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

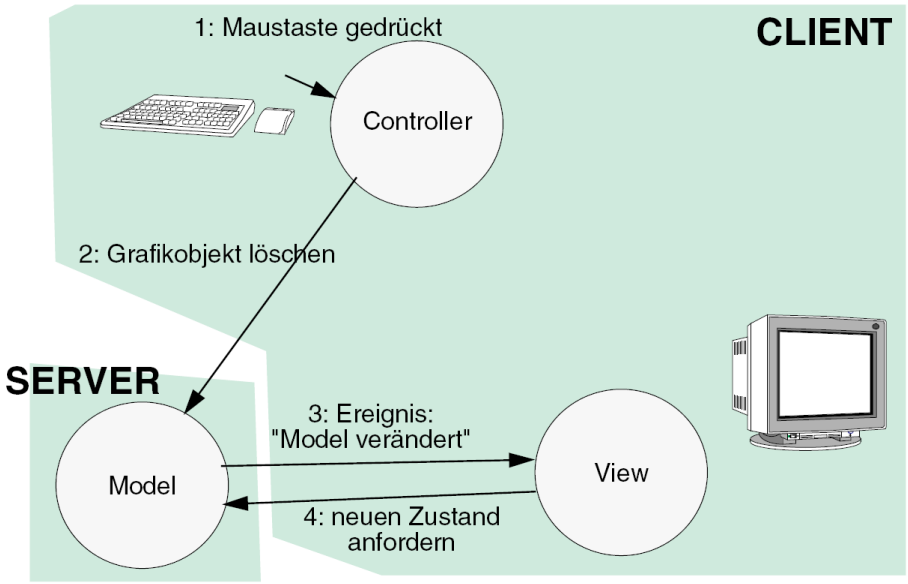
Überblick

Java RMI
- Entwurfsmuster
- Java Security
- Java Remote Method Invocation
- Aufgabe 2

Entwurfsmuster (Design Patterns)

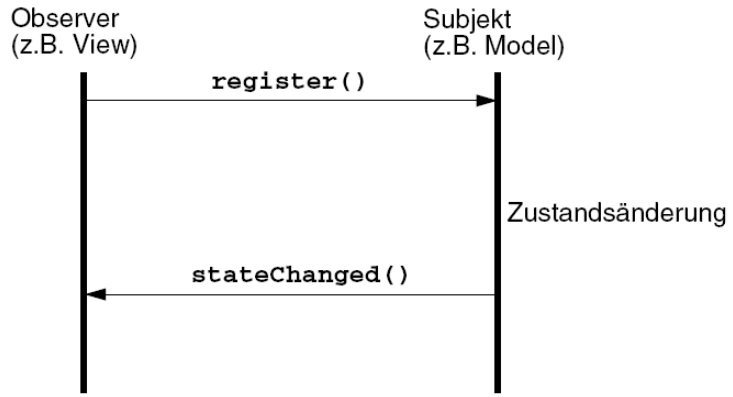
- Software-Entwicklung
  - Wiederkehrende Problemstellungen beim Entwurf von Software
  - Entwurfsmuster stellen wiederverwendbare Lösungsansätze dar
  - "*Elements of Reusable Design*"
- Beispiele
  - Model-View-Controller
  - Observer
  - Iterator
  - Proxy
  - Command
  - Factory
  - ...

Model-View-Controller



Observer

- Beobachter (*Observer*, *Listener*) achtet auf Zustandsänderungen eines Subjekts
- Beispiel: Beobachtung von Modell-Änderungen im MVC-Muster



Iterator

- „Durchlaufen" einer Menge von Objekten
- Iterator verwaltet die aktuelle Position

```
class Iter implements java.util.Enumeration {
    private int pos = 0;
    private Object[] objs;

    public Iter(Object[] objectSet) {
        objs = new Object[objectSet.length];
        System.arraycopy(objectSet, 0, objs, 0, objs.length);
    }

    public boolean hasMoreElements() {
        while((pos < objs.length) && (objs[pos] == null)) {
            pos++;
        }
        return (pos < objs.length);
    }

    public Object nextElement() { return objs[pos++]; }
}
```


