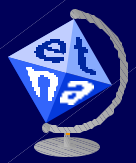
Flexible Vermittlung von skalierbaren Dienstobjekten in verteilten Systemen

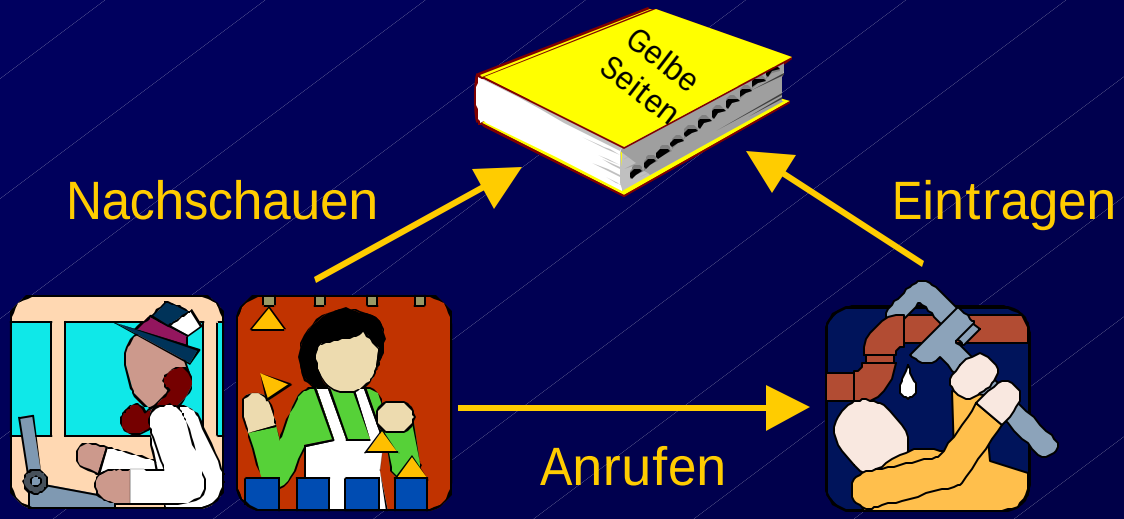
Arnd Große
Stefan Dolk
Rainer Ruggaber

1. Motivation
2. Trading
3. Skalierbare Dienstobjekte
4. SOFT-Trader
5. Java-Aspekte
6. Zusammenfassung



Alltägliches Beispiel

- Der Wasserhahn ist defekt und ich suche einen Klempner

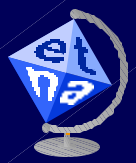


Technische Motivation

- Zerschlagung monolithischer Systeme in komponentenbasierte Anwendungen
- Komponenten
  - große Anzahl
  - verteilt im Netz
  - unterliegen Änderungen
- Informationssysteme bestehen aus über Rechnergrenzen hinweg kooperierenden Komponenten

Probleme:

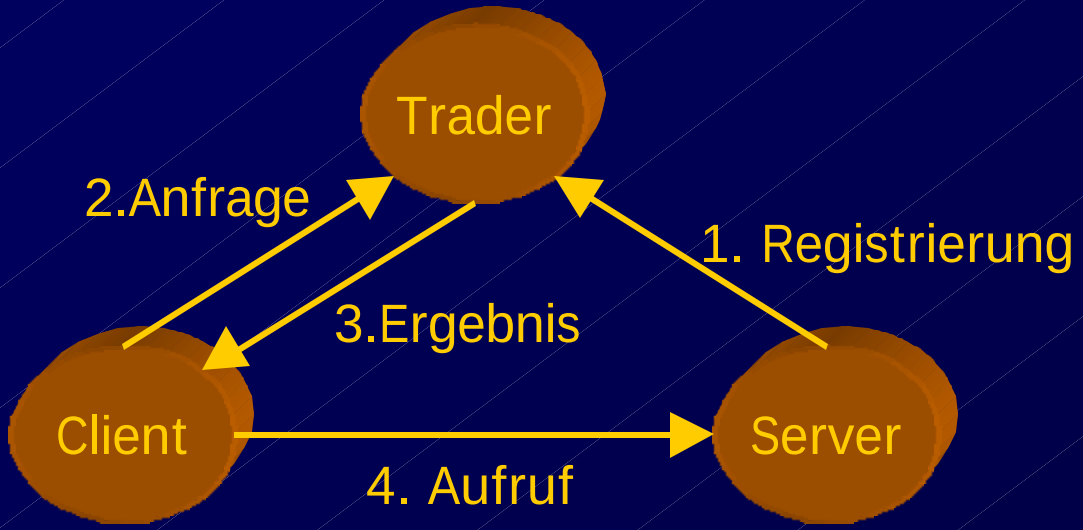
- Komplexität bei der Konfiguration der Anwendungen
- Auffinden geeigneter Komponenten



