Vorlesung „ Methoden des Software Engineering“

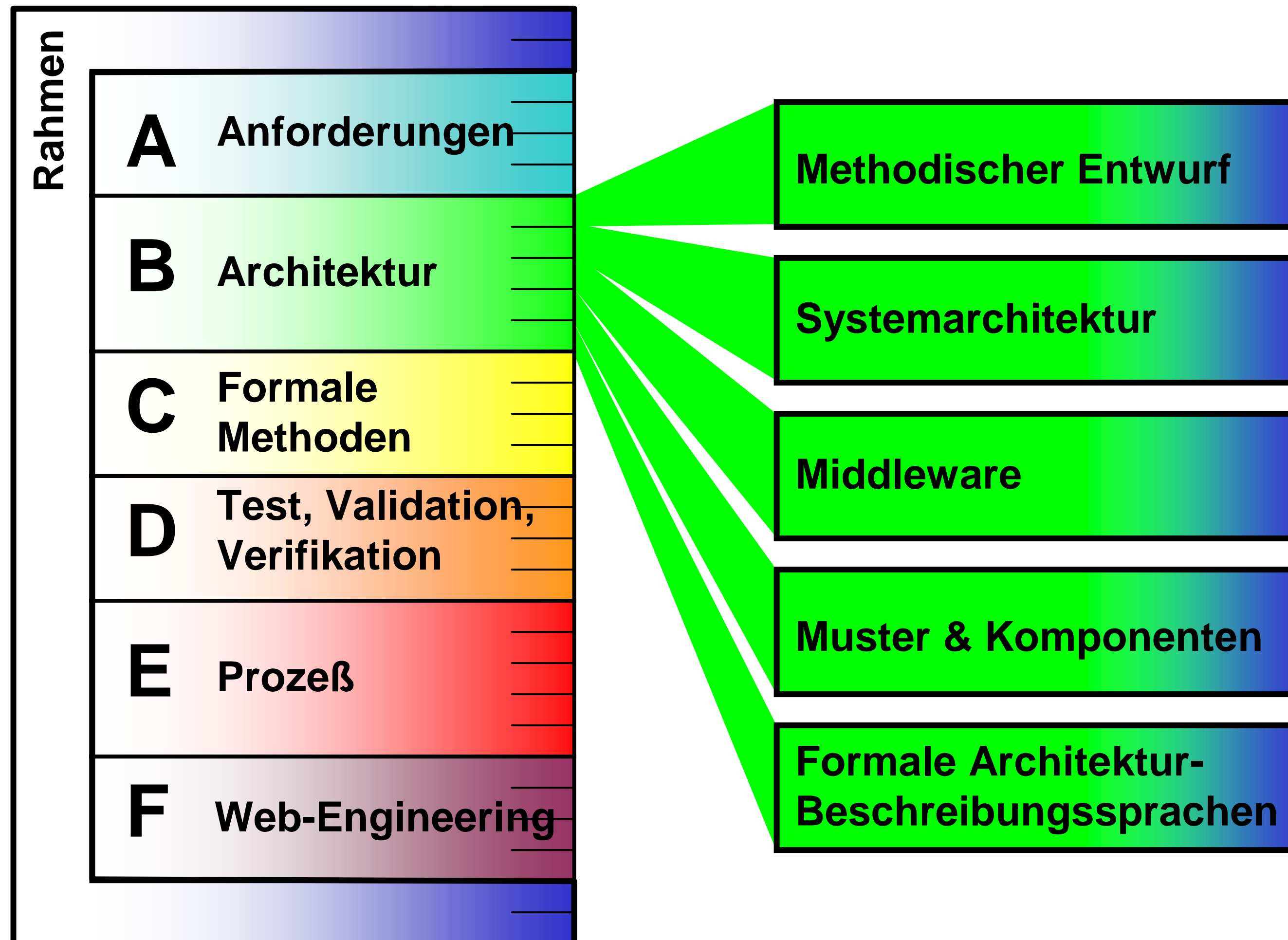
Block B „Software Architektur“

Architekturbeschreibungssprachen

Martin Wirsing

Einheit B.5, 25.11.2004

Gliederung Block B



Kernpunkte heute

- 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.
- Es gibt auch wissensbasierte Ansätze, Architektur-Modelle zu untersuchen.
- Wir suchen ständig Studenten, die in diesem Bereich mitarbeiten wollen.

Themen heute

- ADL-Einführung
  - Definition
  - Historischer Überblick
  - Konzepte
- Wright
- 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 zu mit formalen Methoden quantitativ oder qualitiativ 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.

- Diese Forschungsrichtung kam Mitte der 80´er Jahre im akademischen Milieu auf, lange bevor der Begriff in der allgemeinen Diskussion verbreitet war.

Wiederholung: Was ist Software-Architektur?

