Jedes Modul berechnet eine Funktion
- Beispiele:
  - Im Kleinen:
    - Strukturierung von Code in Funktionen und Prozeduren in der klassischen Programmierung
    - Strukturierung einer Klasse in Methoden in der objektorientierten Programmierung
  - Im Großen (Modularisierung ganzer Systeme): Structured Design (Stevens, Myers, Constantine 1974, Page-Jones 1988)
- Güte der Modularisierung:
  - gut für rein funktionale, zustandsfreie Probleme
  - gut zur Submodularisierung von Klassen und abstrakten Datentypen
  - sonst zu schwach

Funktionsorientierte Modularisierung – 2: Vorgehen

- Lange Programme in inhaltlich zusammenhängende Einheiten untergliedern
- Funktionen, die an verschiedenen Stellen eines Programms benötigt werden,
  - herauslösen und zu einem separaten Funktionsmodul machen (z.B. einer Methode in objektorientierten Sprachen)
  - Vorgang wird auch Faktorisierung (factoring) genannt
- Auf hohe Kohäsion und geringe Kopplung achten
- Wird eine bestehende Modularisierung restrukturiert, so spricht man auch von Refactoring

Mini-Übung 5.2

Beurteilen Sie Kohäsion und Kopplung:

a) Das Modul berechnet das Alter eines Mitarbeiters aus seinem Geburtsdatum. Zu diesem Zweck wird dem Modul der Mitarbeiter-Stammdatensatz (mit insgesamt rund 50 Feldern) zur Verfügung gestellt.

b) Das Modul druckt wahlweise die Wochenumsatzstatistik, die Monatsumsatzstatistik oder die Jahresumsatzstatistik. Die Auswahl wird über ein Flag gesteuert. Die Daten befinden sich in Dateien; der jeweilige Dateiname wird als Parameter übergeben.

c) Das Modul saldiert den Monatsumsatz, die Überzeitguthaben der Mitarbeitenden und die Zahl der beratenen Kunden.

Datenorientierte Modularisierung

- Modularisierungskriterium: Modul fasst eine Datenstruktur und alle darauf möglichen Operationen zusammen
- Beispiel: Abstrakter Datentyp (ADT)
- Güte der Modularisierung: gut
- Problem: ADT sind streng disjunkt; Gemeinsamkeiten im Leistungsangebot verschiedener ADT können nicht zusammengefasst werden
- Vorgehen:
  - Zusammengehörige Entwurfsentscheidungen identifizieren
  - und deren Umsetzung in je einem Modul kapseln
  - ⇨ Anwendung des Geheimnisprinzips

Objektorientierte Modularisierung

- Modularisierungskriterium: Modul repräsentiert Objekt des Problembereichs oder benötigtes Informatik-Element
- Beispiel: Klassen im objektorientierten Entwurf
- Güte der Modularisierung:
  - gut, wenn Klassen als ADT konzipiert werden.
  - Mäßig bis schlecht, wenn Klassen offen konzipiert werden
- Vorteil: Extrem flexibel und ausdrucksmächtig
- Vorgehen:
  - Anwendungsorientierte Module: Gegenstände der Anwendung modellieren und kapseln
  - Lösungsorientierte Module: Entwurfsentscheidungen kapseln

Komponentenorientierte Modularisierung

- Modularisierungskriterium: Jedes Modul ist eine stark gekapselte Menge zusammengehöriger Elemente, die eine gemeinsame Aufgabe lösen und als Einheit von Dritten verwendet werden
- Beispiel: Werkzeugsatz zur Bearbeitung von Verbunddokumenten
- Güte der Modularisierung: sehr gut
- Vorteil: Als in sich geschlossene Einheit verwendbar
- Problem: Weniger flexibel als Klassen, Verwendung in unbekanntem Kontext stellt sehr hohe Ansprüche an die Qualität der Schnittstellen wie der Implementierung
- Vorgehen: Systemteile, die als Einheit durch Dritte wiederverwendet werden können, zusammenfassen

Schnittstelle und Implementierung

- Verwendbarkeit
  - Die Schnittstelle eines Moduls muss nach außen sichtbar und dokumentiert sein
- Geschlossenheit
  - Das Modul ist ausschließlich über die Schnittstelle zugänglich
  - Die Implementierung des Moduls ist nach außen verborgen
- ⇨ Idealerweise sind Schnittstelle und Implementierung getrennt
- Prozeduren/Methoden: keine oder schwache Trennung
- Klassen in objektorientierten Programmiersprachen: dito

Trennung von Schnittstelle und Implementierung

- Beispiel Modula-2:
- DEFINITON MODULE (Schnittstelle)
- IMPLEMENTATION MODULE (Implementierung)
  - Syntaktisch getrennt
  - Aber noch eng gekoppelt: Zu jeder Schnittstelle genau eine Implementierung gleichen Namens
- Beispiel Java:
- **interface** abc ... (Schnittstelle)
- **class** xyz **implements** abc ... (Implementierung)
  - Schwach gekoppelt: mehrere Implementierungen zu einer Schnittstelle möglich
  - Eine Klasse kann gleichzeitig mehrere Schnittstellen implementieren

Spezifikation der Leistungen eines Moduls

Notwendiges Minimum: Namen/Signaturen der verwendbaren Operationen, Typen, Konstanten und ggf. Variablen, zum Beispiel (Wirth 1985):

```
DEFINITION MODULE InOut;
 ...
 CONST EOL = 15C;
 VAR Done: BOOLEAN;
 ...
 PROCEDURE ReadString (VAR s: ARRAY OF CHAR);
 PROCEDURE ReadInt (VAR x: INTEGER);
 ...
```

Besser: Zusätzlicher, erläuternder Kommentar:

```
PROCEDURE ReadString (VAR s: ARRAY OF CHAR);
(* Reads a text that is being typed on the keyboard into s
*)
```

Spezifikation der Leistungen eines Moduls – 2

Noch besser: Rigorose, teilformale oder formale Spezifikation der Schnittstelle

```
PROCEDURE ReadString (VAR s: ARRAY OF CHAR);
(* PRE: -
   POST: Let str be the string typed on the keyboard,
         terminated by a character <= " " (blank)
         and l = HIGH(s) + 1
         IF length (str) <= l
           THEN s = str (excluding the terminating character)
           ELSE s contains the first l characters of str;
                The remaining characters are ignored: they
                are neither read nor displayed
         END IF.
*)
```

Algebraische Spezifikation einer Schnittstelle

Beispiel: Formale algebraische Spezifikation der Schnittstelle für ein einfaches Konto in Java

```
interface EinfachesKonto
{
    public void Einzahlen (int betrag);
    public void Abheben (int betrag);
    public int Kontostand ();

   //      Axiome:
   //      ∀ k ∈  EinfachesKonto, b ∈ int
   //      (1)        new().Kontostand() = 0
   //      (2)        b ≥ 0 → k.Einzahlen(b).Abheben(b) = k
   //      (3)        b ≥ 0 → k.Einzahlen(b).Kontostand() = k.Kontostand() + b
   //      (4)        b ≥ 0 → k.Abheben(b).Kontostand() = k.Kontostand() - b
}
```

