Service Discovery and Description

Svetlana Domnitcheva, Kay Römer, Michael Rohs

Doktorandenseminar Ubiquitäre Information

ETH Zürich, WS 2000/01

UbiComp

- Vernetzung aller Dinge
- Alltagsgegenstände werden zu „Smart Things“
- Dynamik: ständiges Kommen und Gehen
- Vielfalt: viele verschiedene Dinge mit verschiedenen Funktionen
- Erst Kooperation vieler Smart Things ergibt sinnvolle Funktion!

Beispiel: Roomcontrol

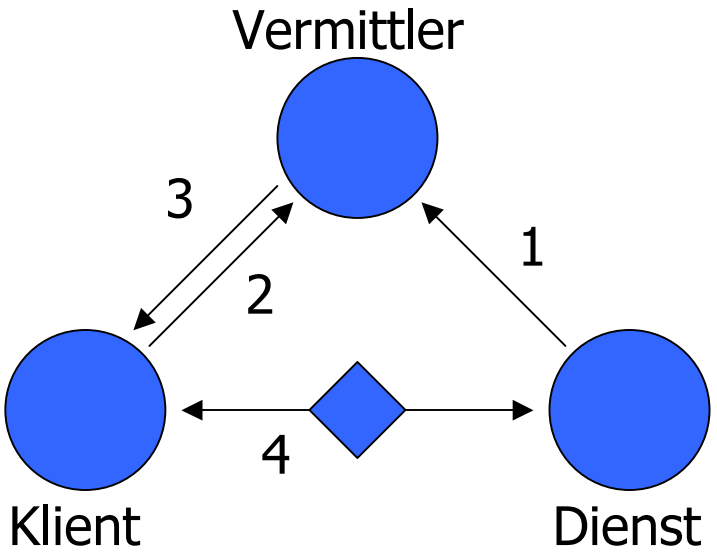
- Persönliche Wohnraumkonfiguration (Licht, Temperatur, Lautstärke, ...)
- Vom Anwender getragenes Gerät konfiguriert Raum beim Betreten automatisch
- Kooperation bei mehreren Anwesenden
- Ohne Eingriffe des Anwenders!

Service Discovery

- Frage: Wie findet Anwender-Gerät Steuerung für Licht, Temperatur etc. und andere Anwender-Geräte beim Betreten?
- Antwort: Service Discovery

Ablauf des Service Discovery

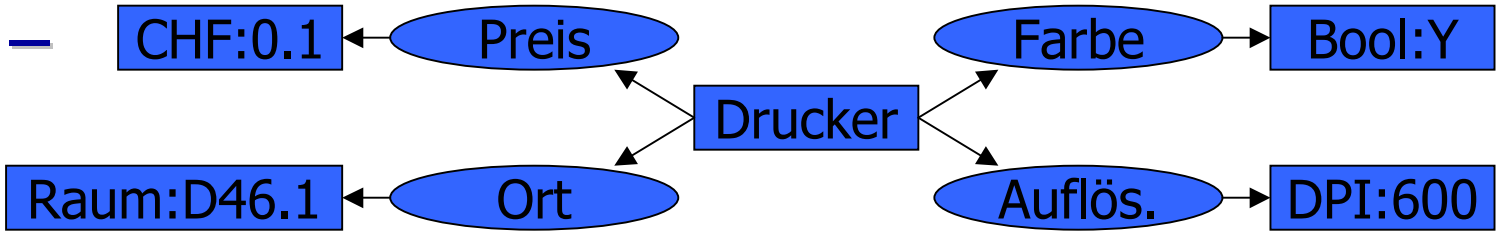
1. Service registriert sich bei Vermittler
   - Dienstbeschreibung
2. Klient fragt bei Vermittler an
   - Anfragebeschreibung
3. Vermittler liefert Referenzen auf Services
   - Matching
4. Klient nutzt Dienst
   - Adaption des Dienstes an Klient



Verschiedene Ebenen

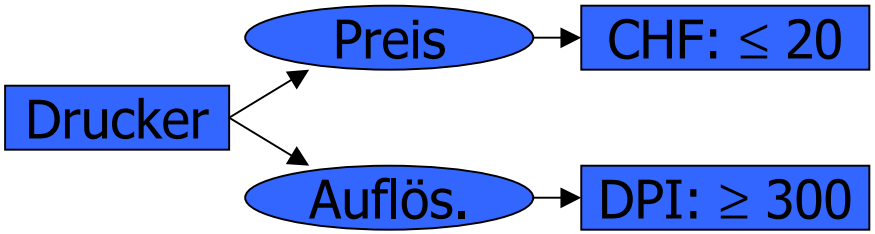
- Benutzerebene
  - Mensch-Maschine-Kommunikation
  - Vergleiche Internet-Suchmaschine
  - Kann „Intelligenz“ des Menschen verwenden, z.B. um unbekannte UIs zu verstehen
- Systemebene
  - Maschine-Maschine-Kommunikation
  - Kann nur auf die „Intelligenz“ der Maschine zählen

Dienstbeschreibung

- Beispiele:
  - Farbdrucker mit 600 DPI für 10 Rappen pro Seite in Raum D46.1
  - class Drucker { void print (String txt); }
    Color=Y, DPI=600, Preis=10, Raum=D46.1
  - 

- Beschreibt funktionale und nichtfunktionale Aspekte (Preis, Ort, ...)

Anfragebeschreibung

- Beispiele:
  - Suche Drucker mit mindestens 300 DPI für höchstens 20 Rappen pro Seite
  - class Drucker { void print (String txt); }
    DPI ≥ 300 AND Preis ≤ 20
  - 

- Vergleiche DB Query Languages

Matching

- Bestimmt zu einer Anfrage passende Dienste
- Formal:
  - Klient erwartet (sucht) Dienst A
  - Server bietet Dienst B
  - Vermittler bestimmt, ob A und B „kompatibel“ sind

Subtyp-Relation

- A ist kompatibel zu B bedeutet:
  - B bietet mindestens die Funktionalität, die A bietet, oder
  - Klient kann „ohne Schaden“ B verwenden, obwohl er eigentlich A erwartet, oder
  - B ist Subtyp von A ($B \leq A$)
- Vermittler implementiert Subtyp-Relation
- Genaue Definition der Subtyp-Relation hängt vom Kontext ab

Beispiele für Subtypen

- Farbdrucker ≤ Drucker
- Dimmer ≤ Lichtschalter
- Katze ≤ Tier
- class B extends A { ... }: B ≤ A
- long ≤ short
- struct C { short x; }
  struct D { long y; short z; }
  D ≤ C

Probleme

```
class Schalter {
  void an();
  void aus();
}
```

```
class Switch {
  void on();
  void off();
}
```

- Formal gilt Switch ≤ Schalter
- Wie kann der Vermittler das feststellen?
- Welche Dienstbeschreibung?
- Wie Switch anstelle Schalter verwenden?

Mehr Probleme

```
class Schalter {
  void an();
  void aus();
}
```

```
class Dimmer {
  // 0=aus, 1=an
  void setze (float wert);
}
```

- Dimmer ≤ Schalter
- Vermittler?
- Dienstbeschreibung?
- Dimmer anstelle Schalter?

Noch Mehr Probleme

```
class Raum {
  // Licht ...
  // Temperatur ...
}
```

```
class Licht { ... }
class Temperatur { ... }
```

- Licht+Temperatur ≤ Raum
- Vermittler?
- Dienstbeschreibung?
- Licht+Temperatur anstelle Raum?

Explizite Semantik

Ursache der Probleme:

- Fundamentaler Unterschied zwischen Schnittstelle und Semantik von Diensten
- Wir sind gewohnt nur die Schnittstelle eines Dienstes zu definieren und haben die Semantik im Kopf
- SD benötigt explizite Semantik
- Je mehr explizite Semantik, desto flexiblere SD möglich

Semantik?

- Attribute
  - Liste von Name/Wert-Paaren
- Mathematische Logik
  - Invarianten, Vor- und Nachbedingungen
- Semantische Netze
  - 

- Natürliche Sprache

Verschied. Anforderungen

- Die Extreme:
  - Closed world: alle Dienste sind im Voraus bekannt und können standardisiert werden
    -> SD trivial, wenig explizite Semantik
  - Open world: neue und modifizierte Dienste jederzeit möglich, kein Vorabwissen, Standardisierung der Dienste unmöglich
    -> SD komplex, viel explizite Semantik
- Mischformen möglich
- UbiComp tendiert zur open world!