Iteratoren in Java — Beispiel

- Beispiel-Menge: `java.util.HashSet`

```
Set<String> set = new HashSet<String>();
set.add("a");
set.add("b");
set.add("c");
set.add("d");
```

- `java.util.Iterator`

```
Iterator<String> setIterator = set.iterator();
while(setIterator.hasNext()) {
    System.out.print(setIterator.next() + " ");
}
```

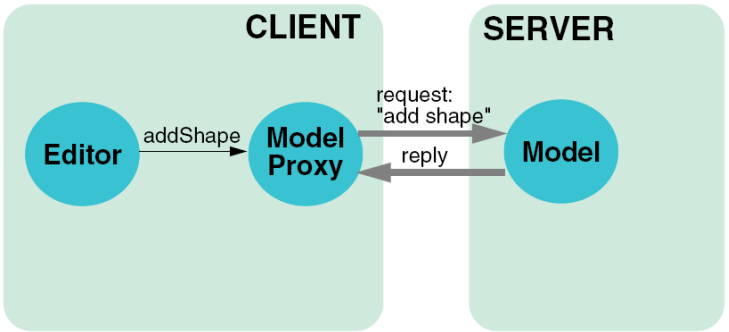
- `for`-Schleife

```
for(String s: set) {
    System.out.print(s + " ");
}
```

- Ausgabe (in beiden Fällen) unsortiert, z.B.: `d a c b`

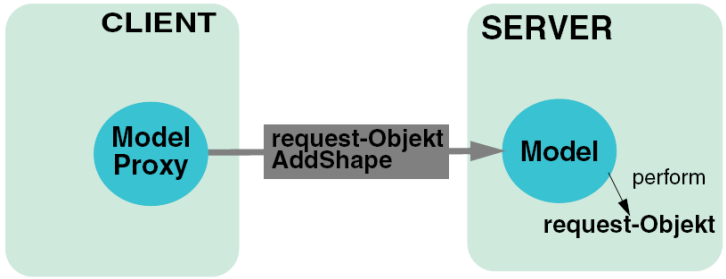
Proxy

- Lokaler Stellvertreter für ein entferntes Objekt
- Implementiert die selbe Schnittstelle wie das „echte" Objekt
- Beispiel (aus Aufgabe 1): `LibraryFrontend` mit entfernter `SimpleDB`



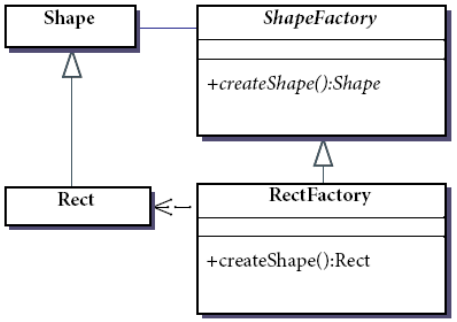
Command

- Transfer einer Anfrage zum Server
- Zustand enthält Informationen vom Client
- Parameter enthalten Informationen vom Server
- Server führt Aktion durch Aufruf einer `perform()`-Methode aus
- Beispiel (aus Aufgabe 1): Kommunikation des `LibraryFrontend` mit der Datenbank



Factory

- Definition einer Klassenschnittstelle zum Erzeugen von Objekten
- Konkrete Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist



Java Security — Überblick

- Java 1.1
  - Schwarz-Weiß-Ansatz
    - *Trusted*: lokale Applikationen & signierte Applets
    - *Non-trusted*: unsignierte Applets
  - Berechtigungen
    - Trusted: vollständiger Zugriff auf sämtliche System-Ressourcen
    - Non-trusted: stark beschränkter Zugriff auf System-Ressourcen
- Seit Java 2
  - Einsatz von Security-Policies
  - Vorteil: Zugriffsberechtigungen lassen sich feingranular erteilen, z.B.
    - nur für bestimmte Klassen
    - nur für bestimmte Ressourcen
    - nur für bestimmte Operationen (`read`, `write`, `connect`,...)
  - Erweiterung der Zugriffskontrolle auf jeglichen Java-Code (Applikationen, Beans, Servlets,...)
- Wichtigste Komponente: der Java-Security-Manager

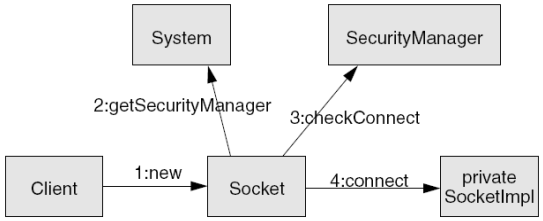
- Funktionsweise
  - Vorabüberprüfung, ob eine bestimmte Operation ausgeführt werden darf
  - Falls ja: Zugriffskontrollmethode kehrt zurück
  - Falls nein: Zugriffskontrollmethode wirft `java.lang.SecurityException`
- Zentrale Methode zur Zugriffskontrolle

```
public void checkPermission(Permission perm)
```

  - Kapselung der Zugriffsanfrage im `Permission`-Parameter
  - Überprüfung, ob Zugriffsanfrage aufgrund der geltenden Security-Policy gewährt werden darf
- Weitere Methoden
  - `checkConnect()`: Überprüfung, ob eine Socket geöffnet werden darf
  - `checkWrite()`: Überprüfung, ob in eine Datei geschrieben werden darf
  - `checkPrintJobAccess()`: Überprüfung, ob eine Druckeranweisung abgesetzt werden darf
  - ...

- Benutzung des Security-Managers
  - Security-Manager installieren: `System.setSecurityManager()`
  - Zu schützende Methode muss Zugriffskontrolle explizit aufrufen
  - Referenz auf Security-Manager erzeugen: `System.getSecurityManager()`
- Was wird durch den Security-Manager geschützt?
  - Zugriffe auf das lokale Dateisystem
  - Zugriffe auf das Betriebssystem
  - Ausführen von Programmen
  - Lesen von System-Informationen
  - Zugriffe auf das Netzwerk
  - Thread-Manipulationen
  - JVM: Linken von nativem Code, Verlassen des Interpreters, Erzeugung eines `ClassLoader`
  - Erzeugung von Fenstern
  - ...

- Zugriffskontrolle bei Erzeugung (inklusive `connect()`) eines Sockets



- Implementierung eines eigenen Security-Managers

```
public class SimpleSecurityManager extends SecurityManager {
   public void checkRead(String file) {
      Calendar cal = GregorianCalendar.getInstance();
      if(cal.get(Calendar.DAY_OF_WEEK) == Calendar.MONDAY) {
         throw new SecurityException("I don't like Mondays!");
      }
   }
}
```

- Allgemein
  - Festlegung des Security-Manager-Verhaltens
  - Standard-Policy in `$JAVA_HOME/jre/lib/security/java.policy`
  - Benutzer-Policy in `$HOME/.java.policy`
  - Angeben einer zusätzlichen Policy mit der JVM-Eigenschaft `java.security.policy`

```
java -Djava.security.policy=<URL> <Klassenname>
```

- Beispiel: Allen-ist-alles-erlaubt-Policy

```
grant {
    permission java.security.AllPermission "", "";
};
```

Fernmethodenaufrufe

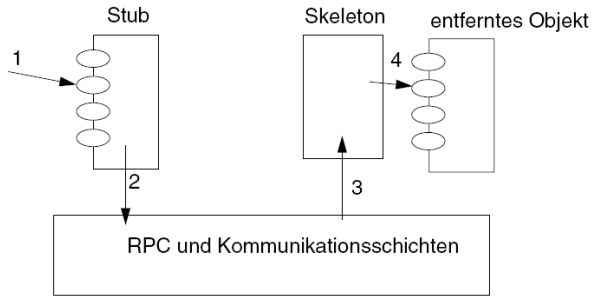
- Ermöglicht Abstraktion in einem verteilten System

