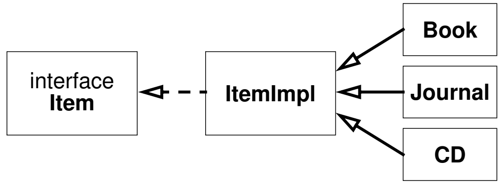
D Überblick über die 3. Übung

- Lösungsskizze der Aufgabe1
- Hinweise zur Aufgabe 2
- Netzwerkprogrammierung
- Entwurfsmuster
- Unit-Tests mit JUnit

D.1 Lösung der 1. Aufgabe

- Item.java
- ItemImpl.java
- LibraryDBImpl.java
- LibraryFrontend.java

1 Item.java




```
interface Item {
    String getTitle();
    void setTitle(String title);
    void borrow() throws AlreadyBorrowedException;
    void giveback() throws NotBorrowedException;
    void incLendingcount();
    int getLendingcount();
}
```


2 ItemImpl.java

```
public abstract class ItemImpl implements Item, Cloneable {
        private String title;
        private boolean borrowed;
        private int lendingcount;
        private int id;

        public ItemImpl() { lendingcount = 0; }
        public ItemImpl(String title) { this(); this.title = title; }

        // implementation of getter and setter
        // setId, getId, getTitle, setTitle
        // incLendingcount, getLendingcount()
        ...

        public void borrow() throws AlreadyBorrowedException {
                if (borrowed) throw new AlreadyBorrowedException();
                borrowed = true;
        }
        public void giveback() throws NotBorrowedException {
                if (!borrowed)
                        throw new NotBorrowedException();
                borrowed = false;
        }
                                                        //- Fortsetzung folgt -
```

2 ItemImpl.java (2)

```
    Item cloneMe() {
        try {
            Item klon = (Item)this.clone();
            return (Item)klon;
        } catch(CloneNotSupportedException e) {
            return null;
        }
    }
}
```

■ Book.java

```
public class Book extends ItemImpl
{
    public Book() {
        super();
    }
    public Book(String title) {
        super(title);
    }
}
```


3 LibraryDBImpl.java

```
import java.util.*;

class LibraryDBImpl implements LibraryDB {
    private int current_id = 0;
    private Hashtable lockedItems = new Hashtable();
    private Hashtable freeItems = new Hashtable();

    public int register(Item item) {
        current_id++;
        ((ItemImpl)item).setId(current_id);
        freeItems.put(new Integer(current_id), item);
        return current_id;
    }

    public Item get(int id) throws NotFoundException {
        ItemImpl itemimpl = (ItemImpl)freeItems.get(new Integer(id));
        if (itemimpl == null)
            itemimpl = (ItemImpl)lockedItems.get(new Integer(id));
        if (itemimpl == null)
            throw new NotFoundException();
        return (Item)itemimpl.cloneMe();
    }

                                                //- Fortsetzung folgt -
```


3 LibraryDBImpl.java (2)

```
    public synchronized Item lock(int id) throws AlreadyLockedException,
                                                 NotFoundException {
        ItemImpl itemimpl = (ItemImpl)freeItems.get(new Integer(id));
        if (itemimpl != null) {
            freeItems.remove(new Integer(id));
            lockedItems.put(new Integer(id), itemimpl);
            return (Item)itemimpl;
        }
        itemimpl = (ItemImpl)lockedItems.get(new Integer(id));
        if (itemimpl != null)
            throw new AlreadyLockedException();
        throw new NotFoundException();
    }

    public synchronized void unlock(Item item) throws NotLockedException{
        ItemImpl itemimpl = (ItemImpl)item;
        if (lockedItems.containsValue(itemimpl)) {
            int id = itemimpl.getId();
            lockedItems.remove(new Integer(id));
            freeItems.put(new Integer(id), itemimpl);
        }else
            throw new NotLockedException();
    }
}
```


4 LibraryFrontend.java

```
import java.io.*;
import java.util.*;

class LibraryFrontend {
    private int current_id = 0;
    LibraryDBImpl library = new LibraryDBImpl();

    private int searchItem(String title) throws NotFoundException {
        int i = 1;
        while (true) {
            Item item = library.get(i);  // throws NotFoundException
            if (title.equals(item.getTitle())) break;
            i++;
        }
        return(i);
    }

                                                //- Fortsetzung folgt -
```

