

Vorlesung „Methoden des Software Engineering“

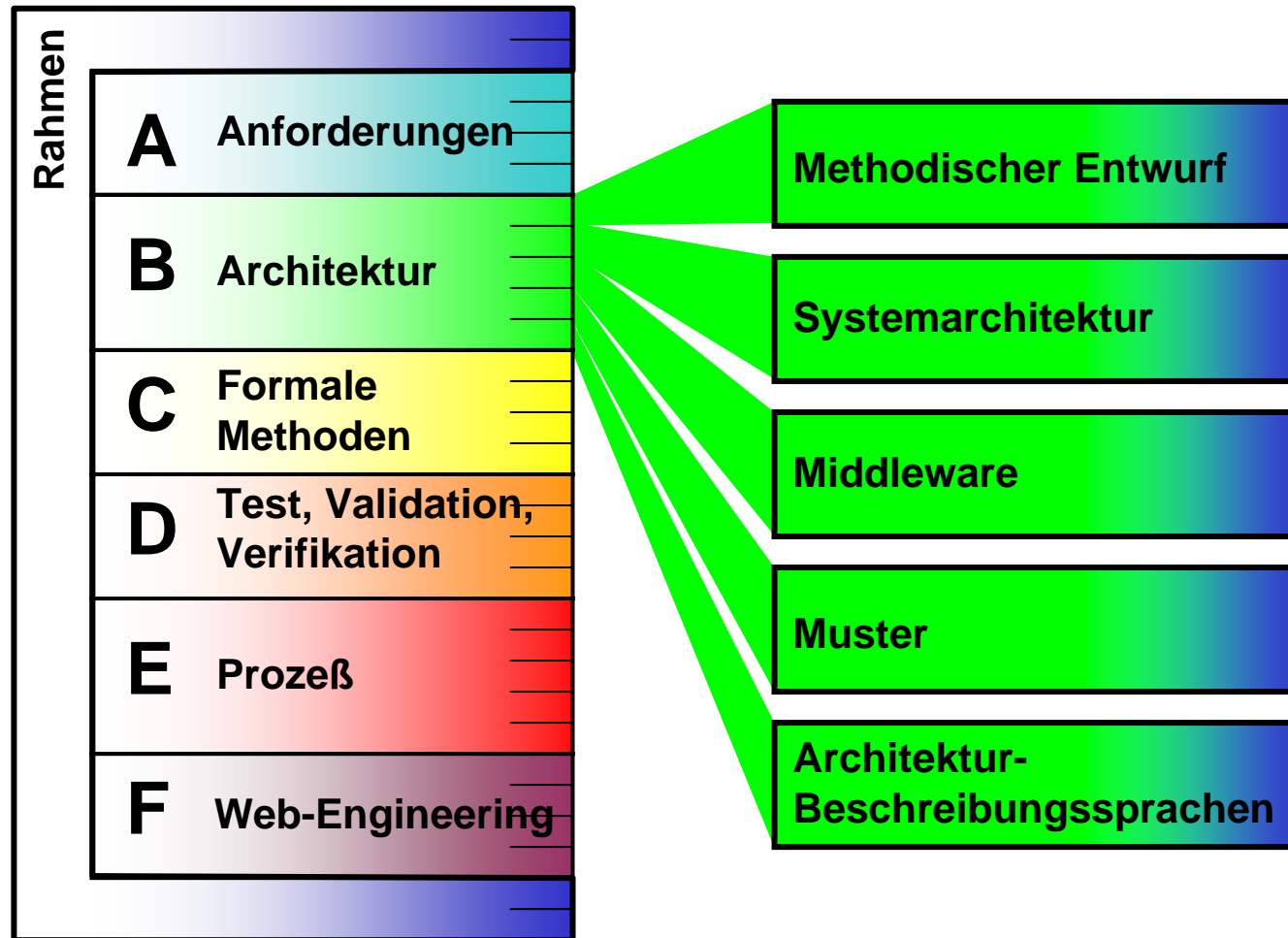
Block B „Software Architektur“

Komponenten und Architekturbeschreibungssprachen

Florian Hacklinger, Martin Wirsing

Einheit B.4, 16.11.2006

Gliederung Block B



Kernpunkte heute

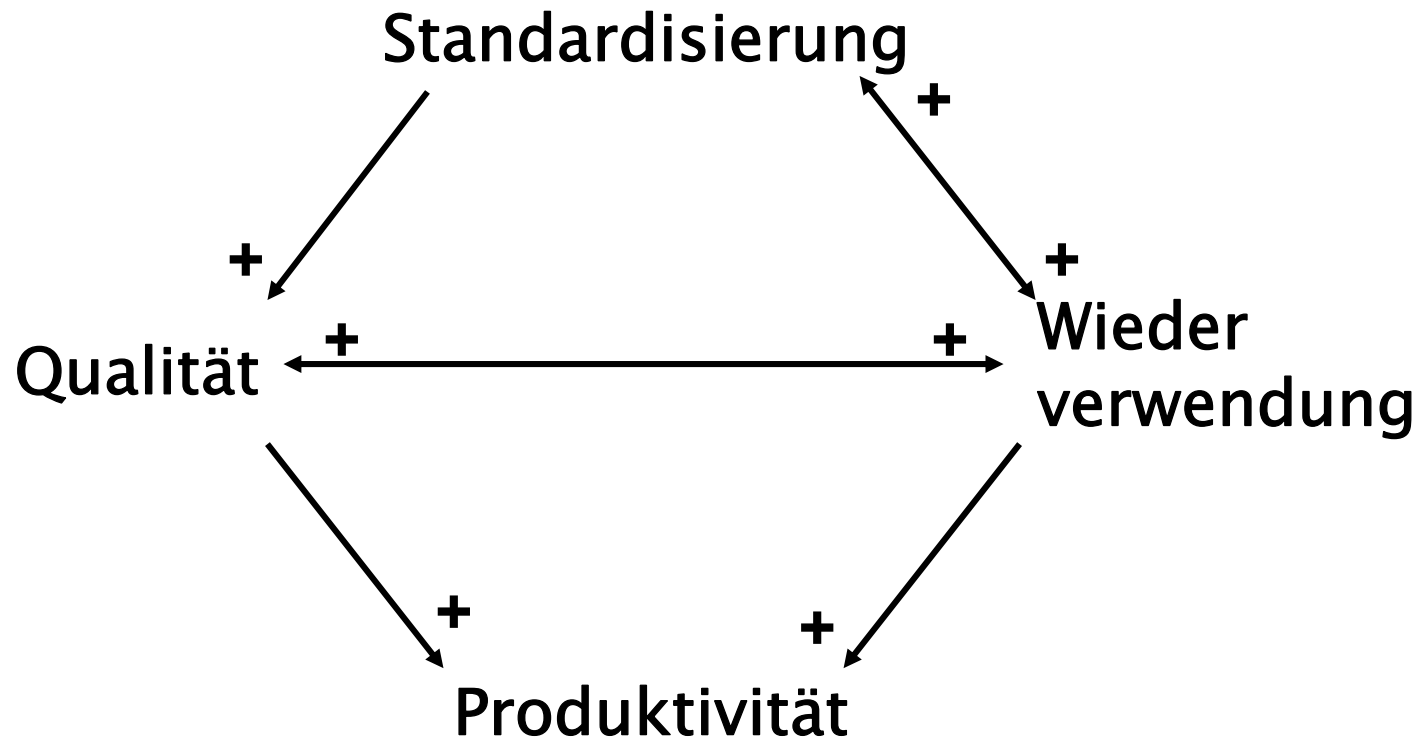
- **Komponenten sind abgeschlossene, wiederverwendbare Einheiten.**
 - Manchmal spricht man von **C**ommercial **O**ff **T**he **S**helf-Komponenten.
- **Es gibt sehr verschiedene Arten von Komponenten.**
 - Sie unterscheiden sich nach Größe, Stellung im Sw.-Lebenszyklus, Abgeschlossenheit und Art ihrer Verwendung.
- **Eine Architecture Description Language ist eine formale Sprache zur Beschreibung von Software-Strukturen.**
- **Solche Modelle können mit formalen Methoden untersucht werden.**
 - Solche Methoden stehen heute vor der Praxisreife. Das Thema wurde und wird am Lehrstuhl PST intensiv bearbeitet.
 - In den kommenden beiden Blöcken der Vorlesung werden Sie die dazu nötigen formalen Methoden kennen lernen.
- **Wir suchen ständig Studenten, die in diesem Bereich mitarbeiten wollen.**