Signaturen: Syntax

Axiome: Semantik

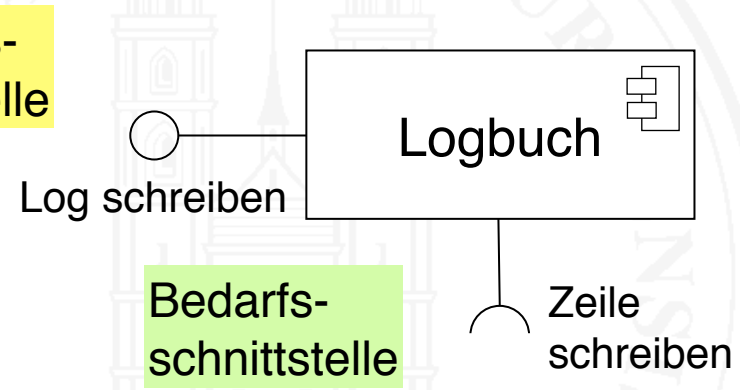
Angebots- und Bedarfsschnittstellen

Zwei Arten von Modulen:

- Dienstleistungsmodul
  - Stellt Leistungen für Dritte bereit
  - Leistungen in einer Angebotsschnittstelle definiert
  - Jede Implementierung der Komponente erbringt die angebotenen Leistungen vollständig selbst
  - Ausnahme: nutzt gegebenenfalls Leistungen des Betriebssystems
- Agentenmodul
  - Stellt Leistungen für Dritte bereit
  - Benötigt zur Erbringung dieser Leistungen die Leistungen von Drittkomponenten
  - Angebots- und Bedarfsschnittstellen erforderlich

Angebots- und Bedarfsschnittstellen – 2

Beispiel eines Agentenmoduls (notiert in UML 2):



Schnittstellenvererbung

- Problem:
  - Schaffung eines systematischen Zusammenhangs zwischen ähnlichen Schnittstellen
  - Wiederverwendung bestehender Schnittstellen

- Mittel: Vererbung (wie bei Klassen)

Beispiele – 1

- Von der Schnittstelle EinfachesKonto wird eine Schnittstelle Spielkasino abgeleitet, weil die Definition der Ein- und Auszahloperationen teilweise wiederverwendet werden kann
  - Die Schnittstellen haben keinen systematischen Zusammenhang
  - Finger weg von dieser Art von Wiederverwendung
  - Zu Grunde liegendes Prinzip: Steinbruch[1)]
- Von der Schnittstelle EinfachesKonto wird eine Schnittstelle Sparkonto abgeleitet.
  - Sparkonto hat «Saldo ≥ 0» als zusätzliche Invariante
  - Sparkonto ist ein Spezialfall von EinfachesKonto
  - Zu Grunde liegendes Prinzip: Spezialisierung

1) Als Bild ist hier nicht der Natursteinbruch gemeint, sondern der Antikensteinbruch, d.h. die früher übliche Wiederverwendung von Steinen aus verfallenen Bauten der Antike

Beispiele – 2

- Aber: Korrekte Implementierungen von Sparkonto sind keine korrekten Implementierungen von EinfachesKonto

Warum ist das so?

- Die Implementierungen sind nicht substituierbar

- Von der Schnittstelle EinfachesKonto wird eine Schnittstelle KontoMitVerbuchung abgeleitet, welche bei jeder Mutation von Saldo zusätzlich die Verbuchung dieser Mutation zusichert
  - KontoMitVerbuchung ist eine echte Subschnittstelle von EinfachesKonto: Jede korrekte Implementierung von KontoMitVerbuchung ist gleichzeitig auch eine korrekte Implementierung von EinfachesKonto
  - Zu Grunde liegendes Prinzip: Substituierbarkeit

Arten der Vererbung

Das Prinzip der Vererbung kann in gleicher Weise wie auf Klassen auch auf Schnittstellen angewendet werden:

- **Steinbruch**: Die Vererbung dient nur dazu, Schreibaufwand zu sparen, indem Teile einer bestehenden Schnittstelle übernommen werden. Im übrigen wird beliebig ergänzt und abgeändert.
- **Spezialisierung**: Sei S' eine Subschnittstelle von S. Die von S' offerierten Leistungen sind ein Spezialfall der von S offerierten Leistungen.
- **Substituierbarkeit**: Sei S' eine Subschnittstelle von S. Jede korrekte Implementierung von S' ist gleichzeitig auch eine korrekte Implementierung von S. Dementsprechend ist jede Implementierung von S durch eine beliebige (korrekte) Implementierung von S' ersetzbar.

Schnittstellendefinition mit Verträgen

- Schnittstelle ist Vertrag zwischen Modul und Modulverwender
- Beschreibung des Vertrags mit Zusicherungen (assertions)
- Vier Arten von Zusicherungen
  - Voraussetzungen (preconditions, requirement, Schlüsselworte: pre, require)
  - Ergebniszusicherungen (postconditions, Schlüsselworte: post, ensure)
  - Invarianten (invariants)
  - Verpflichtungen (obligations)
- "Design by Contract" (Meyer 1988, 1992)

Vertragserfüllung

Vertragserfüllung bedeutet:

- Verwender muss
  - Voraussetzungen erfüllen
  - Übernommene Verpflichtungen einhalten
- Modul muss
  - Ergebniszusicherungen erfüllen
  - Invarianten garantieren
  - aber unter der Annahme der Vertragstreue des Modulverwenders!

Schnittstellendefinition mit Verträgen – Eigenschaften

- Leichter lesbar als algebraische Spezifikation
- Präziser und eindeutiger als einfacher Kommentartext
- Voraussetzungen und Resultate klar formulierbar
- Benötigt in der Regel Zustandsvariablen
  - Gefahr implementierungsabhängiger Spezifikationen
  - ⇨ Nur solche Zustandsvariablen verwenden, welche eine Entsprechung im Problembereich / der Anwendungsdomäne haben

Sprache für die Formulierung von Verträgen

- Natürliche Sprache ist nur beschränkt geeignet
  - mehrdeutig
  - unpräzise
- Rein formale Sprache häufig zu wenig verständlich
- Besser: Teilformale, deklarative Sprache, basierend auf
  - Prädikaten – soweit möglich
  - Fallunterscheidungen
  - Natürlicher Sprache – wo nötig
  - (möglichst wenig) Zustandsvariablen