Socket-Kommunikation	keine Abstraktion
↓	
Fernaufrufe (RPC)	prozedurale Abstraktion
↓	
Fernmethodenaufrufe (RMI)	Abstraktion auf Objektebene

- Fernmethodenaufrufe: Aufruf von Methoden an Objekten auf anderen Rechnern
- Transparente Objektreferenzen zu entfernten Objekten

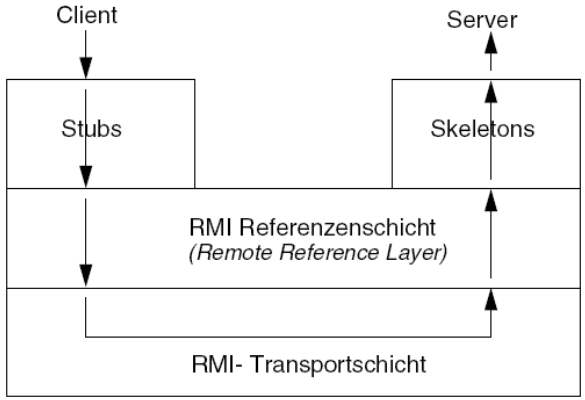
Fernmethodenaufrufe

- *Stub*: Stellvertreter des entfernten Objekts
- *Skeleton*: Ruft die Methoden am entfernten Objekt auf

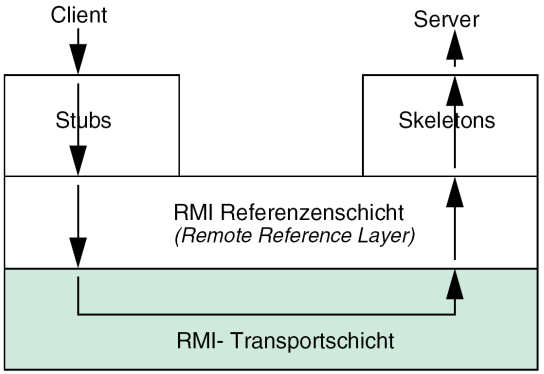


- Anfrage: Objekt ID, Methoden ID, Parameter

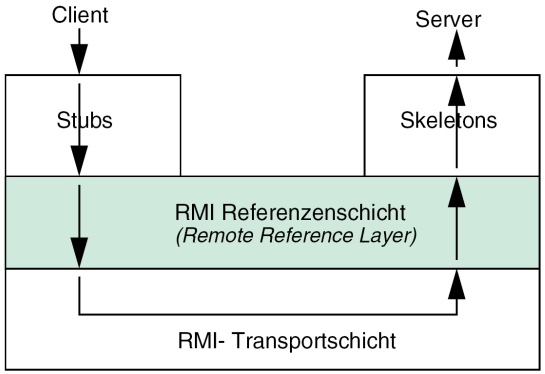
Java RMI



- Datenübertragung zwischen den Rechnern
- Implementierung
  - Aktuell: Verwendung von TCP/IP-Sockets
  - Generell: verschiedene Transportmechanismen denkbar

