

- IDL Value Types
- Life Cycle Service
- Life Cycle Service-Implementierung
- Aufgabe 4

1 Eigenschaften

- Flexible Zustandsbeschreibung
 - ◆ Abstrakte Value Types
 - ◆ Null-Werte
 - ◆ Beliebige Objektgraphen (Listen, Bäume, ...)
- Instanzen werden immer *by-value* übertragen
 - ◆ Objekte stets lokal vorhanden
 - ◆ Keine Fernaufrufe bei Zugriff
 - ◆ Keine (direkte) Registrierung am ORB
- Value Types sind **keine** CORBA-Objekte
 - ◆ Keine Erbschaftsbeziehung zu CORBA::Object
 - ◆ Keine Unterstützung der normalen Objektreferenz-Semantik
 - ◆ Stattdessen: Basis-Typ `ValueBase`

G.1 IDL Value Types

- Problem von Interface-Beschreibungen
 - ◆ Definition begrenzt auf Schnittstellen
 - ◆ Keine Aussage über Objektzustände
 - ◆ Konsequenz: Objekte können in heterogenen Umgebungen im Allgemeinen nicht *by-value* übertragen werden
- Lösung: Value Types
 - ◆ Abstrakte Beschreibung des Objektzustands
 - ◆ Plattformunabhängige Weitergabe von Objekten *by-value* möglich
- Literatur
 - CORBA Specification, Chapter 5: "Value Type Semantics"
 - <http://www.omg.org/cgi-bin/apps/doc?formal/02-06-41.pdf>

2 Value Types und Interfaces

- Value Types können Interfaces "unterstützen"
 - ◆ Explizite Verknüpfung eines Value Type mit einem Interface
 - ◆ Aussage über Zustand von Objekten, die dieses Interface implementieren
 - ◆ Beispiel:

```
interface Account {
    long balance();
    void deposit(in long value);
    void withdraw(in long value);
};

valuetype AccountContainer supports Account {
    private long account_state;
};
```

3 Value Types in Java

- Problem: Mehrfachvererbung im Servant nötig
 - ◆ Vom Skeleton: Marshalling und Unmarshalling
 - ◆ Vom Value Type: Zustandstransfer
- Lösung: Tie-Konzept
 - ◆ Stellvertreterobjekt für Value Type-Implementierung
 - ◆ Tie-Objekt
 - erbt von Skeleton
 - delegiert alle Aufrufe an Value Type-Implementierung
 - ◆ Beispiel

```
// Server-Seite
AccountContainerImpl container = new AccountContainerImpl();
Servant servant = new AccountPOATie(container);

// Client-Seite
Account account = ...; // Referenz auf CORBA-Objekt holen
account.balance(); // Weiterleitung auf container.balance()
```

G.2 Life Cycle Service

- Im Allgemeinen: Zugriffstransparenz auf Objekte erwünscht
- Ausnahmen: Applikationen für
 - ◆ Objektverwaltung
 - ◆ Lastverteilung
 - ◆ Fehlerbehandlung
- Life Cycle Service
 - ◆ Spezifizierter CORBA-Dienst
 - ◆ Grundlage anderer CORBA-Dienste
 - ◆ Erzeugen, Kopieren, Migrieren und Löschen von Objekten
- Literatur
 - Life Cycle Service Specification
 - <http://www.omg.org/cgi-bin/apps/doc?formal/02-09-01.pdf>

3 Value Types in Java

- Automatische Erzeugung von Tie-Klassen
 - ◆ JDK: explizit


```
// nur Server
idlj -fserverTie <idl-Datei>

// alle
idlj -fallTie <idl-Datei>
```
 - ◆ JacORB: implizit


```
idl <idl-Datei>
```
- Mapping des Value Type Basis-Typs
 - ◆ ValueBase <-> java.io.Serializable
 - ◆ Standardmechanismen zur Serialisierung einsetzbar

1 Problemstellungen

- Erzeugung von Objekten
 - ◆ Was entscheidet darüber, wo ein Objekt erstellt wird?
 - Client
 - Policy
 - ...
 - ◆ Wer genau erzeugt die Objektinstanz?
 - ◆ Wie findet ein Client denjenigen?
- Kopieren und Migrieren von Objekten
 - ◆ Wer ist für das Ausführen der Aktion zuständig?
 - ◆ Wo befindet derjenige sich?
 - Ursprungsort
 - Zielort
 - ...
 - ◆ Was passiert mit der transitiven Hülle des Objekts?