4 LibraryFrontend.java (2)

```
void registerItem(String classname, String title)
                                throws NotFoundException {
    try {
        Class itemClass = Class.forName(classname);
        Item item = (Item)itemClass.newInstance();
        item.setTitle(title);
        library.register(item);
    } catch(Exception e) { throw new NotFoundException(); }
}

void borrowItem(String title) throws NotFoundException,
                                AlreadyBorrowedException,
                                AlreadyLockedException {
    int i = searchItem(title);
    Item item = library.lock(i);
    try {
        item.borrow();  // throws AlreadyBorrowed
        item.incLendingcount();
    } finally {
        try { library.unlock(item); } catch (Exception e){}
    }
}

                                    //- Fortsetzung folgt -
```


4 LibraryFrontend.java (3)

```
int returnItem(String title) throws  NotFoundException,
                                AlreadyLockedException,
                                NotBorrowedException {
    int lendingcount;
    int i = searchItem(title);
    Item item = library.lock(i);
    try {
        item.giveback();
        lendingcount = item.getLendingcount();
    } finally {
        try { library.unlock(item);} catch (Exception e){}
    }
    return lendingcount;
}

                                    //- Fortsetzung folgt -
```


4 LibraryFrontend.java (4)

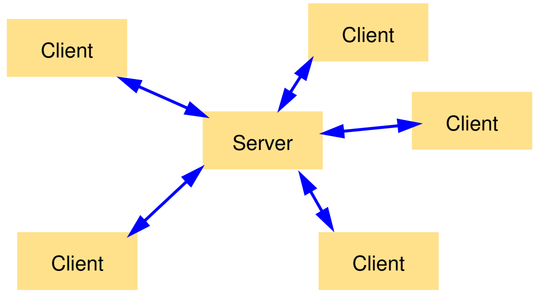
```
    static public void main(String args[]) {
        LibraryFrontend frontend = new LibraryFrontend();
        InputStreamReader is = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(is);
        StringTokenizer st;
        while (true) {
            System.out.print("> ");
            try {
                String myline = br.readLine();
                st = new StringTokenizer(myline);
                if (st.hasMoreTokens()) {
                    String command = st.nextToken();
                    if (command.equals("register")) {
                        ...
                    } else if (command.equals("quit")) {
                        ...
                    } else {
                        System.out.println("ERROR: unknown command");
                    }
                }
            } catch (IOException e) { ... break; }
        }
    }
}
```


D.2 Hinweise zur 2. Aufgabe

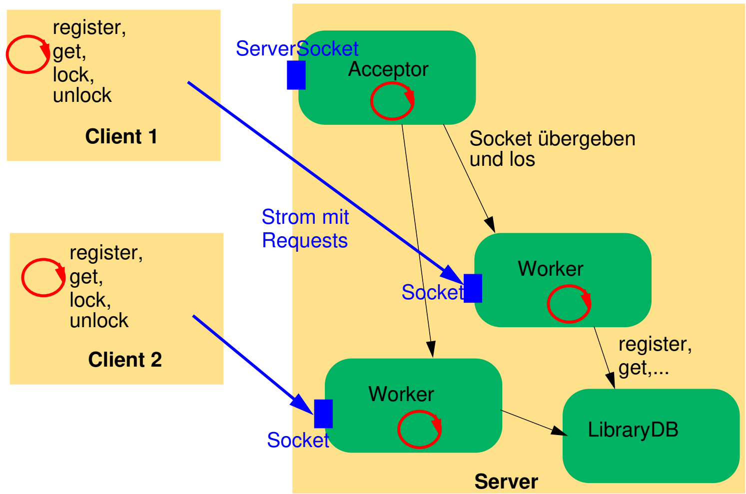
1 Objektidentität in verteilten Programmen:

- Wenn Objekte durch einen ObjectStream transferiert werden, verlieren sie ihre Identität
- Referenzen können nicht mehr zur Identitätsprüfung eingesetzt werden
- Lösung: eine Objekt ID
  - verändert sich nicht zwischen verschiedenen Rechnern und mehrmaligen Ausführungen des Programmes