Service Discovery Architekturen

- Service Location Protocol (SLP)
  - Internet Engineering Task Force (IETF)
- Jini
  - Sun Microsystems
- Berkeley Service Discovery Service
  - University of California, Berkeley

Service Location Protocol

- IETF Protokoll für Service Discovery
- Für IP-basierte Netze
- Komponenten
  - Service Agents (SAs)

    Bietet einen Dienst an, auch stellvertretend für eigentlichen Dienstanbieter
  - User Agents (UAs)

    Versuchen Dienste im Netz zu lokalisieren
  - Directory Agents (DAs)

    Sammeln Informationen über verfügbare Dienste

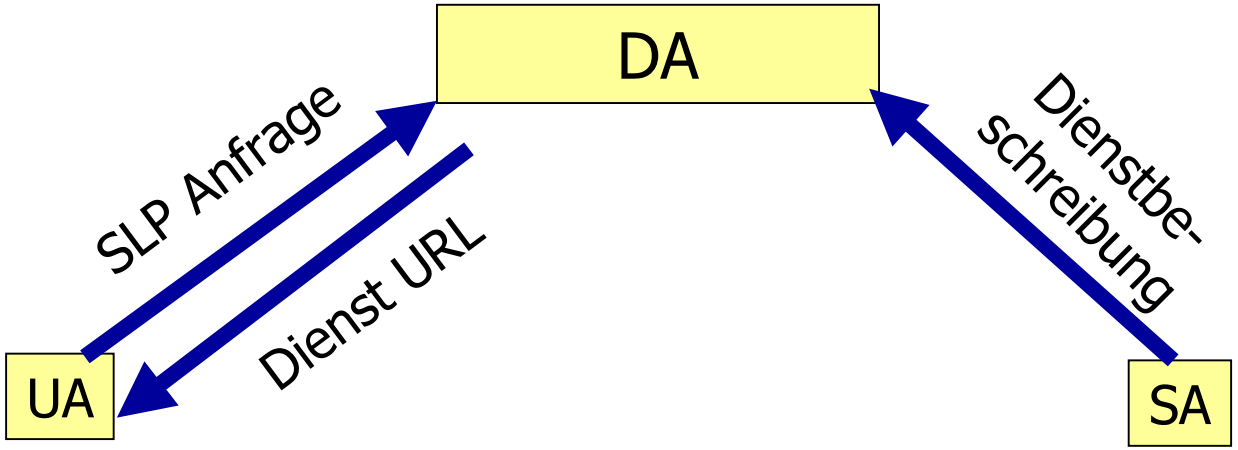
    Nicht zwingend notwendig

SLP Servicebeschreibugen

- Services definiert durch
  - Typname
  - URL
  - Attribute (Schlüssel-Wert-Paare)
- Beispiel

Lifetime: 10800
URL: service:lpr://www.printer.org:123/queue
Attributes: (PAGES_PER_MINUTE = 20),
(UNRESTRICTED_ACCESS),
(LANGUAGE = POSTSCRIPT, HPGCL),
(LOCATION = 12th FLOOR)

SLP Matching



lpr//(& (LANGUAGE = POSTSCRIPT)
(UNRESTRICTED_ACCESS)
(LOCATION = 12th FLOOR)
(PAGES_PER_MINUTE >= 10))/

Antwort:
service:lpr://www.printer.org:123/queue

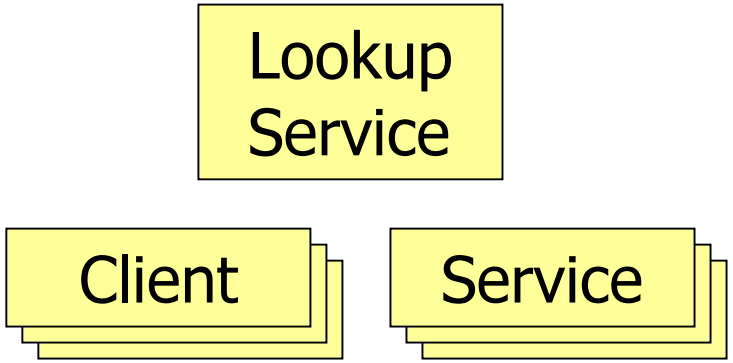
Lifetime: 10800
URL: service:lpr:// www.printer.org:123/queue
Attributes: (PAGES_PER_MINUTE = 20),
(UNRESTRICTED_ACCESS),
(LANGUAGE = POSTSCRIPT, HPGCL),
(LOCATION = 12th FLOOR),
(PAPER SIZE=LETTER)

Jini

- Java-zentriertes Service-Discovery- und Advertisement-System
- Schlüsselkonzepte
  - **Discovery**
  - **Join**
  - **Lookup**
  - Verteilte Ereignisse
  - Leasing
  - Transaktionen

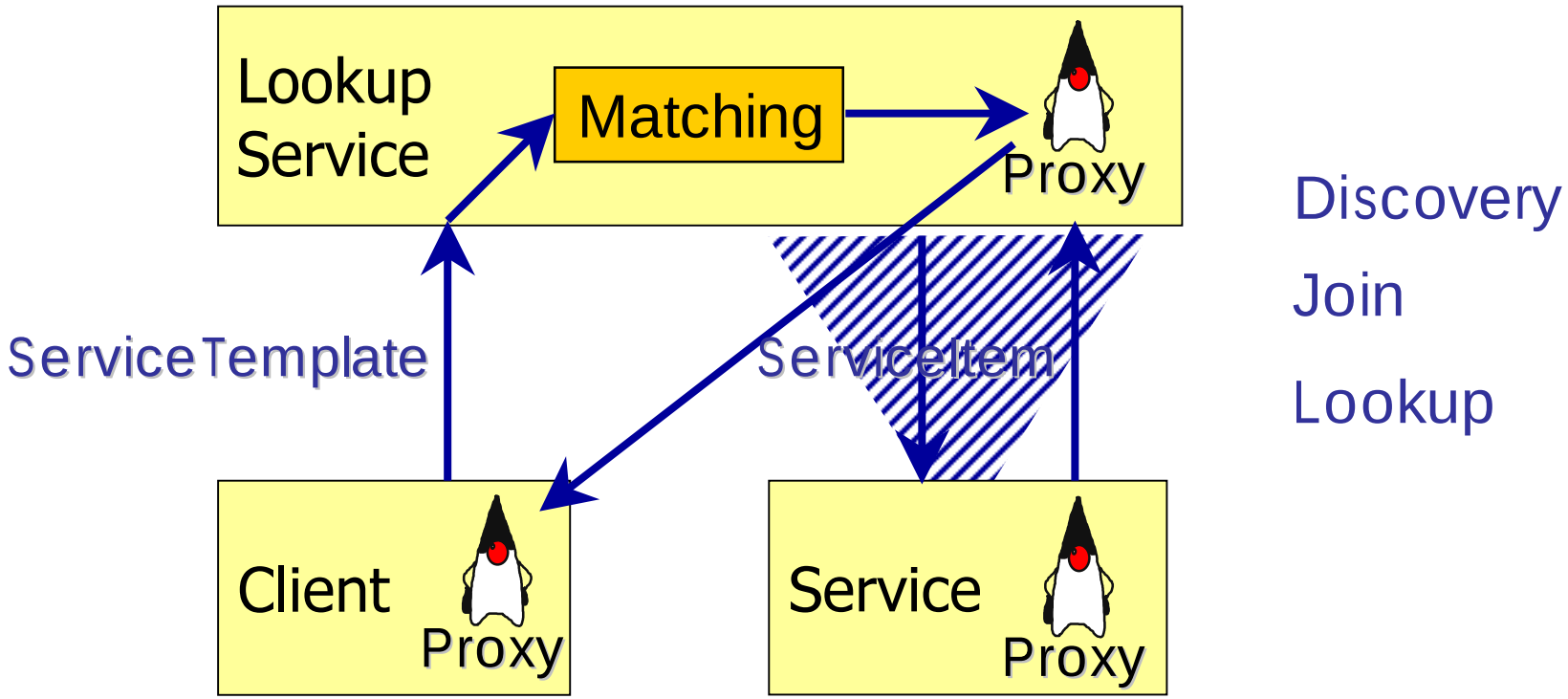
Jini Architektur

- Lookup Service (LUS)
  - Zentrale Registrierungs- und Vermittlungsinstanz für Dienste
  - Enthält Informationen über verfügbare Dienste
- Dienste
  - Spezifiziert durch Java-Interfaces
  - Registrieren sich mit Proxy-Objekten („Treibern“) und Attributen beim LUS
- Klienten
  - Kennen Java-Interfaces der Dienste, aber nicht deren Implementierung
  - Finden Dienste über LUS
  - Nutzen Dienste über Proxy-Objekte

