-Testen verteilter Anwendungen

Seminar „Simulation und Bildanalyse mit Java“ im SS04

Konstantin Tjo, Urs Pricking

Übersicht

- Einführung in verteilte Anwendungen
- RMI (*Remote Method Invocation*)
- CORBA (*Common Object Request Broker Architecture*)
- EJB (*Enterprise JavaBean*)

Einführung in verteilte Anwendungen

„Verteilung ist die logische oder physikalische räumliche Entfernung von Objekten zueinander. Zwei Objekte, die zur gemeinsamen Kommunikation nicht den gewöhnliche Methodenaufruf verwenden können, sondern Mechanismen der entfernten Kommunikation nutzen müssen, sind zueinander verteilt.“

Einführung in verteilte Anwendungen

- Realisierung über Client-Server-Architektur
  - Server
    - stellt Dienste zur Verfügung (Daten, Funktionen,...)
    - sollte bekannt geben, welche Funktionalität er bietet
    - erhält Daten, wendet Funktionen an und gibt Ergebnis zurück
  - Client
    - benötigt Dienste
    - muss mit dem Server Kontakt aufnehmen können

Einführung in verteilte Anwendungen

- Verteilungsmechanismen
  - Sockets
  - Remote Procedure Calls
  - RMI
  - CORBA
  - EJB

Einführung in verteilte Anwendungen

- Ziel
  - Transparenz der Verteilung von Objekten
    - Ort und Aufrufmechanismus des entfernten Objekts sollen für den Aufrufer verborgen bleiben
    - entfernte Objekte sollten wie lokale Objekte behandelt werden

Einführung in verteilte Anwendungen

- Probleme aus Testersicht
  - Probleme der Nebenläufigkeit
  - Auffinden anderer Objekte im Netzwerk (Naming Service)
  - Unsicherheit der Kommunikation

RMI

- Was ist RMI?
  - Remote Method Invocation =
    „Entfernter Methoden Aufruf“ bzw.
    „Entfernte Methoden Ausführung“
  - RMI stellt den Mechanismus zur Verfügung, mit dem Server und Client miteinander kommunizieren und Informationen austauschen können
  - auf Java beschränkt und erfordert die Kenntnis über den Ort des entfernten Objekts

RMI

- Eine RMI-Anwendung besteht aus drei Teilen:
  - Einem Server, welcher Methoden zu Verfügung stellt, die von entfernten Objekten nutzbar sind
  - Mindestens einem Client, welcher die Methoden des Servers aufruft
  - Einem Register, welches zwischen Client und Server vermittelt (*rmiregistry*)