2 Herangehensweise

- Life Cycle-Objekte werden per *Factory* erzeugt
 - ◆ Eine Factory ist ein normales CORBA-Objekte
 - ("Factories are objects that create other objects.")
- ```
typedef Object Factory;
```
- ◆ Jede Factory kann nur lokale Objekte erzeugen
- Client findet passende Factory per zuständigem *Factory Finder*
    - ◆ Verzeichnisdienst für Factory-Objekte
    - ◆ Factory-Objekte registrieren sich bei Factory Finder
  - Alle weiteren Life Cycle-Operationen werden am Objekt selbst initiiert
    - ◆ Kopieren
    - ◆ Migrieren
    - ◆ Löschen

## 4 GenericFactory

- Generisches Interface zur Objekterzeugung
- Schnittstelle

```
interface GenericFactory {
 // Objekt erzeugen
 Object create_object(in Key k, in Criteria the_criteria)
 raises(NoFactory, InvalidCriteria,
 CannotMeetCriteria);

 // Überprüfung, ob Key von Factory akzeptiert wird
 boolean _supports(in Key k);
};
```

- ◆ Key: Name des zu erstellenden Objekts
- ◆ Criteria: Übergabe zusätzlicher (optionaler) Parameter

```
typedef struct NVP {
 CosNaming::Istring name;
 any value;
} NameValuePair;
typedef sequence <NameValuePair> Criteria;
```

## 3 FactoryFinder

- Schnittstelle

```
interface FactoryFinder {
 // Client trifft engültige Entscheidung
 Factories find_factories(in Key factory_key)
 raises(NoFactory);

 // Server trifft engültige Entscheidung
 Factory find_factory(in Key factory_key)
 raises(NoFactory);
};
```

- ◆ Key: Suchschlüssel
- ◆ NoFactory: Exception, falls keine passende Factory vorhanden ist
- Verwendung durch Client
  - ◆ direkt: bei Objekterzeugung
  - ◆ indirekt: bei Objektduplizierung, -migration

## 5 LifeCycleObject

- Super-Interface aller Life Cycle-Objekte
- Schnittstelle

```
interface LifeCycleObject {
 LifeCycleObject copy(in FactoryFinder f, in Criteria c)
 raises(NoFactory, NotCopyable, InvalidCriteria,
 CannotMeetCriteria);

 void move(in FactoryFinder f, in Criteria c)
 raises(NoFactory, NotMovable, InvalidCriteria,
 CannotMeetCriteria);

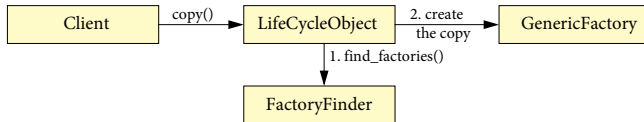
 void remove() raises(NotRemovable);
};
```

- ◆ copy() und move() benötigen Referenz auf einen FactoryFinder
- ◆ Criteria-Objekt
  - enthält Informationen über Zielort der Kopie bzw. des Objekts
  - wird üblicherweise direkt an die Factory weitergegeben

## 6 Erzeugung von Objektkopien

### ■ Ablauf

1. Client ruft `copy()` direkt am Objekt auf
2. Objekt holt mittels `FactoryFinder` eine zu `Criteria` passende `Factory`
3. `Factory` erzeugt neues Objekt & initialisiert es mit dem Zustand des Originals
4. Referenz auf Kopie wird als Rückgabewert von `copy()` an Client übergeben



### ■ Beachte: Objektkopie besitzt eigene Identität

- ◆ Unterschiedliche CORBA-Referenzen
- ◆ Getrennte Zustände

## 8 Löschung von Objekten

### ■ Ablauf

1. Client ruft `remove()` direkt am Objekt auf
2. Das Objekt hört auf zu existieren

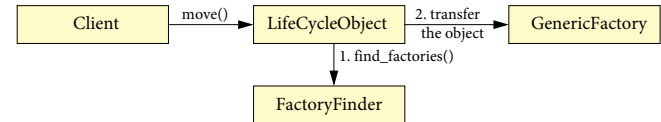
### ■ Beachte:

- ◆ Die CORBA-Referenz muss invalidiert werden
- ◆ Belegte Ressourcen sind nicht notwendigerweise (sofort) wegzuräumen

## 7 Migration von Objekten

### ■ Ablauf

1. Client ruft `move()` direkt am Objekt auf
2. Objekt holt mittels `FactoryFinder` eine zu `Criteria` passende `Factory`
3. `Factory` am Zielort "holt" sich das Objekt



### ■ Beachte: Identität des Objekts bleibt erhalten

- ◆ Die bisherige CORBA-Referenz bleibt weiterhin gültig
- ◆ Alle nachfolgenden Aufrufe erreichen das Objekt am neuen Standort

## G.3 Life Cycle Service-Implementierung

### ■ Problem: Der Life Cycle Service ist in den wenigsten ORBs implementiert

- ◆ Kontrollfluss und Implementierungsdetails nur oberflächlich spezifiziert
- ◆ Offene Punkte, z.B.
  - Gültigkeit von Referenzen nach Objektmigration
  - Zustandstransfer in heterogenen Umgebungen

### ■ Ansatz: Plattformunabhängige Implementierung

- ◆ Zustandstransfer mit Hilfe von Value Types
- ◆ Entwickler muss Objektzustand definieren

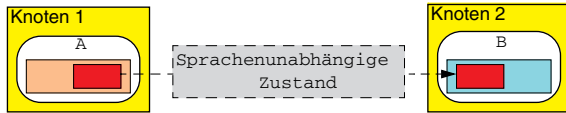
### ■ Literatur

Rüdiger Kapitza, Holger Schmidt, Franz J. Hauck  
Platform-Independent Object Migration in CORBA

<http://www4.informatik.uni-erlangen.de/Publications/pdf/2005/2005-09-13-LNCS3760.pdf>

## 1 Zustandstransfer in heterogenen Umgebungen

- Problemstellungen
  - ◆ Unterschiedliche Sprachumgebungen
  - ◆ Unterschiedliche Implementierungen
- Anforderungen
  - ◆ Sprachenunabhängiges Datenformat
  - ◆ Ausreichend abstrakte Zustandsrepräsentation



- Lösung: IDL Value Types
  - ◆ Kapselung des Zustands in der Implementierung

## 2 Zustandstransfer mittels Value Types

- Zusätzliche GenericFactory-Methode

```
interface GenericFactory {
 // Spezifizierte Methoden
 Object create_object(...) raises ...;
 boolean _supports(...);

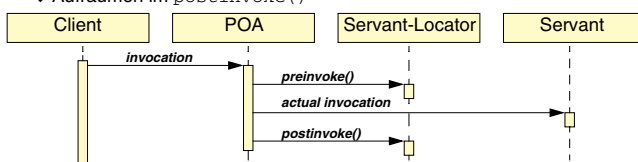
 // Objektkopie erzeugen
 Object createCopyFromValueType(in ValueBase value_type)
 raises (NoFactory);
};
```

- ◆ createCopyFromObject () erzeugt Objekt und initialisiert es mit dem übergebenen Value Type
- ◆ Anwendungsfälle: copy (), move ()

- In Java: *by-value*-Übertragung des Value Types per Serializable
  - ◆ Object createCopyFromValueType (Serializable value\_type) throws NoFactory;

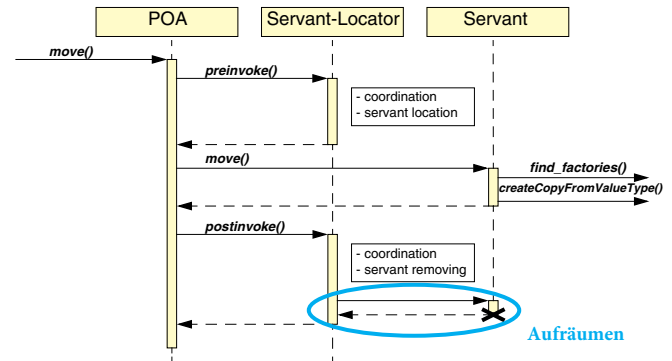
## 3 Koordinierung

- Problemstellungen
  - ◆ Exklusiver Objektzugriff notwendig, um Konsistenz garantieren zu können
  - ◆ Aufräumen nach Objektmigration
    - Referenz umbiegen
    - Alten Servant löschen
- Transparente Umsetzung
  - ◆ Life Cycle-Operationen werden exklusiv ausgeführt
  - ◆ Servant Locator besitzt pro Servant einen synchronisierten Aufrufzähler
  - ◆ Aufräumen im postinvoke ()