„Architecture is defined [...] as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution." (IEEE Architecture Working Group, P1471)

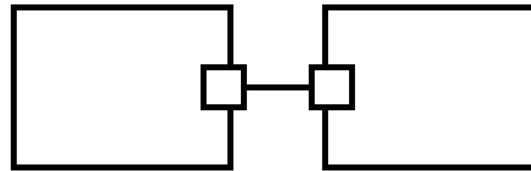
- Darin werden drei Aspekte besonders betont:
  - abstrakte Systemstrukturen und -elemente, und
  - Entwicklungsprinzipien bzw. Systemzweck („Rationale").
- Nicht betrachtet werden hingegen
  - Einbettung in den Kontext und die vorhandene Infrastruktur, und
  - die Betroffenen („Stakeholder"), insbesondere der Architekt.
- Diese Definition ist offenbar sehr stark von der Sichtweise von Software-Architektur beeinflusst, die aus dem akademischen Umfeld stammt.

Beispiele für Architekturbeschreibungssprachen

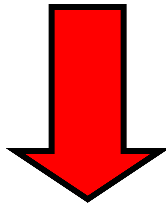
ADL	Ursprung	Urheber	semantischer Formalismus
SARA	1986	Estrin et al.	Petrinetze
Wright	1997	Allen	CSP
Rapide	1995	Luckham	PoSets
Darwin	1996	Kramer, Magee	CCS/π-Kalkül & C/Java
ROOM	1995	Selic	C-Code
UML-artig	1999	div.	Petrinetze, Maude, ...
Java/A	2002	Hacklinger	Java-Code

ADL-Ansatz

konkrete Syntax



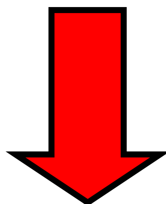
+ Klassendiagramme
+ Interaktionsdiagramme
+ Zustandsautomaten
+ Aktivitätsdiagramme
+ ...