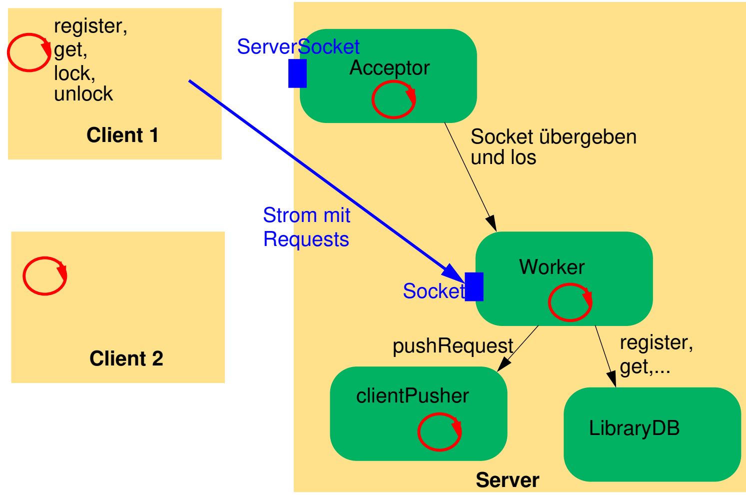
2 Architektur für Aufgabe 2



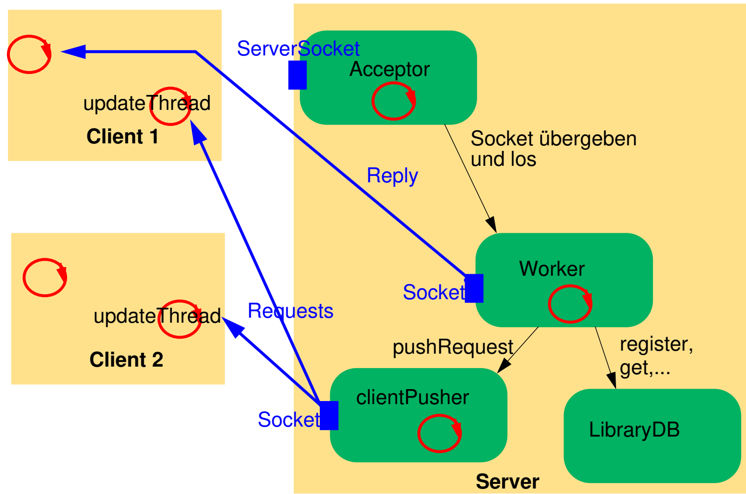
2 Architektur für Aufgabe 2 (2)



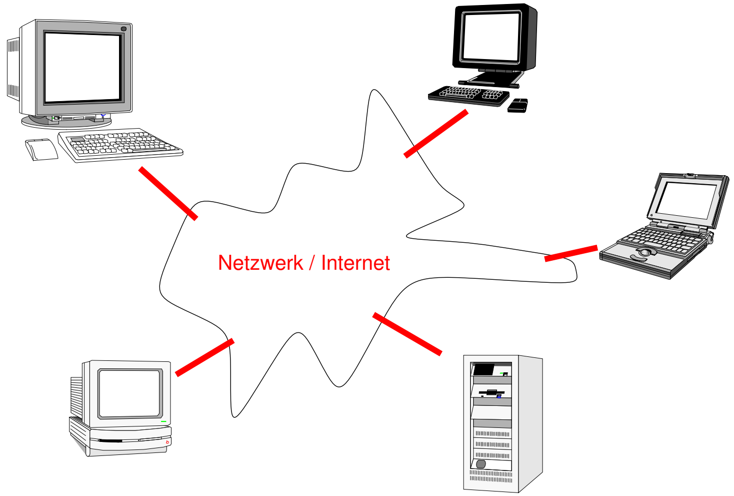
2 Architektur für Aufgabe 2 (3)