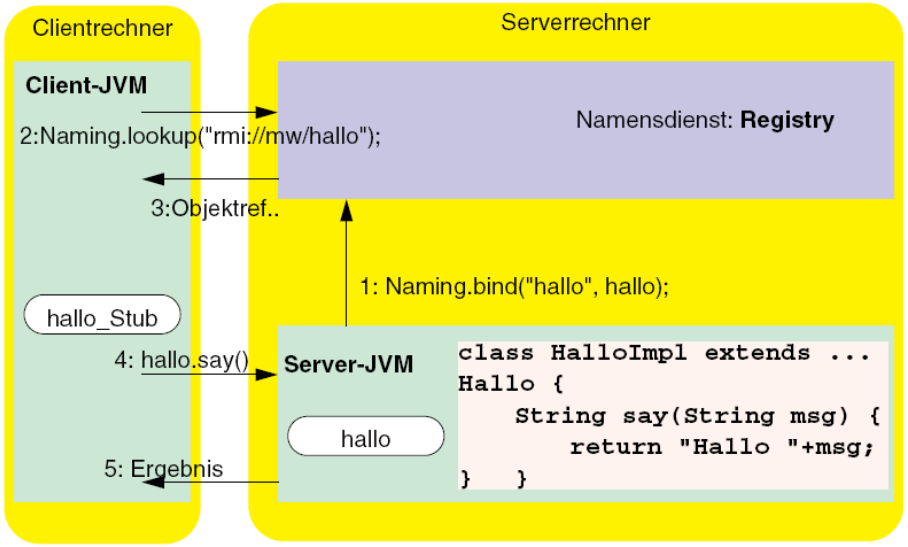


- Verwaltung von Remote-Referenzen
- Implementierung der Aufrufsemantik, z.B.
  - Unicast, Punkt-zu-Punkt
  - Aufruf an einem replizierten Objekt
  - Strategien zum Wiederaufbau der Verbindung nach einer Unterbrechung



- Stub (auf Client-Seite)
  1. erhält einen `ObjectOutputStream` von der RMI-Referenzenschicht
  2. schreibt die Parameter in diesen Strom
  3. teilt der RMI-Referenzenschicht mit, die Methode aufzurufen
  4. holt einen `ObjectInputStream` von der RMI-Referenzenschicht
  5. liest das Rückgabe-Objekt aus diesem Strom
  6. liefert das Rückgabeobjekt an den Aufrufer
- Skeleton (auf Server-Seite)
  1. erhält einen `ObjectInputStream` von der RMI-Referenzenschicht
  2. liest die Parameter aus diesem Strom
  3. ruft die Methode am implementierten Objekt auf
  4. holt einen `ObjectOutputStream` von der RMI-Referenzenschicht
  5. schreibt das Rückgabe-Objekt in diesen Strom

- *Remote-Objekt* (entferntes Objekt): Ein Objekt, das aus einer anderen JVM heraus genutzt werden kann
- *Remote-Schnittstelle*
  - Beschreibt die Methoden eines entfernten Objekts
  - Muss von `java.rmi.Remote` abgeleitet sein
  - Einzige Möglichkeit mit RMI auf ein entferntes Objekt zuzugreifen
  - Die Klasse eines entfernten Objekts muss mindestens eine Remote-Schnittstelle implementieren
- Remote-Exception (`java.rmi.RemoteException`)
  - Muss im `throws`-Clause jeder Methode einer Remote-Schnittstelle angegeben sein
  - Beim Auftreten einer Remote-Exception weiß der Aufrufer nicht, ob die Methode komplett, teilweise oder gar nicht ausgeführt wurde
- Parameter-Übergabe
  - *by reference*: Bei allen Parametern, die `java.rmi.Remote` implementieren
  - *by value*: sonst



- Abbildung von Objektnamen auf Objektreferenzen (→ Namensdienst)
- Zugriff erfolgt über die Klasse `java.rmi.Naming`
  - `void Naming.bind(String name, Remote obj)`

    Registriert ein Objekt unter einem eindeutigen Namen, falls das Objekt bereits registriert ist, wird eine Exception ausgelöst
  - `void Naming.rebind(String name, Remote obj)`

    Registriert ein Objekt unter einem eindeutigen Namen, falls das Objekt bereits registriert ist, wird der alte Eintrag gelöscht
  - `Remote Naming.lookup(String name)`

    Liefert die Objektreferenz zu einem gegebenen Namen
  - `void Naming.unbind(String name)`