abstrakte Syntax
Begriffe/Konzepte
Metamodell

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

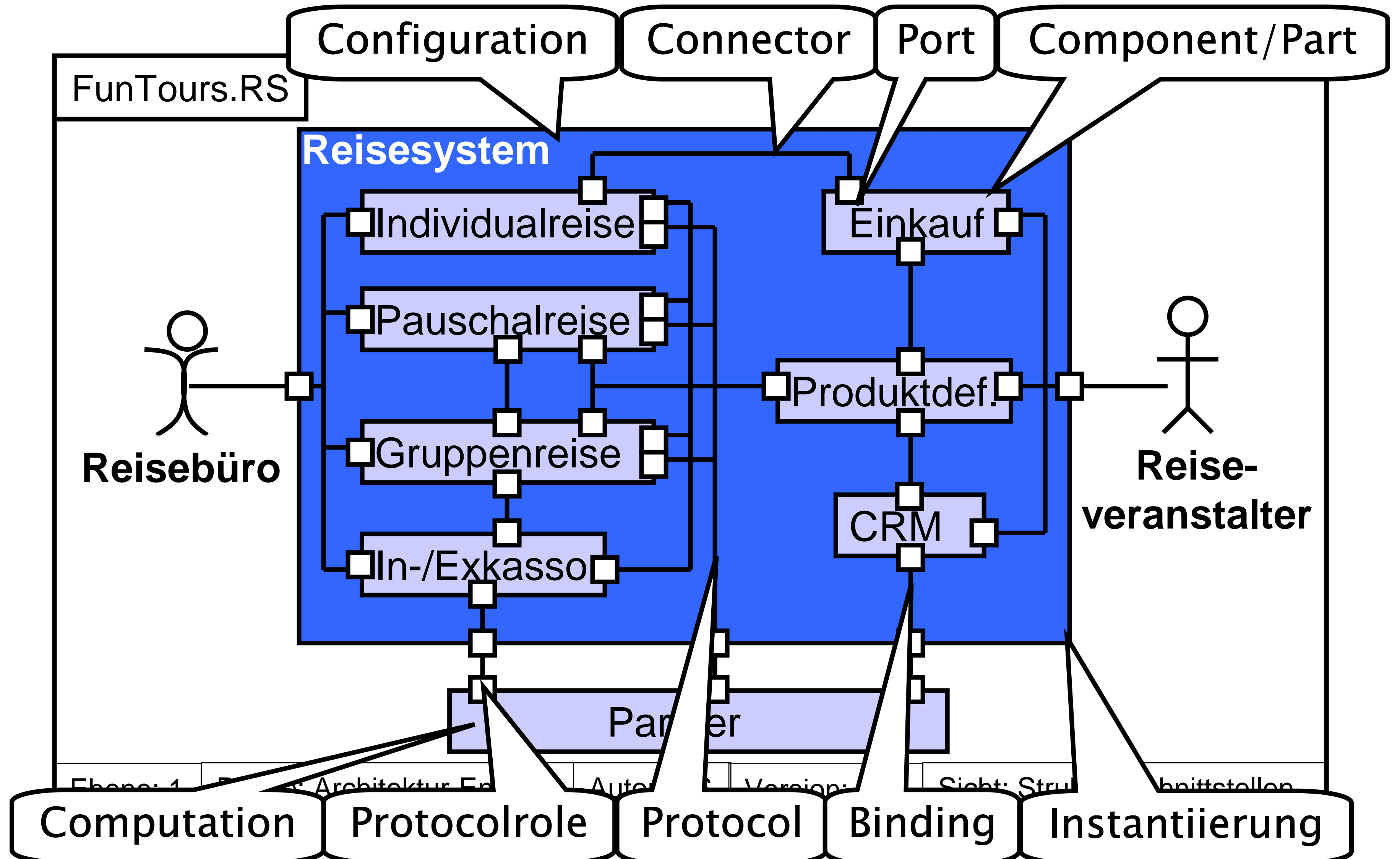
neuerdings auch: UML2



Semantik

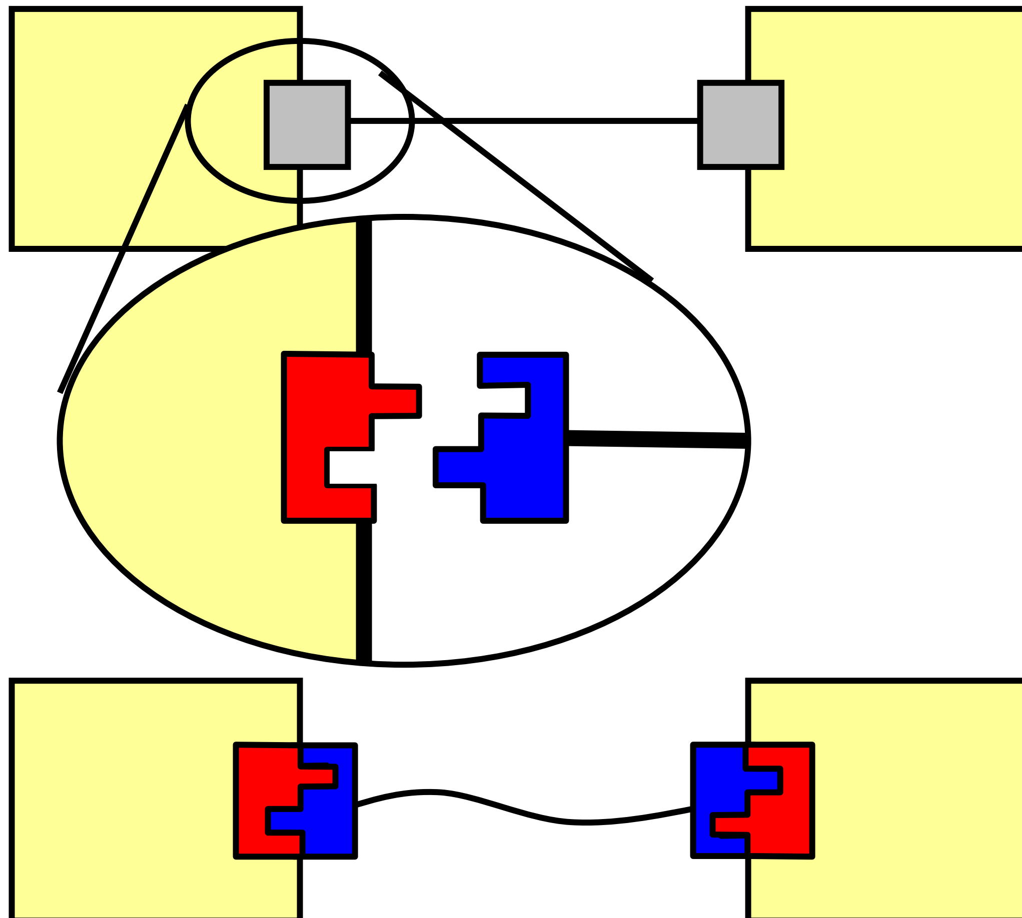
Nahezu alle Formalismen sind verwendet worden, solange sie Verhalten und Nebenläufigkeit ausdrücken können (PN, CSP, CCS/ACP, Logiken, ...)

ADL-Konzepte (Überblick)