2 Architektur für Aufgabe 2 (4)



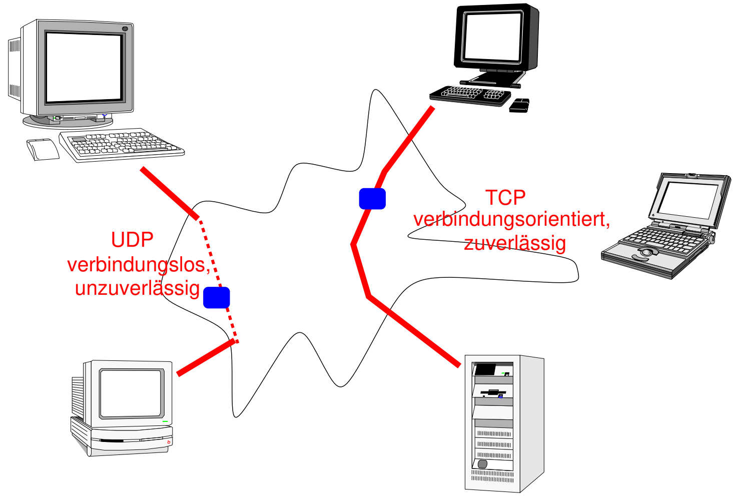
D.3 Netzwerkprogrammierung



1 Adressierung: InetAddress

- IP-Adresse:
  - DNS-Form: www4.informatik.uni-erlangen.de
  - durch Punkte getrenntes Quadtupel: 131.188.34.42
- `java.net.InetAddress` enthält IP-Adressen
- `InetAddress` hat keinen öffentlichen Konstruktor.
  Instanzen können folgendermassen erzeugt werden:
  - `getLocalHost()`
  - `getByName(String hostname)`
  - `getAllByName(String hostname)`
- `InetAddress` stellt Konvertierungsmethoden zur Verfügung:
  - `byte[] getAddress()`: IP-Adresse in Bytearray
  - `String getHostAddress()`: Stringdarstellung
  - `String getHostName()`: Rechnername (DNS-Form)

D.4 Sockets



1 Verbundungsorientierte Sockets

- `java.net.Socket`
  - TCP/IP
  - zuverlässig
  - Repräsentiert einen Kommunikationsendpunkt bei einem Client oder einem Server
- Erzeugen eines neun Sockets:

```
socket= new Socket("www4.informatik.uni-erlangen.de",80);
```

- Ein Kommunikationsendpunkt ist definiert durch *Rechnername und Port* (Ports: 16 bit, < 1024 privilegiert)
- `close` schließt den Socket.

2 ServerSocket

- **`java.net.ServerSocket`**
  - wird serverseitig verwendet um auf Verbindungsanfragen von Clients zu warten
- **`accept`** wartet auf Verbindungsanfragen
- für eine neue Verbindung wird ein neues **`Socket`**-Objekt zurückgegeben:

```
ServerSocket serverSocket = new ServerSocket(10412);
Socket socket = serverSocket.accept();
```

- **`close`** schließt den Port.

3 Ein- / Ausgabe über Sockets

- Lesen von einem Socket mittels **`InputStream`**:

```
InputStream inStream = socket.getInputStream();
```

- Schreiben auf einen Socket mittels **`OutputStream`**:

```
OutputStream outStream = socket.getOutputStream();
```

- aus diesen Strömen können leistungsfähigere Ströme erzeugt werden:

```
DataOutputStream out =
  new DataOutputStream(new BufferedOutputStream(outStream));
```


4 TCP Client/Server

Server

```
ServerSocket serverSocket = new ServerSocket(5124);
```

Client

4 TCP Client/Server (2)

Server

```
ServerSocket serverSocket = new ServerSocket(5124);
Socket socket = serverSocket.accept(); // accept blockiert
```

Client

4 TCP Client/Server (3)

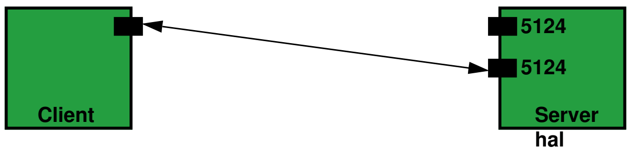
Server

```
ServerSocket serverSocket = new ServerSocket(5124);
Socket socket = serverSocket.accept(); // accept blockiert
```

