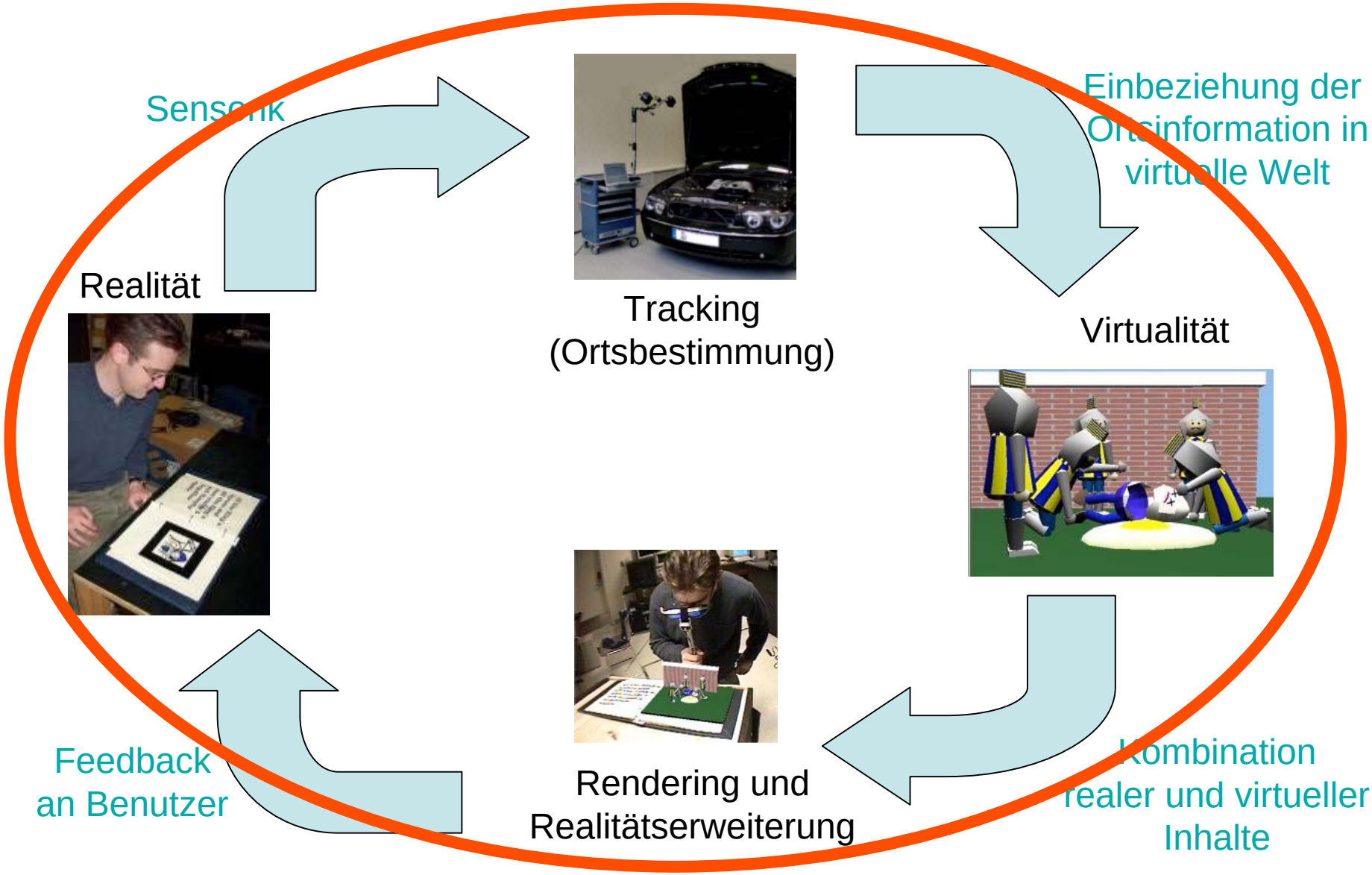
Softwaresysteme für Augmented Reality

Vorlesung „Augmented Reality”

Prof. Dr. Andreas Butz

WS 2006/07

Ein Generisches AR-System



Überblick

- Software Engineering – Überblick
- Architekturstile
  - Monolithische Systeme
  - Komponentenbasierte Systeme
    - Client/Server
    - Peer-to-Peer
    - Blackboard
    - Pipe and Filter
- Verteilte Systeme: Kommunikationsarten
  - RPC/RMI
  - Events
  - Shared Memory

Software Engineering – Bausteine

- Requirements Elicitation
- Requirements Analysis
- System Design
- Implementation
- Testing

! Je nach Prozessmodell werden verschiedene Kombinationen/Iterationen dieser Bausteine ausgeführt

SE – Requirements Elicitation

- Probleme:
  - Kunde weiß oft nicht, was er will
  - Wer ist Kunde (Auftraggeber, Endanwender)?
  - Verschiedene Erfahrungsniveaus
- Gängige Methoden:
  - Interviews, Meetings
  - Surveys, Fragebogen
  - Beobachtung