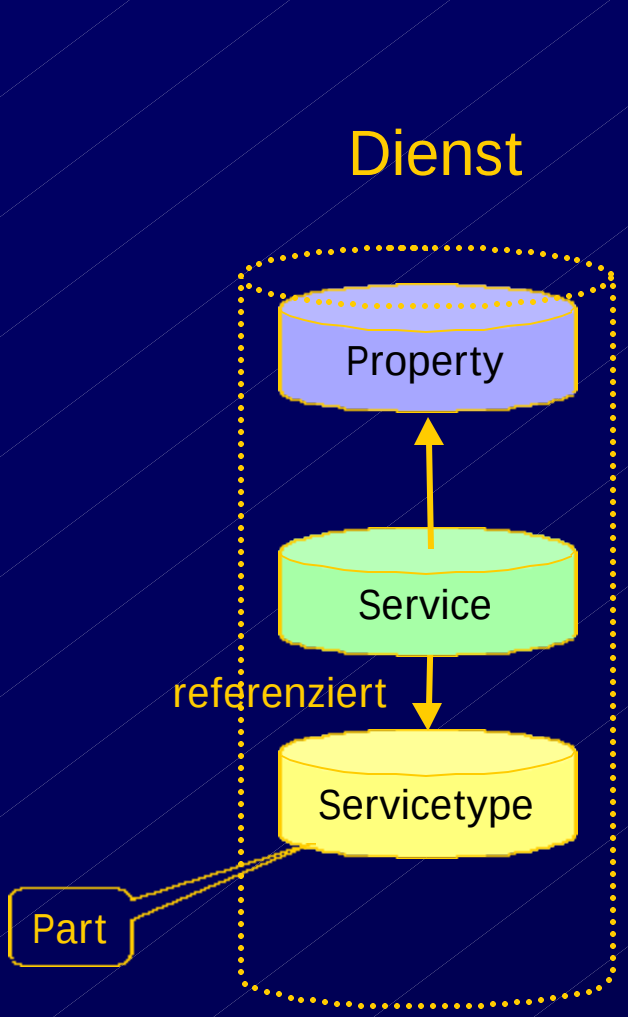
Trader

Lösung dieser Komplexität durch Trader

- unabhängiger Dienst zur Verwaltung und Vermittlung von Dienstangeboten zwischen Server und Client



Dienstmodell Standard-Trader



Eigenschaften

Referenz

Diensttyp

Nicht vorhanden!
Unterschied zwischen den Gelben Seiten und einem Trader

Telefonnummer

Kategorie in den Gelben Seiten

Kritik Standard-Trading

- Standard-Trading vermittelt nur Dienstreferenzen:
  - unflexibel
  - entfernte Aufrufe
    - große Latenzzeiten
    - schmalbandige Verbindungen
    - Verbindungsabbruch

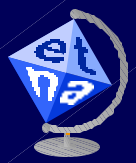
➔ Standard-Trading ist nicht geeignet für feingranulare komponentenbasierte Anwendungen

➔ Ich muß den defekten Wasserhahn selbst ausbauen und zum Klempner schicken der ihn repariert und mir zurückschickt!

SOFT (Scaleable Object Flow Trees)

- Flow Trees
  - Beschreibung des Trader-Interworking
  - effiziente Algorithmen zur Informationsverteilung

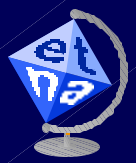
- Scaleable Objects
  - neue Parts für die Vermittlung:
    - Referenz, Diensttyp und Eigenschaften (wie bisher)
    - Implementierung
    - Zustand



Gewinn durch SOFT

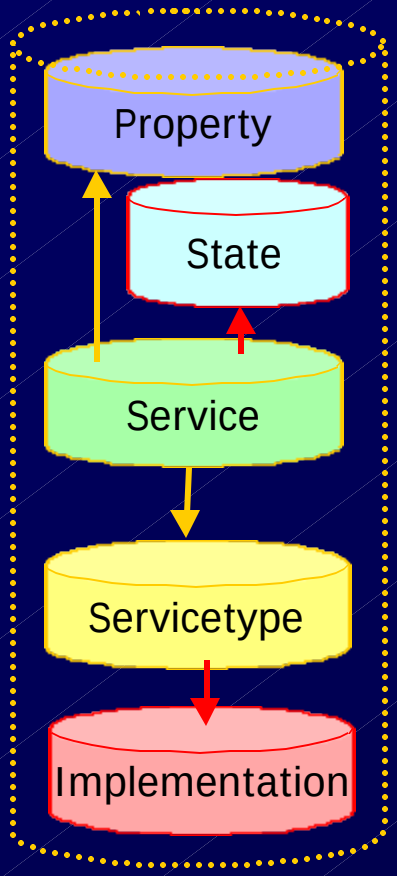
- Implementierung
  - lokale Dienstinstantiierung
    - Reduzierung der Latenzzeit durch lokale Aufrufe
    - abgekoppeltes Arbeiten
  - Softwareaktualisierung