Client

```
Socket socket = new Socket("hal",5124);
```


4 TCP Client/Server (4)



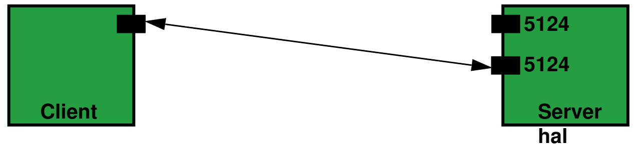
Server

```
ServerSocket serverSocket = new ServerSocket(5124);
Socket socket = serverSocket.accept(); // accept kehrt zurück
```

Client

```
Socket socket = new Socket("hal",5124);
```


4 TCP Client/Server (5)



Server

```
ServerSocket serverSocket = new ServerSocket(5124);
Socket socket = serverSocket.accept();
InputStream in = socket.getInputStream();
OutputStream out = socket.getOutputStream();
```

Client

```
Socket socket = new Socket("hal",5124);
InputStream in = socket.getInputStream();
OutputStream out = socket.getOutputStream();
```


D.5 Verbindungslose Sockets

- `java.net.DatagramSocket`
  - UDP/IP
  - unzuverlässig: Datagramme können verloren gehen!
  - geringe Latenzzeit
  - Konstruktoren:

    `DatagramSocket(int port)`
    bindet an den lokalen Port **`port`**

    `DatagramSocket()`
    bindet an irgendeinen lokalen Port
  - Methoden:

    `send(DatagramPacket packet)`
    sendet ein Paket, die Zieladresse muss im Paket eingetragen sein

    `receive(DatagramPacket packet)`
    empfängt ein Paket, die Absenderadresse ist im Paket enthalten

1 Empfänger

- Pakete an einem bestimmten Port empfangen:

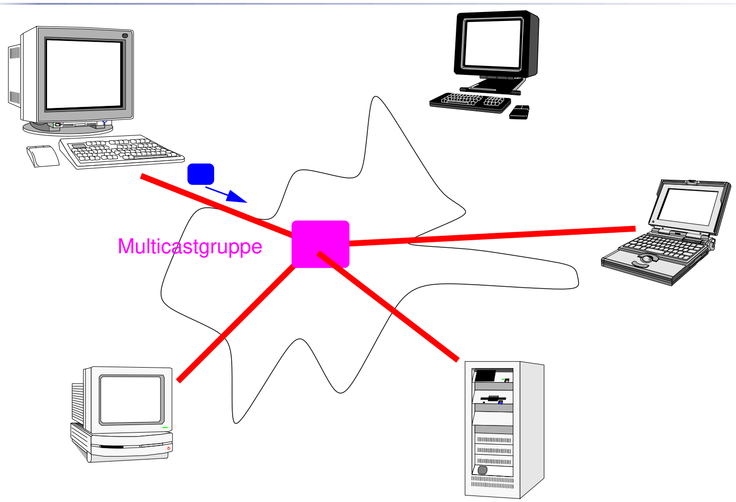
```
DatagramSocket socket = new DatagramSocket(10412);
byte[] buf = new byte[1024];
DatagramPacket packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);
InetAddress from = packet.getAddress();
int bytesReceived = packet.getLength();
```


2 Sender

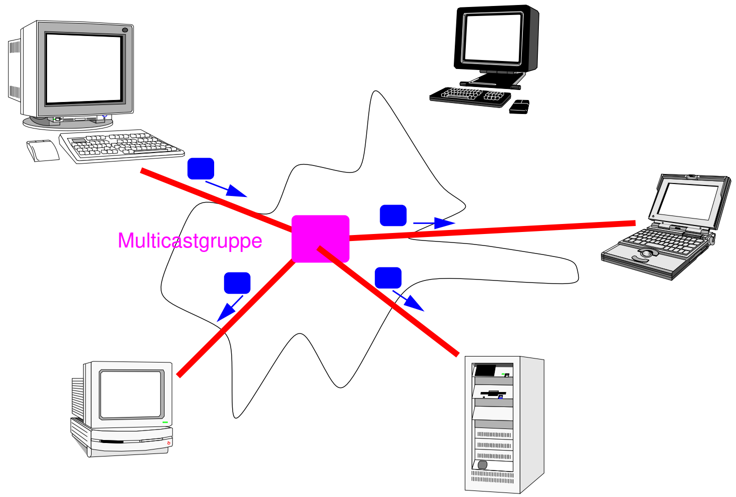
- Pakete von einem beliebigen Port verschicken:

```
DatagramSocket socket = new DatagramSocket();
byte[] buf = new byte[1024];
buf[0] = ...

InetAddress addr = InetAddress.getByName("faui40");
int port = 10412;
DatagramPacket packet;
packet = new DatagramPacket(buf, buf.length, addr, port);
socket.send(packet);
```