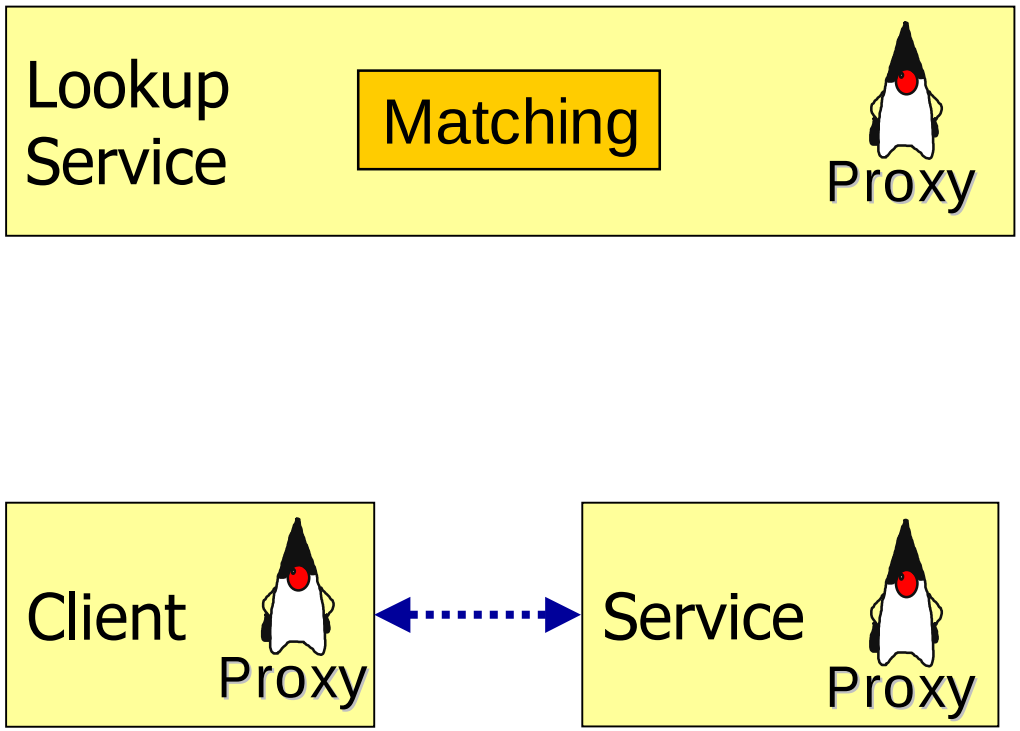


Closed-World Assumption

Jini Grundprinzip



Jini Grundprinzip



Discovery

Join

Lookup

Lease

Jini Join

- Management von Service-Registrierungen in Lookup-Diensten
- Beim LUS werden **ServiceItems** registriert:

```
public class ServiceItem implements Serializable
{
    public ServiceID serviceID;
    public Object service;
    public Entry[] attributes;
}
```

UUID

Proxy

beschreibende Attribute

Address, Comment, Location, Name, ServiceInfo, ServiceType, Status, ...

Jini Lookup

- LUS vermittelt zwischen Klient und Dienst
- Klient stellt Anfragen in Form von **ServiceTemplates**:

```
public class ServiceTemplate implements Serializable
{
    public ServiceID serviceID;
    public Class[] serviceTypes;
    public Entry[] attributes;
}
```

- **null** im Template wird als Wildcard interpretiert

Jini Match-Semantik

Anfrage:

```
class ServiceTemplate {
    ServiceID serviceID;
    Class[] serviceTypes;
    Entry[] attributes;
}
```

Im LUS gespeichert:

```
class ServiceItem {
    ServiceID serviceID;
    Object service;
    Entry[] attributes;
}
```

- null-Werte im Template sind Wildcards
- sonst Treffer, falls
  - template.serviceID = item.serviceID
  - service *ist-Subtyp* jedes serviceTypes
  - für alle Attribute t im Template gibt es Attribute i im Item mit

    i *ist-Subtyp* von t

    jedes Feld (ungleich null) in t ist *serialisiert-gleich* i

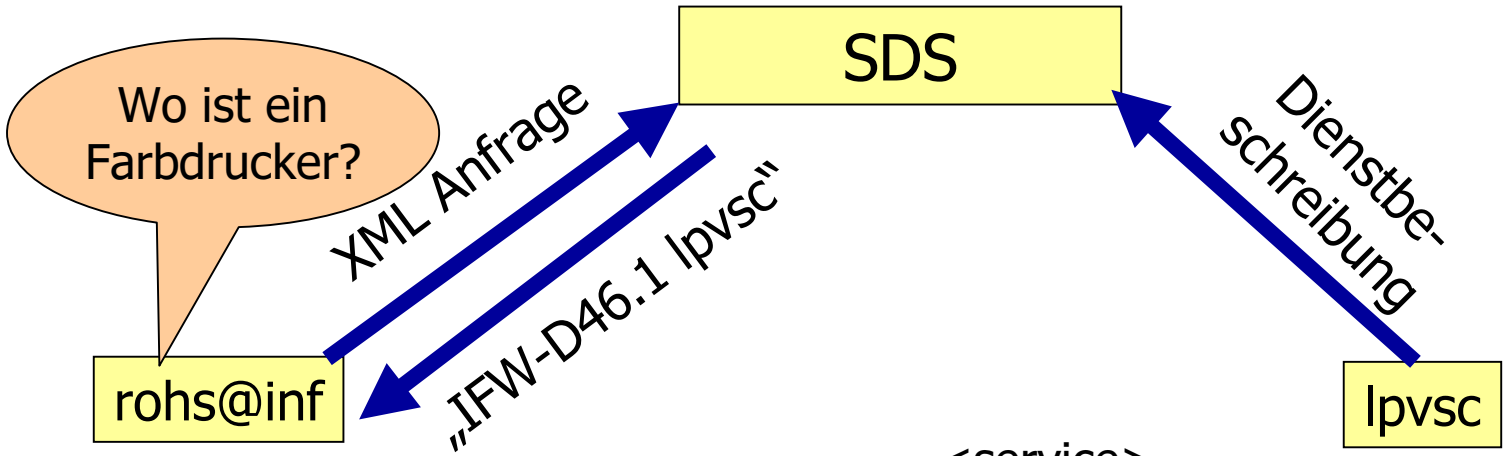
Jini Verteilte Ereignisse

- Lookup-Dienst speichert
  - ServiceTemplate und
  - Referenz auf Klient
- Klient wird benachrichtigt, beim Übergang von
  - Match → No-Match
  - No-Match → Match
  - Match → Match
- Verteilte Ereignisse für beliebige Dienste einsetzbar

Berkeley Service Discovery Service

- **Idee:** Ein sicherer Verzeichnisdienst, der Klienten das Ausfindigmachen von Diensten mit Hilfe von ausdrucksstarken Anfragen erlaubt
- **Anfragen in XML**
- **Dienstbeschreibungen in XML**
- **Fokus auf Sicherheit**
- Wide Area Features
- Fault Tolerance