Themen heute

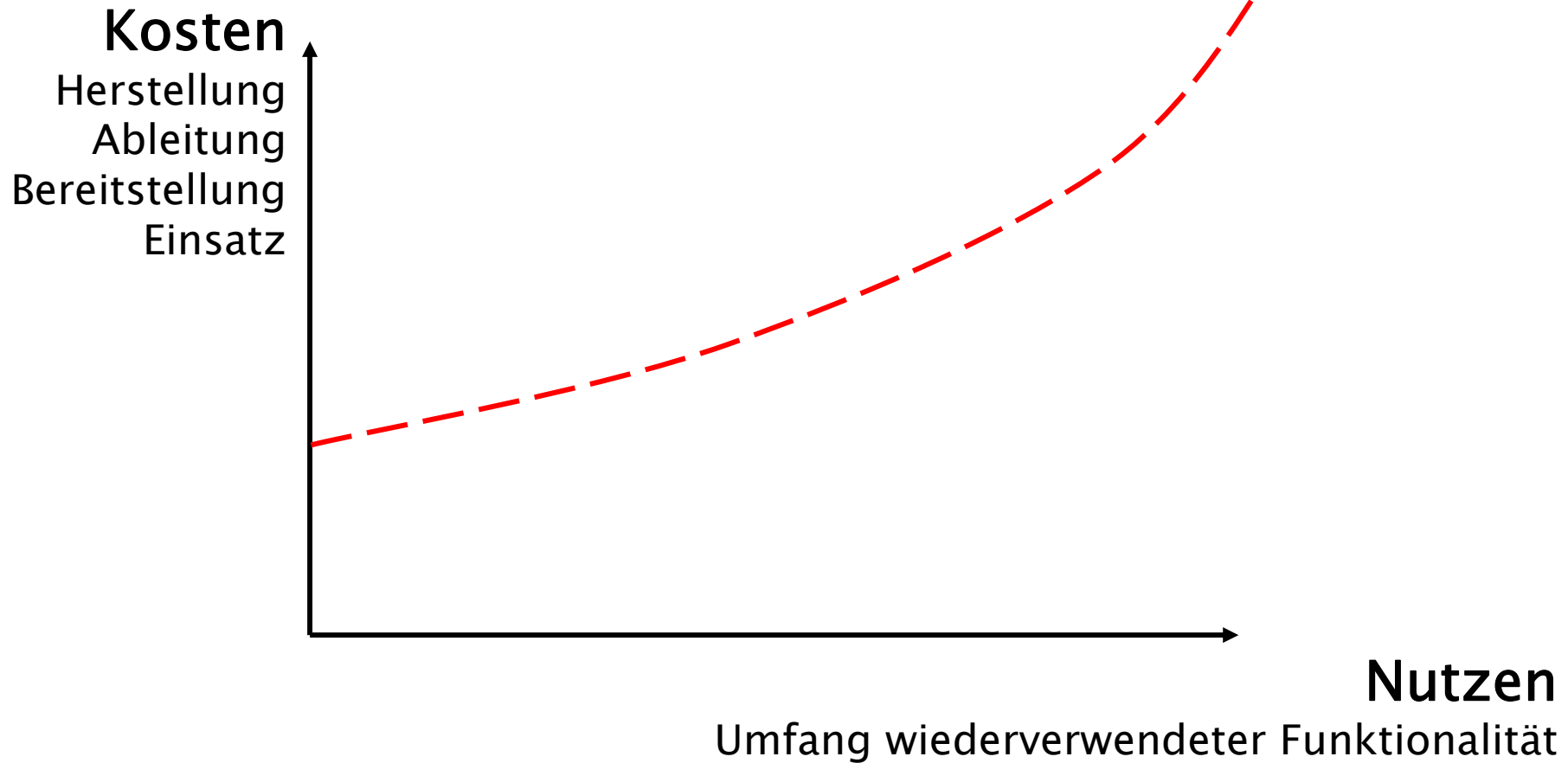
- Einführung Komponenten
- ADL-Einführung
- Java/A

Motivation – Warum überhaupt Komponenten?