D.6 Multicast-Sockets



D.6 Multicast-Sockets



D.6 Multicast-Sockets

- **`java.net.MulticastSocket`**
  - verbindungslos (Unterklasse von **`DatagramSocket`**)
  - verwendet IP-Adressen der *Klasse D* (**`224.0.0.1`** bis **`239.255.255.255`**)
  - nach dem Erzeugen des Sockets kann man Pakete verschicken
  - Um Pakete zu empfangen muss man der Gruppe beitreten mittels **`joinGroup()`**
  - Die Paketverbreitung wird mit Hilfe des Parameters “time-to-live” (TTL) gesteuert.
- Beispiel:

```
InetAddress group = InetAddress.getByName("228.5.6.7");
MulticastSocket socket = new MulticastSocket(6789);
socket.setTimeToLive((byte)2);
socket.joinGroup(group);
```

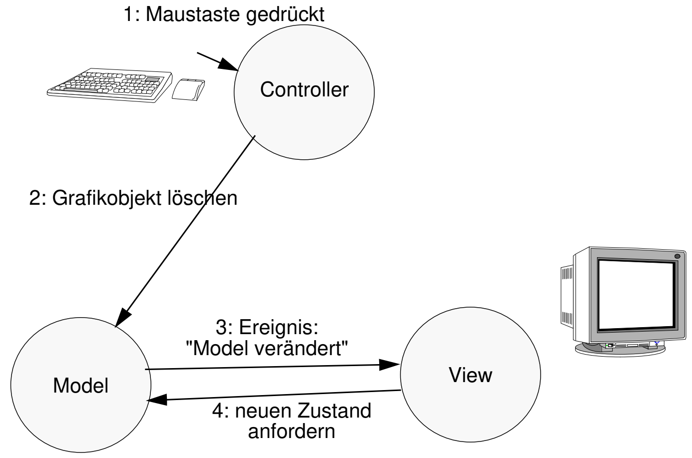

D.7 Zusammenfassung

- **`Socket`**: Endpunkt (Server oder Client) einer TCP-Kommunikation
  - enthält Zieladresse / -port und lokalen Port
  - Daten werden mittels Strömen (Streams) gelesen und geschrieben
- **`SocketServer`**: ein Server-Endpunkt, erzeugt Instanzen von **`Socket`**
- **`DatagramSocket`**: UDP-Kommunikation
  - **`send()`**/**`receive()`** um Daten zu verschicken / empfangen.
  - Zieladresse ist in einem **`DatagramPacket`**-Objekt enthalten.
- **`MulticastSocket`**: Multicast-UDP-Kommunikation
  - verwendet einen reservierten Bereich von IP-Adressen
  - bevor man Daten empfängt, muss man mittels **`joinGroup()`** einer Multicastgruppe beitreten

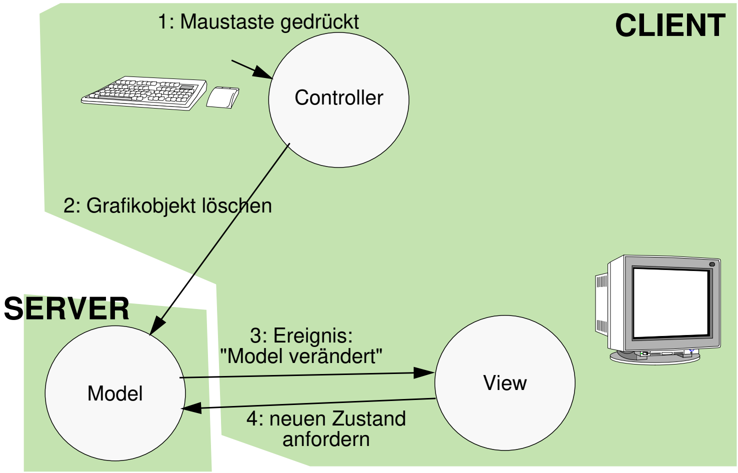
D.8 Design Patterns (Entwurfsmuster)