Beispiel: Ein einfaches Konto

```
interface EinfachesKonto
{
    // public EinfachesKonto ();
    // PRE –
    // POST int saldo = 0
    public void Einzahlen (int betrag);
    // PRE betrag ≥ 0
    // POST saldo = saldo@PRE + betrag
    public void Abheben (int betrag);
    // PRE betrag ≥ 0
    // POST saldo = saldo@PRE - betrag
    public int Kontostand ();
    // PRE –
    // POST result = saldo and saldo = saldo@PRE
}
```

Syntaktisch Kommentar, da es in Java-Schnittstellen keine Konstruktoren gibt

Erforderlich, damit der Anfangszustand von saldo spezifizierbar ist

Wert einer Zustands-variable bei Prüfung der Voraussetzung

Mathematische Gleich-heit, *keine* Zuweisung

Voraussetzungen und Ergebniszusicherungen

- Spezifizieren die Wirkung einer Operation / Methode einer Schnittstelle
- Voraussetzungen
  - Müssen zum Zeitpunkt des Aufrufs durch den Aufrufer erfüllt sein
  - Werden von der Implementierung der Schnittstelle nicht geprüft
- Ergebniszusicherungen
  - Beschreiben die Effekte der Operation
  - Müssen von jeder Implementierung der Schnittstelle erfüllt werden
  - Aber unter der Annahme, dass die Voraussetzungen erfüllt sind

Mini-Übung 5.3

Benötigt wird ein dreistelliger Dezimalzähler, der von 0 bis 999 hochzählt und dann wieder bei Null beginnt. Es werden drei Operationen benötigt: Reset, Increment und Display

Entwerfen Sie eine Schnittstelle Zähler mit diesen drei Operationen, und zwar

a) mit vertragsorientiertem Entwurf

b) mit algebraischer Spezifikation

Invarianten

- Objekte haben Eigenschaften, die nicht verändert werden dürfen
  - Beispiel: ein Quadrat hat vier gleiche Seiten und ist rechtwinklig
- Wenn eine Methode eine Zustandsvariable nicht verändert, so muss dies explizit zugesichert werden
  - Beispiel: POST result = saldo **and** saldo = saldo@PRE
- Operationen / Methoden hängen zusammen
  - Beispiel: (konto.Einzahlen(n)).Abheben(n) = konto
- Invarianten lösen diese Probleme
  - Beschreiben Eigenschaften der Schnittstelle, die unter allen Operationen invariant sind
  - Entlasten die Ergebniszusicherungen der Operationen
  - Spezifizieren Zusammenhänge zwischen Operationen

Beispiel einer Invariante

```
interface EinfachesKonto
{
// INVARIANT with e = Summe aller mit Einzahlen eingezahlten Beträge,
//                a = Summe  aller mit Abheben abgehobenen Beträge
//                holds saldo = e - a
...
```

- Garantiert, dass der Saldo nur durch Einzahlen und Abheben verändert wird
- Ermöglicht, die Bedingung saldo = saldo@PRE in der Ergebnis-zusicherung von Kontostand wegzulassen

➪ Eine Invariante bezieht sich immer auf die ganze Schnittstelle, nicht auf eine einzelne Operation / Methode

Verpflichtungen

- „Wer A sagt, muss auch B sagen“ (Volksweisheit)
- Mit dem Aufruf einer Operation / Methode übernimmt der Aufrufer häufig Pflichten, zum Beispiel
  - Aufräum- oder Terminierungsoperationen aufzurufen
  - Ein Protokoll von Aufrufen einzuhalten
- Verpflichtungen
  - Spezifizieren Pflichten, die der Aufrufer mit dem Aufruf einer Operationen übernimmt
  - Brauchen in der Darstellung oft temporale Logik

Beispiel: Einfaches Sperrprotokoll

```
// Wer eine Sperre setzt, muss sie später auch wieder freigeben
interface EinfacheSperre
{
  //public EinfacheSperre ();
  //PRE –
  //POST boolean gesperrt = false
  public boolean Sperren ();
  //PRE –
  //POST if gesperrt@PRE then result = false
  //              else gesperrt and result = true endif
  //OBLIGATION sometimes Freigeben()
  public void Freigeben ();
  //PRE –
  //POST gesperrt = false
}
```

Mini-Übung 5.4

Beurteilen Sie die Qualität der folgenden beiden Entwurfsfragmente:

double Sqrt (**double** x);
// PRE –
// POST **if** $x \geq 0$ **then for all** $\varepsilon < 10^{-12}$ **and** $\varepsilon < 10^{-6}x$ $|(Sqrt(x))^2 - x| < \varepsilon$
// **else** Fehlermeldung "x ist negativ" **endif**

double Kalibrieren (**double** a, **double** b);
// Die Korrekturfaktoren a und b werden vom Benutzer eingegeben
// PRE $a \neq 0$
// POST **result** = (this/a)*Fkorr(b)

Was, wann und wo prüfen?

- Vertragsorientierter Entwurf: Voraussetzungen werden nicht geprüft

  Metapher: Vertragstreue Partner
- Vorteil: klare Verantwortlichkeiten, schlanker Code
- Problem: Woher weiß ich, ob mein Partner vertragstreu ist?

  ➪ Gegebenenfalls Zusicherungen dynamisch prüfen
- Defensives Programmieren: Prüfe, was immer du kannst

  Metapher: “Designed for the unexpected”
- Vorteil: Mehr Sicherheit
- Problem: redundante Mehrfachprüfungen
  - blähen den Code auf
  - behindern die Lesbarkeit des Codes

Gefährlich: Implizite Voraussetzungen

- Eine Operation / Methode macht faktisch Voraussetzungen
- Die Voraussetzungen werden weder geprüft noch sind sie dokumentiert
- Der Aufrufer muss entweder die Implementierung kennen oder durch Experimente herausfinden
- Standardvorgehen bei C-Bibliotheken
- Bevorzugte Angriffsstelle für Hacker (Pufferüberlauf-Angriffe)

Prüfregeln für vertragsorientierten Entwurf

- Voraussetzen immer dann, wenn dem Aufrufer die Erfüllung der Voraussetzungen zugemutet werden kann
- Prüfen immer dann, wenn mit Falscheingaben gerechnet werden muss (zum Beispiel bei Benutzereingaben)
- Prüfen nur, wenn eine sinnvolle Behandlung von Fehlern möglich ist
- Bewusste Entwurfsentscheidungen treffen ➔ Nicht dem Geschmack der Programmierer überlassen

Prüfen der Ergebnisse

- Voraussetzungen werden nicht geprüft
- Leistungserbringer prüft Ergebnisse und erzeugt bei falschen oder unzulässigen Ergebnissen Fehlerbedingungen
- Leistungserbringer behandelt den Fehler nicht, sondern gibt nur Fehlerbedingungen an Aufrufer zurück
- Aufrufer interpretiert Fehlerbedingungen und handelt danach
- Nachteil: Umständlich, erschwert Lesbarkeit des Codes des Aufrufers
- Vorteil: Der Aufrufer kennt den Kontext besser:
  bessere Fehlermeldung möglich
- Aber: wenn schon, dann besser mit Ausnahmebehandlung lösen (siehe unten)