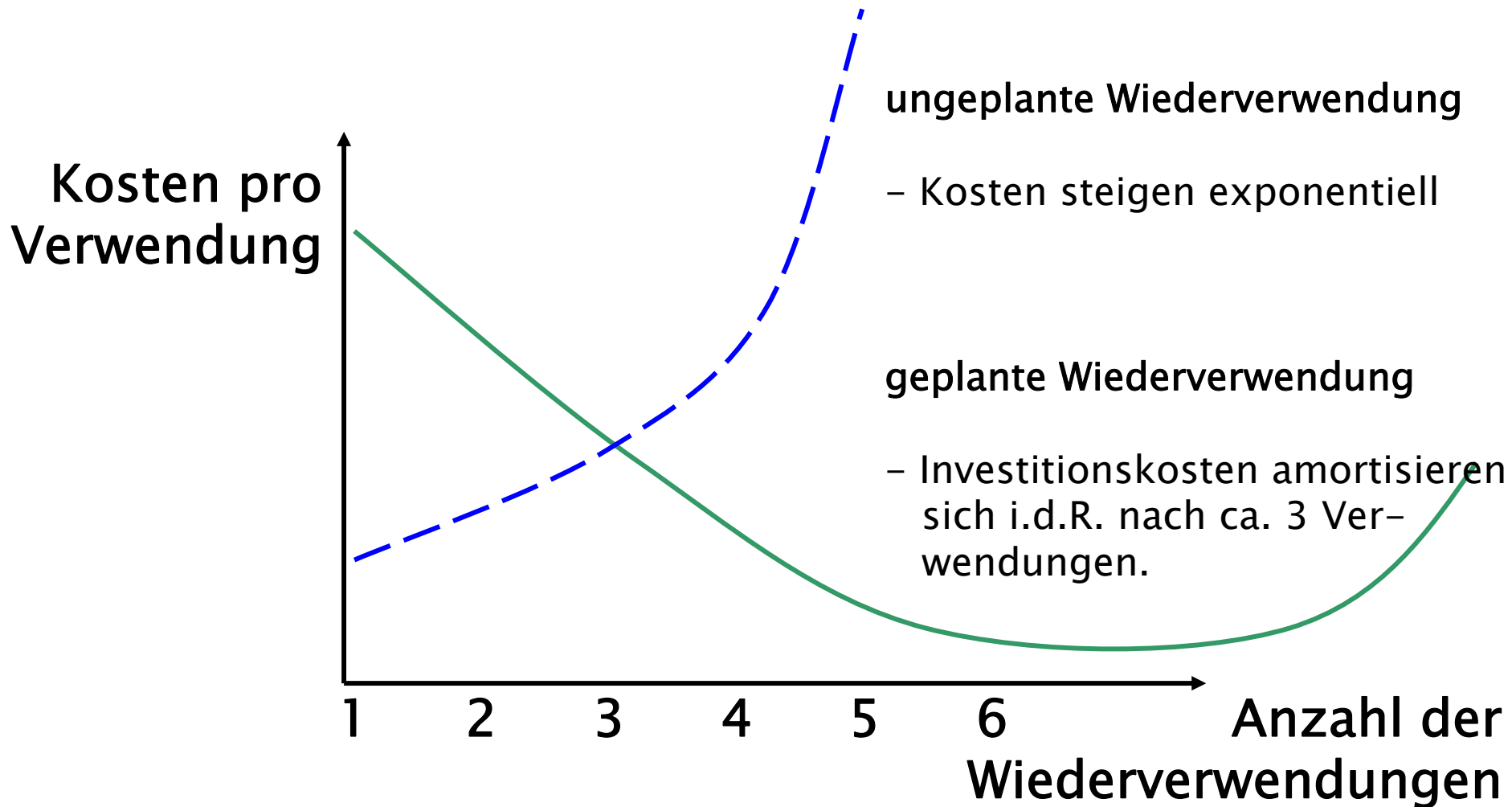
Der Einsatz von Komponenten ist gleichbedeutend mit der Standardisierung von Bauelementen.



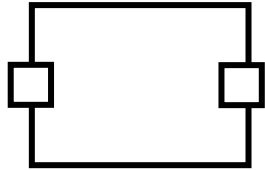
Die Kosten der Wiederverwendung steigen überproportional mit dem Nutzen



Positiver Return Of Investment nach drei Wiederverwendungen



Was ist eine Komponente?



- Eine **Software-Komponente** ist eine Einheit der Komposition mit durch einen Kontrakt spezifizierten Schnittstellen und nur expliziten Kontext-Abhängigkeiten. Eine Software-Komponente kann unabhängig verteilt werden und zur Komposition durch Dritte verwendet werden. [Szyperski, Pfister, ECOOP 96]

Schnittstelle

- Eine **Schnittstelle** beschreibt die Verwendung der Komponente in einem Produktiv-System.
- **Aspekte:**
 - Die Beschreibung der Schnittstelle enthält eine **funktionale Spezifikation** der von der Komponente angebotenen Operationen, Konstanten, Signale, ...
 - d.h. Signatur und möglichst eine Verhaltensbeschreibung (Vor- Nachbedingungen, Zustandsmaschinen, ...)
 - **Beispiel:** Eine Keller-Komponente bietet die Methoden push und pop an und es muss klar sein, dass pop(push(o)) wieder o zurückgibt. Das kann auch eine Bedingung an die Speicherschnittstelle sein, welche von der Komponente importiert wird.
 - Außerdem kann/sollte die Beschreibung der Schnittstelle Angaben über die **nichtfunktionalen Eigenschaften** machen inklusive von Verteilung, Konfiguration, Installation der Komponente.
 - Neben den in der Schnittstelle beschriebenen Angeboten der Komponente muss auch die **Umgebung** (auch Kontextabhängigkeit genannt) spezifiziert werden. Das sind typischerweise die von der Komponenten benötigten Operationen sowie nichtfunktionale Angaben inkl. Verteilung, Installation und Aktivierung von Komponenten.
 - Die Schnittstelle und Kontextbeschreibung dienen als **Vertrag zwischen Entwickler und Kunde**.

Gebräuchliche Komponentenarchitekturen

- **JavaBeans**
- **Enterprise JavaBeans**
- **CORBA Component Model**

Außerdem:

- **COM, DCOM, ActiveX, .Net (aus der Microsoft-Welt)**
 - nicht portabel, proprietär, aber .Net sehr leistungsfähig
- **Hersteller-spezifische Speziallösungen**
 - **GUI-Builder Klassenbibliotheken**

JavaBeans

“A Java Bean is a reusable software component that can be manipulated visually in a builder tool.” [Sun 97]

- **JavaBeans** ist eine Schnittstellen-Spezifikation für wiederverwendbare Software-Komponenten in Java
 - definiert die Java-Komponenten
 - und wie sie zusammenarbeiten.
- Eine Bean ist eine beliebige **Java-Klasse**, die den JavaBeans-Konventionen folgt.
- Beans kommunizieren typischerweise über Ereignisse und bieten zusätzliche **Methoden zur Ermittlung und Einstellung ihrer Eigenschaften** an (Introspection, Property Sheet).
- Viele JavaBeans können als **grafische Komponenten** mit GUI-Editoren interaktiv zu Applikationen zusammengesetzt werden. Andere JavaBeans haben **keine sichtbare Oberfläche**, z.B. wenn sie in JSPs (Java Server Pages) verwendet werden.