Berkeley SDS Lookup

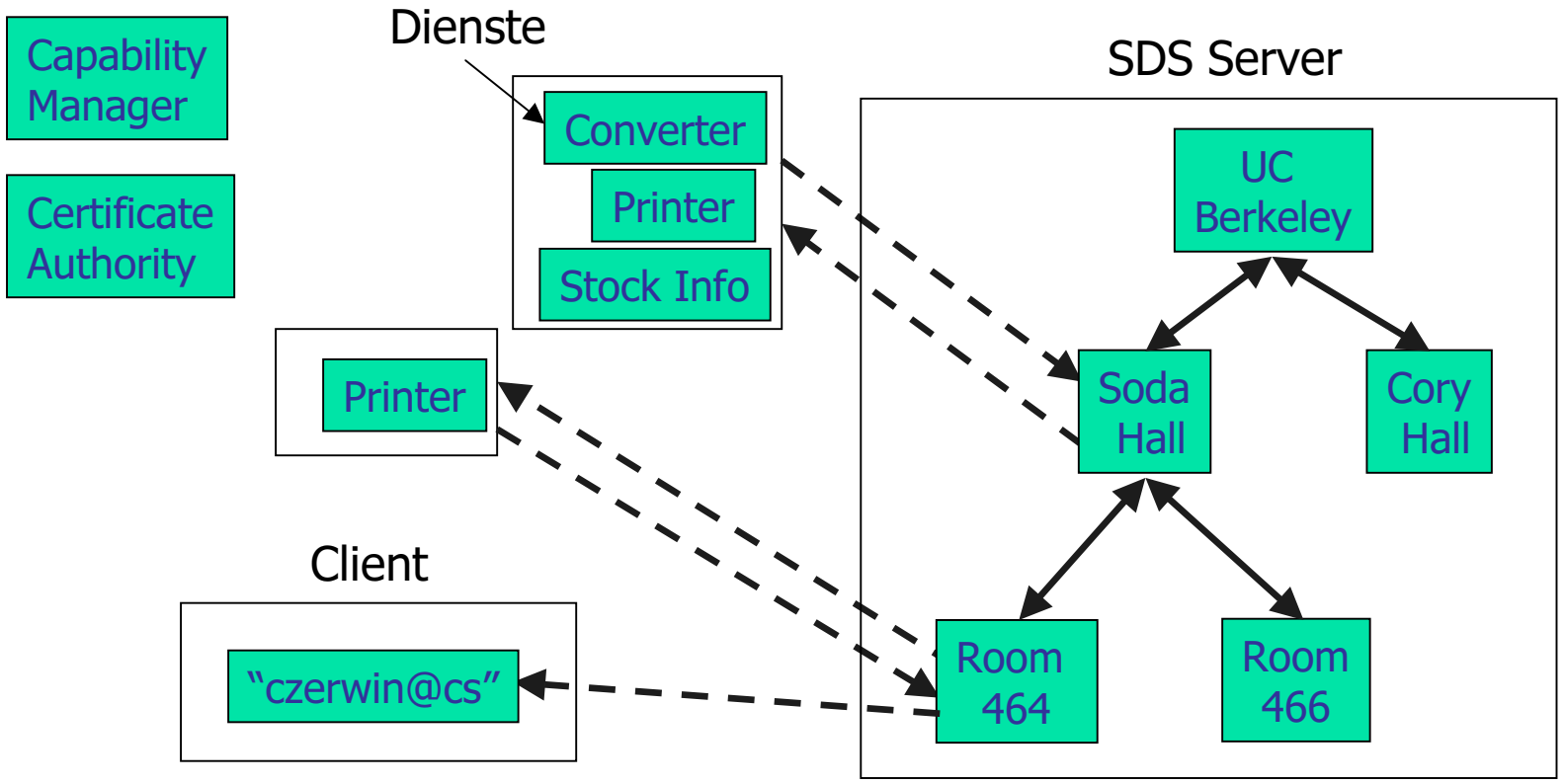


<query>
<type> io.printer </type>
<color> yes </color>
</query>

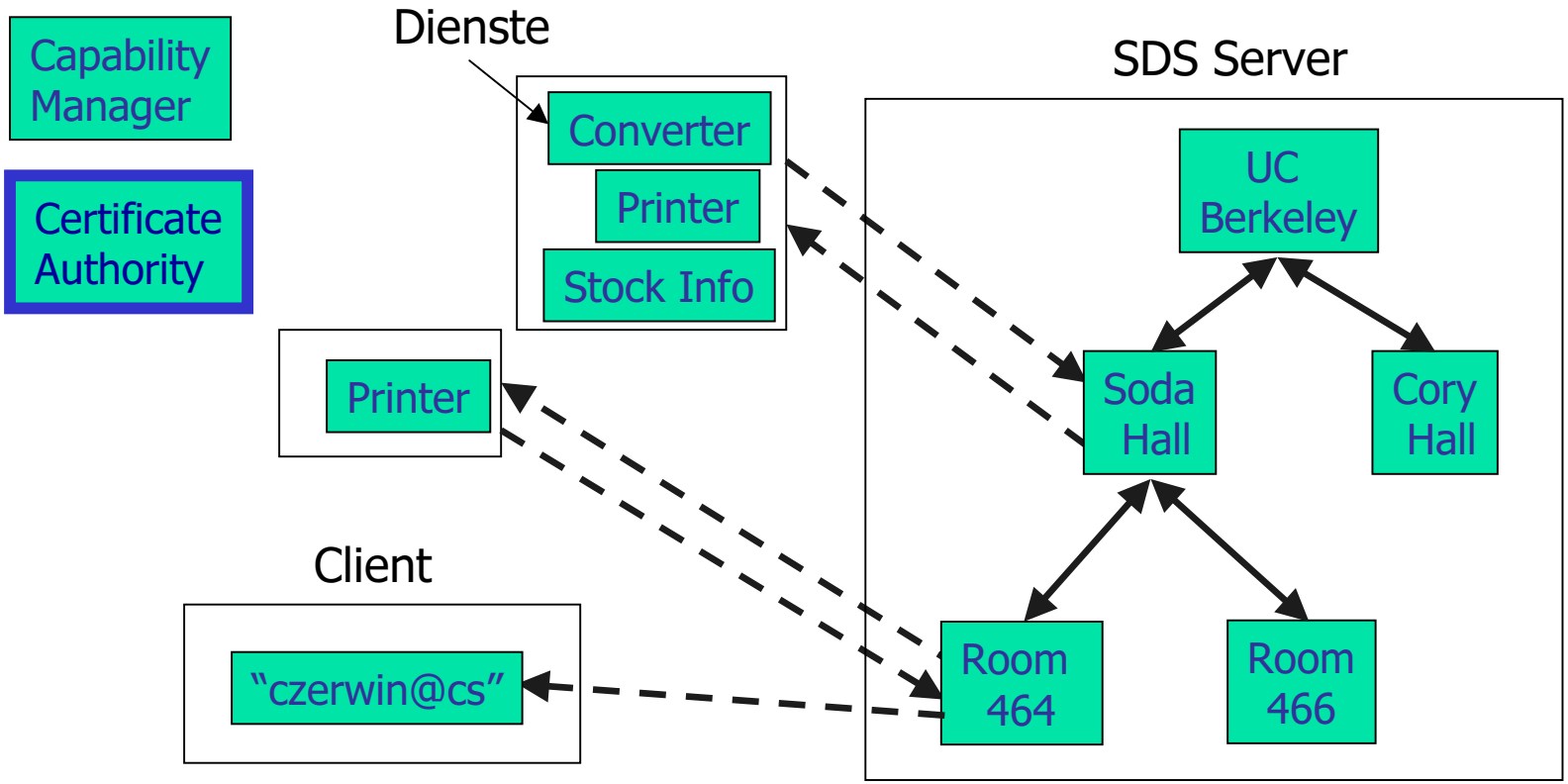
Antwort:
Dienstbeschreibung und URL

<service>
<name> IFW-D46.1 lpvsc </name>
<type> io.printer </type>
<location> IFW-D46.1 </location>
<color> yes </color>
<postscript> yes </color>
<contact>
<url> rmi://lpvsc.inf </url>
</contact>
</service>