RMI

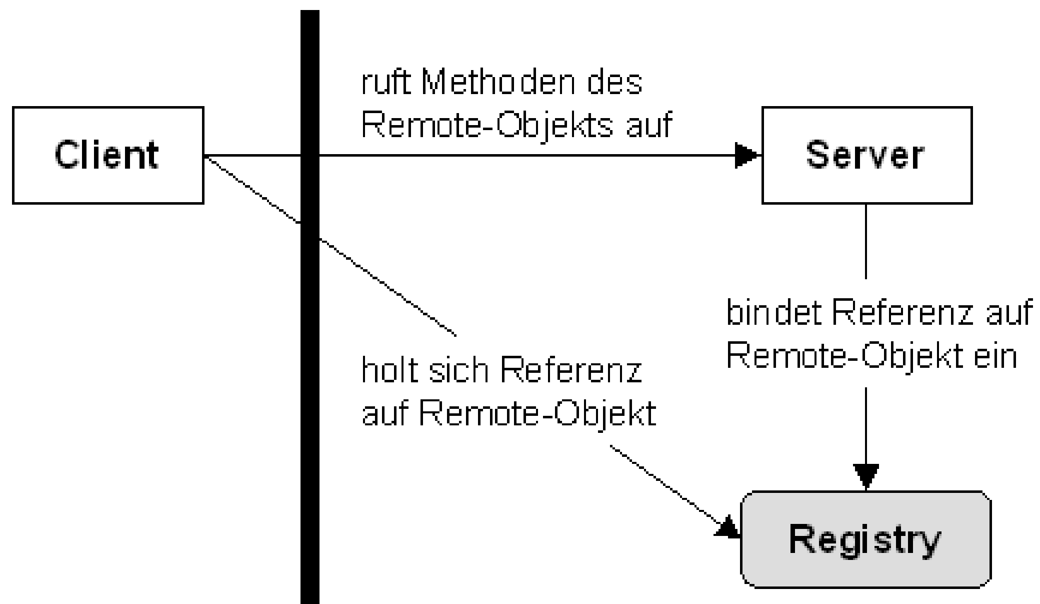
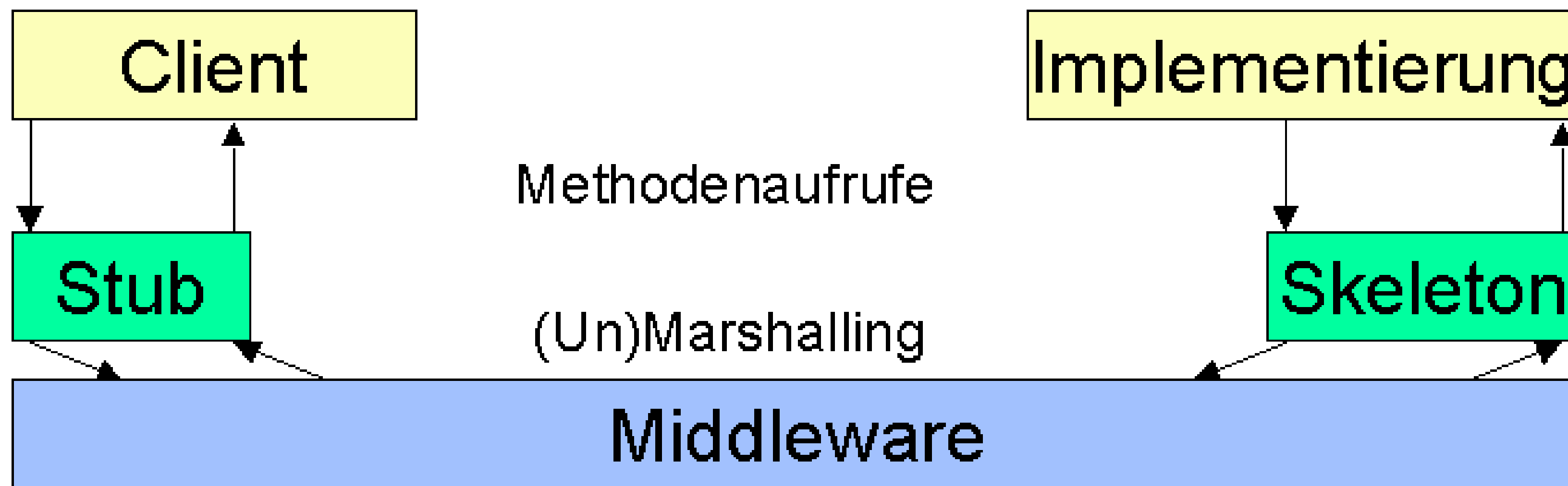


Abbildung 1: Einfaches Ablaufschema einer RMI Anwendung

RMI

- Ablauf:
  - Client ruft Methode eines entfernten Objekts auf
  - Argumente werden von einem *Stub* (Client-Vertreter) in ein transportfähiges Format umgewandelt (Marshalling) und an den Server gesendet
  - Ein *Skeleton* (Server-Vertreter) nimmt den Aufruf entgegen, decodiert ihn (Unmarshalling) und ruft die Methode des Zielobjekts auf. Rückgabewerte werden ebenso zurückgeschickt

RMI



RMI

- Testvoraussetzungen:
  - Stubs und Skeletons müssen erzeugt worden sein (z. B. mit Hilfe von *rmic*)
  - RMI-Registry muss laufen (z. B. durch einen Kommandozeilenstart mit „*rmiregistry*“)

