Client/Server-Programmierung

WS 2017/2018

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 17. November 2017

Client/Server-Programmierung

WS 2017/2018

5 Java Komponenten-Modelle

Inhalt

- Komponenten-Modelle
- Java Beans
- Enterprise Java Beans (EJB 3)

- Burke / Monson-Haefel
- Farley / Crawford / Flanagan, Kap. 6 (EJB 2.1!)
- Orfali / Harkey, Kap. 27-34 (EJB 2.1!)
- Sriganseh / Brose / Silverman
- `http://docs.oracle.com/javase/7/docs/api/java/beans/package-summary.html`
- `http://docs.oracle.com/javaee/6/tutorial/doc`

5.1 Komponenten-Modelle

- Was sind Software-Komponenten?

  Software-Komponenten *sind ausführbare Software-Einheiten, die unabhängig hergestellt, erworben und konfiguriert werden und aus denen sich funktionierende Gesamtsysteme zusammensetzen lassen.*

- Im Vordergrund: Zusammensetzungsaspekt
- Eine Komponente ist:
  - eine funktional und technisch **abgeschlossene**, ausführbare **Einheit**
  - **unabhängig** als Einheit **entwickelbar** und **konfigurierbar**
  - **wiederverwendbar**
  - nur über genau **festgelegte Schnittstellen** ansprechbar

Begriffe

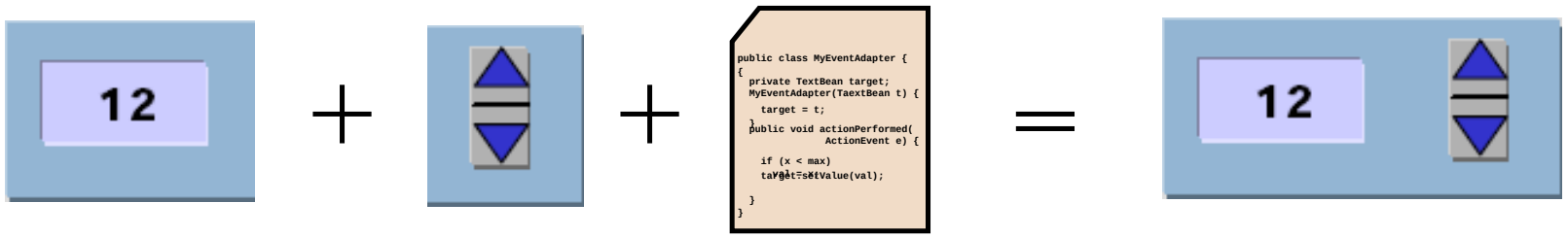
- **Visuelle Komponente**
  - stellt etwas auf dem Bildschirm (der Anwendung) dar
- **Nichtvisuelle Komponente**
  - ist für Benutzer der Anwendung nicht sichtbar
- **Komponentenmodell**: definiert
  - Architektur der Komponenten u. Struktur ihrer Schnittstellen
  - Mechanismen zur Zusammenarbeit mit Container und anderen Komponenten
- ***Container***: Ausführungsumgebung für Komponenten
  - Kontext, in dem Komponenten gruppiert und miteinander verbunden werden
  - stellt Management- und Kontroll-Dienste zur Verfügung

Software-Entwicklung mit Komponenten

- Zwei Arten von Programmierern:
  - Komponenten-Entwickler (*component developer*)
    - implementiert eine Komponente
  - Komponenten-Anwendungsentwickler (*component assembler*)
    - entwickelt eine Anwendung (oder neue Komponenten) durch Zusammenfügen existierender Komponenten
    - i.d.R. mit (graphischer) Werkzeug-Unterstützung!
- Bei Komponenten daher zwei verschiedene Nutzungsphasen:
  - *Design Time*: während der Anwendungsentwicklung
  - *Run Time*: während der Ausführung der Anwendung

Software-Entwicklung mit Komponenten: Beispiel

- Zusammenbau einer Komponente aus Teilkomponenten:



- Verwendung der Komponente in einer Anwendung:

Werkzeug-Unterstützung (Bsp: Visual Café)