SE – Requirements Analysis

- Ziel: Problemdomäne in technische Domäne abbilden
- Aufgaben:
  - Abgrenzung des Problems
  - Modellierung des Problems
  - Definition und Strukturierung der Anforderungen
    - Functional Requirements: beschreiben Interaktion zwischen dem System und der Umgebung
    - Non-Functional Requirements: beschreiben die für den Benutzer sichtbaren Systemaspekte (z.B. Performanz, Usability)

SE – System Design

- Ziel: Erstelle ein Modell des Problems, das später implementiert werden kann
- Systemmodelle:
  - Datenmodell
  - Funktionsmodell
  - Verhaltensmodell
  - Datenflussmodell
- Architektur:
  - Hierarchische Struktur von Prozeduren und Daten
  - Definition von Schnittstellen zum Datenfluss

SE – Implementation & Testing

- Implementierung
  - Umsetzen des Systemdesigns in eine Programmiersprache
- Testen
  - Verifikation der Implementierung anhand der Anforderungsspezifikation

Architectural Style

A style consists of a vocabulary of design elements, a set of well-formedness constraints that must be satisfied by any architecture written in the style, and a semantic interpretation of the connectors.

[Moriconi 1994]

- Beschreibt allgemeine Lösung des Systemaufbaus
- Wird während des Systemdesigns festgelegt
- Domänenunabhängig
- Abstraktester Beschreibungsteil einer Architektur
- Reicht nicht aus, um eine Architektur zu beschreiben!

Arch. Style: Monolithische Systeme

- Alle Systemteile im selben Programm
- Standardansatz von „Hacks“
- Beispiele:
  - (Alte) Mainframeprogramme
  - Viele DOS/Windows 3.x Anwendungen
  - Lösungen zum de Übungsaufgaben ;-)

Arch. Style: Diskussion Monolithe

Pro:

- Schnell programmiert (aber nur für sehr einfache Probleme)
- Einfaches Einmal-Deployment

Con:

- Schwierige Wartbarkeit
- Sehr schlechte Wiederverwendung
- Nebenläufigkeit schwer zu realisieren

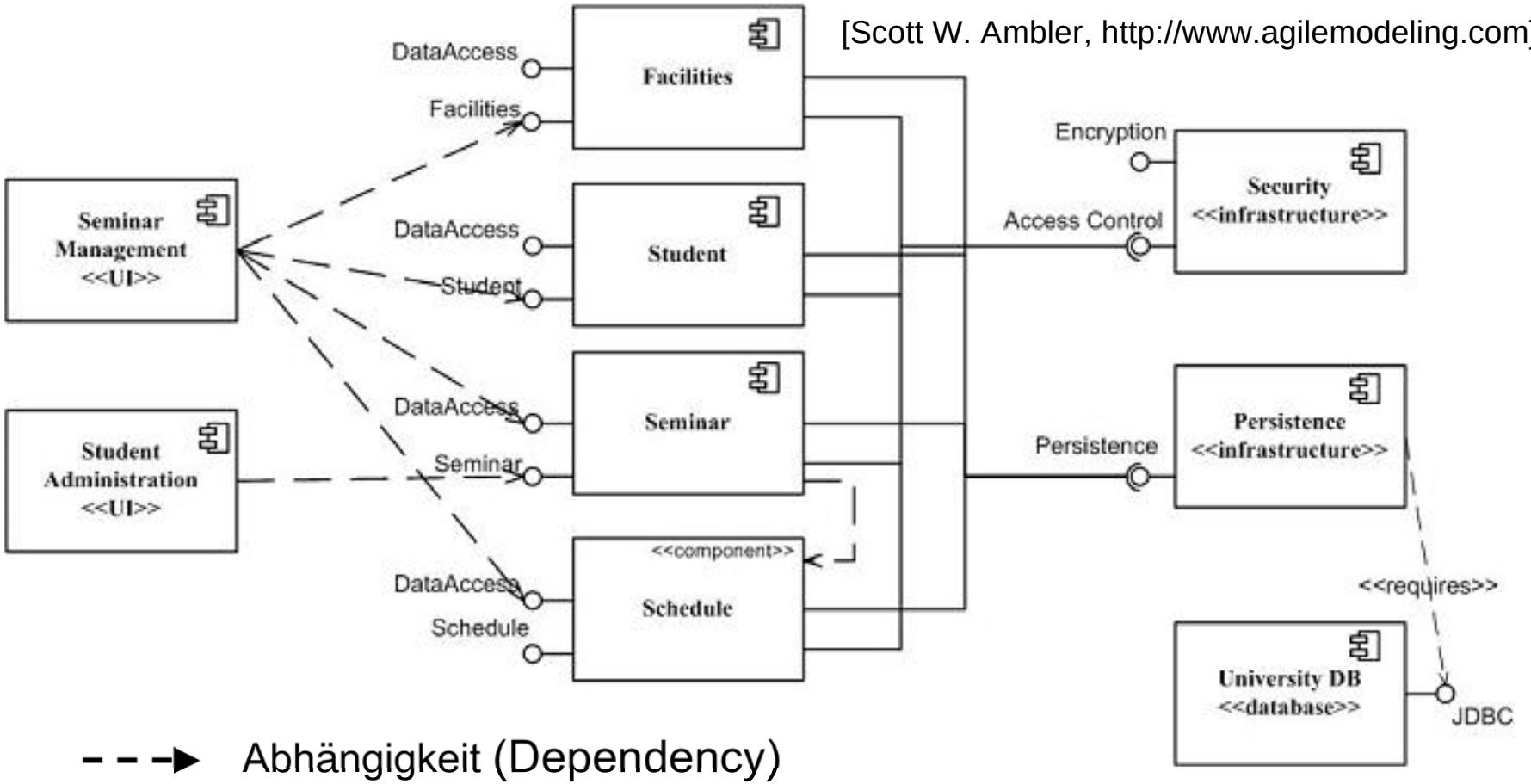
Don‘t try this at home.