Beispiel für Ergebnisprüfung

```
public abstract class EinfachesKonto
{
    public boolean ok;   // Falsch nach Aufrufen mit ungültigem Ergebnis
    ...
    public abstract void Einzahlen (int betrag);
    // PRE –
    // POST   if betrag ≥ 0 then (saldo = saldo@PRE + betrag) and ok
    //              else (saldo = saldo@PRE) and not ok endif
    ...
}
```

Für den Aufrufer bedeutet das Konstruktionen der Art:

```
...
k.Einzahlen (betrag);
if (!k.ok) ...        // Fehler behandeln
```

Ausnahmebehandlung

- Voraussetzungen werden nicht geprüft
- Leistungserbringer prüft Ergebnisse und erzeugt bei falschen oder unzulässigen Ergebnissen Ausnahmen (exceptions)
- Laufzeitsystem übergibt Steuerung an Ausnahmebehandler des Aufrufers
- Falls kein Behandler vorhanden, wird Ausnahme in der Aufrufhierarchie hochgereicht
- Behandler behandelt Ausnahmen
  - Fehlermeldungen
  - Abbruch oder geordnete Rückkehr in den Programmablauf
- Nachteil: Nicht in allen Programmiersprachen verfügbar
- Vorteil: Code für Normal- und Ausnahmesituationen sauber trennbar
  Keine Variablen zur Weitergabe von Prüfergebnissen nötig

Ausnahmebehandlung, Beispiel

```
public void Einzahlen (int betrag) throws BetragNegativ;

    // PRE –
    // POST  if betrag ≥ 0 then (saldo = saldo@PRE + betrag)
    //       else (saldo = saldo@PRE) and exception BetragNegativ
    //       endif
```

Mini-Übung 5.5

In Mini-Übung 5.4 haben wir diskutiert, warum die Verträge der folgenden beiden Methoden schlecht bzw. gefährlich sind. Entwerfen Sie bessere Verträge.

double Sqrt (**double** x);
// PRE –
// POST **if** $x \geq 0$ **then for all** $\varepsilon < 10^{-12}$ **and** $\varepsilon < 10^{-6}x$ $|(Sqrt(x))^2 - x| < \varepsilon$
// **else** Fehlermeldung "x ist negativ" **endif**

double Kalibrieren (**double** a, **double** b);
// Die Korrekturfaktoren a und b werden vom Benutzer eingegeben
// PRE $a \neq 0$
// POST **result** = (this/a)*Fkorr(b)

Dynamische Prüfung von Zusicherungen

- Geeignet formulierte Zusicherungen in Programmen sind maschinell prüfbar
- Mächtiges Mittel zur dynamischen Prüfung von Programmen
  - Als eigenständiges Prüfverfahren
  - Zur Lokalisierung von Defekten bei der Behebung von beim Testen festgestellten Fehlern (Debugging)
- In manchen Programmiersprachen (z. B. Eiffel) direkt programmierbar
- Sonst mit Hilfskonstrukten zu programmieren, z. B. in Java bis Version 1.3 mit Ausnahmen
- Java 1.4 bietet neu einen primitiven, auf Ausnahmebehandlung basierenden Zusicherungsmechanismus als Bestandteil der Sprache

Dynamische Prüfung in Eiffel

- Hochzähloperation für einen Zähler mit oberer Grenze

```
-- upper: Obere Grenze
-- count: aktueller Zählerwert

add(n: INTEGER) is
  require
    (n > 0) and (count + n <= upper)
  do
    count := count + n
  ensure
    count = old count + n
  end -- add
```

Dynamische Prüfung in Java 1.4 – 1

```
class BoundedCounter {
private int count, lower, upper;
…                                   // add constructor method here
public void add(int n) {
   // assert precondition
   assert (n > 0) && (count + n <= upper) :
       "precondition violated:  n: " + n + " count: " + count
       + " upper: " + upper;

   // Inner class that saves state to verify postcondition
   class DataCopy {
       private int countCopy;
       DataCopy(int value) {countCopy = value; }
       int countAtPRE() { return countCopy; }
   }
   DataCopy copy = new DataCopy(count);
   // Creates an object that saves the value of count@PRE
```

Dynamische Prüfung in Java 1.4 – 2

```
   // productive code
   count = count + n;

   // assert postcondition
   assert count == copy.countAtPRE() + n :
       "postcondition violated:  count: " + count +
       " count@PRE: "  + copy.countAtPRE() + " n: " + n;
 }
…

// assert invariant
assert count >= lower && count <= upper :
       "invariat violated:  count: " + count + " lower: " +
        lower + " upper: " + upper;
}
```

Dynamische Prüfung in Java 1.4 – 3

```
//Main program for testing
public static void main (String[] args) {
   BoundedCounter bc = new BoundedCounter(0, 0, 100);
   bc.add(25);
   bc.add(50);
   bc.add(33);
}
```

Ausführungsprotokoll:

```
$ javac -source 1.4 BounderCounter.java
$ java BoundedCounter
$ java -ea BoundedCounter
Exception in thread "main" java.lang.AssertionError: precondition violated:
    n: 33 count: 75 upper: 100
       at BoundedCounter.add(BounderCounter.java:20)
       at BoundedCounter.main(BounderCounter.java:53)
```

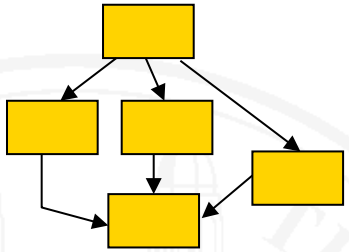
Verträge für Bedarfsschnittstellen

Vertrag ist invers zu Verträgen für Angebotsschnittstellen

- Für jede benötigte Operation sind zu spezifizieren
  - Die Voraussetzungen, welche die benötigte externe Operation höchstens machen darf
  - Die Ergebniszusicherungen, welche die benötigte Operation mindestens machen muss
- Notwendige Invarianten: Eigenschaften, welche jede Implementierung der benötigten Komponente unverändert lassen muss