JavaBeans Beispiel: SimpleBean

```
public class SimpleBean extends Canvas implements Serializable
{
    private Color color = Color.green;
    public Color getColor() { return this.color; }
    public void setColor (Color c) { this.color = c; }

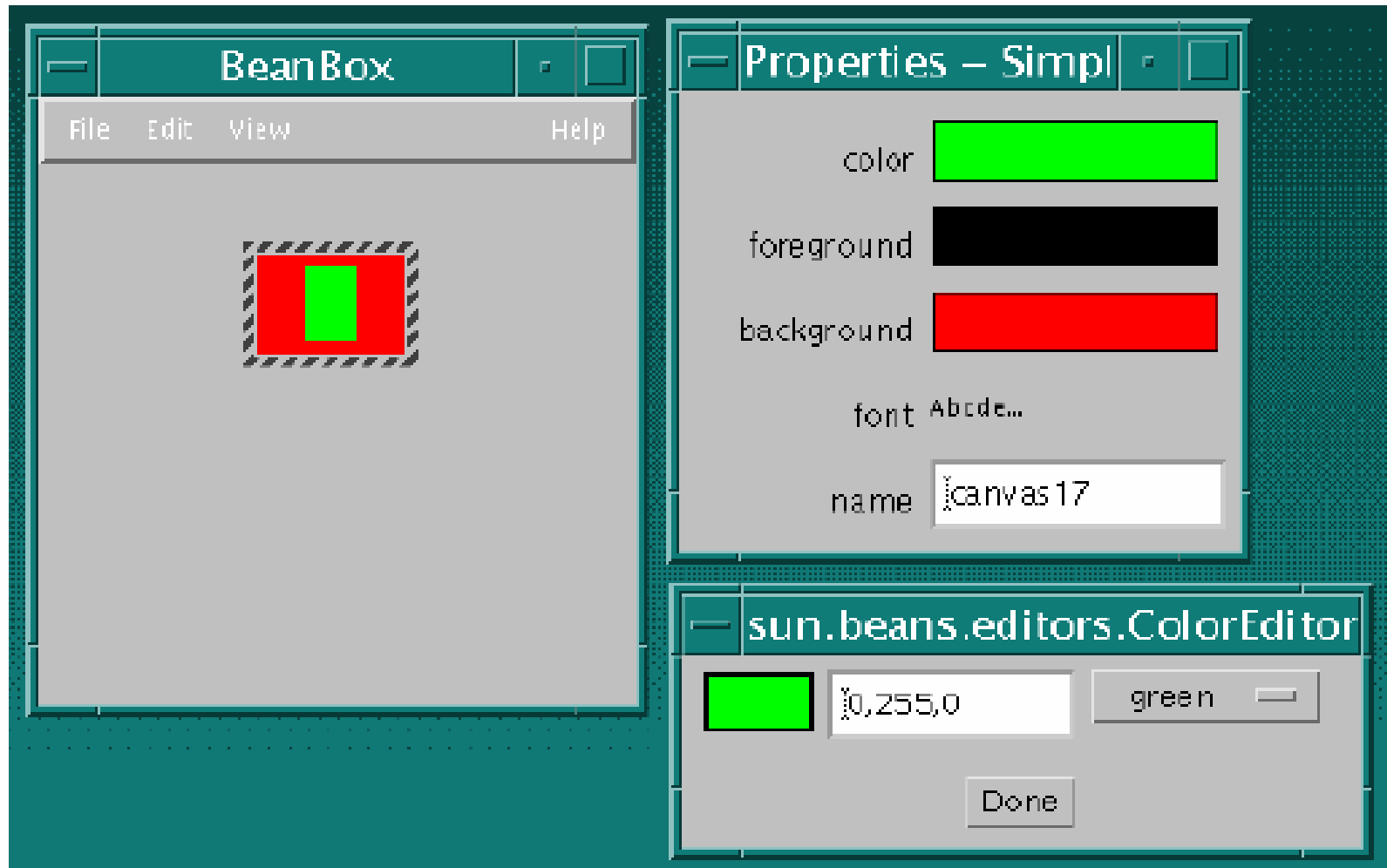
    public void paint(Graphics g) {
        g.setColor(color);
        g.fillRect(20, 5, 20, 30);
    }

    //Constructor: sets inherited properties
    public SimpleBean() {
        setSize(60,40);
        setBackground(Color.red);
    }
}
```

JavaBeans Beispiel: SimpleBean

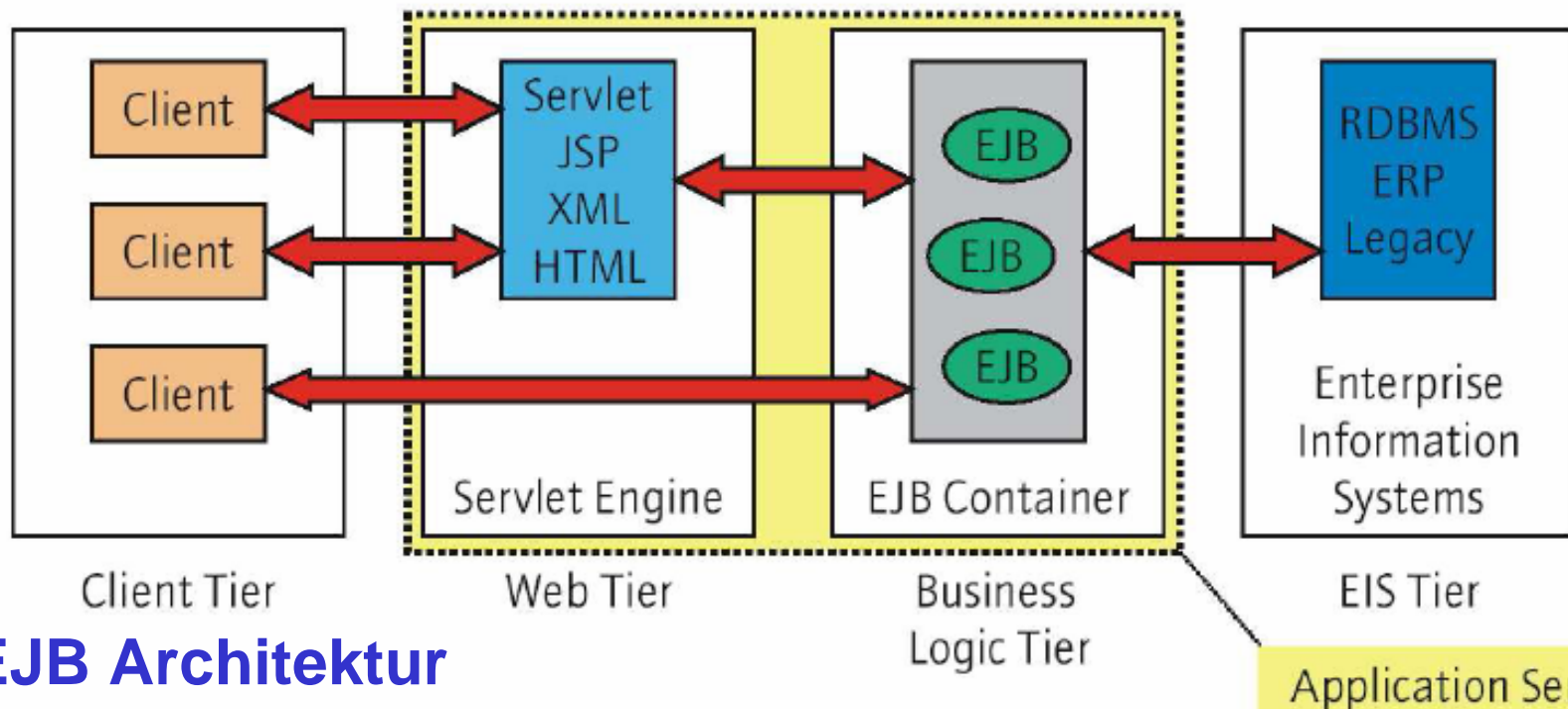
Property Color der Bean "SimpleBean,,:

SimpleBean wird in BeanBox geladen, Feld color anklicken öffnet ColorEditor



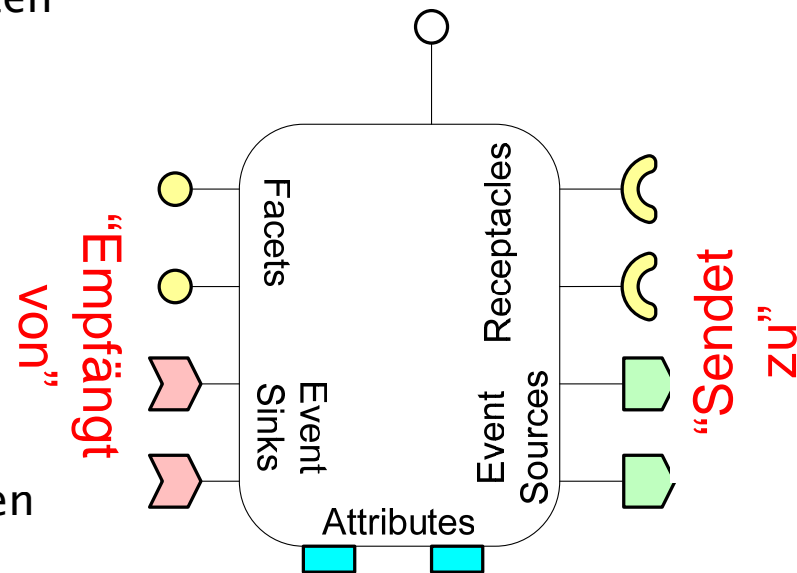
Enterprise JavaBeans (EJB)

- Enterprise JavaBean ist eine Spezifikation zur Entwicklung und Verteilung von komponenten-basierten verteilten Geschäftslogik-Anwendungen.
- EJBs sind gemanagte Objekte und benötigen eine spezielle Laufzeitumgebung (EJB-Container) in einem EJB-kompatiblen Java Application Server.
- EJBs bedingen einen höheren Einarbeitungsaufwand als JavaBeans, bieten dafür aber genau definierte Strukturen und Aufgabenteilungen.