Vorteile eines Komponenten-Modells

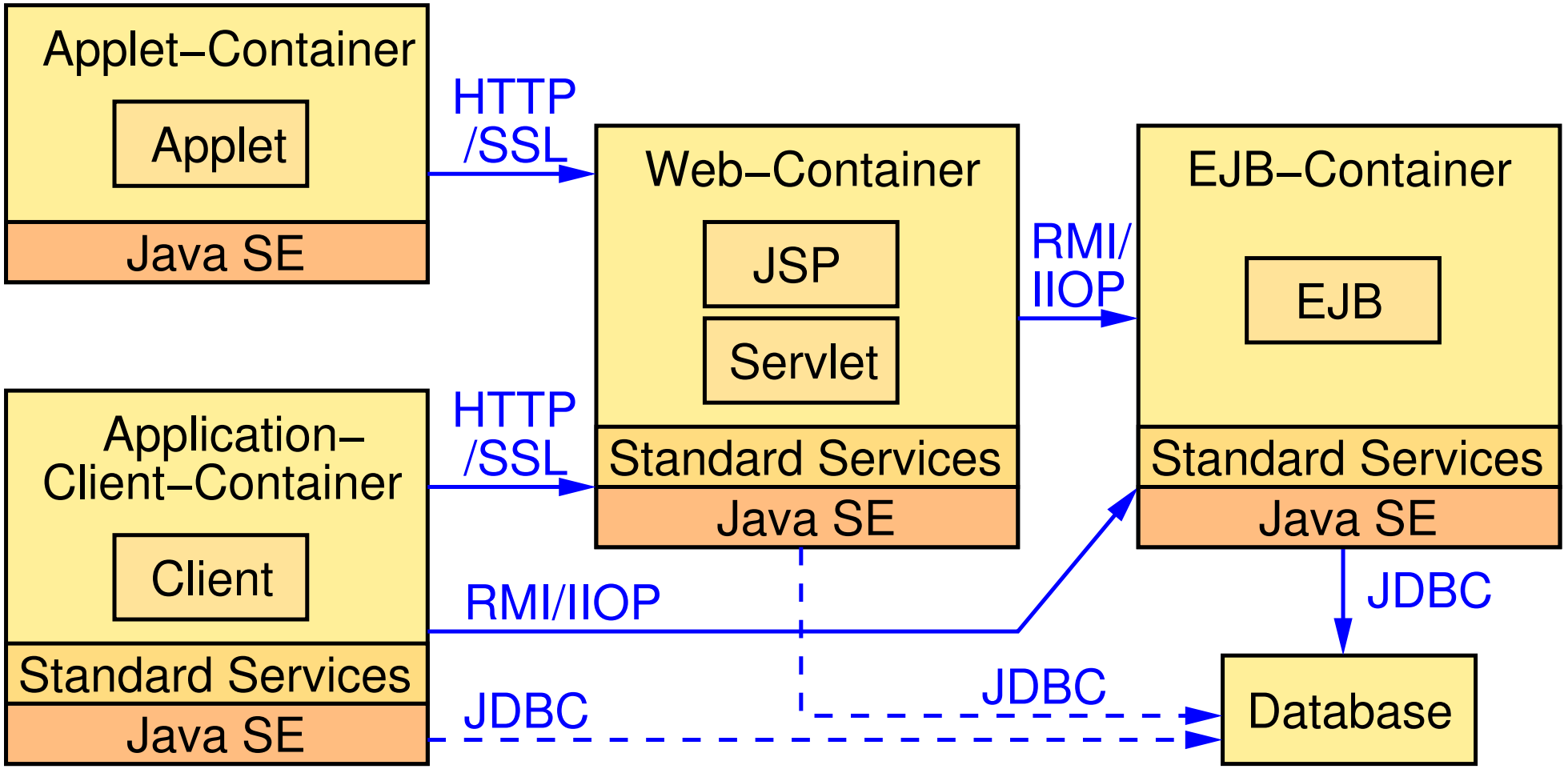
- Wiederverwendung von Software
- Vereinfachte Designphase für Anwendungen
- Vereinfachte Implementierungsphase für Anwendungen
  - „Zusammenstecken“ von Komponenten
- Bessere Skalierbarkeit
- Unterstützung durch Entwicklungswerkzeuge

- Ziel: Aufbau eines Markts von Komponenten

5.2 Komponentenmodelle in Java EE

- Java EE: Java *Enterprise Edition*
- Java EE definiert Familie von Komponentenmodellen:
  - für Client-*Tier*: JavaBeans, Applets
  - für Webserver-*Tier*: Servlets, JSP
  - für Anwendungsserver-*Tier*: Enterprise JavaBeans (EJB)
- Daneben bietet Java EE weitere Dienste, z.B.
  - Namensdienst (JNDI)
  - Nachrichten-Infrastruktur (Java Messaging Service, JMS)
  - Datenbank-Zugriff (JDBC)
  - Transaktionsdienste, Sicherheitsdienste, ...