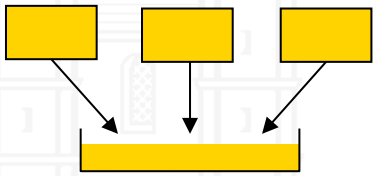
Formen der Zusammenarbeit

- Leistungserbringung



- Informationsaustausch

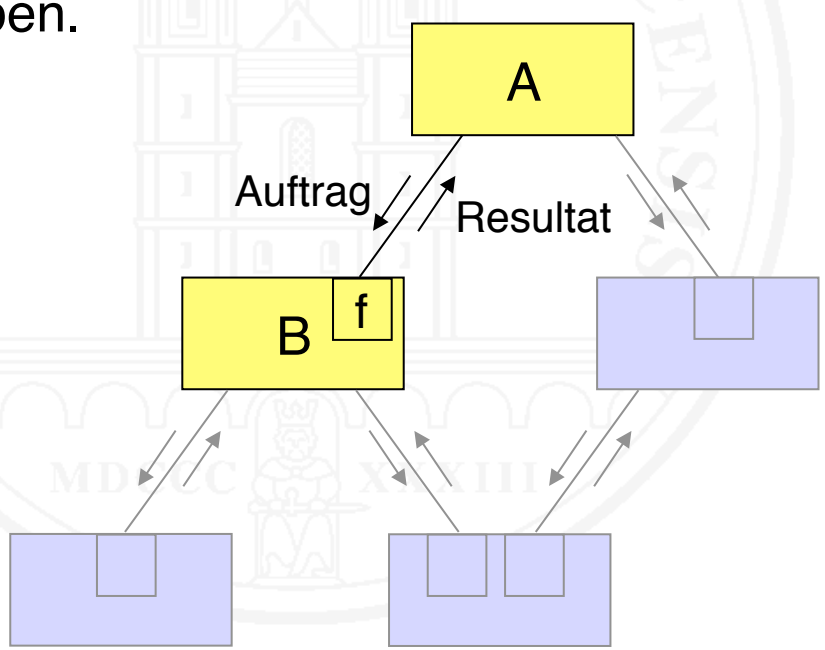
- Informationsteilhabe



- Die Art der Zusammenarbeit definiert wesentlich den Entwurfsstil
- Die Zusammenarbeit muss dokumentiert werden

Leistungserbringung – 1

- Motiv: Delegieren von Aufgaben
- Situation 1
  - A will eine benötigte Funktion f durch B ausführen lassen.
  - Mit Ausnahme des Funktionswerts soll der Systemzustand unverändert bleiben.

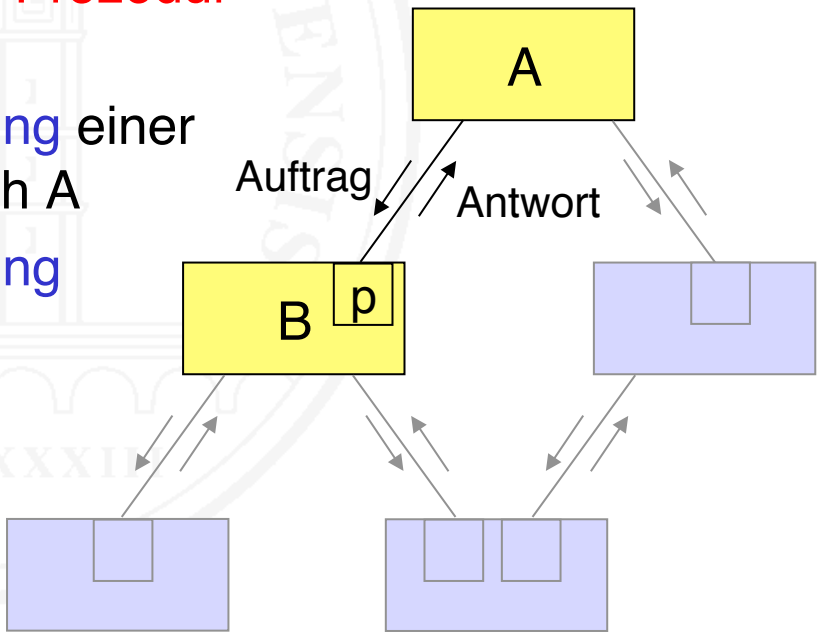


Leistungserbringung – 2

- Mittel
  - Statisch gebundener Aufruf einer Funktionsprozedur oder Methode f in B durch A
  - Dynamisch gebundene Anwendung einer Methode f (mit Rückgabewert) auf das Objekt B durch A
  - Dynamisch gebundene Anwendung einer Methode f aus einer Oberklasse von B auf das Objekt B durch A : super.f
- Bemerkungen
  - Ausführung von f darf den Systemzustand nicht verändern mit Ausnahme des Funktionswerts
    - ⇨ Die Verwendung einer Funktion ist nebenwirkungsfrei
  - Als Parameter und Resultat werden in der Regel Daten oder Objekte übergeben

Leistungserbringung – 3

- Situation 2
  A will eine benötigte Operation durch B ausführen lassen. Die Ausführung kann (oder soll) den Systemzustand verändern.
- Mittel
  - Statisch gebundener Aufruf einer Prozedur oder Methode p in B durch A
  - Dynamisch gebundene Anwendung einer Methode p auf das Objekt B durch A
  - Dynamisch gebundene Anwendung einer Methode p aus einer Oberklasse von B auf das Objekt B durch A : super.f



Leistungserbringung – 4

- Bemerkungen
  - Direkt veränderbar sind:
    - Ausgabeparameter von p
    - Zustand des Moduls, der p enthält, bzw. des Objekts, auf das p angewendet wird.
  - Indirekt veränderbar sind
    - Alle Zustände von Elementen, die von Operationen veränderbar sind, an die p (direkt oder transitiv) Arbeit delegiert.
  - Beliebige Nebenwirkungen möglich
  - Als Parameter können Daten, Operationen und Objekte übergeben werden
  - Rückruf / Delegation durch Übergabe von Operationen und Objekten möglich

Informationsaustausch

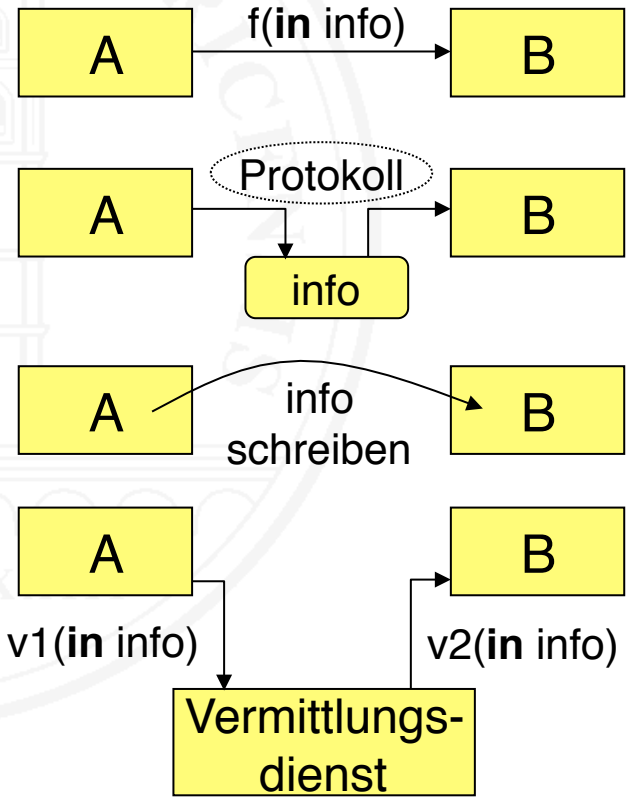
- Motiv: Organisation der Zusammenarbeit zwischen Komponenten nach dem Prinzip
  - der Wertschöpfungskette oder der Fließbandarbeit (Bringprinzip)
  - einer Kette von Einkäufern (Holprinzip)
  - von Lieferverträgen (Abonnementsprinzip)
- Information: Daten, Operationen oder Objekte

Mini-Übung 5.6

Nach welchem Prinzip arbeiten Pipes in UNIX?