CORBA Komponentenmodell

- Das CORBA Component Model (CCM) ist ein auf CORBA aufsetzendes Komponentenmodell.
- Das CORBA-Komponentenmodell führt den Metatyp CORBAComponent in CORBA ein.
- Eine CORBA-Komponente kapselt ihren inneren Aufbau durch Interfaces, die über Ports angeboten werden. Portarten sind u.a.
 - **Event Source**, **Event Sink** zum Senden und Empfangen von Ereignissen
 - **Facet** zum Anbieten von Schnittstellen und **Receptacle** zum Zugriff auf andere Komponenten
- **Attribute** einer Komponente dienen zu Konfigurationszwecken.
- Die Laufzeitumgebung von CORBA-Komponenten ist der **Container**. In erster Linie verbirgt der Container die Heterogenität der benutzten Hard- und Software.

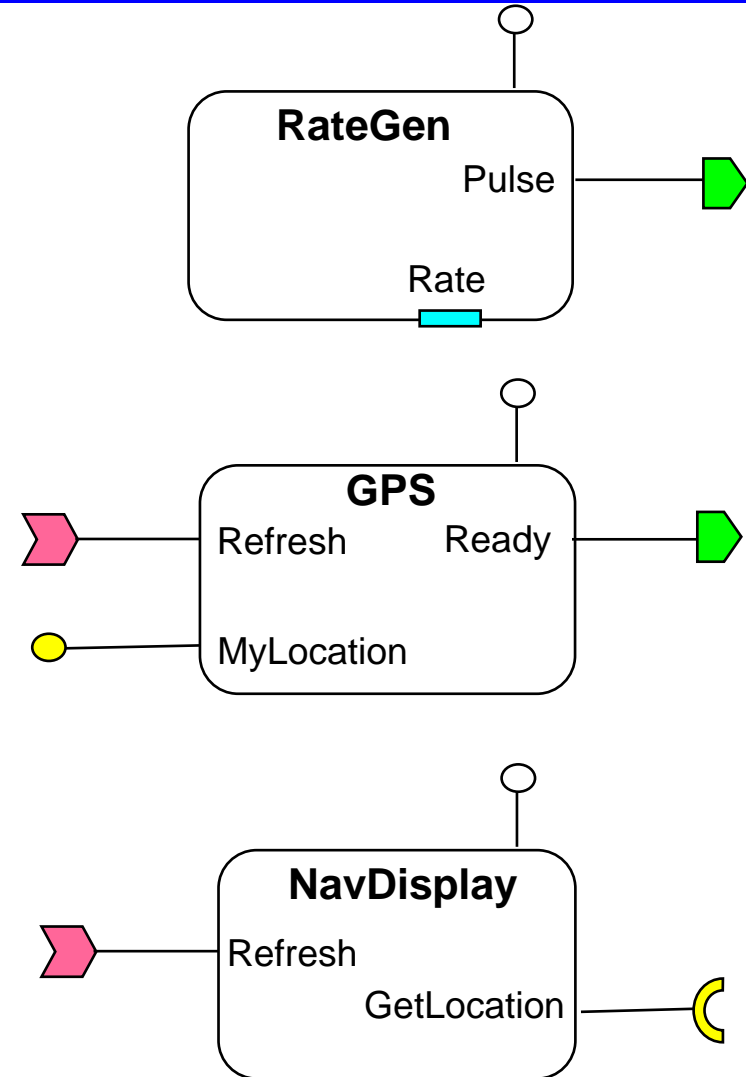


CCM Beispiel: Impulsgeber, GPS und Navigationsanzeige

```

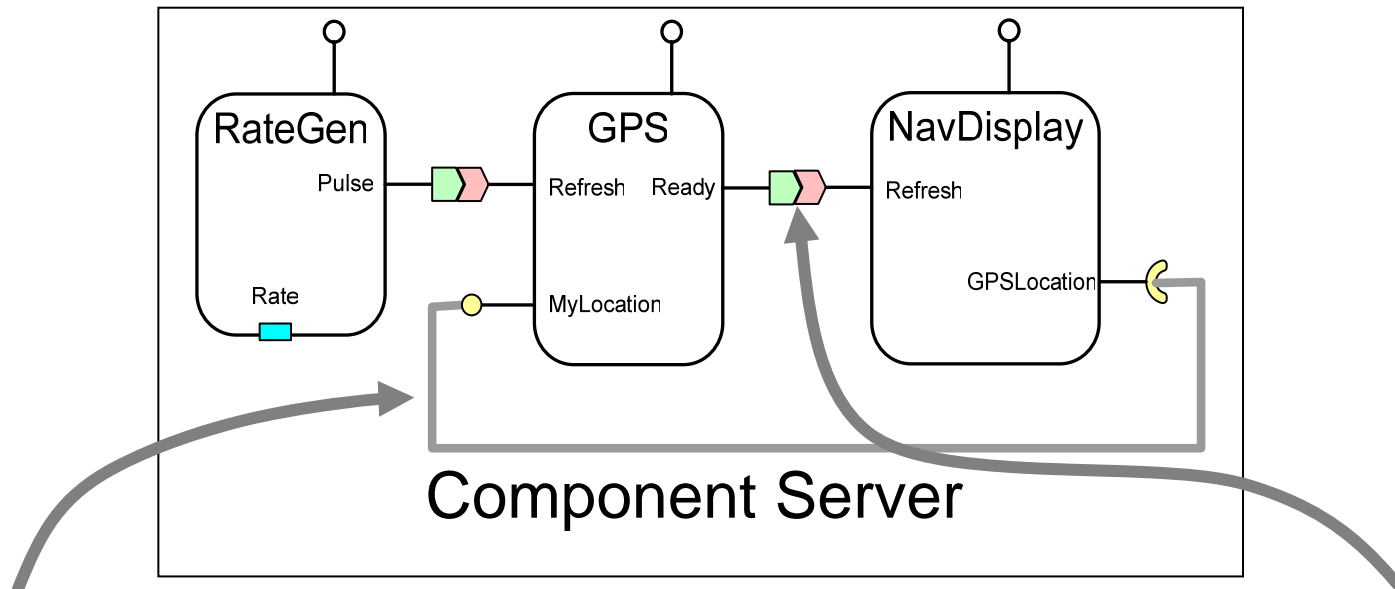
typedef unsigned long rateHz;
eventtype tick
{ public rateHz Rate; };
component RateGen
{ publishes tick Pulse;
  attribute rateHz Rate;
  ...
};
interface position
{ long get_pos (); };
component GPS
{ provides position MyLocation;
  publishes tick Ready;
  consumes tick Refresh;
  ...
};
component NavDisplay
{ uses position GPSLocation;
  consumes tick Refresh;
  ...
};