- "elements of reusable design"
- Beispiele:
  - Model-View-Controller
  - Observer
  - Iterator
  - Proxy
  - Command
  - Factory

1 Model-View-Controller

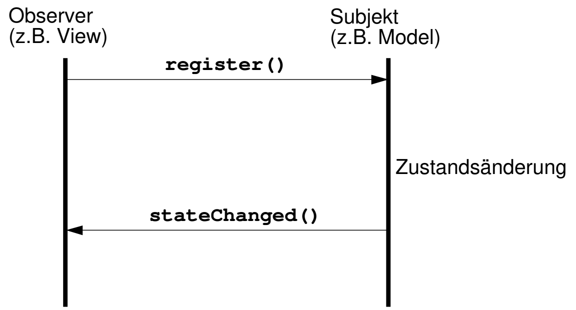


1 Model-View-Controller (2)



2 Observer

- wird z.B. im MVC-Muster verwendet um Änderungen des Modells zu beobachten



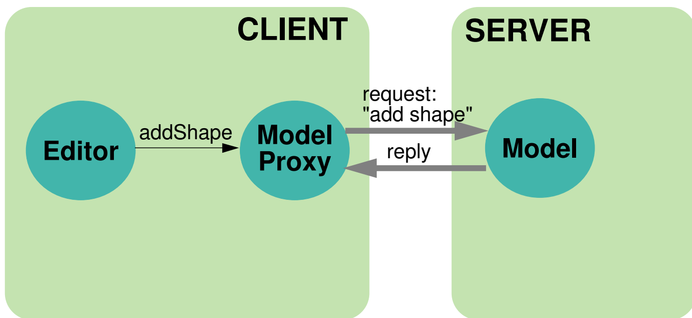
3 Iterator

- wird verwendet um durch eine Menge von Objekten zu "laufen"
- Ein Iterator ist dafür verantwortlich die aktuelle Position zu verwalten.

```
class Iter implements java.util.Enumeration {
  int cursor;
  Shape[] shapes;

  public Iter(Shape[] shapes) {
    this.shapes = new Shape[shapes.length];
    System.arraycopy(shapes, 0, this.shapes, 0, shapes.length);
  }

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

  public Object nextElement() { return shapes[cursor++]; }
}
```

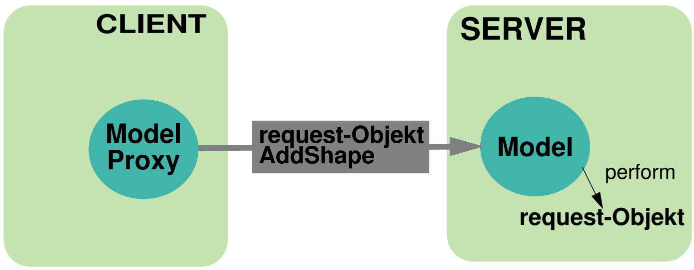

4 Proxy

- wird verwendet um einen lokalen Stellvertreter zu einem entfernten Objekt zu haben
- implementiert die gleiche Schnittstelle wie das “echte” Objekt
- WhiteBoard mit entferntem Modell:



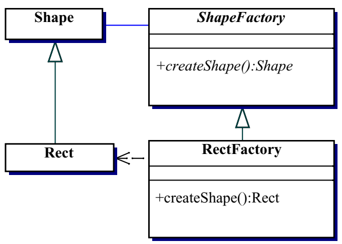
5 Command

- wird verwendet um eine Anfrage zum Server zu transferieren
- Der Zustand enthält Informationen vom Client
- Die `perform()`-Methode wird vom Server aufgerufen
- Die Parameter enthalten Informationen vom Server



6 Factory

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



D.9 Testen mit JUnit

- Test-First-Ansatz
  - Tests werden erstellt bevor der eigentliche Produktionscode implementiert wird
  - sobald ein Test erfolgreich aufgerufen werden kann ist der zugehörige Produktionscode ausreichend
  - Entwicklung verläuft in Mikro-Iterationen von 10-15 min
- Unit-Test
  - testet eine Methode einer Klasse
  - überprüft das Verhalten einer Methode unter Rand- und Normalbedingungen
  - ist jederzeit wiederholbar

