Middleware - Übung

Tobias Distler, Michael Gernoth, Rüdiger Kapitza

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

Wintersemester 2009/2010

Überblick

Mobile Objekte
- IDL Value Types
- CORBA Life-Cycle Service
- Aufgabe 4

Value Types — Übersicht

- Problem von Interface-Beschreibungen
  - Definition begrenzt auf Schnittstellen
  - Keine Aussage über Objektzustände
  - → 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/doc?formal/02-06-41.pdf`

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`

Schnittstellen

Value Types können eine Schnittstelle „unterstützen"

- Explizite Verknüpfung eines Value Type mit einer Schnittstelle
- Aussage über den Zustand von Objekten, die diese Schnittstelle implementieren
- Übliche Bezeichnung: `<Schnittstellenname>Container`
- Beispiel

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

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

Value Types in Java

- Problem: Eigentlich 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()
```

Value Types in Java

- Automatische Erzeugung von Tie-Klassen per IDL-Compiler
  - `idlj`: explizit

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

// alle
idlj -fallTie <idl-Datei>
```

  - `idl`: implizit

```
idl <idl-Datei>
```

- Mapping des Value Type Basis-Typs
  - `ValueBase` ⇔ `java.io.Serializable`
  - Standardmechanismen zur Serialisierung einsetzbar

Mobile Objekte

Übersicht

- 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/doc?formal/02-09-01.pdf`

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?

Herangehensweise in CORBA

- 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 **direkt am Objekt** selbst initiiert
  - Kopieren
  - Migrieren
  - Löschen

Factory-Finder

`FactoryFinder`

- Schnittstelle

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

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

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

Generische Factory

GenericFactory

- Generische Schnittstelle zur Objekterzeugung
- Schnittstelle

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

    // Ueberpruefung, 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;
```

Life Cycle-Objekte

LifeCycleObject

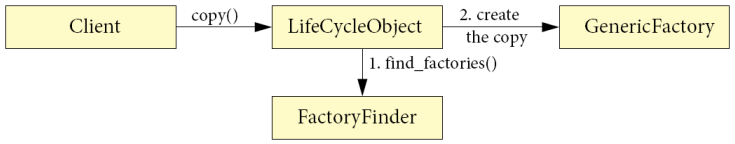
- Super-Interface aller Life Cycle-Objekte
- Schnittstelle

```
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

Erzeugung von Objektkopien

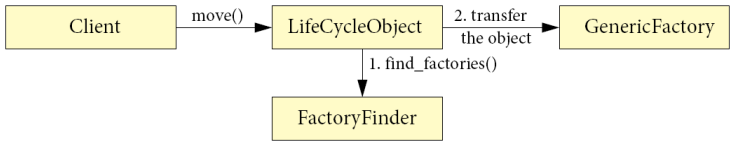
- Ablauf
  1. Client ruft `copy()` direkt am Objekt auf
  2. Objekt holt mittels `FactoryFinder` eine zu `Criteria` passende Factory
  3. Factory erzeugt neues Objekt & initialisiert es mit dem Zustand des Originals
  4. Referenz auf Kopie wird als Rückgabewert von `copy()` an Client übergeben



- Beachte: Objektkopie besitzt eigene Identität
  - Unterschiedliche CORBA-Referenzen
  - Getrennte Zustände

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

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

Implementierung Übersicht

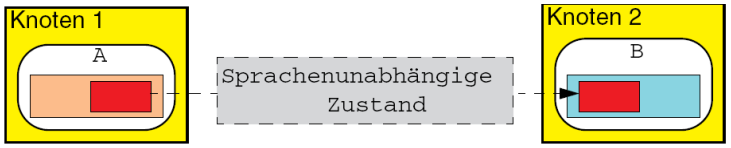
- Problem: Der Life-Cycle Service ist in den wenigsten ORBs implementiert
  - Kontrollfluss und Implementierungsdetails nur oberflächlich spezifiziert
  - Offene Punkte, z.B.
    - Zustandstransfer in heterogenen Umgebungen
    - Gültigkeit von Referenzen nach Objektmigration
- Ansatz: Plattformunabhängige 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`