Berkeley SDS Architektur



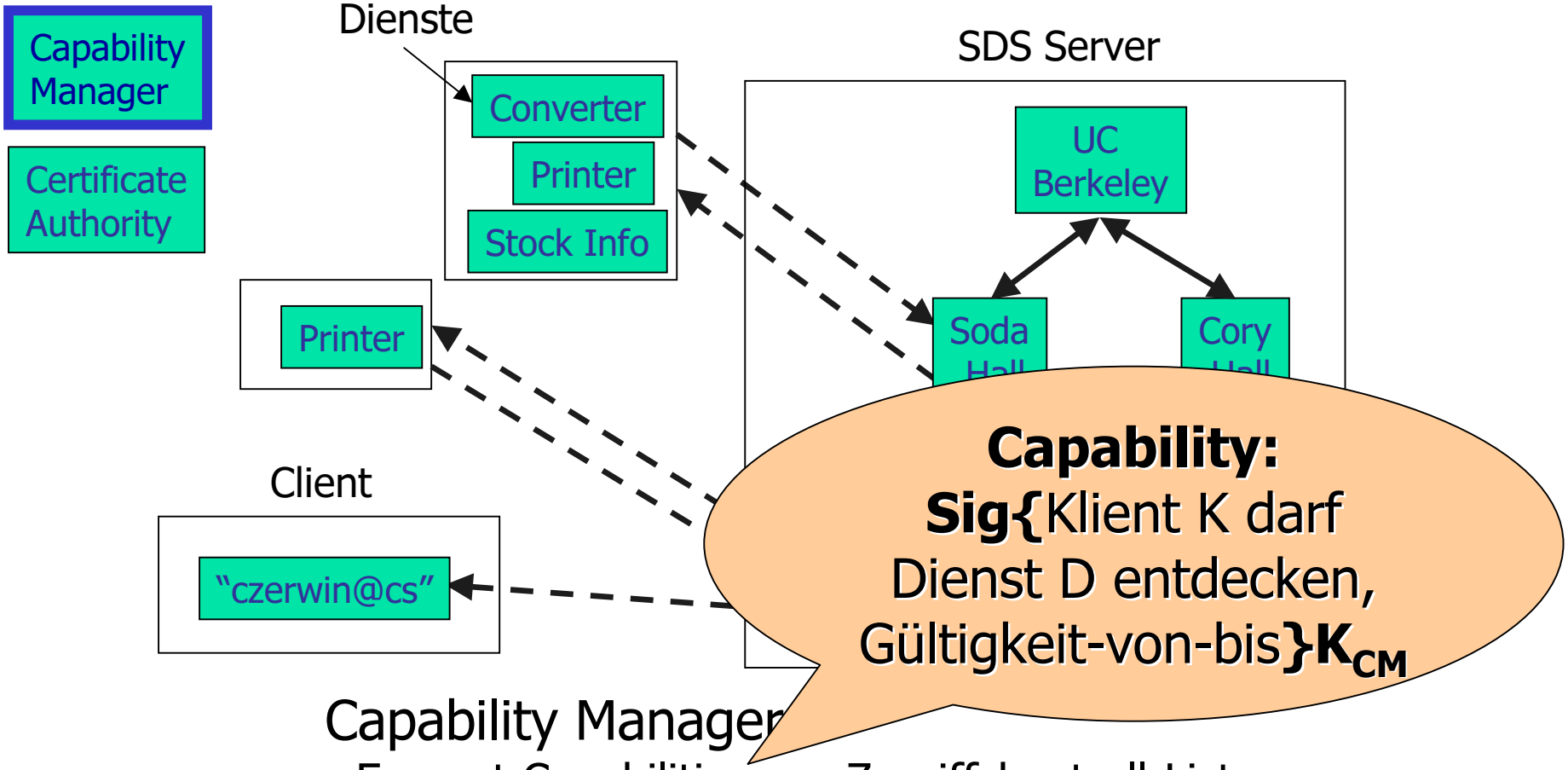
Berkeley SDS Architektur



Certificate Authority

- Authentifizierung von Benutzern und Diensten
- Verteilung von Zertifikaten

Berkeley SDS Architektur



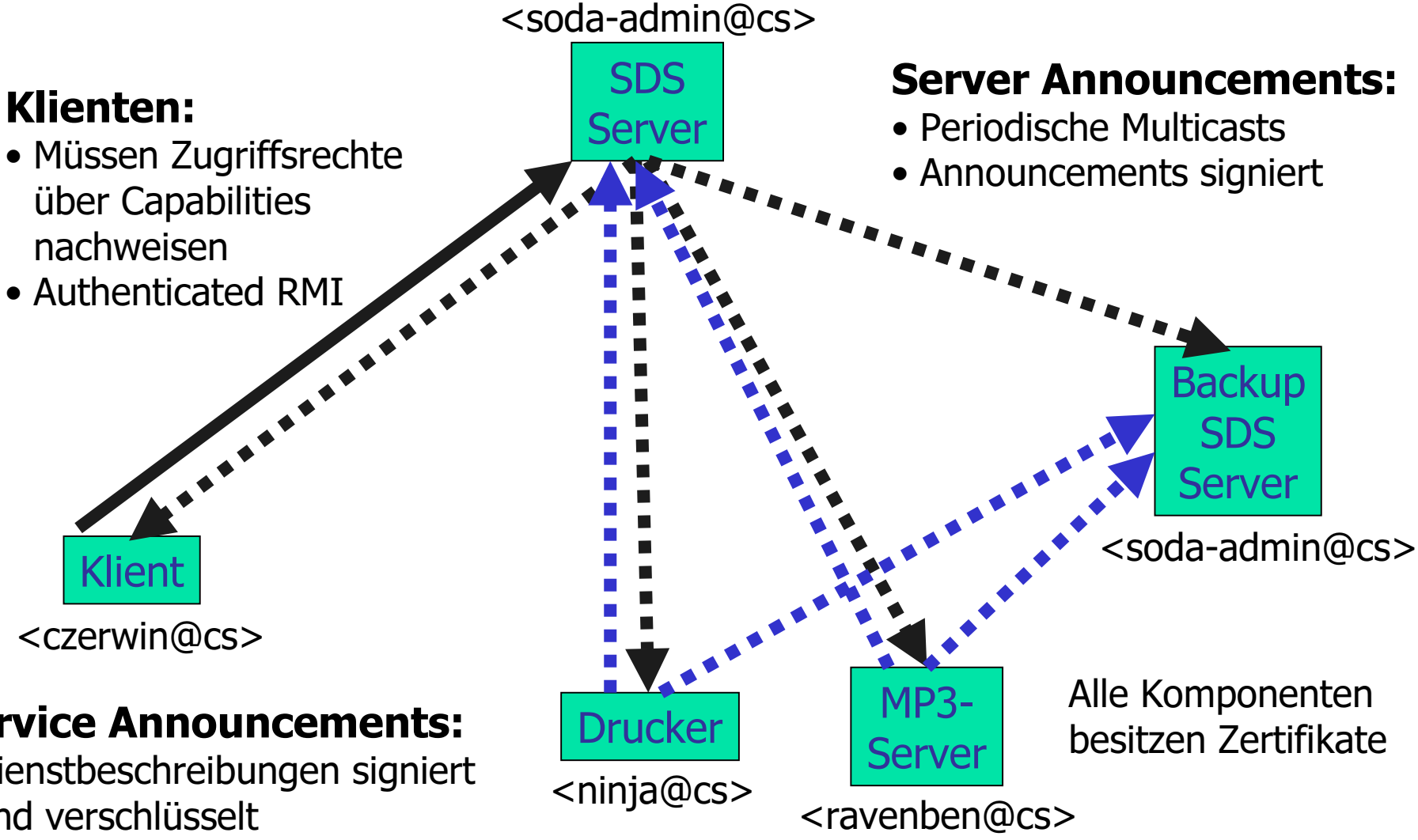
Capability Manager

- Erzeugt Capabilities aus Zugriffskontroll-Listen
- Verteilt Capabilities

Sicheres Service Discovery?

- Zugriffskontrolle
  - Welche Benutzer dürfen welche Dienste entdecken?
- Authentisierung aller Komponenten
  - Wer darf Dienste anbieten?
  - Wer darf SDS-Server sein?
- Sichere Kommunikation
  - RSA + Blowfish
  - Service Announcements verschlüsselt und signiert
  - Sichere Interaktion zwischen Benutzern und Discovery Service über *Authenticated RMI*

Sicherheit in SDS



Fazit

- Closed-World Assumption
  - Klienten müssen Dienstschnittstellen kennen
  - Gemeinsames Wissen zwischen Dienstanbieter und –nutzer auf Dienstebene nötig
  - Standardisierung aufwendig, da für jeden Dienst
  - Etwas Flexibilität durch Vererbung von Interfaces
- Dienstbeschreibung durch
  - Standardisierte Interfaces
  - Attribute
- Anfragen durch
  - Exaktes Matching oder Prädikate (SLP)

Open World

- *Kein gemeinsames Wissen* auf Dienstebene vorhanden
- Standardisierung auf einer abstrakteren Ebene
- Semantische Beschreibungen erforderlich:
  - Weltwissen (general-purpose ontology, taxonomy; „commonsense summer", CYC-Projekt, Teil von Tacitus-Project)
  - Wissen für Anwendungsbereiche
- Mögliche Ansätze
  - Analogical Reasoning
  - AI-Trader
  - Semantic Subtyping

Analogical Reasoning

- Identifikation und Darstellung des Dienstes
  - *Struktur, Funktionalität, andere wichtige Eigenschaften*
- Suche nach möglichen Kandidaten
  - *mit gewissem Mass der Übereinstimmung*
- Auswahl des besten Dienstes
  - z.B. durch gewichtete Dienst-Eigenschaften
- System lernt!
- Menschliche Hilfe wird benötigt