RMI

- Folgende Aspekte werden überprüft :
  - Server:
    - Implementierung von *callService()*
    - Korrekte Verwendung von *rebind()* und unbind()
    - RMI-Fähigkeit
  - Client:
    - Korrekte Verwendung von *lookup()* beim Client
    - Implementierung von *callTwice()* beim Client
    - Error-Handling
    - RMI-Fähigkeit

RMI

- Test-First-Entwicklung eines Servers:
  - Interface (Schnittstelle):

```
import java.rmi.*;
  public interface MyRemoteServer extends Remote {
      String LOOKUP_NAME = “MyRemoteServer“;
      String callService() throws RemoteException;
  }
```

RMI

– erster lokaler Test

```
public class MyRemoteServerTest extends
  TestCase {
  public MyRemoteServerTest(String name) {...}
  public void testCallService() {
   MyRemoteServer server = new
   MyRemoteServerImpl();
   assertEquals("OK", server.callService());
  }
}
```

RMI

- Auslagerung der statischen Naming-Schnittstelle in ein Interface:

```
import java.rmi.*;
import java.net.*;
public interface MyNaming {
    void rebind(String name, Remote obj) throws
        RemoteException, MalformedURLException;
    void unbind(String name) throws RemoteException,
        NotBoundException, MalformedURLException;
}
```

RMI

- Wiederholung zu Mock-Objekten:
  - Objektattrappen
  - legen erwartetes Verhalten fest
  - verifizieren korrektes Verhalten

RMI

- Mock-Implementierung der Schnittstelle:

```
import java.rmi.*:
public class MockNaming implements MyNaming {
  public void rebind(String name, Remote obj) {...}
  public void unbind(String name) {...}
  public void expectRebind(String name, Class
   RemoteType) {...}
  public void expectUnbind(String name) {...}
  public void verify() {...}
}
```

RMI

- Test unter Verwendung des Naming-Objekts:

```
public void testCallService() throws Exception {
    MockNaming mockNaming = new MockNaming();
    mockNaming.expectRebind(
            MyRemoteServer.LOOKUP_NAME,
                        MyRemoteServerImpl.class);
    MyRemoteServer server =
      MyRemoteServerImpl.createServer(mockNaming);
    assertEquals("OK", server.callService());
    mockNaming.verify();
}
```

RMI

- das korrekte Freigeben des Servers prüfen:

```
public void testReleaseService() throws Exception {
     mockNaming.expectRebind(
          MyRemoteServer.LOOKUP_NAME,
          MyRemoteServerImpl.class);
     mockNaming.expectUnbind(
          MyRemoteServer.LOOKUP_NAME);
     server =
       MyRemoteServerImpl.createServer(mockNaming);
     ((MyRemoteServerImpl)server).release();
     mockNaming.verify();
}
```

}

RMI

- MyNaming-Implementierung unter Verwendung von *java.rmi.Naming*:

```
import java.rmi.*;
import java.net.*;
public class RMINaming implements MyNaming {
   public void rebind(String name, Remote obj)
     throws RemoteException, MalformedURLException {
             Naming.rebind(name, obj);
   }
   public void unbind(String name) throws RemoteException,
     NotBoundException, MalformedURLException {
             Naming.unbind(name);
   }
}
```

RMI

- Testfall für die Verwendung von RMI:

```
public void testRealService() throws Exception {
  RMINaming naming = new RMINaming();
  server = MyRemoteServerImpl.createServer(naming);
  MyRemoteServer client =
   (MyRemoteServer)Naming.lookup
           (MyRemoteServer.LOOKUP_NAME);
  assertEquals("OK", client.callService());
  (MyRemoteServerImpl)server).release();
}
```

RMI

- Server-Implementierung (1):

```
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
public class MyRemoteServerImpl extends
  UnicastRemoteObject implements MyRemoteServer {
  private MyNaming naming;
  private MyRemoteServerImpl(MyNaming naming) throws
  RemoteException {
   super();
   this.naming = naming;
  }
  public String callService() {
   return "OK";
  }
```

RMI

- Server-Implementierung (2):

```
    public static MyRemoteServer createServer
    (MyNaming naming) throws RemoteException,
     MalformedURLException {
    MyRemoteServer server = new MyRemoteServerImpl (naming);
    naming.rebind(LOOKUP_NAME, server);
    return server;
    }
    public void release() throws RemoteException,
     MalformedURLException, NotBoundException {
    naming.unbind(LOOKUP_NAME);
    }
}
```