Komponentenbasierte Architekturen

- Idee: Baue Programme aus fertig fabrizierten Komponenten (Douglas McIlroy, *Mass Produced Software Components*, 1968)
- Häufige Implementierung:
  - Komponenten sind Objekte
  - Definierte Schnittstellen in einer Interface Description Language (IDL)
- Komponenten können auf einen (z.B. Microsoft COM) oder mehrere Rechner (z.B. CORBA, DCOM) verteilt werden

Notation von Komponenten (UML 2)

[Scott W. Ambler, http://www.agilemodeling.com]

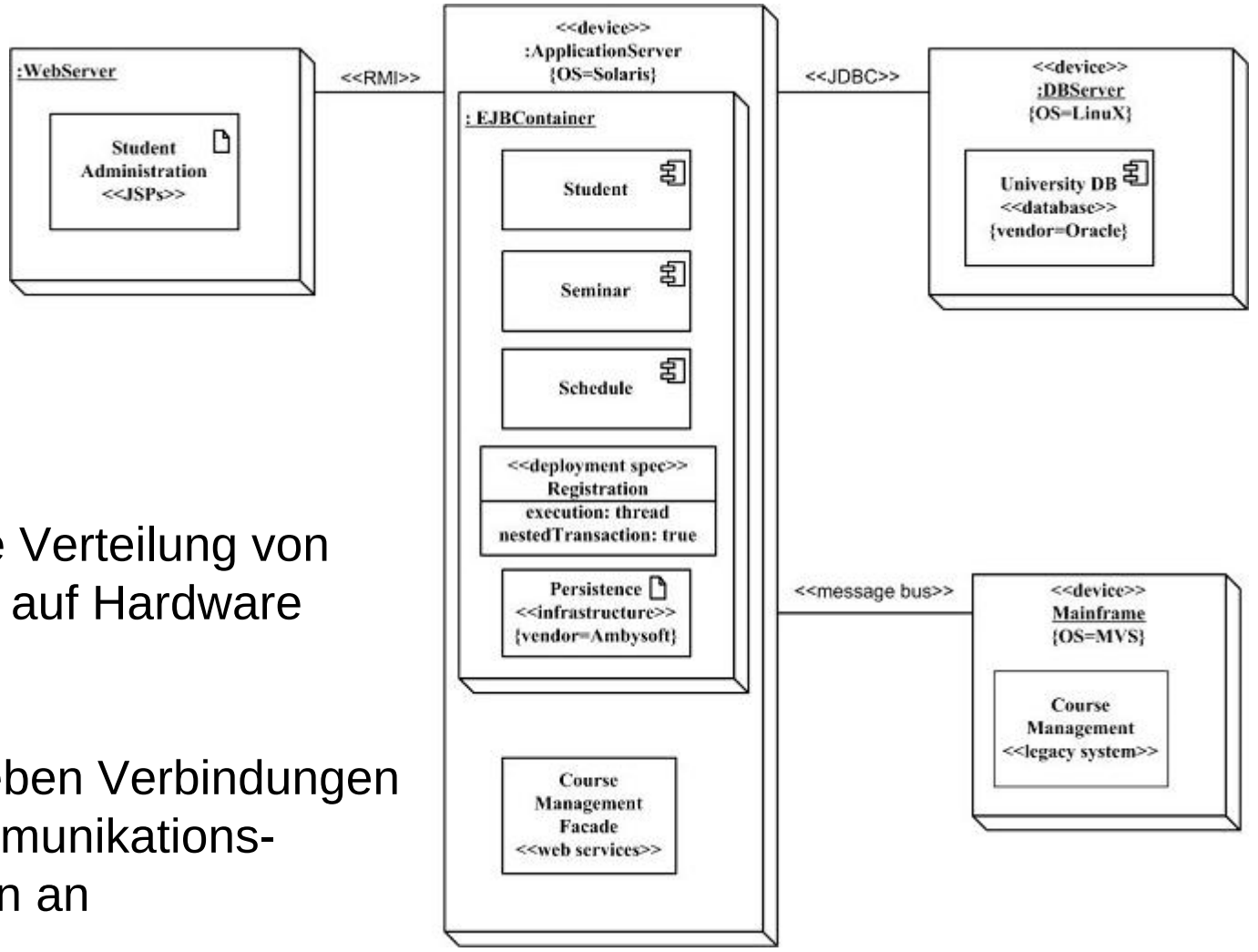


Abhängigkeit (Dependency)