ADL-Konzepte: Protokollrolle

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

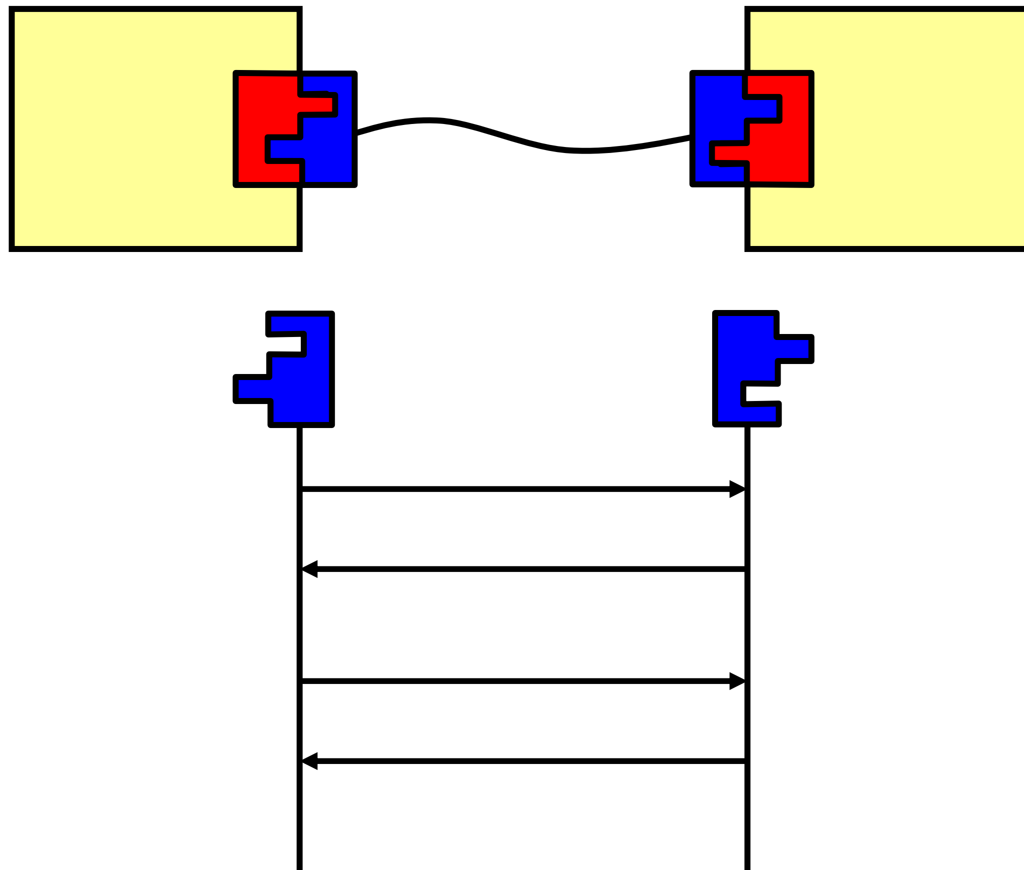


ADL-Konzepte: Protokollrolle

- 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?

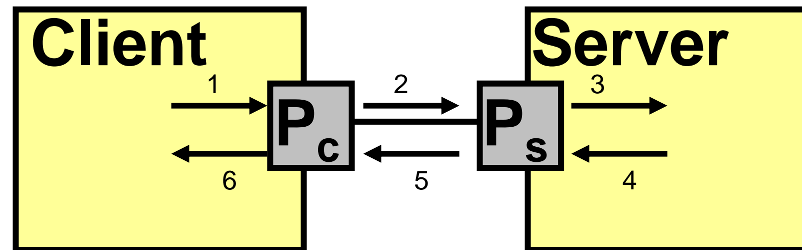


Themen heute

- ADL-Einführung
  - Definition
  - Historischer Überblick
  - Konzepte

- Wright
- Java/A

Laufendes Beispiel



P_c P_s

idle — **request** → idle

waiting ← **reply** — busy

idle idle

idle

snd(put) **rcv(reply)**

rcv(get) **snd(request)**

waiting

idle

snd(reply) **rcv(get)**

rcv(request) **snd(put)**

busy

Wright

- Architekturbeschreibungssprache
- Entworfen von Robert Allen, CMU 1997
- Dient zur Beschreibung von Architekturkonfigurationen und Architekturstilen
- Beschreibt eine Architekturkonfiguration
  - als eine Menge von Komponenten und Konnektoren,
  - deren Verhalten formal in CSP spezifiziert wird.
- Bemerkung: CSP (Communicating Sequential Processes) ist eine Sprache zur Beschreibung des Verhaltens und der Interaktionen nebenläufiger Systeme, eingeführt 1985 von Tony Hoare.

Wright-Komponenten

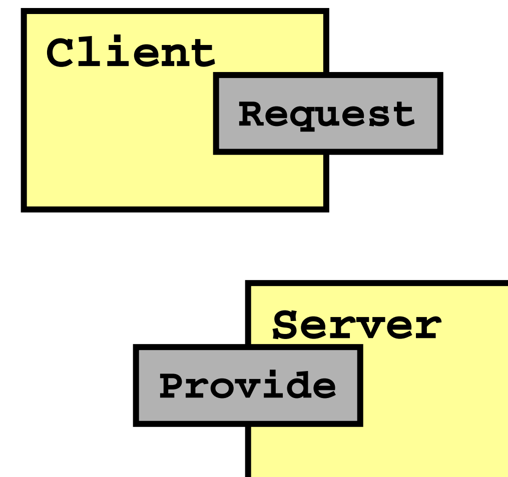
Eine Komponente

- beschreibt (den Typ) einer lokalen unabhängigen Berechnung
- Ist gegeben durch
  - eine **Schnittstelle**, die aus einer Menge von Ports besteht, und
  - eine **Berechnung**, die das Verhalten der Komponente beschreibt

Beispiel:

```
component Client
      port request = [ request protocol]
      computation  = [ some computation ]

component Server
      port provide = [ provide protocol]
      computation  = [ some computation ]
```



Wright-Konnektoren

- Ein Konnektor
  - beschreibt (den Typ) einer Interaktion zwischen einer Menge von Komponenten
  - Ist gegeben durch
    - eine **Menge von Rollen** (sogenannte Protokollrollen) und
    - ein **Protokoll** (engl. “glue”), das das Verhalten der Interaktion festlegt.
- Beispiel:

Beschreibt das von client erwartete Verhalten

```
connector C-S-connector
    role client   = [ client protocol]
    role server   = [ server protocol]
    glue          = [ glue protocol]
```

Laufendes Beispiel

```
component Client  =
  port request = [ request protocol]
  computation  = [ some computation ]
```

```
component Server =
 port provide = [ provide protocol]
 computation  = [ some computation ]
```

```
connector C-S-connector =
 role client   = [ client protocol]
 role server   = [ server protocol]
 glue          = [ glue protocol]
```

Wright-Konfigurationen

- Eine Konfiguration beschreibt eine Systemarchitektur
- Eine Konfiguration besteht aus
  - einer Menge von Komponenteninstanzen
  - verbunden durch Konnektoren,

    wobei
    - der Zugriff auf Port/Rolle durch die übliche "Dot"-Notation
    - die Bindung zwischen Port und Konnektorrolle als Umbenennung

    beschrieben wird.

- Beispiel

Instanzen (instances)

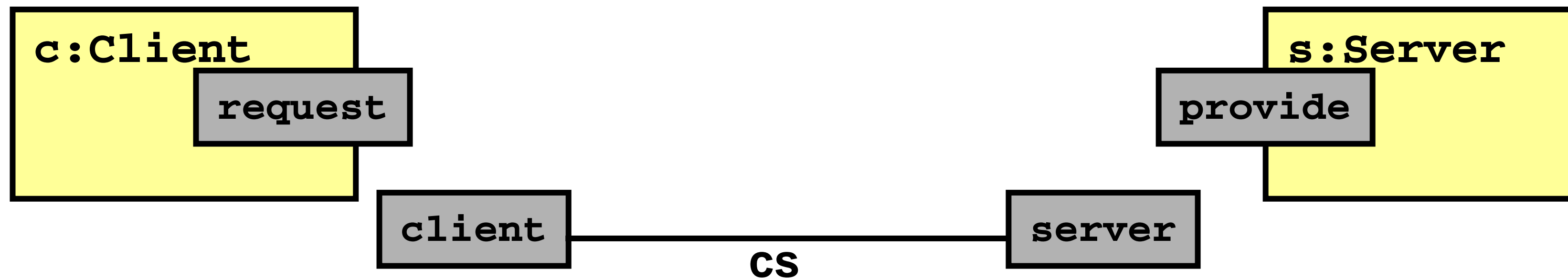
```
s: Server
c: Client
cs: C-S-connector
```

Verbindungen (attachments)

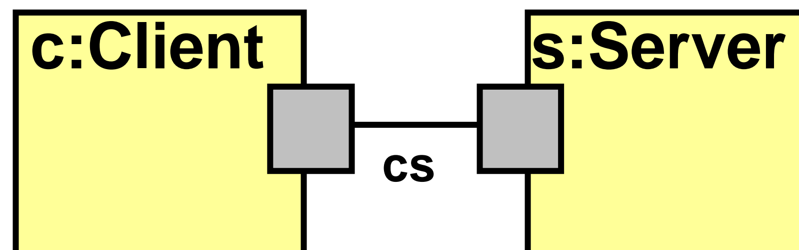
```
s.provide as cs.server
c.request as cs.client
```

Laufendes Beispiel

Instanzen von Komponenten und Konnektoren:



Konfiguration:



s.provide as **cs.server**

c.request as **cs.client**

Wright-Beispiel

Modell

```
Configuration CSExample
   component Server
        port provide = [ provide protocol]
        computation  = [ some computation ]
   component Client
        port request = [ request protocol]
        computation  = [ some computation ]
   connector C-S-connector
        role client  = [ client protocol]
        role server  = [ server protocol ]
        glue         = [ glue protocol]
```

Instances (Instanzen)

```
s: Server
c: Client
cs: C-S-connector
```

Attachments (Bindungen)

```
s.provide as cs.server
c.request as cs.client
```

CSP - Verhaltensbeschreibung

- Ein CSP-Programm besteht aus einer Menge von Prozessdeklarationen und einem Prozess.
- Ein Prozess wird beschrieben durch (Empfangs- und Sende-) Aktionen und Operationen zur Komposition von Prozessen.
- Ein Ereignis bezeichnet das individuelle Auftreten einer Aktion.
- Ein Prozess besteht semantisch aus einer Menge von möglichen Spuren (engl. Trace), wobei
  - eine Spur gegeben ist durch

    eine endliche Folge a1 … an von Ereignissen.

  Außerdem betrachtet man noch die Fehlerspuren (engl. failure trace), d.h. eine Spur a1 … am und einer Menge von (Fehler-) Ereignissen {b1,…,bk}, die nach Ausführung von a1 … am abgewiesen werden können.

  (Hier beschränken wir uns meist der Einfachheit halber auf Spuren.)

CSP - Verhaltensbeschreibung

- Aktionen und Ereignisse
  - Empfangsaktion wird beschrieben durch
    - `e bzw. e?` für Aktionen ohne Datenübertragung oder
    - e?x (d.h. Name, Fragezeichen, Variable)
      für Aktionen mit Datenübertragung
  - Sendeaktion wird beschrieben durch
    - $\overline{e}$ bzw. e! für Aktionen ohne Datenübertragung oder
    - `e!x` (d.h. Name, Fragezeichen, zu sendender Ausdruck)
      für Aktionen mit Datenübertragung
  - Die Aktion √ bezeichnet erfolgreiche Terminierung.