RMI

- Aufgaben beim Testen des Clients:
  - korrekte Verwendung des Naming-Dienst zum Auffinden der entfernten Serverinstanz
  - korrekte Verwendung des entfernten Serverobjekts

RMI

- Test-First-Entwicklung:
  - Erweiterung der *MyNaming*-Schnittstelle um eine *lookup()*-Methode:

```
import java.rmi.*;
import java.net.*;
public interface MyNaming {
    ...
 Remote lookup(String name) throws
    NotBoundException, MalformedURLException,
    RemoteException;
}
```

RMI

- Entsprechend bei MockNaming:

```
import java.rmi.*;
public class MockNaming implements MyNaming {
    ...
    public void expectLookup(String name,
    Remote lookup {...}
    public Remote lookup(String name) {...}
}
```

RMI

- Testen der *lookup()*-Methode (1):

```
public class MyRemoteClientTest extends TestCase {
    public MyRemoteClientTest(String name){...}
    public void testLookup() throws Exception {
      MyRemoteServer remote = new MyRemoteServer() {
          public String callService() {
                 return "";
          }
      };
```

RMI

- Testen der *lookup()*-Methode (2):

```
MockNaming namingClient = new MockNaming();
namingClient.expectLookup(MyRemoteServer.LOOKUP_NAME,
  remote);
MyRemoteClient client = new
    MyRemoteClient(namingClient);
namingClient.verify();
}
}
```

RMI

- Test einer einfachen *callTwice()*-Methode (1):

```
public class MyRemoteClientTest extends TestCase {
  private MockNaming naming;
  public MyRemoteClientTest(String name) {...}
  protected void setUp() {
    naming = new MockNaming();
  }
```

RMI

- Test einer einfachen *callTwice()*-Methode (2):

```
public void testLookup() throws Exception {...}
public void testCallTwice() throws Exception {
 MyRemoteServer remote1 =
    this.createRemoteServer("Test");
 naming.expectLookup(MyRemoteServer.LOOKUP_NAME,
    remote1);
 MyRemoteClient client1 = new MyRemoteClient(naming);
 assertEquals("TestTest", client1.callTwice());
```

RMI

- Test einer einfachen *callTwice()*-Methode (3):

```
MyRemoteServer remote2 =
    this.createRemoteServer("Xyz");
naming.expectLookup(MyRemoteServer.LOOKUP_NAME,
    remote2);
MyRemoteClient client2 = new MyRemoteClient(naming);
assertEquals("XyzXyz", client2.callTwice());
}
```

RMI

- Test einer einfachen *callTwice()*-Methode (4):

```
private MyRemoteServer createRemoteServer(final
    String returnString) {
  return new MyRemoteServer() {
    public String callService() {
          return returnString;
    }
  };
 }
}
```

RMI

- Test bei Verteilungsproblemen (*hier: Reaktion des Clients, falls das Serverobjekt nicht registriert ist*):

```
public void testFailingLookup() throws Exception {
     naming.expectLookupThrowException(
       MyRemoteServer.LOOKUP_NAME,
             new NotBoundException("test"));
     try {
       MyRemoteClient client = new
             MyRemoteClient(naming);
       fail ("NotBoundException expected");
     } catch (NotBoundException expected) {}
}
```

RMI

- Test mit echtem Server:

```
public void testWithRealServer() throws Exception {
    MyNaming naming = new RMINaming();
    MyRemoteServer server =
      MyRemoteServerImpl.createServer(naming);
    MyRemoteClient client = new
      MyRemoteClient(naming);
    assertEquals("OKOK", client.callTwice());
    ((MyRemoteServerImpl) server).release();
}
```

RMI

- Bemerkung:
  - Probleme beim Deployment im Netz werden nicht abgedeckt (Verfügbarkeit der Registry, Vermeidung von Namenskonflikten, Sicherheit,...)