Zur Verfügung gestelltes Interface

Benötigtes Interface

Deployment Diagram (UML 2)



Statische Verteilung von Software auf Hardware

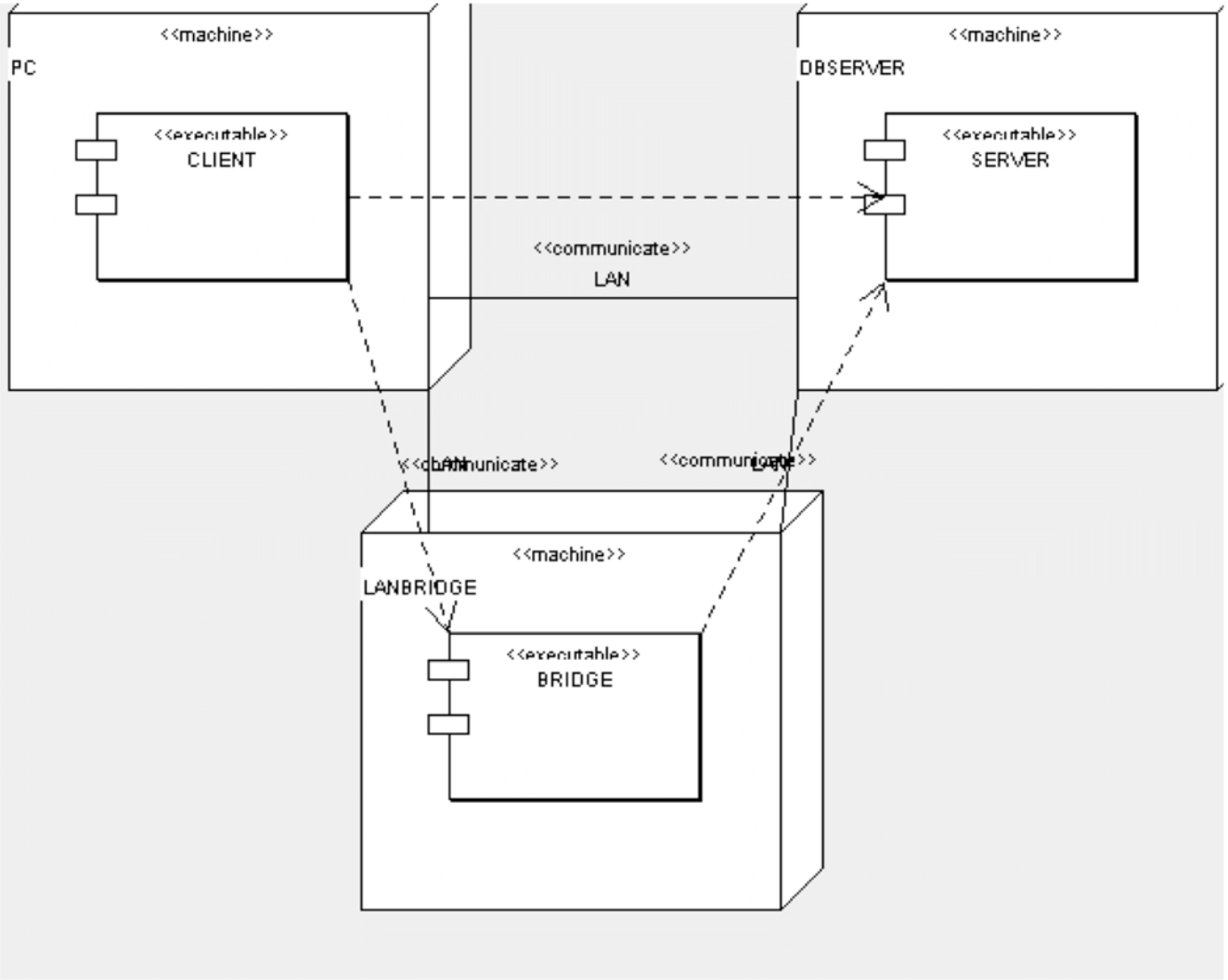
Linien geben Verbindungen und Kommunikations-methoden an

[Scott W. Ambler, http://www.agilemodeling.com]

Arch. Style: Client/Server

- Ein oder mehrere Server bieten Dienste für einen oder mehrere Clients
- Client ruft bekannte Schnittstelle des Servers auf und erhält Antwort
- Benutzer interagieren nur mit dem Client
- Beispiele
  - Webserver und Browser
  - NFS-Server und –Clients
  - Mailserver und Mailclient

Arch. Style: Client/Server Bsp.