Analogical Reasoning

- Transformation der Lösung
  - gefundene Dienst wird mit Hilfe von *analogy map* und *transformation rules* angepasst.
  - Ergebniss ist oft leistungsschwach und *unelegant*
- Transformation der Herleitung
  - Analyse der *Herleitung änliches Dienstes*
  - Modifikation ungeeigneter Schritte und *Synthese des gesuchten Dienstes*
  - *Dekomposition*, wenn nötig
  - *viel effizienter*, aber auch aufwendiger

Synthese von Diensten

- bekannte und neuerzeugte Dienste werden systematisiert und wiederverwendet
- Input: semi-formale Beschreibung des Problems
- Output: z.B., UNIX-Shell-Skript

Beispiel:

(1) „Drucke aus alle Dateien grösser als 10K und deren Besitzer"

```
INPUT: dir WHERE: (Directory dir)
OUTPUT: (List f u)
SUCH-THAT: (and (Belongs f dir))
                (> (Size f) 10K)
                (Owner u f)
WHERE: (and (File f)(User u))
```

Synthese von Diensten

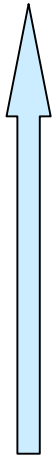
(2) „Lösche alle Prozesse mit *cpu_time>1000 s*“
- Erstelle Liste aller Prozesse
- Wähle diejenigen mit cpu_time>1000s
- `kill`

(3) „Erweitere Namen aller schreibgeschützten Dateien um Suffix `.read`“

- mögliche Lösung für (1)
  - Erstelle Liste aller Dateien
  - Wähle alle >10K
  - `rm`
- Verallgemeinerung: „Bestimme Objekte mit einer bestimmter Eigenschaft und führe auf diesen eine Operation aus “:
  - *Apply C(x) where x ∈ D such that P(x)*

Heuristiken

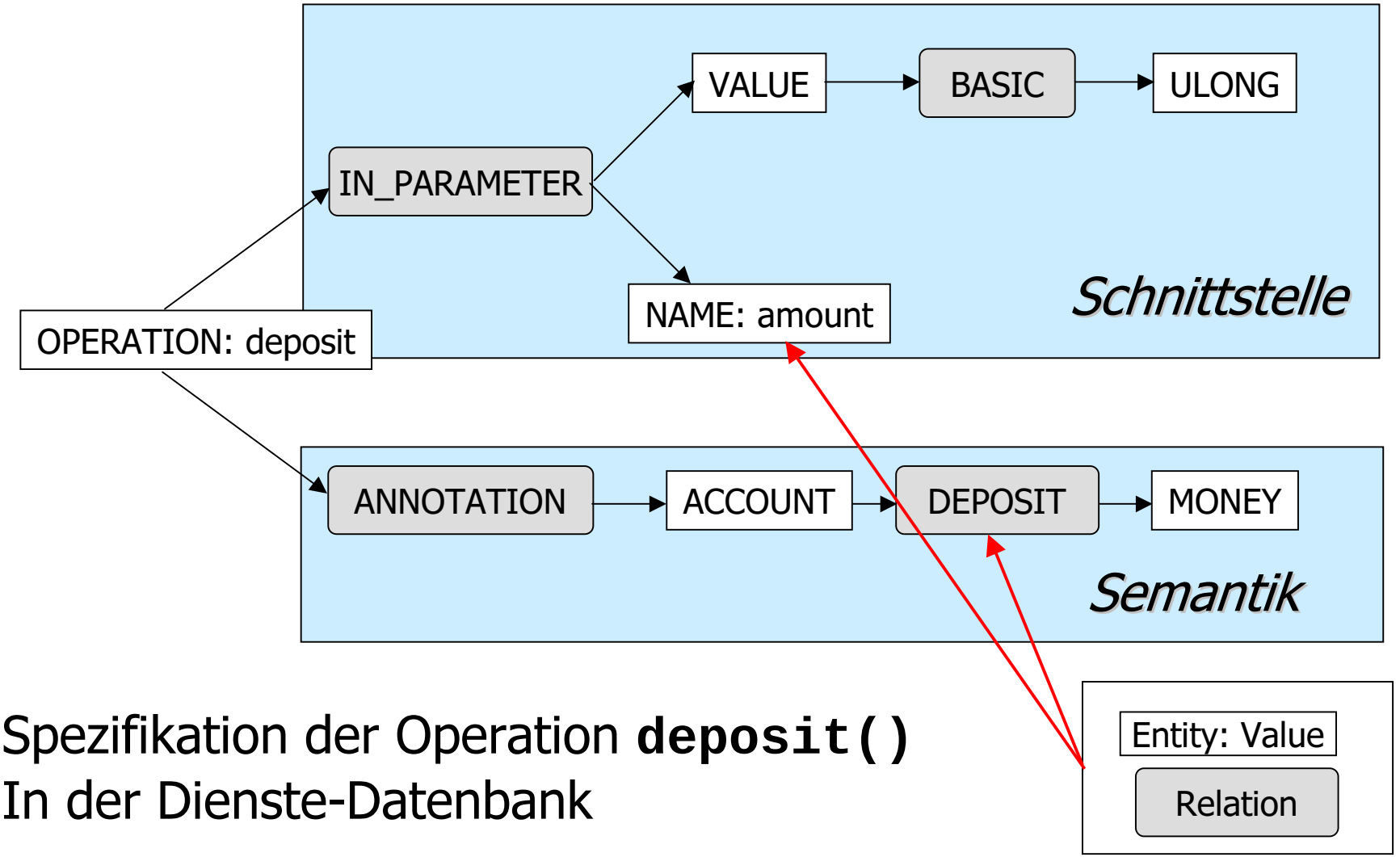
- Die *beste* Analogie wird gesucht
- denkbare Heuristiken
  - *Systematicity* (Verhältnis zw. input und output)
  - *Syntactic structure* (für dynamische Funktionen, Strukturähnlichkeit)
  - *Functional abstraction* (gleiche abstrakte Funktion)
  - *Argument abstraction* (Argumente gehören zur gleichen Hierarchie)



Beispiel: Bank-Service

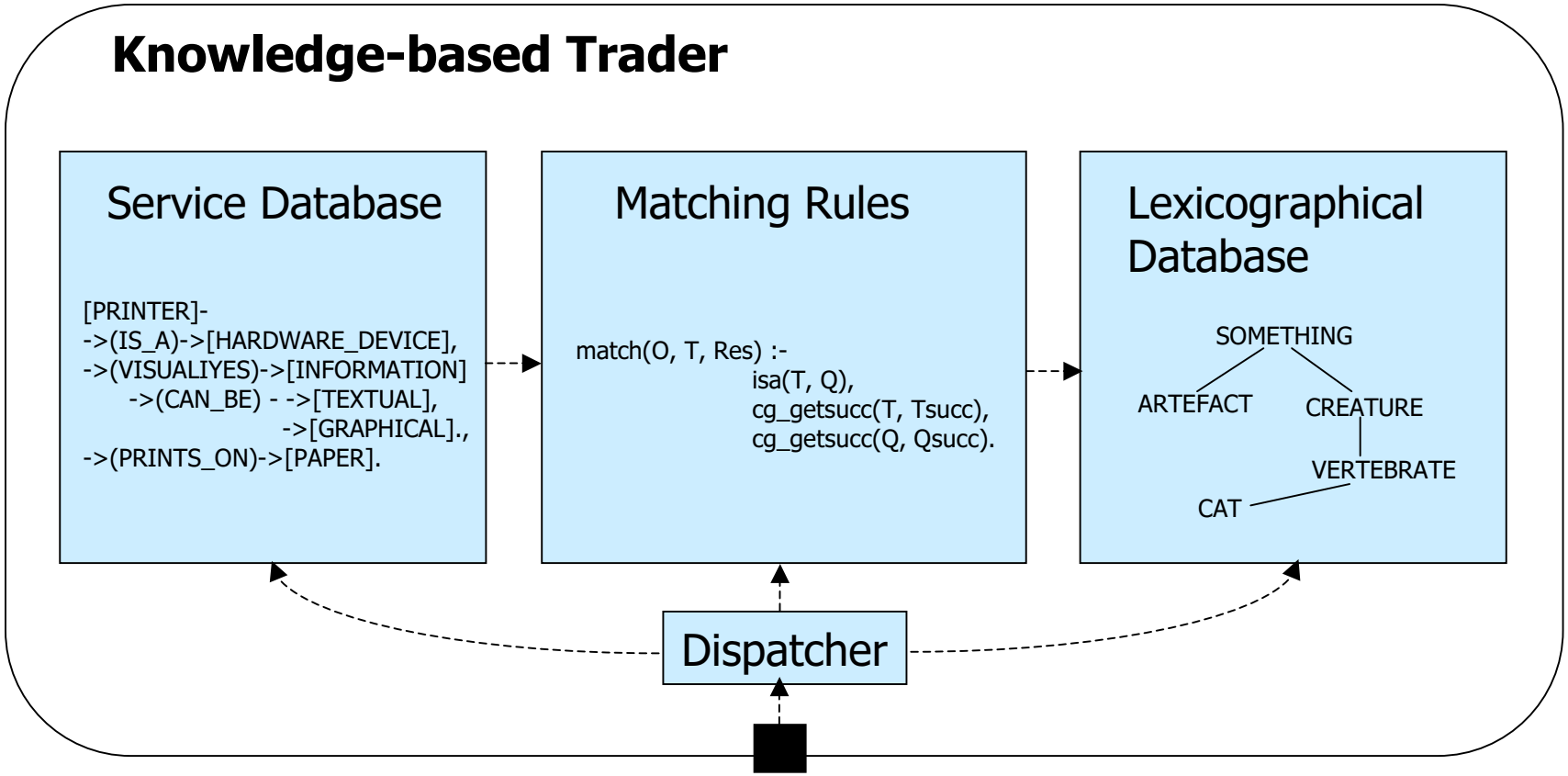
```
/* Operational interface for a bank account */
interface Account {

/* Deposit a certain amount to an account*/
void deposit(in unsigned long amount);

/*Withdraw a certain amount from an account*/
void withdraw(in unsigned long amount);

/*Show the curent balance of an account*/
void balance(out long amount);
}
```