    ➔ Akzeptanz- und Deploymenttests
  - Probleme der Nebenläufigkeit werden nicht berücksichtigt

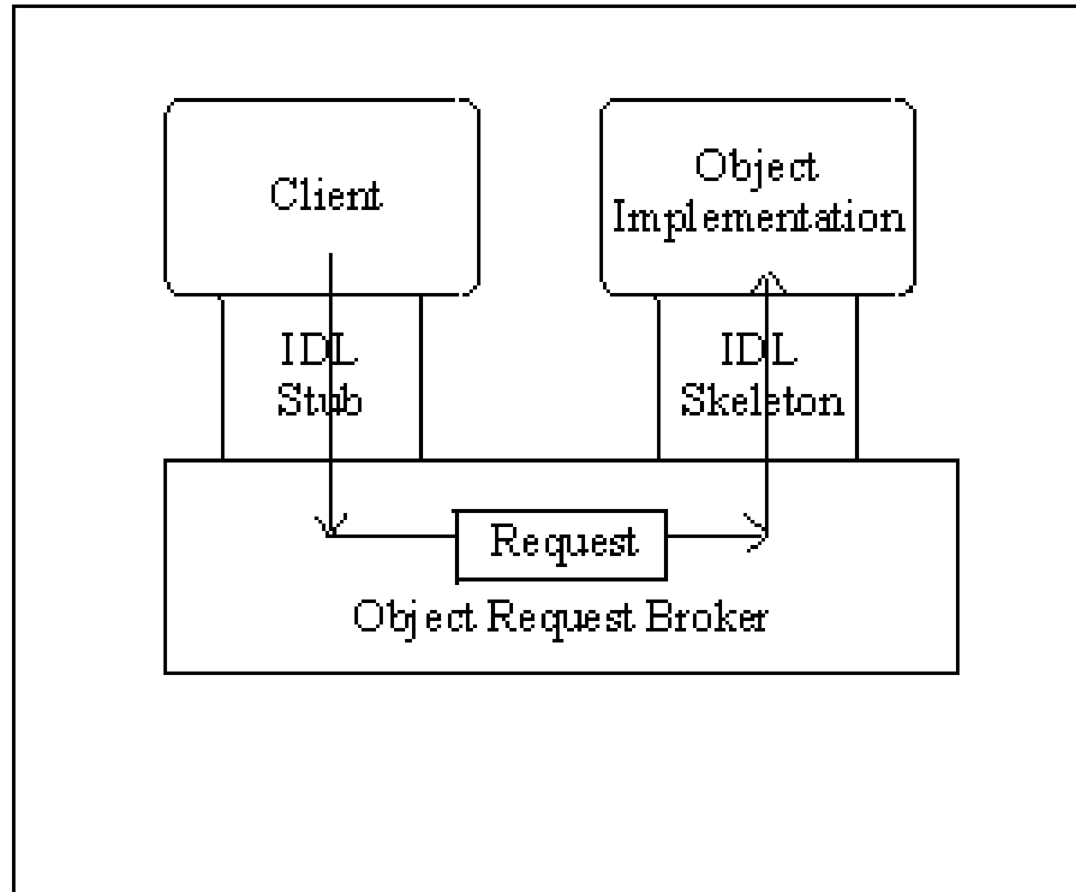
CORBA

- Kurzer Überblick:
  - CORBA = „Common Objekt Request Broker Architecture“
  - offener Standard der Open Management Group (OMG)
  - erlaubt beliebige Implementierungssprache

CORBA

- Realisierung verteilter Anwendungen ähnlich wie bei RMI
- Definition der Schnittstelle aber in *IDL (Interface Definition Language)*
  - *Keine neue Programmiersprache*
  - *Muss in die jeweilige Implementierungssprache übersetzt werden (z.B. durch idltojava)*
- Kommunikation übernimmt der *Object Request Broker (ORB)*