Arch. Style: Diskussion Client/Server

Pro:

- Einfache Struktur
- Zentralisierte Datenverwaltung
- (Tragbarer) Client kann mit geringer Rechenleistung auskommen

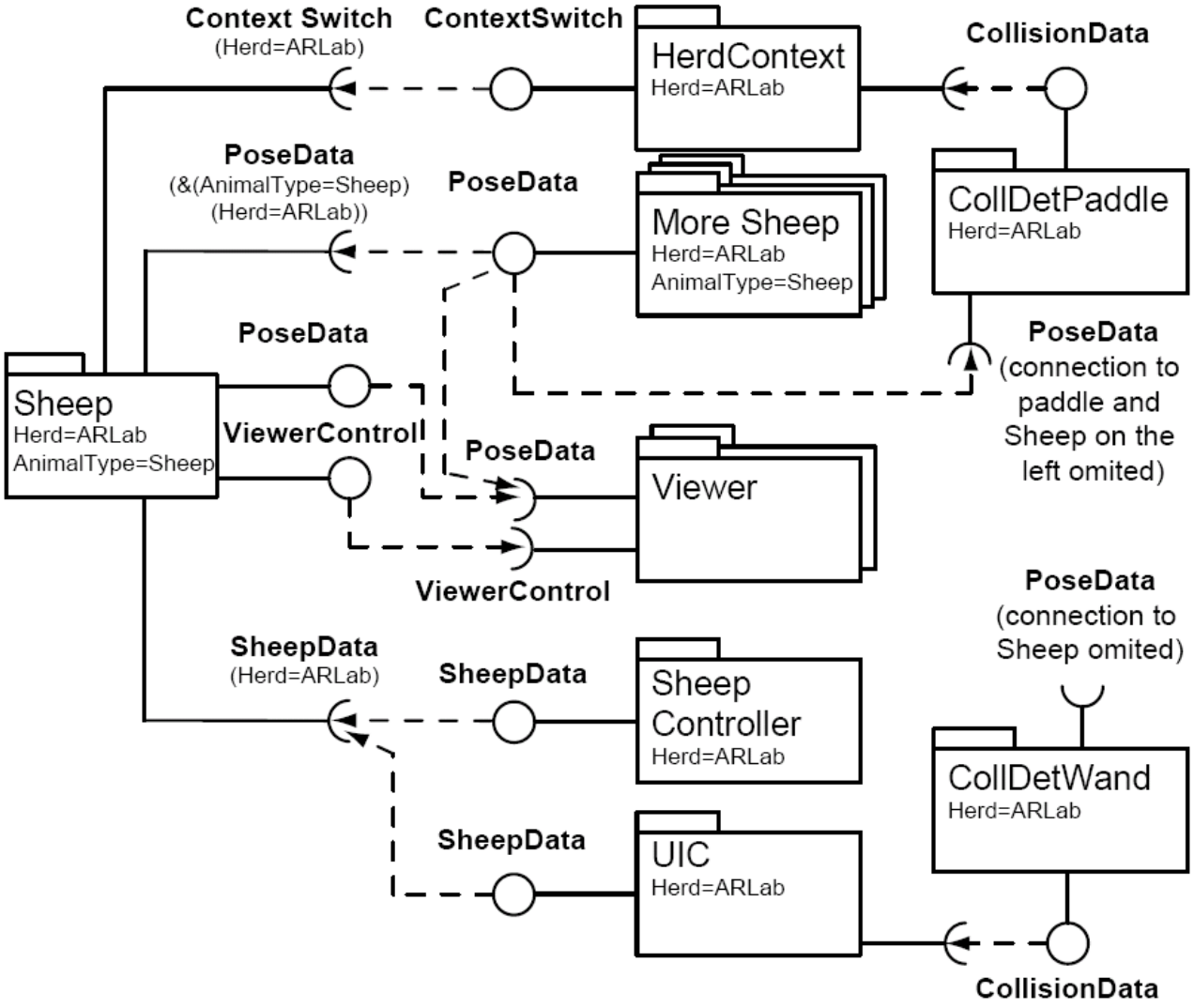
Con:

- Hohe Abhängigkeit von Server (Redundanz nötig)
- Besonders für Bildverarbeitung hohe Netzwerklast
- Existierende Systeme nicht auf Echtzeit ausgelegt (eher auf interaktive Bedienung)

Arch. Style: Peer to Peer (P2P)

- Jede beteiligte Komponente ist zugleich Server und Client
- Mischformen möglich, in denen Server, Clients und (Server+Client)s existieren
- Oft zusammen mit Ad-hoc Verbindungen
- Beispiele:
  - Filesharing Netze
  - Standard-Arbeitsgruppe in Windows-Netzwerk
  - Internet (konzeptionell)

Arch. Style: P2P Bsp.



Arch. Style: Diskussion P2P

Pro:

- Höchste Flexibilität
- Extreme Ausfallsicherheit möglich
- Passt gut in das Paradigma des Ubiquitous Computing