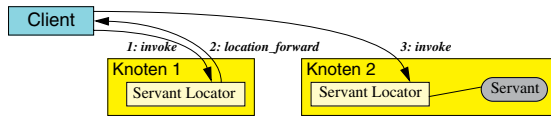
## 3 Beispiel: Objektmigration

- Aufruf von move () an einem Life Cycle-Objekt



### 3 Gültigkeit von Objektreferenzen nach der Migration

- Eine Referenz auf ein Life Cycle-Objekt sollte gültig bleiben, solange dieses existiert
- Lösung 1: Weiterleitung von Referenzen
  - ◆ Servant Locator kann im `preinvoke()` eine `ForwardRequest` werfen
  - ◆ Die `ForwardRequest` enthält Informationen über Zielort der Migration



#### ◆ Nachteile

- Möglicherweise lange Ketten
- Vorherige Knoten müssen weiterhin aktiv bleiben

## G.4 Aufgabe 4

- Bibliotheks-Server
  - ◆ Dienste
  - ◆ Typen
- Life Cycle Service-Funktionalität
  - ◆ Objekterzeugung
  - ◆ Objektduplizierung
  - ◆ Objektmigration

### 3 Gültigkeit von Objektreferenzen nach der Migration

- Eine Referenz auf ein Life Cycle-Objekt sollte gültig bleiben, solange dieses existiert
- Lösung 2: Location Service
  - ◆ Zentraler Dienst zur Verwaltung des Aufenthaltsorts von Objekten
  - ◆ Factories stellen Referenz auf den Location Service zur Verfügung
  - ◆ Aktualisierung des betreffenden Eintrags bei Objektmigration
  - ◆ **Nachteil: Alle Objekte müssen den selben Location Service verwenden**

## 1 Bibliotheks-Server: Dienste

- Library
  - ◆ Datenbank-Funktionalität: `register()`, `get()`, `getAll()`
  - ◆ wie bisher
- Item
  - ◆ Medien-Funktionalität: vgl. `ItemServant`
  - ◆ gleiche Funktionalität wie bisher, andere Implementierung
- ItemFactory
  - ◆ Erzeugung von `LifeCycle`-Objekten
  - ◆ Lokaler Dienst
- FactoryFinder
  - ◆ Auffinden von `ItemFactory`-Objekten
  - ◆ Zentraler Dienst

## 2 Bibliotheks-Server: Typen

- Primary
  - ◆ Zentraler Server: einer pro Bibliothekssystem
  - ◆ Bietet alle 4 Dienste an
  - ◆ Vorhandene Komponenten
    - LibraryDBServant
    - FactoryFinderServant
    - ItemFactoryServant
    - ItemPOA: Servants für Items
- Secondary
  - ◆ Beliebige Anzahl im Bibliothekssystem
  - ◆ Bietet nur 2 Dienste an
  - ◆ Vorhandene Komponenten
    - ItemFactoryServant
    - ItemPOA: Servants für Items

## 3 Life Cycle Service-Funktionalität

- Objekterzeugung
  - ◆ Keine Änderungen am Client
  - ◆ Library legt eigenständig Erstellungsort des Objekts fest, mögliche Entscheidungskriterien:
    - Individuelle Server-Auslastung
    - Zufall
    - ...
- Objektduplizierung
  - ◆ Zusätzliche Methode im Library Frontend: `copyItem(String title)`
  - ◆ Library legt eigenständig Erstellungsort der Objektkopie fest (siehe oben)
- Objektmigration
  - ◆ Zusätzliche Methode im Library Frontend: `clearPrimary()`
  - ◆ Library verteilt alle Objekte, die aktuell auf dem Primary liegen, gleichmäßig auf die vorhandenen Secondary-Server