CORBA



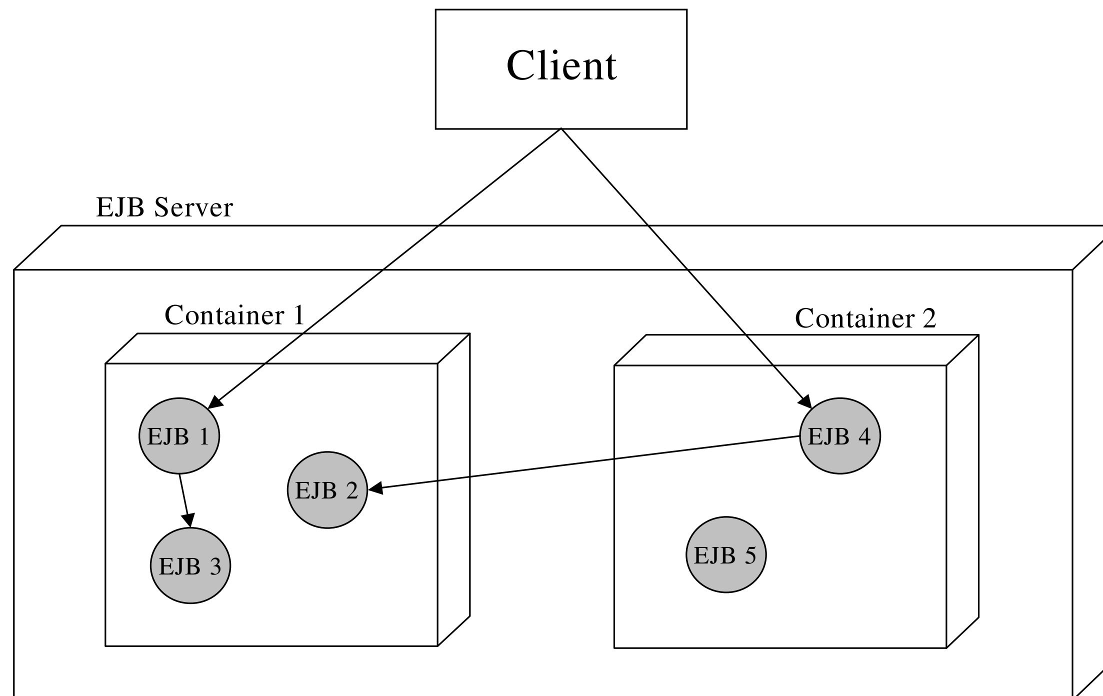
RMI vs. CORBA

- RMI
  - Einsatz in reinen Java-Umgebungen
  - Plattform-unabhängigkeit durch Java
  - leicht einzusetzen
- CORBA
  - Einsatz in heterogenen Umgebungen
  - Plattform-unabhängigkeit durch Sprachen-unabhängigkeit
  - komplizierter, aber flexibler

EJB

- Was sind Enterprise JavaBeans (EJBs)?
  - serverseitige Java-Komponenten
  - benötigen eine eigene Laufzeitumgebung, dem sogenannten *EJB-Container*
    - *Der Container ist verantwortlich für den Lebenszyklus der Komponenten, das Transaktionsverhalten, die Persistenz, Lastverteilung und Sicherheit*
  - Der EJB-Server kann mehrere Container aufnehmen und bietet ihnen den Zugriff auf Systemressourcen