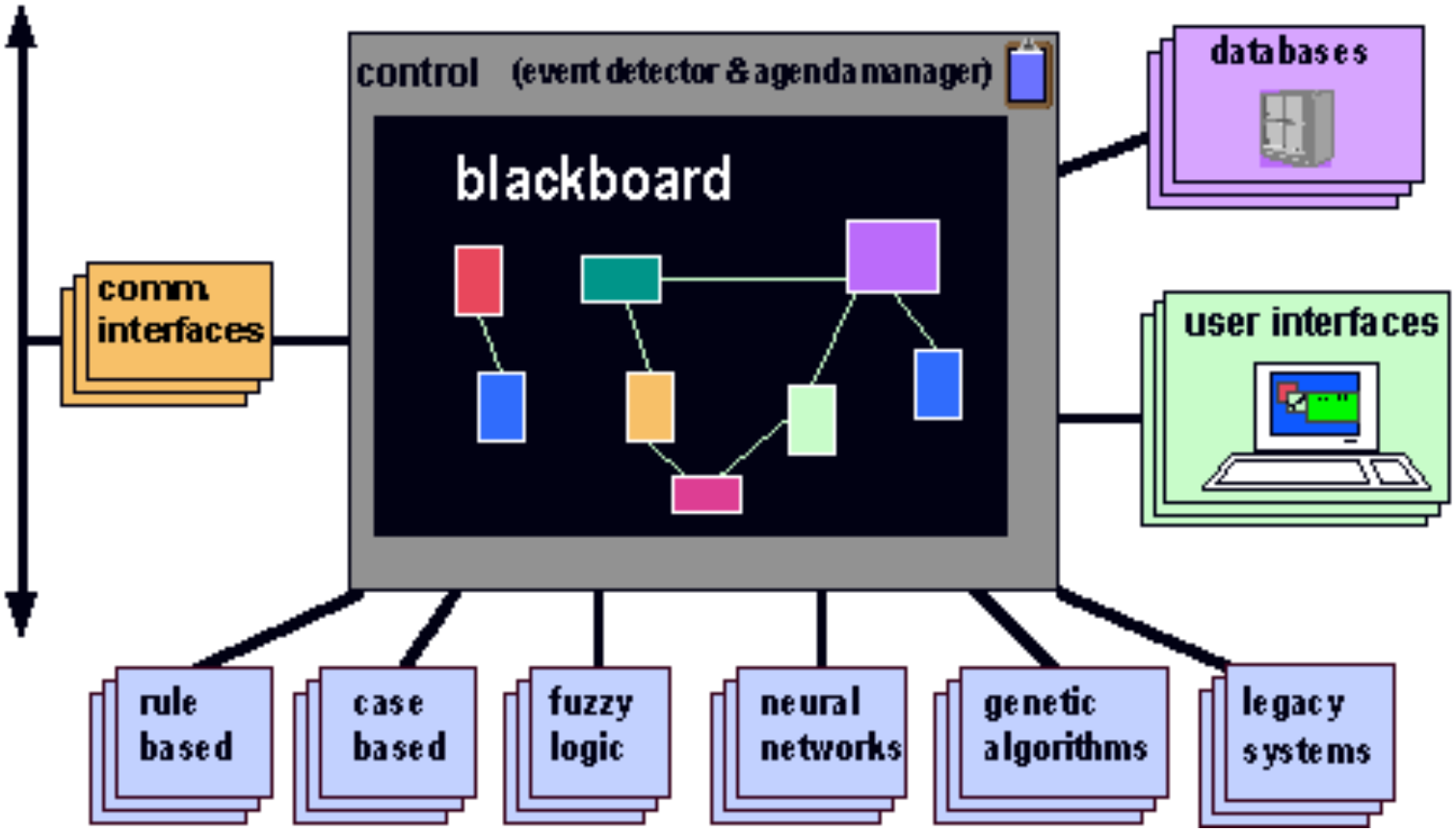
Con:

- Sehr komplex
- Unbekannte Struktur muss antizipiert werden
- Rekonfigurationen zur Laufzeit nötig (bei Ad-hoc Verbindungen)

Arch. Style: Blackboard

- „Schwarzes Brett“ als zentrales Repository
- Tuple Spaces als theoretische Basis
  - In Blackboard werden Tupel gelegt (Menge von Name/Wert Paaren)
  - Abfrage durch partielle Tupel (gegebene Name/Wert Paare müssen matchen, die restlichen werden ausgefüllt)
- Beispiele:
  - Agentensysteme in der KI
  - Java Spaces

Arch. Style: Blackboard Bsp.



[http://www.nb.net/~javadoug/bb.htm]

Arch. Style: Diskussion Blackboard

Pro:

- Sehr lose Kopplung von Komponenten
- Einfaches Multiplexing von Events
- Sehr intuitive Bedienung