- Zustand
  - Vorkonfigurierte Dienste
  - Zwischenergebnisse



Erweiterung des Dienstmodells

SOFT-Trading



Zustand

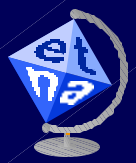
Implementierung

Kenntnis des Problems

Hausbesuche

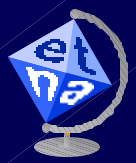
SOFT-Trader

- Transparente, abwärtskompatible Erweiterung
- Definition neuer Schnittstellen
  - **`SOFTexport`**
  - **`SOFTquery`**
- Erweiterung der Constraint-Language
  - Prädikate
    - **`SOFTimpl`**
    - **`SOFTstate`**
  - Identifikator für SOFT-serviceconstraints
  - Bsp.: **`"<<SOFT 1.0>> SOFTimpl and SOFTstate"`**
- Ablauf des Trading bleibt unverändert



Erweiterungen der Trader-IDL

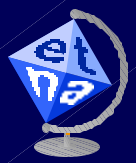
- Registrierung
  - **`export ( Reference, Type, Properties )`**
  - **`SOFTexport ( Reference, Type, Properties, Implementation, State)`**
- Anfrage
  - **`query ( ..., offerSeq, ... )`**
  - **`SOFTquery ( ..., SOFTofferSeq, ... )`**
- Aufruf
  - bleibt unverändert



Java-Realisierung

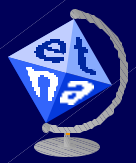
- Hilfsklasse: **`ObjectCreation`**
  - Methode **`getObject`**
    - lokale Installation (Proxy, Code, State)
    - normaler oder statischer Aufruf
    - Ergebnis: **`java.lang.object`**

- Vollständige Transparenz aufgrund des notwendigen Typecasts nicht möglich
  - expliziter Typecast
  - impliziter Typecast in der **`narrow`**-Methode der Helper-Klasse



Java-spezifische Implementierungsaspekte

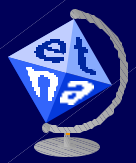
- Vollständigkeit des Implementation-Parts
  - ➔ Anwendungsprogrammierer (.jar-File)
  - ➔ Reflection-API zur Laufzeit
- Dynamisches Laden und Instantiieren von Dienstobjekten zur Laufzeit
  - ➔ Class-Loader-Mechanismus
- Generierung des State-Parts
  - ➔ Serialization-API um den Zustand auszulesen und auf neue Dienstobjekte aufzuprägen



Implementierung

- Implementierung:
  - OrbixWeb 3.0 von IONA
  - Jdk 1.1.6

- Plattformen:
  - DigitalUnix 4.0
  - Solaris 2.6
  - Windows NT/95



Zusammenfassung

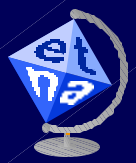
- SOFT-Konzept erweitert Trading-Dienst um die Vermittlung von Implementierung und Zustand
- größere Flexibilität bei der Vermittlung
- Integration in bestehende Anwendungen mit geringem Aufwand
- Lokale Instantiierung oder entfernter Aufruf ist für weitere Anwendung transparent
- Lokale Instantiierung reduziert Latenzzeiten und ermöglicht abgekoppeltes Arbeiten

Fazit: Der Klempner kommt ins Haus und hat die geeigneten Werkzeuge dabei!

Beispiel 1: Entfernter Methodenaufruf

- ```
  Calculator myCalc =
      CalculatorHelper.narrow (
        SOFT.util.ObjectCreation.getProxyObject (
          SOFTofferSeq.value[0] ) );
  int result = myCalc.add (1,1);
  ```

- entspricht:
  ```
  Calculator myCalc = CalculatorHelper.narrow(
      offerSeq.value[0] ) );
  int result = myCalc.add (1,1);
  ```


  ```

Beispiel 2: Lokale Kopie

```
SOFT.util.ObjectCreation o = new
  SOFT.util.ObjectCreation (
      SOFT.util.ObjectCreation.Local,
      ".",
      SOFTofferSeq.value[0]
  );

Calculator myCalc = (Calculator) o.getObject ();
int result = myCalc.add (1,1);

Calculator myCalc2 = (Calculator) o.getObject ();
int result = myCalc2.add (1,1);
```