    Löscht den Namenseintrag
- Die Registrierung ist nur bei der lokalen Registry möglich

Zuweisung eines bestimmten Ports

- Standard-Port für TCP/IP-Verbindungen: 1099
- Alternativer Port, z.B. 10412

```
rmiregistry 10412
```

- Wenn eine Registry an einem bestimmten Port verwendet werden soll, so muss die URL, die bei `bind()`/`rebind()`/`unbind()`/`lookup()` verwendet wird, diesen Port enthalten

  Beispiel:

```
Naming.rebind("rmi://faui40:10412/hallo", hallo);
Naming.lookup("rmi://faui40:10412/hallo");
```


Bank-Beispiel

- Remote-Schnittstelle
- Server
- Server-Initialisierung
- Client
- Starten des Systems

Anforderungen

- Jedes entfernte Objekt muss eine Schnittstelle bereitstellen, die alle Methoden enthält, die das Objekt seinen Clients anbieten soll
- Diese Schnittstelle muss von `java.rmi.Remote` erben
- Alle Methoden können/müssen eine `java.rmi.RemoteException` werfen
- Alle Parameter und Rückgabewerte müssen
  - entweder *serializable* sein (d.h. sie müssen die Schnittstelle `java.io.Serializable` implementieren)
  - oder sie müssen entfernte Objekte sein.

- Remote-Schnittstelle `Bank`

```
public interface Bank extends java.rmi.Remote {
   void deposit(Money amount, Account account)
                throws java.rmi.RemoteException;
}
```

- Remote-Schnittstelle `Account`

```
public interface Account extends java.rmi.Remote {
   void deposit(Money amount) throws java.rmi.RemoteException;
}
```

- Parameter `Money`

```
public class Money implements java.io.Serializable {
   private float value;
   public Money(float value) { this.value = value; }
}
```

- Implementierung der Remote-Schnittstelle
- Methoden müssen keine `RemoteException` werfen
- Zwei Alternativen zur Erzeugung eines Remote-Objekts
  - Der Server wird von `UnicastRemoteObject` abgeleitet

```
public class BankImpl extends UnicastRemoteObject
                      implements Bank {
    // Exception wegen Konstruktor der Oberklasse
    public BankImpl() throws java.rmi.RemoteException {
        super();
    }

    public void deposit(Money amount, Account account) {
        account.deposit(amount);
    }
}
```

  - Der Server wird mit `exportObject()` erzeugt

```
public class BankImpl implements Bank {... }
Remote bank =
       UnicastRemoteObject.exportObject(new BankImpl(), 0);
```

- Eintragen des Remote-Objekts in die Registry mit `bind()` oder `rebind()`, z.B.
  - Protokoll: `rmi`
  - Rechnername: `faui40u`
  - Registry-Port: `10412`
  - Objektname: `bank`

```
Naming.bind("rmi://faui40u:10412/bank", bank);
```

- Setzen des Security-Managers
  - Ohne Security-Manager lädt der RMI-Classloader keine Klassen vom Netz
  - Eigener Security-Manager für RMI: `java.rmi.RMISecurityManager`
  - Setzen des Security-Managers der Server-JVM

```
System.setSecurityManager(new RMISecurityManager());
```

Beispiel — Client

- Aufruf von `lookup()`, um vom Namensdienst eine Referenz auf das entfernte Objekt (den Server) zu erhalten
- Beispiel

```
public class Client {
    public static void main(String[] args) throws
                          java.net.MalformedURLException,
                          java.rmi.NotBoundException,
                          java.rmi.RemoteException {

        Bank bank = (Bank)
           java.rmi.Naming.lookup("rmi://faui40u:10412/bank");
        Account account = new AccountImpl();
        bank.deposit(new Money(10), account);
    }
}
```

Beispiel — Starten des Systems

- Klassenpfad setzen (damit Java die Klassen findet)

```
setenv CLASSPATH /proj/i4mw/<...usw...>
```