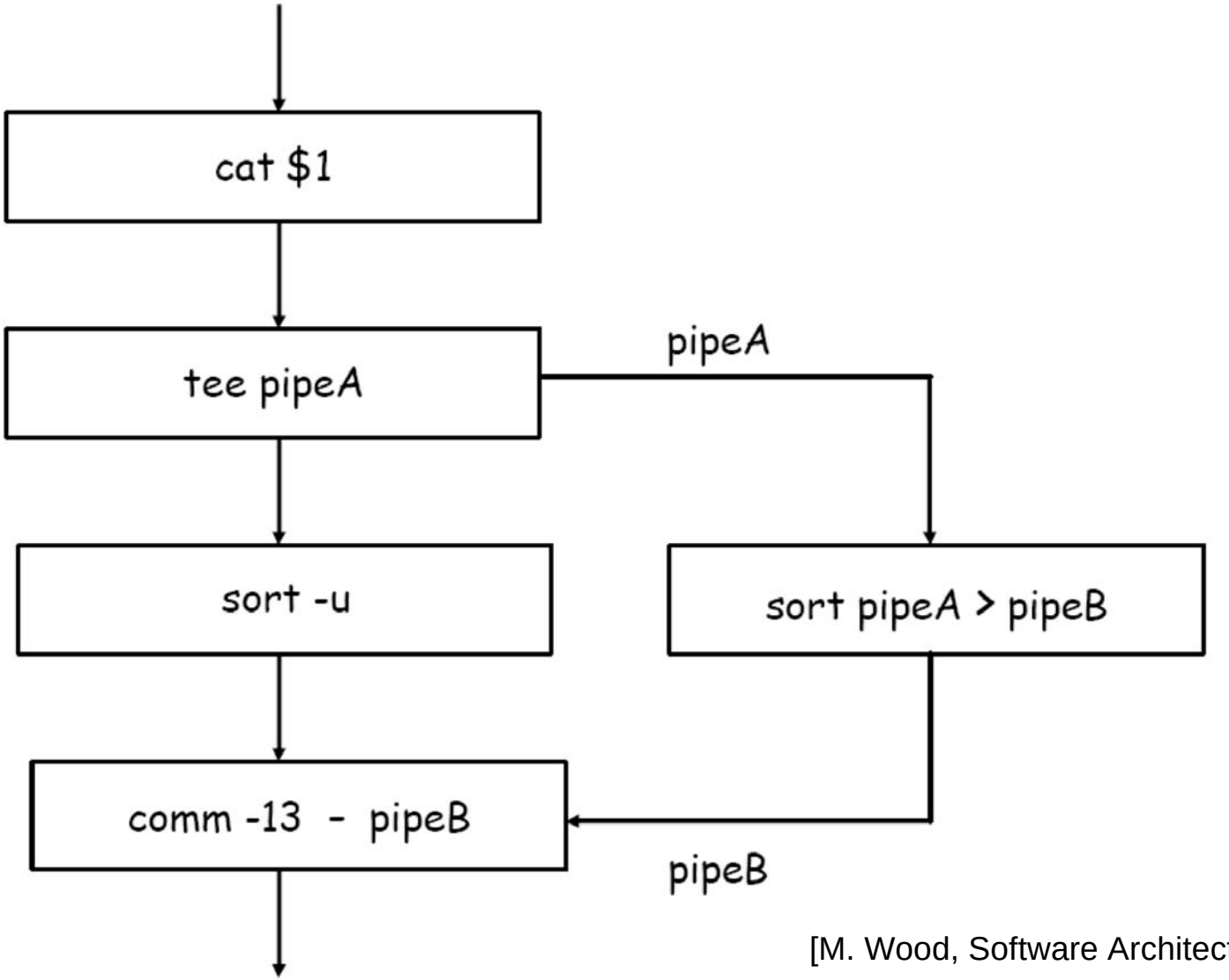
Con:

- Zentrale Komponente (Ausfallprobleme)
- Relativ ineffizient
- Hoher Ressourcenbedarf

Arch. Style: Pipe and Filter

- Datenflussarchitektur
- Jede Komponente liest Datenstrom als Eingabe und produziert Datenstrom als Ausgabe (→ Filterkomponenten)
- Verbindungen zw. Filtern durch Pipes
- Beispiele:
  - UNIX Shell-Skripte
  - Microsoft DirectShow Framework

Arch. Style: Pipe and Filter Bsp.



[M. Wood, Software Architecture Class]

Arch. Style: Diskussion Pipe and Filter

Pro:

- Verständlichkeit
- Hohe Wiederver-wendbarkeit der Komponenten
- Erweiterbarkeit (neue Filter) und Verbesserbarkeit (Filter ersetzen)
- Nebenläufigkeit einfach modellierbar

Con:

- Laufzeit kann recht hoch werden
- Keine globale Systemsicht – ein Filter kennt nur sich selbst
- Wie setzt man das Filternetzwerk (automatisch) auf?

Kommunikation in Verteilten Systemen

- Problem: wie kommunizieren (potentiell über ein Netzwerk verteilte) Komponenten
- Ziele:
  - Einfache API
  - Hohe Performanz
  - Gute Skalierbarkeit

RPC/RMI

- RPC = Remote Procedure Call, bei objektorientierten Systemen oft auch RMI = Remote Method Invocation
- Scheinbar lokaler, direkter Aufruf einer (entfernten) Komponente, die nur durch eine IDL spezifiziert ist
- Folgt Client-Server Modell
- Beispiele:
  - Microsoft DCOM
  - CORBA
  - XML-RPC / SOAP (Web Services: http als Protokoll, IDL in XML spezifiziert)

RPC/RMI: Diskussion

Pro:

- Nach Verbindungs-aufbau einfache Kommunikation (wie unter lokalen Komponenten)
- Hohe Performanz
- Gut für statusbehaftete Komponenten

Con:

- Multiplexing/ Broadcasting sehr schwierig
- Begrenzte Nachrichtengröße und –anzahl