```



Example of Connecting Components

- CCM Komponenten werden in der Initialisierungsphase verbunden:



- **Facet → Receptacle**

```
objref = GPS->provide
("MyLocation");
NavDisplay->connect
("GPSLocation", objref);
```

- **Event Source → Event Sink**

```
consumer = NavDisplay->
get_consumer ("Refresh")
GPS->subscribe
("Ready", consumer);
```

- **Verbundene Objektreferenzen werden in Containers verwaltet.**

© D. Schmidt

Themen heute

- Einführung Komponenten
- **ADL-Einführung**
 - Definition
 - Historischer Überblick
 - Konzepte
- Java/A

Architecture Description Languages (ADLs)

Architektur-Beschreibungssprachen

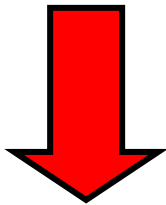
- Eine Architektur-Beschreibungssprache (Architecture Description Language, ADL)
 - beschreibt die Struktur eines Systems auf einer hohen Abstraktionsebene
 - um sie mit formalen Methoden quantitativ oder qualitativ zu untersuchen,
 - sowie zu simulieren oder Code zu erzeugen,
 - und so Aussagen über ein existierendes, oder Vorhersagen über zu bauende Systeme zu liefern.
 - Dabei werden insbesondere die Bausteine/Subsysteme, ihr Verbindungen und ihr Verhalten betrachtet.
- Typischerweise enthält eine ADL Konzepte für
 - Komponenten, Konnektoren, Schnittstellen (oder Ports) und Konfigurationen
- Diese Forschungsrichtung kam Mitte der 80´er Jahre im akademischen Milieu auf, lange bevor der Begriff in der allgemeinen Diskussion verbreitet war.

Beispiele für Architekturbeschreibungssprachen

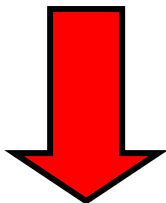
ADL	Ursprung	Urheber	semantischer Formalismus
SARA	1986	Estrin et al.	Petrinetze
Wright	1997	Allen	CSP
Rapide	1995	Luckham	PoSets
Darwin	1996	Kramer, Magee	CCS/ π -Kalkül & C/Java
ROOM	1995	Selic	C-Code
UML-artig	1999	div.	Petrinetze, Maude, ...
Java/A	2002	Hacklinger	Java-Code

ADL-Ansatz

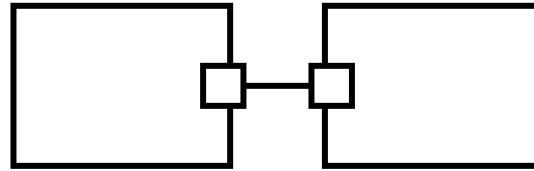
konkrete Syntax



abstrakte Syntax
Begriffe/Konzepte
Metamodell



Semantik



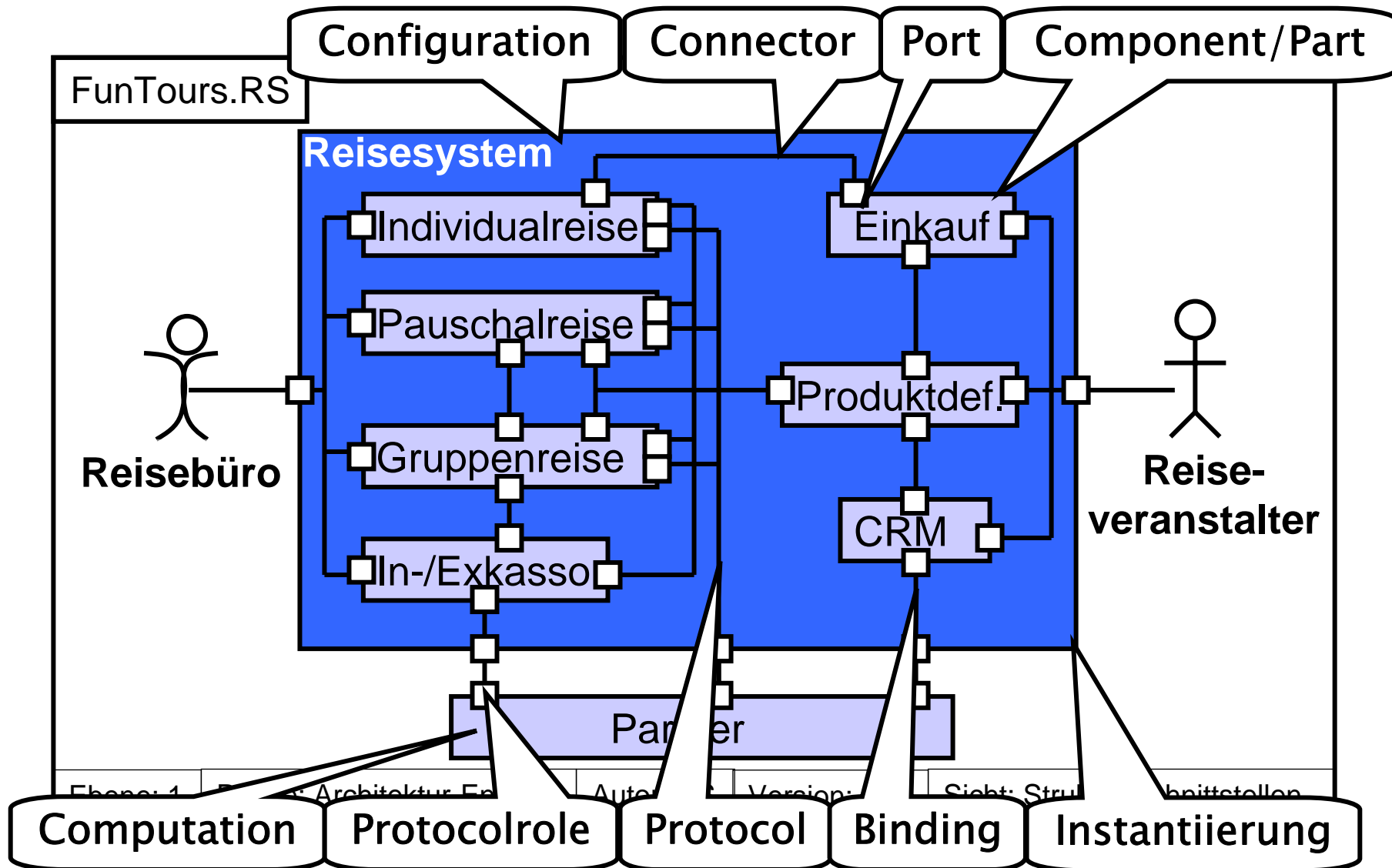
- + Klassendiagramme
- + Interaktionsdiagramme
- + Zustandsautomaten
- + Aktivitätsdiagramme
- + ...