CSP - Verhaltensbeschreibung

- Prozesse
  - `STOP` Prozess, der nichts tut;
  - `e --> P` Prozess, der zuerst die Aktion e ausführt und sich dann wie P verhält;
  - `§ (= √ --> STOP)` Prozess, der sofort erfolgreich terminiert;
  - `P;Q` Prozess, der sich wie P verhält, bis P erfolgreich terminiert, und sich dann wie Q verhält;

  Beispiel: `(e --> f --> §); (g -->§) = (e --> f --> g -->§)`
  - `<Name> = <Prozessausdruck>` Prozessdeklaration;

  Beispiel:

  `P = e --> P` Prozess, der e unendlich oft ausführt;

  `P` **`where`** `P1,P2,…` Prozess mit Hilfsdeklarationen (Häufige Form eines CSP-Programms);

CSP - Verhaltensbeschreibung

- Auswahl
  - `P[]Q` externe Auswahl; d.h.
    die Umgebung entscheidet, ob P oder Q ausgewählt wird.

Beispiel: `e-->P [] f-->Q`

Prozess, der sich verhält wie P, wenn e (zuerst) empfangen wird, und wie Q, wenn f (zuerst) empfangen wird;

Dieser Prozess kann weder e noch f abweisen.

  - `P ⊓ Q` interne Auswahl, d.h.
    der Prozess entscheidet selbst, ob P oder Q ausgeführt wird.

Beispiel: `e!x-->P1 ⊓ f!y-->Q1`

Prozess, der

entweder x über e sendet und sich dann verhält wie P1,

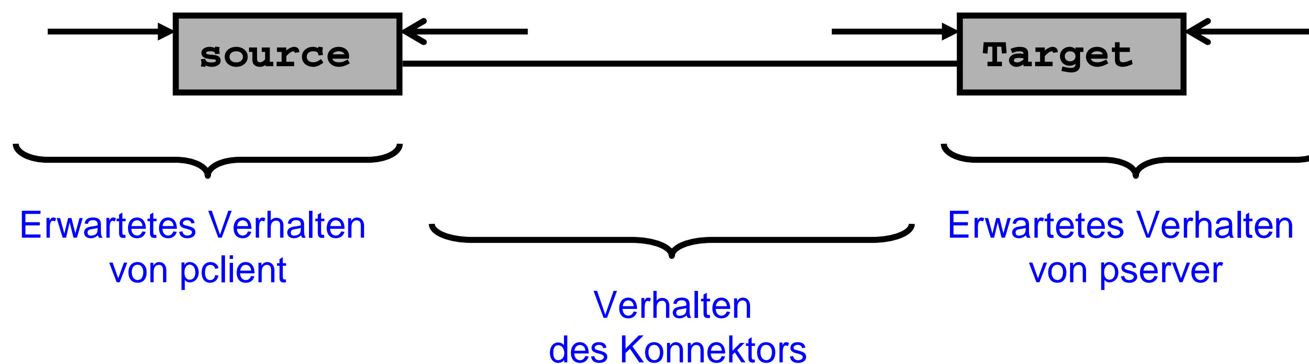
oder y über f sendet und sich dann verhält wie Q1.

Dieser Prozess kann sowohl e!x (wenn er sich für die Ausführung von f!y entscheidet) als auch f!y abweisen.

Wright – Verhaltensbeschreibung

Simple Pipe

```
connector pipe =
  role source = (in!x --> source) Π §
  role target = (out?x --> target) [] §
  glue   = (source.in?x --> target.out!x --> glue) [] §
```



Laufendes Beispiel

component Client

port request = (requ!v --> result?w --> request) Π §

computation = (request.requ!v --> request.result?w --> **computation)** Π §

component Server

port provide = (invoke?v --> return!w --> provide) [] §

computation = (provide.invoke?v --> provide.return!w --> **computation)** [] §

connector C-S-connector

role client = (requ!x --> result?y --> client) Π §

role server = (invoke?v --> return!w --> server) [] §

glue = (client.requ?x --> server.invoke!x -->

server.return?y --> client.result!y --> **glue**) [] §

CSP – Verhaltensbeschreibung

- Alphabet von P Menge der Namen von Aktionen in P

  Beispiel: `a1 --> a2 -->§` Alphabet: `{a1, a2}`

  `b --> c --> §` Alphabet: `{b, c}`

- **`P || Q`** Parallele Komposition

  P und Q werden parallel ausgeführt, wobei gleiche Aktionen koordiniert werden.

  In jedem Ablauf von P||Q können Aktionen von P und Q vorkommen. Beschränkt man einen Ablauf von P||Q auf das Alphabet von P, so ergibt dies einen (möglicherweise unvollständigen) Ablauf von P, und analog für Q.