Informationsaustausch: Bringprinzip – 1

- Situation 1
  A will Informationen an B weiterreichen (Bringprinzip)
- Mittel
  - Aufruf mit Parameterübergabe
  - Zugriff auf globale Variablen
  - Direktmanipulation von Attributen
  - Verwendung eines Vermittlungsdienstes

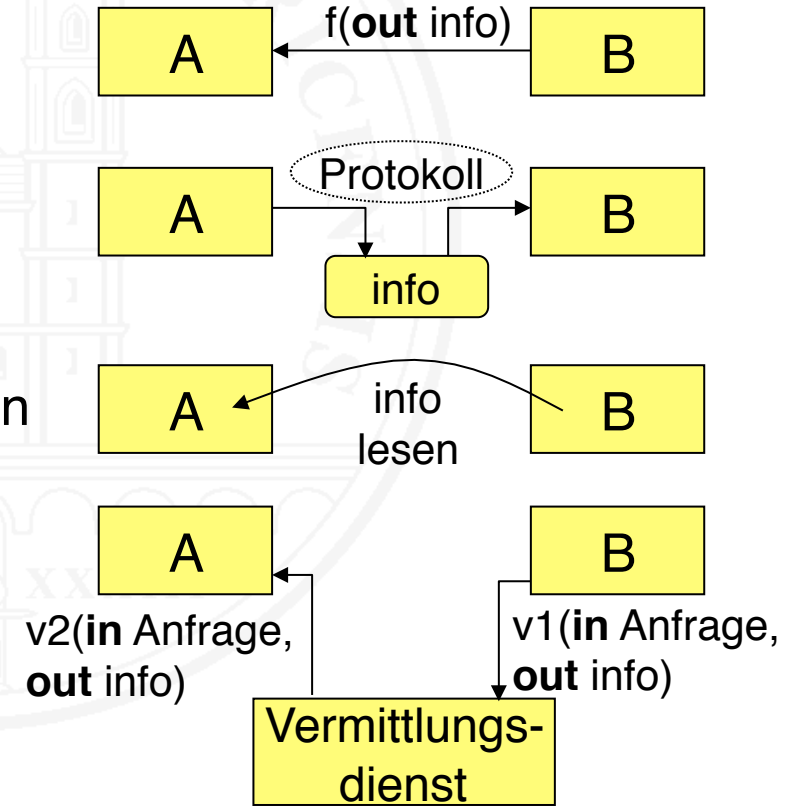


Informationsaustausch: Bringprinzip – 2

- Bemerkungen
  - Übergabe mit Wertparameter: nebenwirkungsfrei, schwache Kopplung
  - Übergabe von Operationen und Objekten: mächtiger und flexibler
    Aber: Nebenwirkungen und Rückwirkungen auf A möglich, stärkere Kopplung
  - Globale (oder teilglobale) Variablen: fast immer Nebenwirkungen, Synchronisation erforderlich, starke Kopplung
  - Direktmanipulation: sehr starke Kopplung ➪ vermeiden
  - Vermittlungsdienst: entkoppelt A und B, ermöglicht geografische Verteilung
    Aber: Funktionen und Objekte nur eingeschränkt übertragbar

Informationsaustausch: Holprinzip – 1

- Situation 2
  A will Informationen von B erhalten (Holprinzip)
- Mittel
  - Aufruf mit Parameterübergabe
  - Zugriff auf globale Variablen
  - Direktmanipulation von Attributen
  - Verwendung eines Vermittlungsdienstes

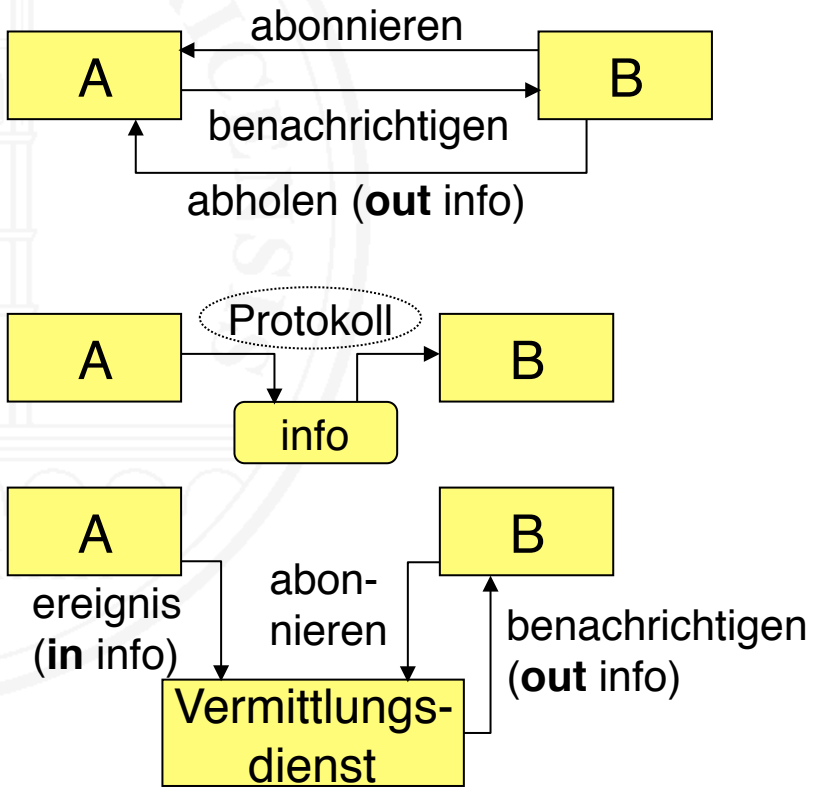


Informationsaustausch: Holprinzip – 2

- Bemerkungen
  - Übernahme als Referenzparameter: nebenwirkungsfrei und schwach koppelnd, wenn Daten nicht verändert werden
    Sonst: massive Nebenwirkungen möglich
  - Übernahme von Operationen und Objekten: mächtiger und flexibler
    Aber: Nebenwirkungen und Rückwirkungen auf A möglich, stärkere Kopplung
  - Globale (oder teilglobale) Variablen: fast immer Nebenwirkungen, Synchronisation erforderlich, starke Kopplung
  - Direktmanipulation: starke Kopplung ➪ nur verwenden, wenn Änderungen in der Struktur der gelesenen Daten wenig wahrscheinlich
  - Vermittlungsdienst: entkoppelt A und B, ermöglicht geografische Verteilung
    Aber: Funktionen und Objekte nur eingeschränkt übertragbar