- Registry auf dem Server-Rechner starten

```
rmiregistry 10412 &
```

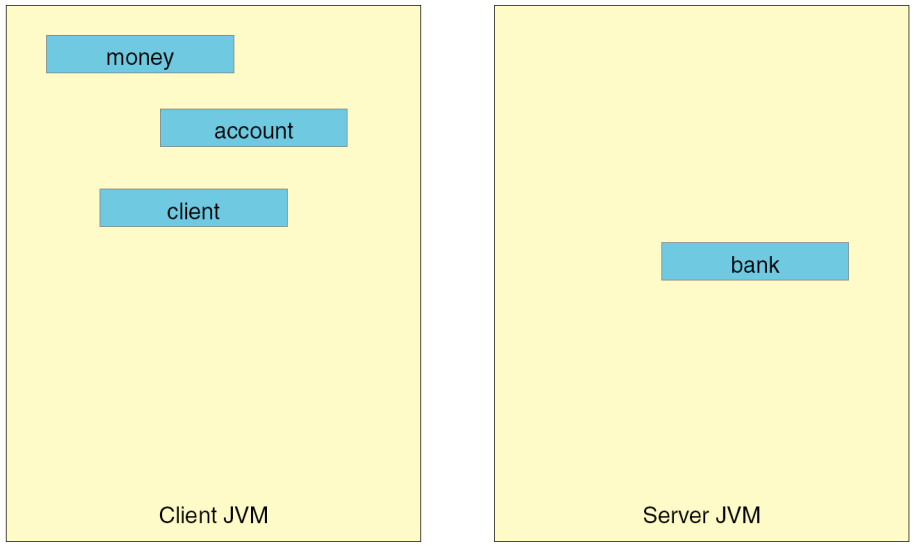
- Server-Objekt auf dem Server-Rechner starten

```
java -Djava.rmi.server.codebase=
                        file:///proj/i4mw/<...usw...>/ BankImpl
```