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

Zustandstransfer mittels Value Types

- Zusätzliche `GenericFactory`-Methode

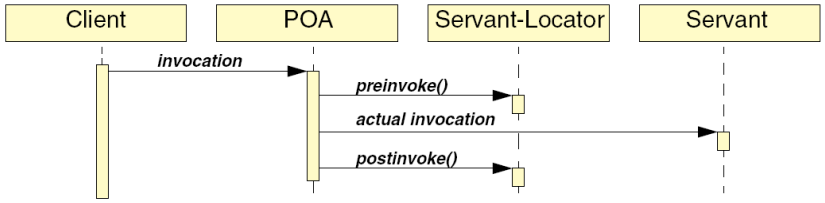
```
interface GenericFactory {
    // Spezifizierte Methoden
    Object create_object(...) raises ...;
    boolean _supports(...);

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

  - `createCopyFromValueType()` 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;
```

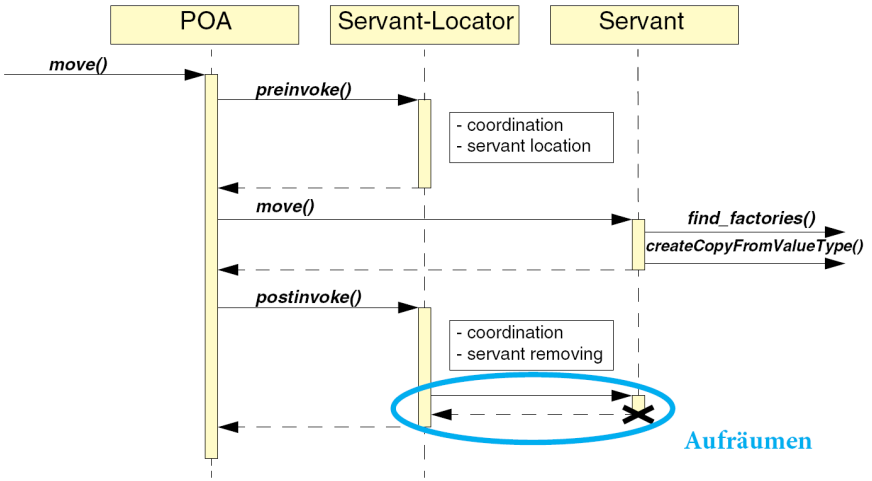
Koordinierung

- 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()`



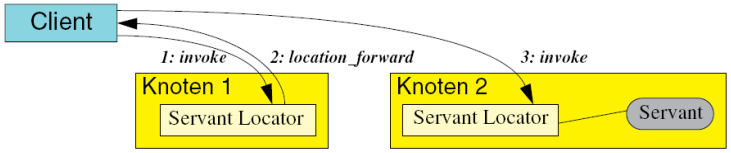
Beispiel: Objektmigration

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



Gültigkeit von Objektreferenzen nach der Migration

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



  - Nachteile
    - Entstehung langer Ketten
    - Vorherige Knoten müssen weiterhin aktiv bleiben

Gültigkeit von Objektreferenzen nach der Migration

- Eine Referenz auf ein Life Cycle-Objekt sollte gültig bleiben, solange das Objekt 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

Mobile Objekte
- IDL Value Types
- CORBA Life-Cycle Service
- Aufgabe 4

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 Life Cycle-Objekten
  - Lokaler Dienst
- FactoryFinder
  - Auffinden von `ItemFactory`-Objekten
  - Zentraler Dienst

Bibliotheks-Server — Server-Typen

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

Life-Cycle Service-Funktionalität

- Objekterzeugung
  - Keine Änderungen am `LibraryFrontend`
  - Library legt eigenständig Erstellungsort des Objekts fest, mögliche Entscheidungskriterien:
    - Individuelle Server-Auslastung
    - Zufall
    - ...
- Objektduplizierung
  - Zusätzliche Methode im Library Frontend

    ```
    void copyItem(String title)
    ```
  - Library legt eigenständig Erstellungsort der Objektkopie fest (siehe oben)
- Objektmigration
  - Zusätzliche Methode im Library Frontend

    ```
    void clearPrimary()
    ```
  - Library verteilt alle Objekte, die aktuell auf dem Primary liegen, gleichmäßig auf die vorhandenen Secondary-Server