Informationsaustausch: Abonnementsprinzip – 1

- Situation 3
  - B abonniert Informationen bei A
  - A benachrichtigt B, worauf B abholt (Abonnementsprinzip)
- Mittel
  - Abonnieren-Benachrichtigen-Holen (Beobachtermuster)
  - Zugriff auf globale Variablen
  - Verwendung eines Vermittlungsdienstes



Informationsaustausch: Abonnementsprinzip – 2

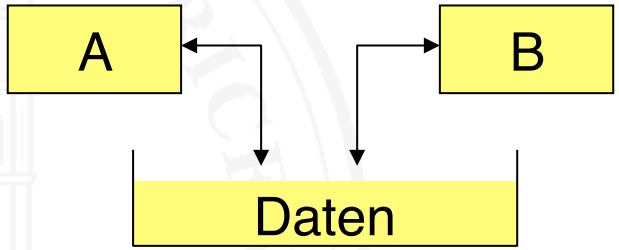
- Bemerkungen
  - Dient zur Trennung eng kooperierender Aufgaben in separate Module (Entkopplung)
  - Kopplung zwischen A und B wird von stark auf mittel reduziert.
  - Verwendung globaler oder teilglobaler Variablen zur Realisierung eines Abonnementsprinzips ist aufwendig und fehlerträchtig
    ➪ vermeiden
  - Verwendung eines Vermittlungsdienstes entkoppelt A und B, ermöglicht geografische Verteilung
    Aber: Funktionen und Objekte nur eingeschränkt übertragbar

Informationsteilhabe

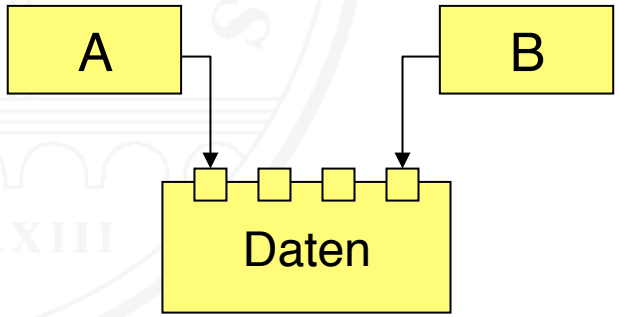
- Motiv: Komponenten sind gleichberechtigte Teilhaber an einer Menge von Information
  - Offene, direkte Teilhabe: gemeinsame Speicher
  - Gekapselte, direkte Teilhabe: Datenabstraktionen
  - Teilhabe über einen gemeinsamen Datenverwalter: Informationsdepot

Informationsteilhabe: gemeinsamer Speicher – 1

- Situation 1
  A und B nutzen einen gemeinsamen Speicherbereich
- Mittel:
  - Direktzugriff auf gemeinsamen Speicher



  - Gemeinsam genutzte Datenabstraktion

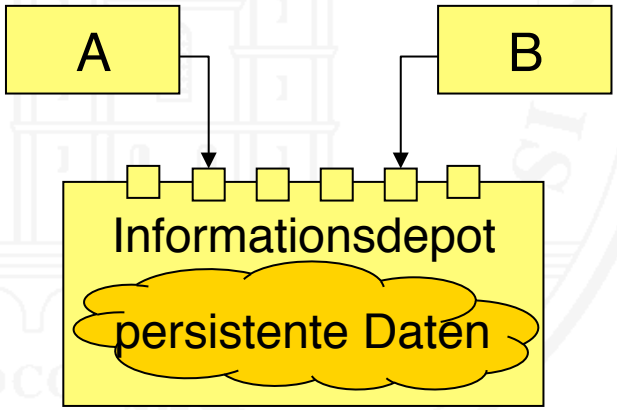


Informationsteilhabe: gemeinsamer Speicher – 2

- Bemerkungen
  - Direktzugriff ist einfach und schnell
    Aber: erfordert Sichtbarkeit der Datenstrukturen und explizite Synchronisation ➪ starke Kopplung
  - Gemeinsam genutzte Datenabstraktion verbirgt Datenstrukturen und Synchronisation

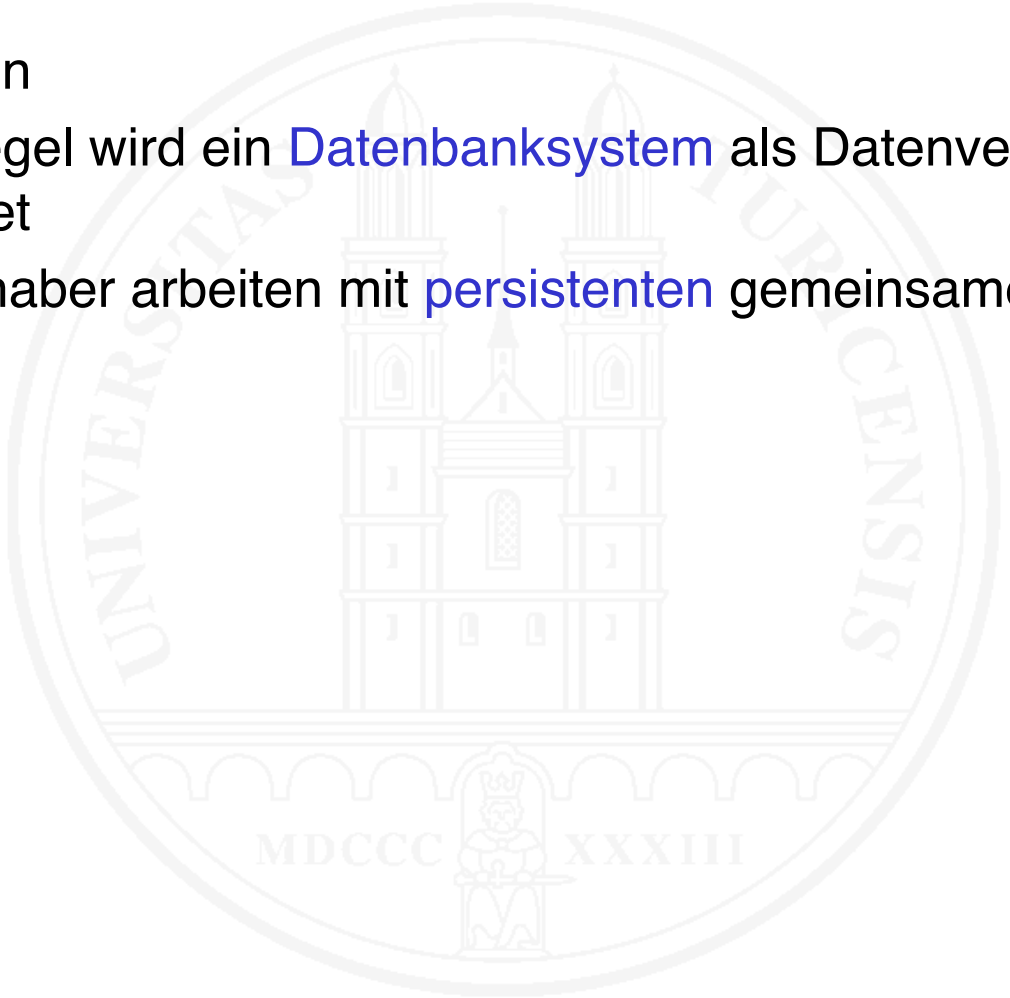
Informationsteilhabe: Informationsdepot – 1