Traditionell unterschiedliche in jeder ADL
Ansätze zur Vereinheitlichung (ACME, ...)

neuerdings auch: UML2

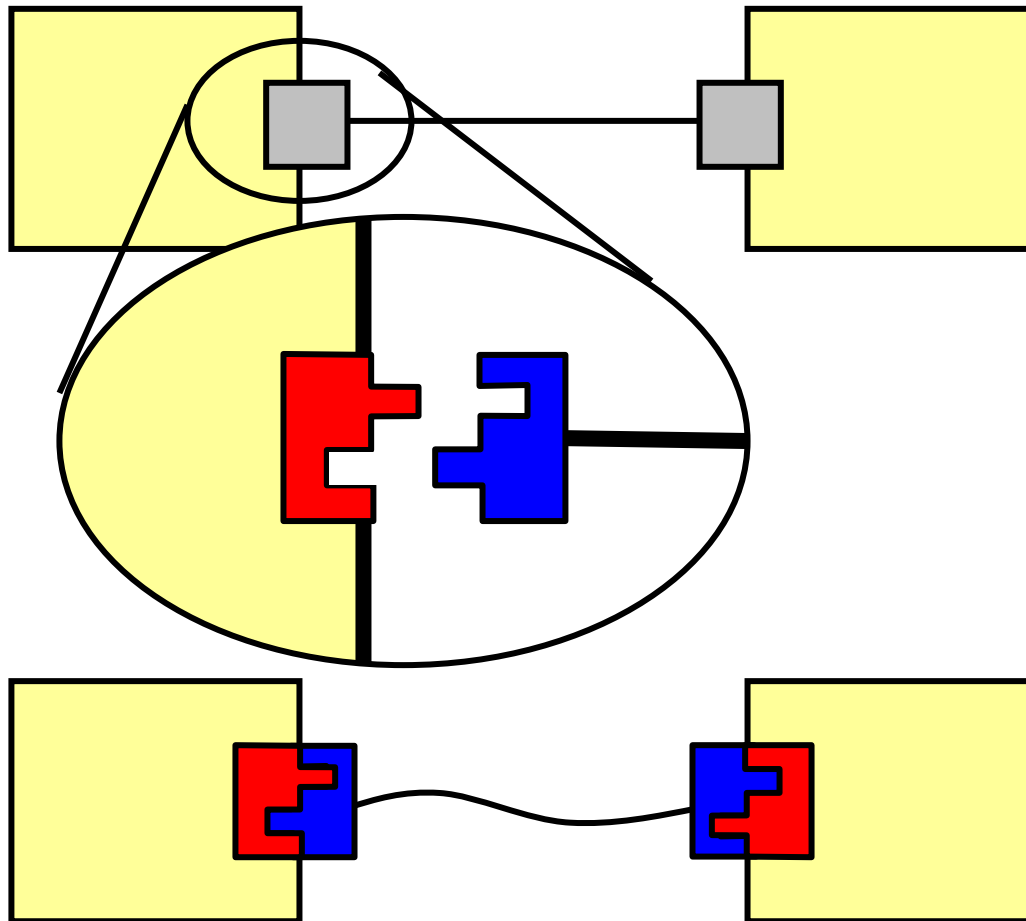
Nahezu alle Formalismen sind verwendet worden,
solange sie Verhalten und Nebenläufigkeit aus-
drücken können (PN, CSP, CCS/ACP, Logiken, ...)

ADL-Konzepte (Überblick)



Weitere ADL-Konzepte: Protokollrolle

Damit Konnektoren nur zueinander passende Ports verbinden, müssen ihre Enden „getypt“ sein.

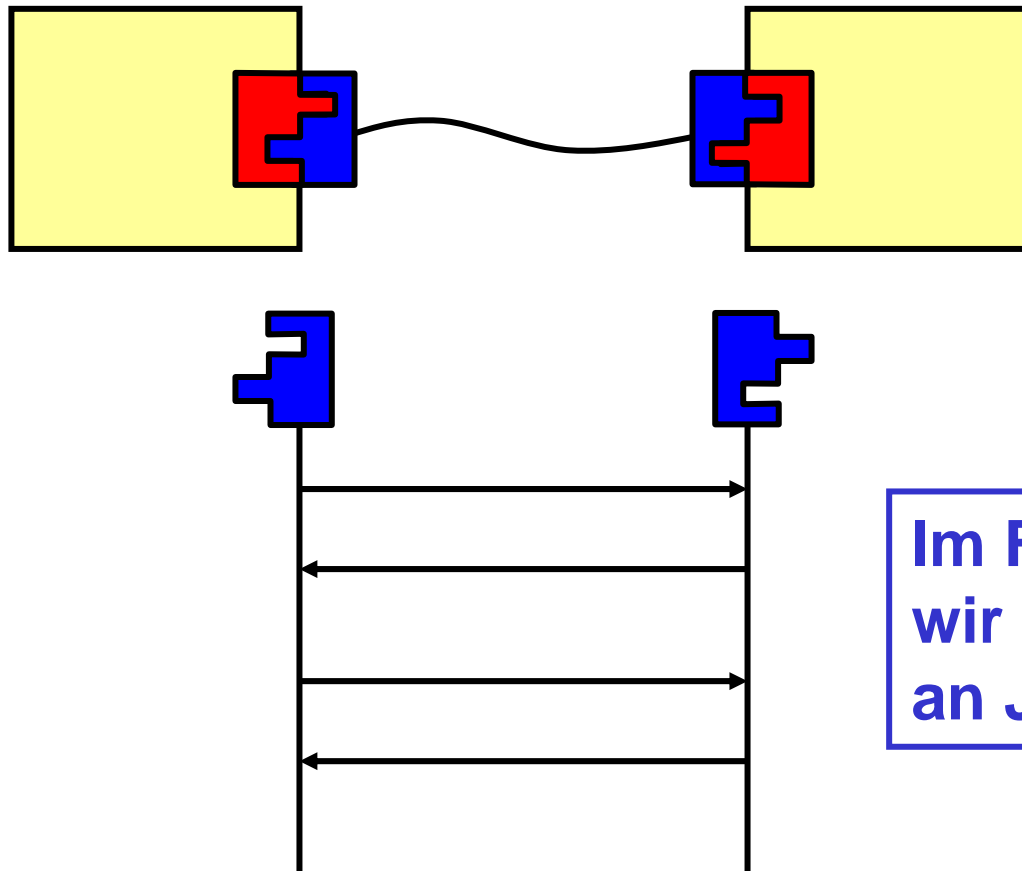


Weitere ADL-Konzepte: Protokollrolle

- Der „Typ“ eines Ports oder eines Konnektor-Endes heißt Protokollrolle.
- Protokollrollen werden beschrieben durch je
 - eine Menge eingehender Signale,
 - eine Menge ausgehender Signale, und
 - ein Verhalten bezüglich dieser Signale.
- Diese Elemente können z.B. durch zwei Interfaces und einen Zustandsautomaten spezifiziert werden.
- Das Zusammenspiel verschiedener Protokollrollen heißt Protokoll.

ADL-Konzepte: Protokoll

Ein Konnektor verbindet Rollen.
Wie beschreibe ich die Interaktion der Rollen?