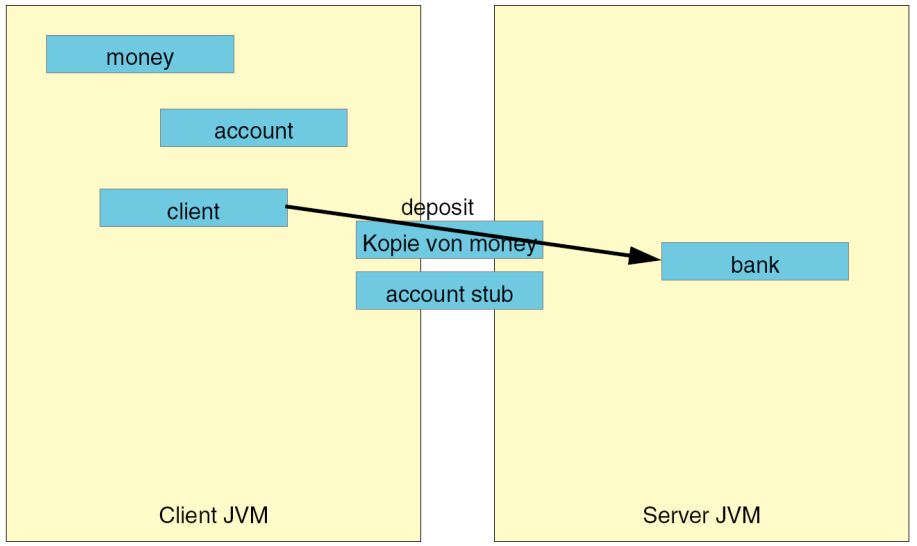
- Client starten

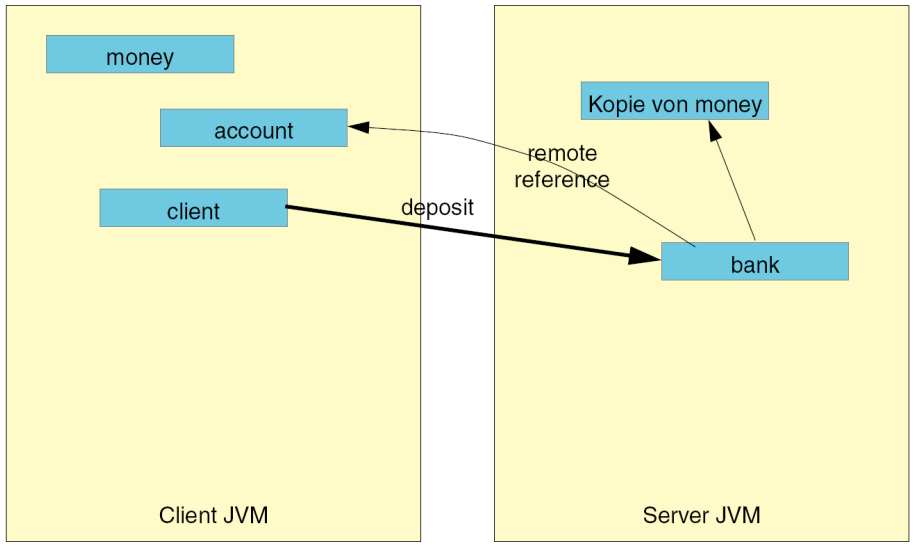
```
java Client
```

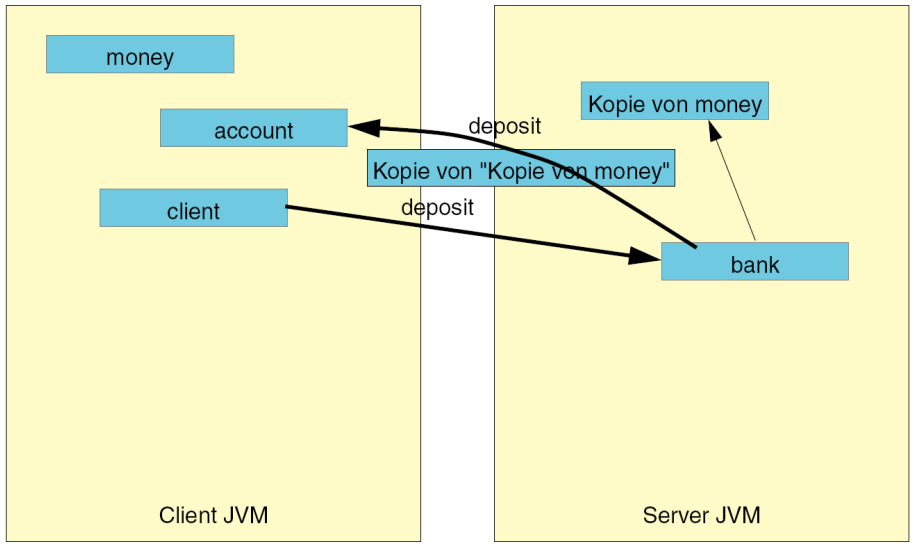
Beispiel — Interaktion



Beispiel — Interaktion







Aufgabe 2

- Ausgangssituation nach Aufgabe 1
  - Kommunikation zwischen `LibraryFrontend` und Datenbank erfolgt über selbstimplementierten Proxy
  - Anweisungen werden per `Command`-Objekt ausgetauscht
  - Ausleih-Operationen arbeiten auf lokalen Kopien → `Items` müssen gesperrt sein, bevor ihr Zustand verändert werden kann
- Änderungen in Aufgabe 2
  - Kommunikation über Fernaufrufe (→ RMI)
  - Fernaufrufe arbeiten direkt auf dem entfernten Objekt → kein Frontend-seitiges Locking mehr notwendig
- Zusatz
  - Rückruf der Datenbank an alle `LibraryFrontends` bei Registrierung eines neuen `Item`
  - Implementierung mittels Observer-Entwurfsmuster