- Situation 2
  Mehrere Teilhaber betreiben ein gemeinsames Informationsdepot (Repository)
- Mittel:
  - Verwaltung und Zugriff durch einen Datenverwaltungsdienst



Informationsteilhabe: Informationsdepot – 2

- Bemerkungen
  - In der Regel wird ein Datenbanksystem als Datenverwaltungsdienst verwendet
  - Die Teilhaber arbeiten mit persistenten gemeinsamen Objekten



Mini-Übung 5.7

Die Authentifizierung eines Benutzers soll über eine persönliche Benutzerkarte mit PIN-Code erfolgen

Entwerfen Sie eine Modularisierung dieser Authentifizierung

a) nach dem Prinzip des Delegierens von Aufgaben

b) nach dem Prinzip der Wertschöpfungskette (Bringprinzip)

Zusammenarbeit und Entwurfsstil

- Die meisten Entwurfs- bzw. Architekturstile verwenden eine charakteristische Form der Zusammenarbeit

Beispiele

- Pipe-and-Filter Stil
  - Informationsaustausch nach dem Bringprinzip
- Layered System Stil
  - Leistungserbringung mit statisch gebundenen Funktionen und Prozeduren
  - Azyklische Aufrufhierarchie
  - Übergabe von Daten als Parameter
  - Informationsteilhabe über gekapselte Datenstrukturen

Zusammenarbeit und Entwurfsmuster

- Manche Entwurfsmuster (Gamma et al. 1995) verwenden eine charakteristische Form der Zusammenarbeit
- Beispiel Abonnementsprinzip im Beobachtermuster

Dokumentation der Zusammenarbeit

- Durch Zusammenarbeit werden aus Modulen bzw. Komponenten Systeme
- Die Festlegung und Dokumentation der Zusammenarbeit ist eine zentrale Entwurfsaufgabe
- Dokumentation der einzelnen Komponenten
  - Leistungsangebot: Angebotsschnittstelle(n)
  - Leistungsbedarf: Bedarfsschnittstelle(n)
- Dokumentation der Komposition
  - Statische Struktur typisch durch Diagramme
  - Dynamischer Ablauf wo nötig durch Zusammenarbeitsprotokolle (meistens mit Automaten / Statecharts
- Der Dokumentationsbedarf hängt vom Grad der Unabhängigkeit der Komponenten ab

Gemeinsam erstellte Komponenten

Bei gemeinsam erstellten und vertriebenen Komponenten

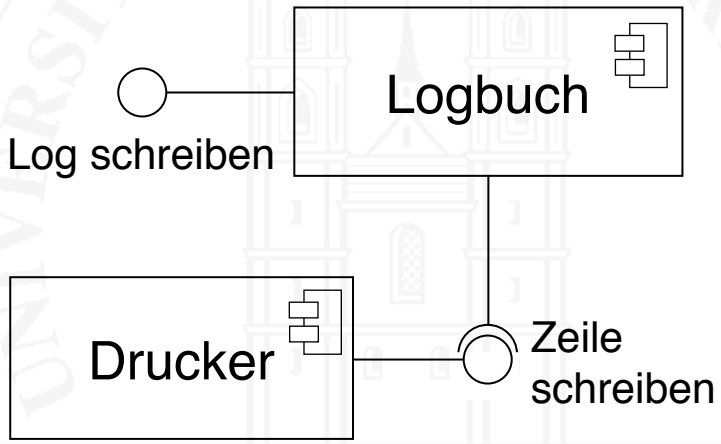
- Komponenten und Zusammenarbeit werden miteinander verzahnt entworfen
- Der Verwendungskontext jeder Komponente ist bekannt
- Entwerfende stimmen Schnittstellen und Zusammenarbeitsbedürfnisse aufeinander ab
  - Angebotsschnittstellen häufig nur syntaktisch definiert
  - Bedarfsschnittstellen in der Regel nicht explizit definiert, ggf. Namen der benötigten Komponenten aufgelistet
  - Dokumentation der Zusammenarbeit durch Kompositions-diagramm(e)

Separat erstellte Komponenten

- Bei separat erstellten und vertriebenen Komponenten
- Jede Komponente steht für sich und kennt bei Erstellung ihren Verwendungskontext nicht
- Präzise Definition der Angebotsschnittstelle(n) notwendig, damit die Komponente verwendbar ist
- Präzise Definition der Bedarfsschnittstelle(n) notwendig, damit die Komposition von Komponenten möglich ist
- Dokumentation der Komposition durch Kompositionsdiagramme

Kompositionsdiagramm

Beispiel eines Kompositionsdiagramms (in UML 2)



Literatur

Siehe Literaturverweise im Kapitel 6 des Skripts. Weitere Literatur:

Szyperski, C. (1998). *Component Software – Beyond Object-Oriented Programming*. Harlow, England etc.: Addison-Wesley.

Wirth, N. (1985). *Programming in Modula-2*. 3rd edition. Berlin, etc.: Springer.

Im Skript [M. Glinz (2005). *Software Engineering*. Vorlesungsskript, Universität Zürich] lesen Sie Kapitel 6.

Im Begleittext zur Vorlesung [S.L. Pfleeger, J. Atlee (2010). *Software Engineering: Theory and Practice*, 4th edition. Upper Saddle River, N.J.: Pearson Education International] lesen Sie Kapitel 5 und 6.

Hinweis: Die dritte und die vierte Auflage des Begleittextes unterscheiden sich in den Kapiteln 5 und 6 erheblich. Die Darstellung in der vierten Auflage ist unseres Erachtens deutlich besser und stimmt wesentlich mehr überein mit der Art wie wir Architektur und Entwurf in dieser Vorlesung lehren.