Im Folgenden zeigen wir die ADL-Konzepte an Java/A

Themen heute

- Einführung Komponenten
- ADL-Einführung
- Java/A

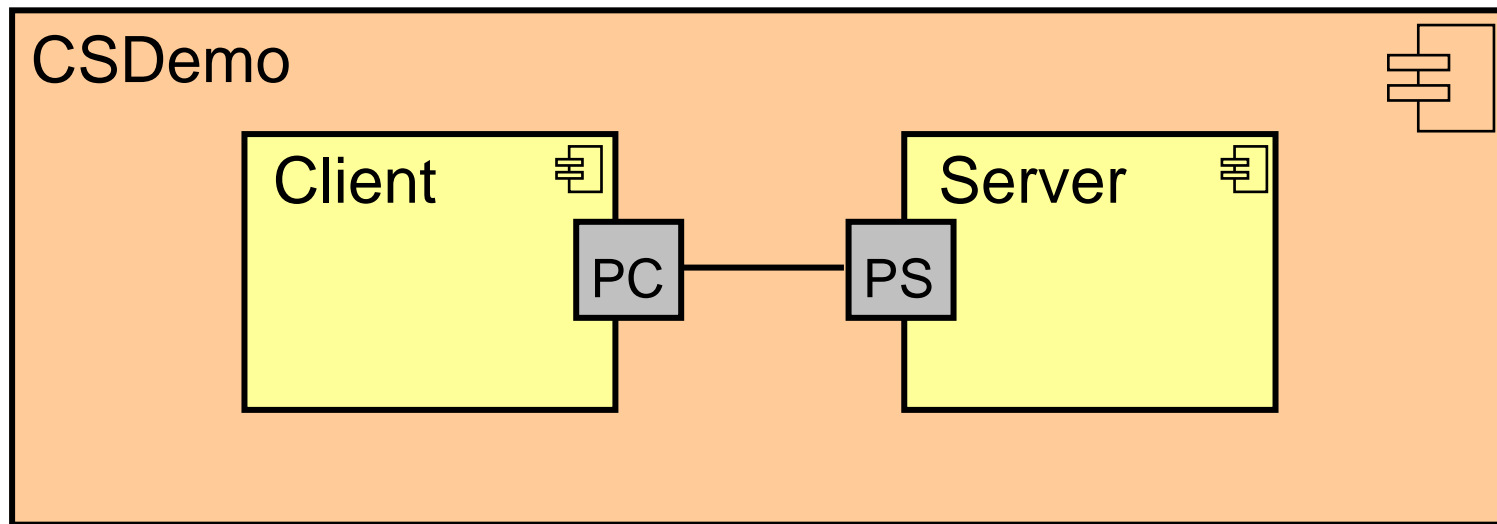
Java/A – „echte“ Komponenten in Java

- Programmiersprache auf Java basierend (Entwicklung an der LFE PST seit 2002)
- mit Compiler, Framework und IDE
- enthält Konstrukte zur Darstellung von
 - Komponenten mit Ports
 - Konfigurationen
- Ports beschreiben
 - angebotene und benötigte Schnittstellen
 - ein Protokoll
- Komponenten sind streng gekapselt und hierarchisch komponierbar

Ziel: Komponenten

- einfach zu implementieren
- wiederverwendbar
- austauschbar
- wartbar / änderbar

Java/A – das laufende Beispiel



- **Komponenten:**
 - einfach: Client, Server
 - hierarchisch: CSDemo
- **Ports: PC und PS**
- **Konnektor zwischen PC und PS**

Java/A – Protokolle (I)

- **Port besteht aus**
 - benötigten und angebotenen Schnittstellen (in Form von Methodendeklarationen)
 - einem Protokoll
 - mit UML-Zustandsmaschinen spezifiziert
 - im Java/A-Quellcode dargestellt durch UTE (UML Text)
- **Protokoll eines Ports beschreibt eine partielle Ordnung von Nachrichten, die ein Port empfängt bzw. sendet.**
- **Verbindung zweier Ports induziert Interaktion zwischen den Zustandsmaschinen:**
 - gibt es Deadlocks?
 - sind spezifische Abläufe möglich?
- **Verbindungen zweier Ports werden verifiziert durch den Modelcheckers HUGO (Integration im Compiler und der IDE)**

Java/A – Protokolle (II)



In diesem Beispiel ist offensichtlich kein Deadlock: Client sendet *request* wartet auf *reply*, der Server wartet auf *request* und sendet danach *reply*.

Java/A – Protokolle (III): UTE

UTE (UML Text) ist eine textuelle Darstellung von UML Zustandsmaschinen.

Das Protokoll des Ports Client.PC:

```
behaviour {
    states {
        initial Initial;
        simple idle;
        simple waiting;
    }
    transitions {
        Initial -> idle;
        idle -> waiting { effect ^request(); }
        waiting -> idle { trigger reply; }
    }
}
```

Java/A – Codebeispiel: *Client*

```
import java.io.*;
simple component Client {
    port PC {
        provided { void reply(); }
        required { void request(); }
        protocol <! ... !>
    }
    void reply() implements PC.reply() {
        System.out.println("Received reply.");
    }
    void start() {
        BufferedReader br = new BufferedReader(new
        InputStreamReader(System.in));
        try {
            while (!br.readLine().equals("quit")) {
                PC.request();
            }
            ...
        }
    }
}
```

Java/A – Codebeispiel *CSDemo*

```
composite component CSDemo {
  assembly {
    components { Client, Server }
    connectors { Client.PC, Server.PS; }

    initial configuration {
      component Client c = new Client();
      component Server s = new Server();
      // provided by the Java/A-framework:
      connector Connector cn = new Connector();
      cn.connect(s.PS, c.PC);
    }
  }
  void start() {
    c.start(); // Start the computation of the client
  }
}
```

Zusammenfassung

- **Komponenten sind abgeschlossene, wiederverwendbare Einheiten.**
- **Gebräuchliche Komponentenansätze sind**
 - JavaBeans, EJB, CCM sowie aus der Microsoft-Welt
 - COM, ActiveX, .Net
- **Architecture Description Languages (ADL) sind formale Sprachen zur Beschreibung von Software-Strukturen.**
- **Bekannte ADLs sind Wright, Darwin, ROOM, UML**
- **Java/A**
 - Programmiersprache integriert mit UML und formaler Analyse
 - zur Zeit noch einfache Konnektoren
 - erste Ansätze zu dynamischen Architekturen und Rekonfiguration
- **Ziel: Modellierung, automatische Analyse, Codegenerierung für Software-Architekturen und dynamische Rekonfiguration**
(hier sind Fortgeschrittenenpraktika und Diplomarbeiten zu vergeben)

Literatur für B.5

- G. Estrin, R.S. Fenchel, R. R. Razouk, M. K. Vernon: SARA: Modeling, Analysis, and Simulation Support for Design of Concurrent Systems. IEEE T_{xn} SE, Vol se-12, No 2, Feb. 1986, pp. 293-311
- R.J. Allen: A Formal Approach to Software Architecture. CMU-CS-97-144 (-> Wright)
- C. Hofmeister, R. Nord, D. Soni: Applied Software Architecture. Addison-Wesley, 2000
- H. Störrle: Models of Software Architecture. Book-on-demand, 2001
- F. Hacklinger: Java/A – Taking Components into Java. IASSE 2004: 163-168