Konzeptgraph



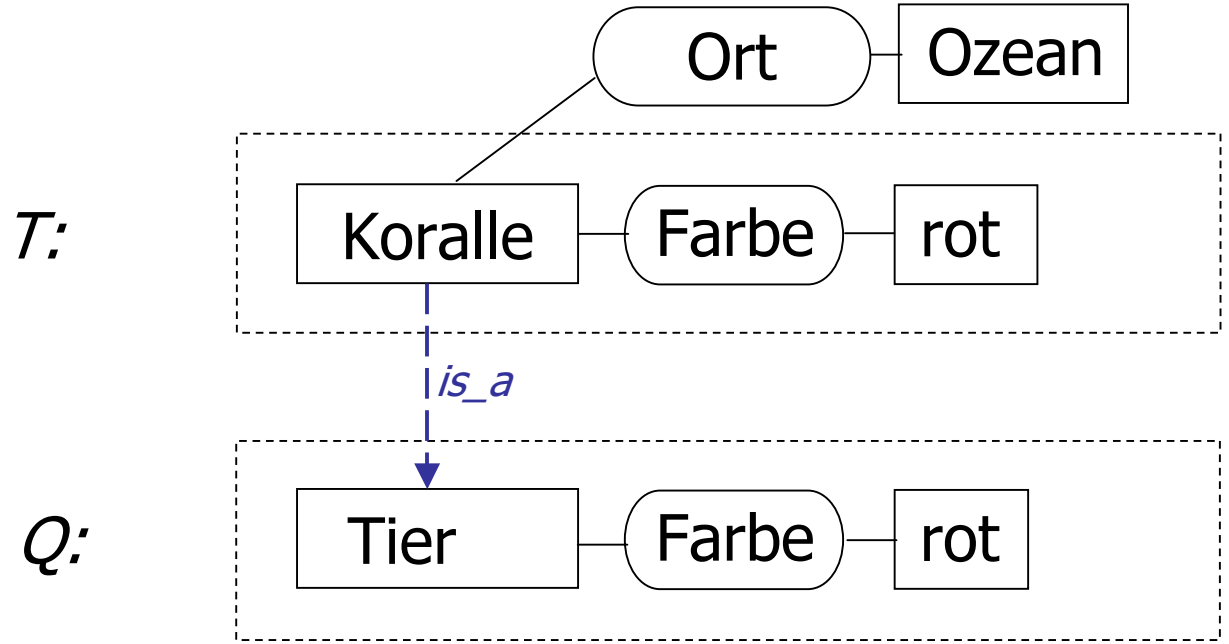
Spezifikation der Operation **`deposit()`**
In der Dienste-Datenbank

AI-Trader


Knowledge-based Trader
Service Database
[PRINTER]-
->(IS_A)->[HARDWARE_DEVICE],
->(VISUALIYES)->[INFORMATION]
->(CAN_BE) - ->[TEXTUAL],
->[GRAPHICAL].,
->(PRINTS_ON)->[PAPER].
Matching Rules
match(O, T, Res) :-
isa(T, Q),
cg_getsucc(T, Tsucc),
cg_getsucc(Q, Qsucc).
Lexicographical Database
SOMETHING
ARTEFACT
CREATURE
VERTEBRATE
CAT
Dispatcher

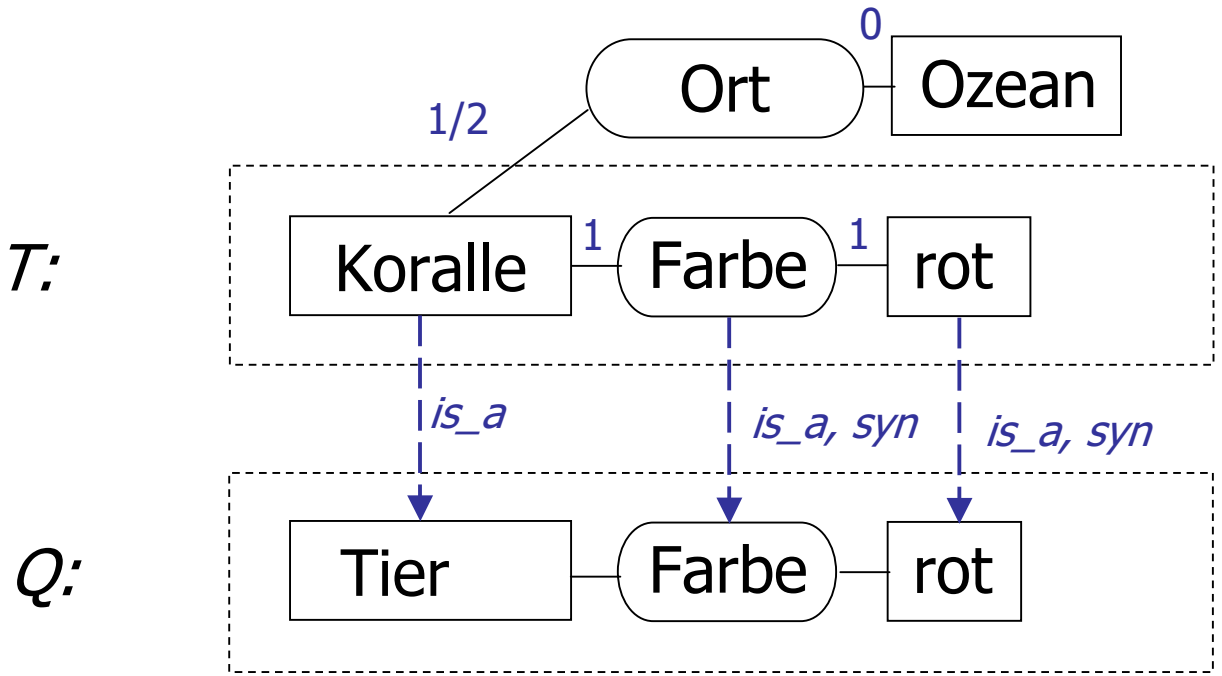
Anfrage

- Dienst-Darstellung als Konzeptgraph in der Datenbank und bei der Nachfrage
- Beispiel:



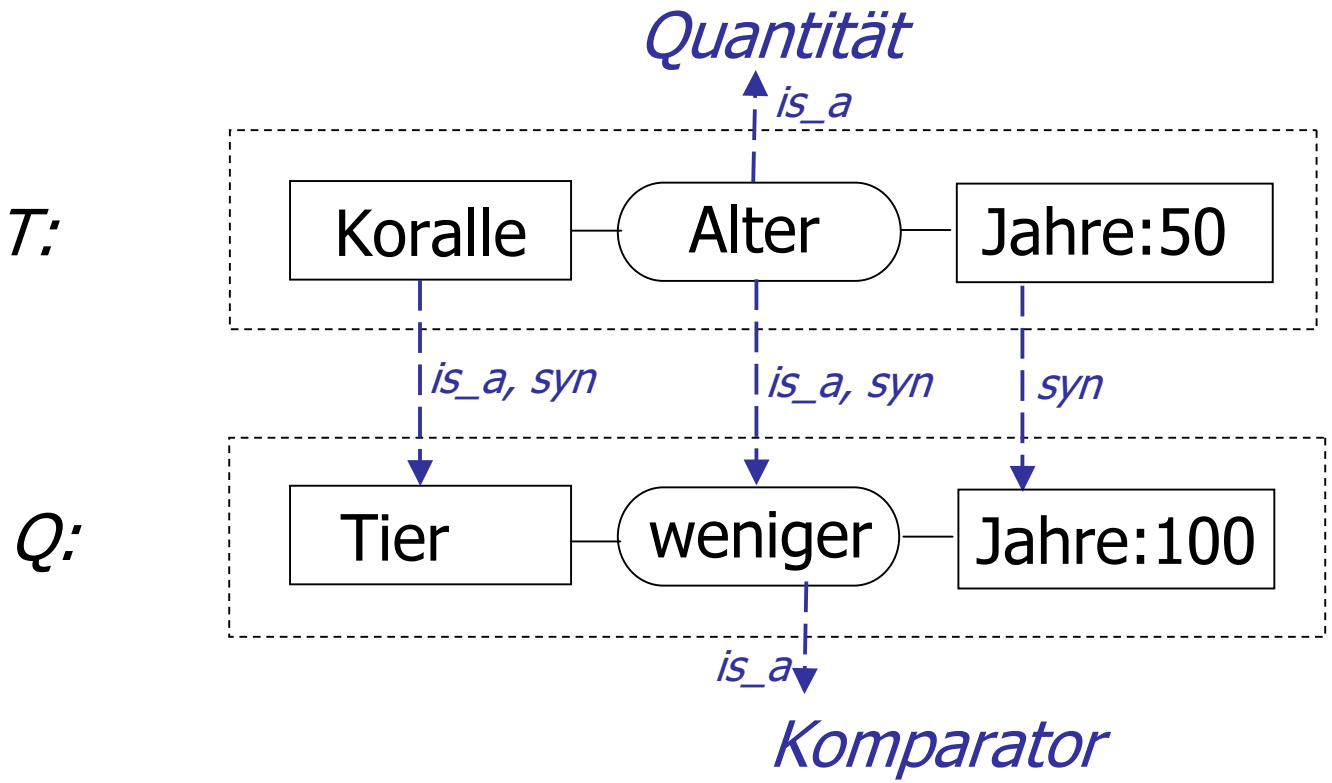
Matching Rules

- Spezialisierung
  - Testet, ob Type ein Spezialfall von Query ist



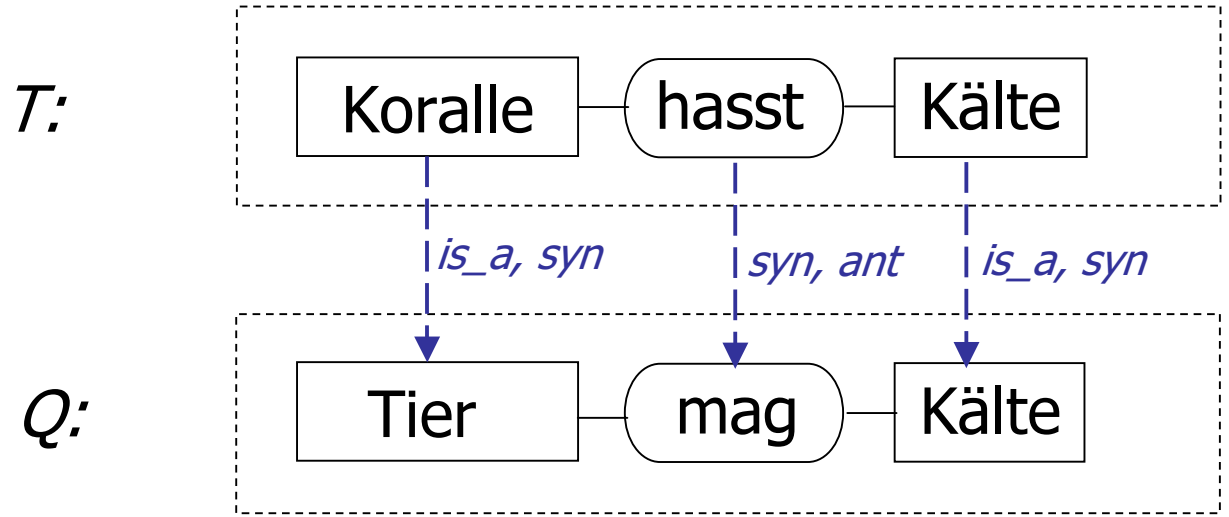
Matching Rules II

- Quantität
  - Testet ob quantitative Eigenschaften von Type mit Query übereinstimmen



Matching Rules III

- Negation
  - Testet ob *Type* und *Query* gegenteilige Eigenschaften haben



Semantic Subtyping

- formale Beschreibung semantischer Eigenschaften (z.B Larch: Garlang, Guttag, 1989)
- B.H. Liskov (MIT), J.M. Wing (CMU)
- Wie definiert man Sub-/Supertyp Verhältnis?
- i.a. *„the objects of the subtype ought to behave the same as those of the supertype as far as anyone or any program using supertype objects can tell"*

Bag & Stack

```
bag {
  put(x);
  get(x);
}
```

≥

```
stack {
  push(y);
  pop(y);
}
```

- "passende" Methoden und Argumente gemäss der Contra/Covariance-Regel

  put(x) ≤ push(y)

  y ≤ x

- Methodensignaturen von Stack und Bag "stimmen überein", aber nicht die Semantik

Subtype Requirement

- *Stärkere Bedingung:* „Any property provable about objects x of type T should be true for objects y of type S where S is a subtype of T„
- Eigenschaften:
  - safety (gleiches Verhalten, keine Störungen)

    Bsp: Invariant properties (bag: *size* < *bound*), history properties (*bound* ändert sich nicht)
  - other (existence, number of objects liveness... etc)

Typbeschreibung

- Name
- Wertebereiche, Invarianten
- Für jede Methode:
  - MethodenName
  - Argumente
  - Verhalten im Sinne von Vor- und Nachbedingungen

Typbeschreibung für Bag

bag=**type**

uses BBag (bag **for** B)
for all *b*: bag

put = **proc** (*i* : int)
requires $|b_{pre}.elems| < b_{pre}.bound$
modifies *b*
ensures $b_{post}.elems = b_{pre}.elems \cup \{ i \} \wedge$
$b_{post}.bound = b_{pre}.bound$

...
end bag

Typbeschreibung für Stack

stack=**type**

uses BStack (stack **for** S)
for all *s*: stack
invariant length(sp.items) <= sp.limit

put = **proc** (*i* : int)
requires length (s_{pre}.*items*)< s_{pre}.*limit*
modifies s
ensures s_{post}.*items* = s_{pre}.*items* || [*i*] ∧
s_{post}.*limit* = s_{pre}.*limit*

...
subtype of bag *(push* **for** *put*, *pop* **for** *get*, *...)*
// Transf. der Methoden
∀*st* : S . *A*(*st*) = (*st.items*, *st.limit*) // Transf. des Zustands
...
end stack

Subtypbestimmung

Stack ist Subtyp von Bag, wenn:

- Invarianten gelten
  - für alle Stacks s gilt $I_{Stack}(s) \Rightarrow I_{bag}(A(s))$
- Methodensignaturen sind "klassische" Subtypen: PUSH $\leq$ PUT
- Methodensemantik:
  - Vorbedingung: für alle Stacks s gilt: $PUSH_{pre}[A(s_{pre})/s_{pre}] \Rightarrow PUT_{pre}$
  - Nachbedinung: für alle Stacks s gilt: $PUT_{post} \Rightarrow PUSH_{post}[A(s_{pre})/ s_{pre}, A(s_{post})/ s_{post}]$

Fazit

- Semantische Beschreibung sehr aufwendig und fehleranfällig
- Mächtigkeit der semantischen Beschreibung (Vor- und Nachbedingungen für Drucker?)
- Weltwissen!!?

Zusammenfassung

- Service Discovery im Kontext von UbiComp
- Subtyp-Relation
- Verschiedene Anforderungen:
  - Closed world: SLP, Jini, SDS
  - Open World: Analogical Reasoning, AI-Trader, Semantic Subtyping
- Viele Fragen offen insbesondere in Bezug auf open world