  Beispiel:

  - `(a1 --> a2 -->§) || (b -->§)`

    Abläufe: `a1 a2 b, b a1 a2, a1 b a2`

  - `(a --> b -->§) || (b --> c --> §)`

    Einziger Ablauf: `a b c` (da b synchronisiert wird)

CSP - Verhaltensbeschreibung

- Alphabet von P Menge der Namen von Aktionen in P

Beispiel:

```
A = (a --> A) ח §                Alphabet: {a}
B = (c --> b --> B) [] §         Alphabet: {b, c}
C = (a --> c --> C) [] §         Alphabet: {a, c}
```

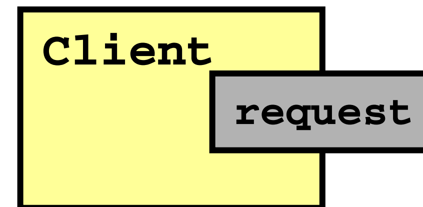
- Mögliche Abläufe:
  - von A
  - von B
  - von C
  - von A||B||C

Verhalten einer Wright-Konfiguration

Das Verhalten einer konsistenten Komponente ist gegeben durch die Instantiierung ihrer “Computation”.

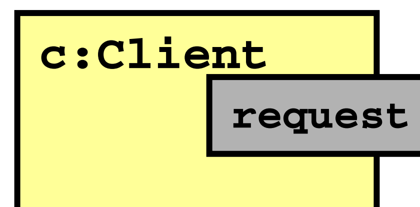
Beispiel: Client

Komponente:




```
component Client
  port request = requ!v --> result?w --> request Π §
  computation  = request.requ!v --> request.result?w --> computation Π §
```

Instanz:




```
Verhalten
c = c.request.requ!v --> c.request.result?w --> c Π §
```

Verhalten einer Wright-Konfiguration

- Das Verhalten einer konsistenten Konfiguration ist gegeben durch die
  - parallele Komposition des Verhaltens aller Komponenten und Konnektoren,
  - nachdem alle Umbenennungen durchgeführt wurden.

Beispiel: Verhalten der Client-Server Konfiguration

Konfiguration:

```
comp = (request.requ!v -->
  request.result?w --> comp) Π §
```

```
comp  = (provide.invoke?v -->
   provide.return!w --> computation) [] §
```

```
glue = (client.requ?x --> server.invoke!x -->
            server.return?y --> client.result!y --> glue) [] §
```

Verhalten:

```
c || s || cs  where
c  = (c.request.requ!v --> c.request.result?w --> c) Π §
s  = (s.provide.invoke?v --> s.provide.return!w --> s) [] §
cs = (c.request.requ?x --> s.provide.invoke!x -->
            s.provide.return?y --> c.request.result!y --> cs) [] §
```

Verfeinerung von Prozessen

- Verfeinerung bei Prozessen bedeutet Einschränkung von internem Nichtdeterminismus, d.h. insbesondere weniger Spuren und weniger Fehlerspuren.
- Ein Prozess Q ist **Spur-Verfeinerung** von P , wenn
  - P und Q das gleiche Alphabet besitzen und
  - Jede Spur von Q eine Spur von P ist.
- In CSP ist ein Prozess Q **Fehlerspur- Verfeinerung** von P, wenn
  - P und Q das gleiche Alphabet besitzen und
  - Jede Fehlerspur von Q eine Fehlerspur von P ist.
  - Und P divergiert, wenn Q divergiert.

Beispiel

`(e! --> §)` ist Spur- und Fehlerspur-Verfeinerung von `(e! --> §)` Π `(f! --> §)`

Aber

`(e? --> §)` ist Spur-Verfeinerung aber NICHT Fehlerspur-Verfeinerung von

`(e? --> §) [] (f? --> §)`

da `(e? --> §)` das Ereignis f? abweist, was `(e? --> §) [] (f? --> §)` NICHT kann.

Bemerkung: Jede Fehlerspurverfeinerung ist auch Spurverfeinerung.

Konsistenz einer Wright-Komponente

- Eine Komponente ist **konsistent bzgl. eines Ports**, wenn die Port-Spezifikation eine Projektion des Verhaltens der Komponente ist.
- Ein Prozess P ist **Projektion** von C, falls C, beschränkt auf das Alphabet von P, eine Fehlerspur-Verfeinerung von P ist.

  Beispiel:

  (1) `Client` und `Server` sind trivialerweise konsistent.

  (2)

```
Component Double
  Port in = (read?x --> in) [](close? --> §)
  Port out = (write!x --> out) Π(close! --> §)
  Computation =(in.read?x --> out.write!(2*x) --> Computation)
                    [](in.close? --> out.close! --> §)
```

  Die Komponente `Double` ist konsistent:

  - `in` ist Projektion von **`Computation`**
    (nach Verbergen von `write!x, close!`)
  - `out` ist Projektion von **`Computation`**
    (nach Verbergen von `read?x, close?` ist die Auswahl zwischen `write!x` und `close!` intern)

Konsistenz eines Wright-Konnektors

- Ein Konnektor ist **konsistent**, wenn die parallele Komposition seines Verhaltens C (Computation) mit den Prozessen Pi seiner Rollen Ri (i =1, …, n) verklemmungsfrei ist, d.h. wenn C || P1 || … || Pn verklemmingsfrei ist.
- Ein Prozess P ist verklemmingsfrei, wenn an keiner Stelle seiner Spuren alle Ereignisse verweigert werden können (außer nach der Terminierung).
- **Beispiel für Verklemmung**: Server verlangt Initialisierung, Client nicht

```
Connector Faulty
   Role client = (request! --> result? --> client) Π §
   Role server = (init? --> request? --> result! --> server)[] §
   Glue =(client.init? --> server.init! --> Glue)
            [](client.request? --> server.request! --> Glue)
            [](server.result? --> client.result! --> Glue) [] §
```

Nach `request! client.request? server.request!` warten `client, server` und **`Glue`** auf Eingabe: Verklemmung!