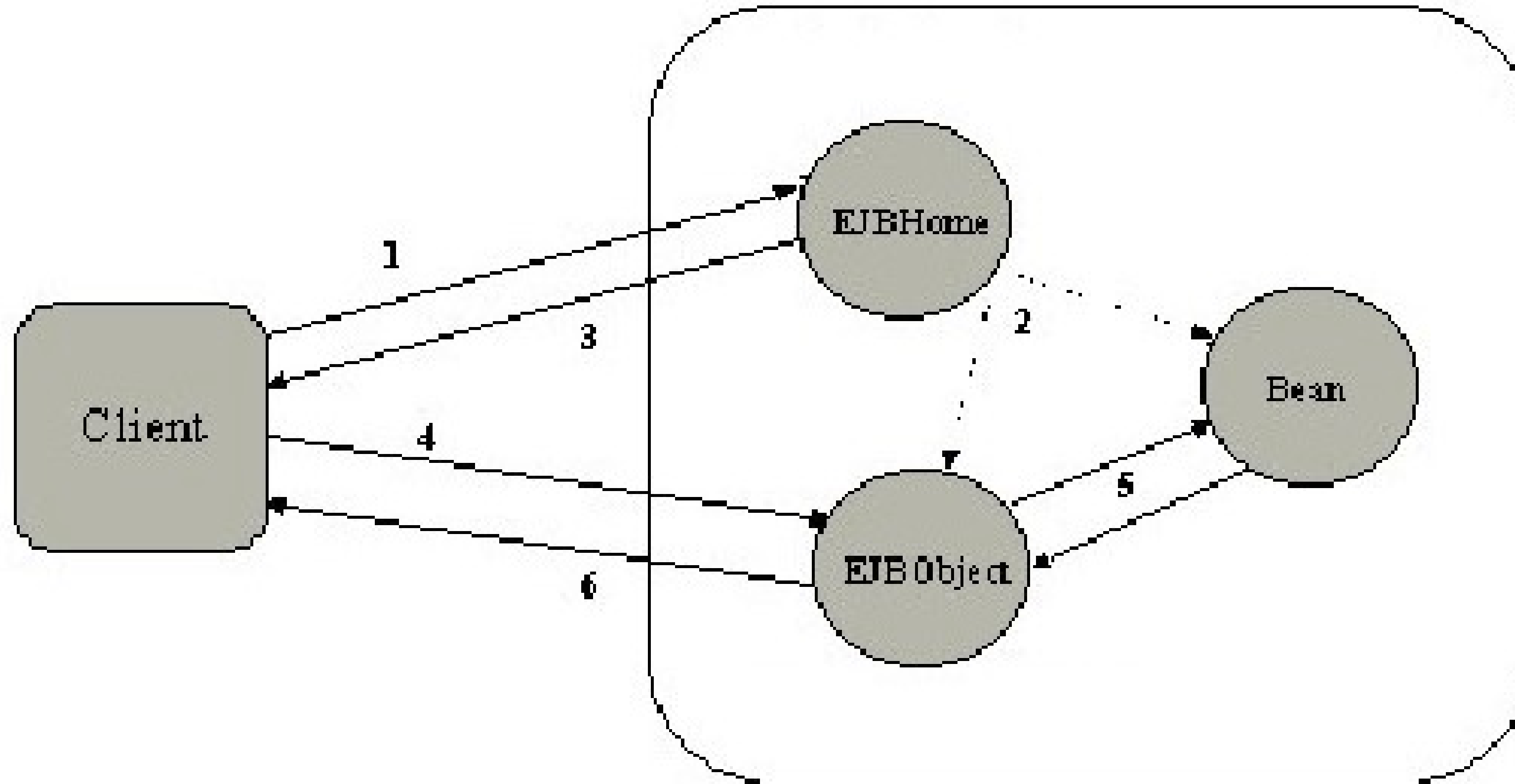
EJB



EJB

- Aufbau einer EJB-Komponente:
  - Remote-Interface:
    - Nutzung der EJB-Funktionalität über diese Schnittstelle
  - Home-Interface:
    - Erzeugen, Auffinden und Löschen der Beans
  - Bean-Implementierung:
    - Implementiert die im Remote-Interface spezifizierten Methoden
  - Deployment-Deskriptor:
    - Informationen und Anpassungen zum betrachteten Bean

EJB



EJB

- Problematik des Testens von EJBs:
  - Deployment einer EJB-Komponente dauert mehrere Minuten
  - Isolation von einzelner Komponente nicht möglich, da diese von der spezifischen Konfiguration im Deployment-Deskriptor abhängt

  ➔ Ersetzen des EJB-Containers durch Mock-Objekte nicht sinnvoll

  ➔ Testen von EJBs nur innerhalb ihres Containers sinnvoll

EJB

- Bemerkung:
  - Einführung von EJBs zu Beginn eines Projekts ist nur selten zu rechtfertigen:
    - „If you start developing a new system, chances aren‘t very high that you have a good reason to start using EJBs“ (Vera Peeters)

Quellen:

- J. Link: *Unit Tests mit Java,* dpunkt.verlag, 2002
- *http://java.sun.com/docs/books/tutorial/rmi/index.html*
- *http://java.sun.com/j2ee/docs.html*
- *http://www.omg.org*