Event-Kommunikation

- Grundidee: Programme warten auf Events, machen etwas, warten wieder
- Einsatz vor allem in GUI-Anwendungen
  → initiale Events kommen vom Benutzer
- Implementierung durch zentrale Event-Loop:

```
while (true) {
        waitForNextEvent();
        processEvent();
}
```

- Häufig: Kaskaden von Events

Events: Diskussion

Pro:

- Hervorragend für interaktive Anwendungen (GUIs)
- Natürliche Unterstützung von Nebenläufigkeit
- Multiplexing von Events problemlos (v.a. durch Broad- und Multicasts über UDP)

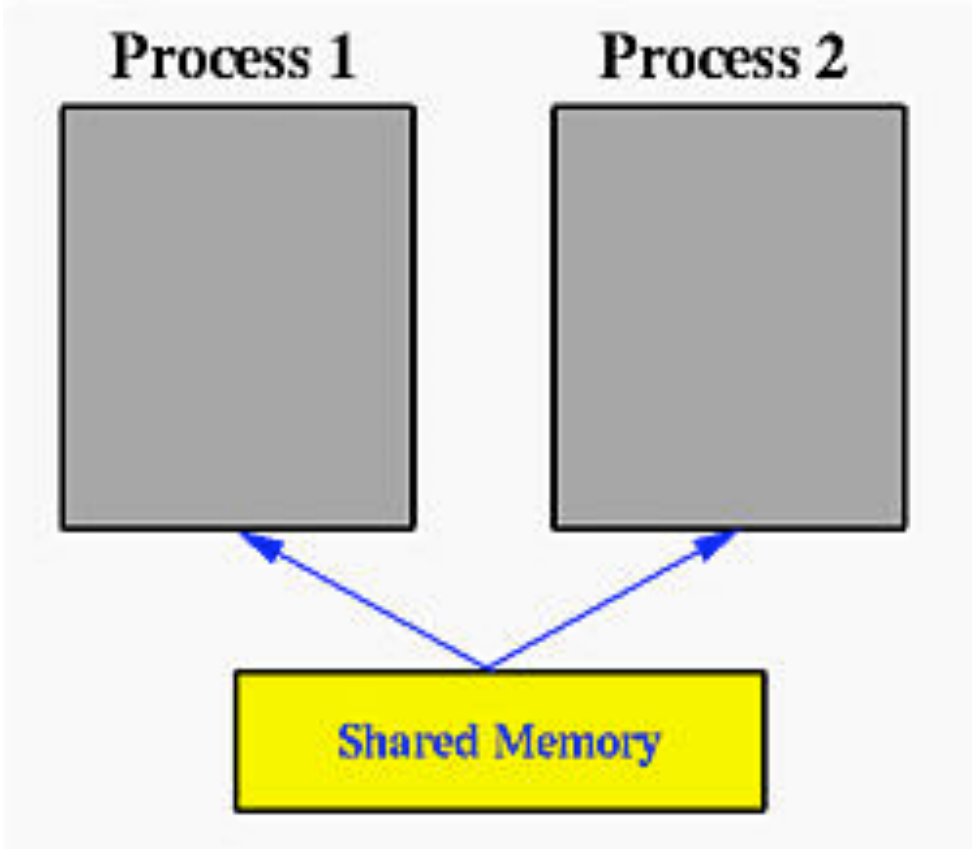
Con:

- Komponenten sollten zustandslos sein (v.a. bei unzuverlässiger Eventzustellung)
- Events können sich stauen, dann starke Verzögerungen in Abarbeitung möglich
- I.a. keine Garantien für Bearbeitungszeit eines Events möglich

Shared Memory

- Grundidee: Mehrere Prozesse teilen sich gemeinsamen Speicherbereich
- Vor allem für sehr breitbandige Übertragungen geeignet (Videodaten!)
- Analogie zur Blackboard Architektur
- Beispiele:
  - Multithreading: mehrere Threads greifen auf gemeinsamen Speicher zu
  - Parallelrechner (SMP): mehrere CPUs greifen auf gemeinsamen Speicher zu

Shared Memory: Bsp.



[http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/shm/what-is-shm.html]

Shared Memory: Diskussion

Pro:

- Höchst effizient
- Quasi beliebige Datengröße
- Beliebige Daten können ausgetauscht werden

Con:

- Extrem enge Kopplung
- Nur lokal möglich
- Synchronisation (z.B. über Semaphore) zwingend nötig

Streaming-Protokolle

- Grundidee: Übertrage (zeitsynchronisierten) Datenstrom
- Einsatz i.A. nur bei Audio/Videodaten sinnvoll
- Meist verlustbehaftete Übertragung (z.B. über UDP)
- Beispiele:
  - RTSP (RealTime Streaming Protocol, Kontrollinformationen für eigentliche Streams)
  - RealMedia/RealAudio
  - Windows Media
  - Quicktime (Framework für Vielzahl von Formaten)

Streaming-Protokolle: Diskussion

Pro:

- Sehr gut für Videodaten geeignet (Bildverarbeitung/ Remote Rendering)
- Variable Kompression der Daten, je nach Protokoll/Framework

Con:

- I.A. verlustbehaftete, unzuverlässige Übertragung
- Komplexe Technik