- **Beispiel**: `C-S-Connector` ist konsistent.

Konsistenz einer Wright-Konfiguration

- Eine Konfiguration ist konsistent, wenn
  - alle Komponenten und Konnektoren konsistent sind,
  - alle Rollen von Konnektoren kompatibel mit den zugehörigen Ports sind, d.h. vereinfacht, wenn jeder Port seine zugehörige Rolle (bzgl. Fehlerspuren) verfeinert,
  - alle Konnektoren und Komponenten korrekt instantiiert sind.
- Beispiel: Die Client-Server Konfiguration ist konsistent.
- Beispiel (Kompatibilität zwischen Rolle und Port)

```
Role Gen = (write!x --> Gen) Π (close! --> §)
```

  - wird verfeinert von

```
Port Out = write!1 --> write!2 --> write!17 --> close! --> §
```

  - wird NICHT verfeinert von

```
Port BadOut = (write!x --> BadOut) Π §
```

da die Spuren von BadOut das Ereignis close! nicht enthalten.

Themen heute

- ADL-Einführung
  - Definition
  - Historischer Überblick
  - Konzepte
- Wright
- Java/A

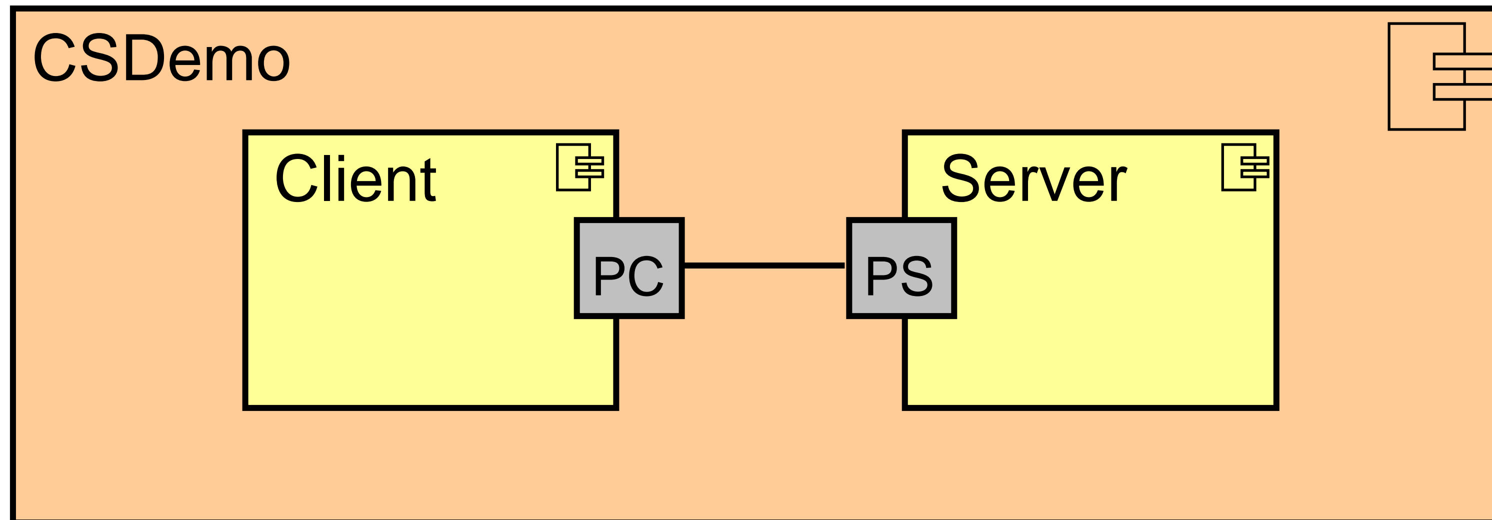
Java/A – „echte“ Komponenten in Java

- 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 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 {
    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

- Vergleich der Ansätze
  - Wright
    - Abstrakte, formale Beschreibungssprache unterstützt durch Simulation und formale Analyse
    - Defizite bei Codegenerierung
    - Statische Architektur
  - 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

- C.A.R. Hoare: Communicating Sequential Processes. Prentice Hall International Series in Computer Science, 1985.
- G. Estrin, R.S. Fenchel, R. R. Razouk, M. K. Vernon: SARA: Modeling, Analysis, and Simulation Support for Design of Concurrent Systems. IEEE Txn 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

Ausblick auf Block C: Formale Methoden