D.9 Testen mit JUnit

- Vorteile der Kombination aus Unit-Tests und Test-First-Ansatz:
  - verändert den Programmcode der Applikation nicht
  - gewährleistet definierte Funktion auch nach großen Modifikationen des Programmcodes
  - weniger debugging am kompletten System
  - kleinerer Programmcode
  - reduziert den Wartungsaufwand
- JUnit:
  - Unit-Tests Framework für Java
  - weiterführende Informationen und Dokumentation unter http://www.junit.org

1 TestCase

- Sammelt alle Unit-Tests zu einer Klasse.
- Erbt von **`junit.framework.TestCase`**.
- Verfügt per Konvention über den Klassennamen **`<Name>Test.java`**.
- Beispiel:

```
import junit.framework.*;

public class MoneyTest extends TestCase {

    public MoneyTest (String name){super(name);}

    public void testAmount(){
        Money money = new Money(3.00);
        assertTrue(3.00 == money.getAmount());
    }
}
```


1 TestCase

- Erzeugen der Testbasis (*Fixture*) mit **`setUp()`**:

```
import junit.framework.*;

public class MoneyTest extends TestCase {
    private Money money;

    public MoneyTest (String name){super(name);}

    protected void setUp(){money = new Money(3.00);}

    public void testAmount(){
        assertTrue(3.00 == money.getAmount());
    }
}
```

- Aufräumen der Testbasis mit **`tearDown()`**.

2 Asserts

- **`assert()`**-Methoden überprüfen Testbedingungen:
  - **`assertTrue()`** stellt fest, ob eine Bedingung wahr ist
  - **`assertEquals()`** verifiziert, ob zwei Objekte gleich sind
  - **`assertSame()`** verifiziert, ob zwei Referenzen auf das gleiche Objekt verweisen
  - **`assertNull()`**
  - **`assertNotNull()`**
- Beispiel:

```
assertEquals("rounded amount",2,money.getAmount(),0.002);
```

- Ausgabe im Fehlerfall:

```
junit.framework.AssertionFailedError:
      rounded amount expected:<2.0> but was:<1.995>
       at MoneyTest.testRounding(MoneyTest.java:16)
```


2 Asserts

- **`fail()`** kann verwendet werden wenn es keine passende **`assert()`** Methode gibt
- Beispiel:

```
public void testIndexOutOfBoundsException() {
    Vector v= new Vector(10)

    try{
       Object o= v.elementAt(v.size());
       fail("Should raise ArrayIndexOutOfBoundsException");
    } catch (ArrayIndexOutOfBoundsException e) {}
}
```

3 Testsuiten

- Aufruf eines einzelnen Tests:

```
TestResult result = (new MoneyTest(“testAmount”)).run();
```

- Sammlung der Testfälle eines TestCase:

```
public class MoneyTest extends TestCase {

    public static Test suite(){
        TestSuite suite = new TestSuite();
        suite.addTest( new MoneyTest(“testAmount”) );
        suite.addTest( new MoneyTest(“testSimpleAdd”) );
        return suite;
    }
    ....
}
```

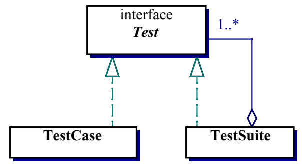

3 Testsuiten

- Sammlung der Testfälle mehrere TestCases:

```
public class AllTests extends TestCase {

    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(MoneyTest.suite());
        suite.addTest(CustomerTest.suite());
        return suite;
    }
    ....
}
```


3 Testsuiten

- UML-Darstellung der Abhängigkeiten



4 TestRunner

- Starteten eines jUnit Tests?
  - ◆ batch-TestRunner : junit.textui.TestRunner
  - ◆ swing-TestRunner: junit.swingui.TestRunner
- Beispiel:

```
public class AllTests extends TestCase {
    ...
    public static void main(String args[]){
        junit.textui.TestRunner.run(suite());
    }
}
```

4 TestRunner

- Beispiel: junit.swingui.TestRunner