Verteilte Anwendungen mit Java EE



5.3 JavaBeans

- JavaBeans ist Spezifikation **eines** Komponentenmodells für Java
  - definiert mehrere neue Java-Klassen und Schnittstellen
  - besteht aber i.W. nur aus Namenskonventionen bzw. Entwurfsmustern
- Ziel: Zusammenfügen von Komponenten mit Hilfe von visueller Programmierung
  - graphische Werkzeuge zur Anwendungserstellung
- Im Prinzip ist jede Java-Klasse eine JavaBean!
  - Kompositions- und Anpassungsfähigkeit aber nur garantiert, wenn Konventionen eingehalten werden
- I.d.R. besteht eine Bean aber aus mehreren Klassen

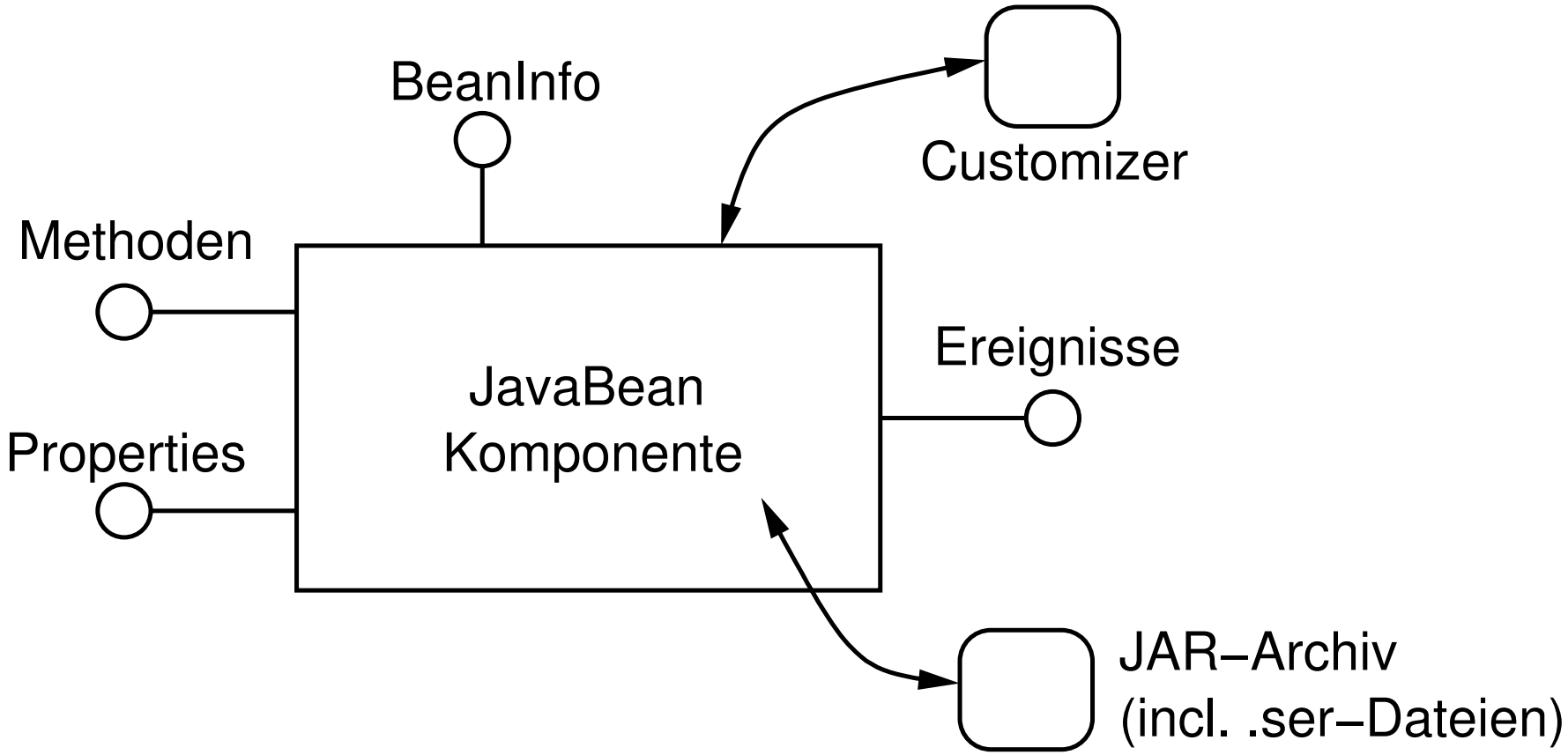
Was bietet eine JavaBean?

- **Methoden**
  - wie jede andere Java-Klasse auch ...
- **Eigenschaften (*Properties*)**
  - Attribute, über die mit Get- und Set-Methoden zugegriffen werden kann
- **Ereignisse (*Events*)**
  - werden von der Bean erzeugt
  - können über *Event Listener* an andere Beans weitergegeben werden

Was bietet eine JavaBean? ...

- **Introspektion**
  - Bean stellt über eine `BeanInfo`-Klasse Information über ihre Methoden, Eigenschaften und Ereignisse bereit
    - automatisch oder manuell erzeugt
- **Anpassung (*Customization*)**
  - erlaubt Konfiguration der *Properties* über graphische Werkzeuge
- **Persistenz**
  - *Properties* der Bean können dauerhaft gespeichert und bei Erzeugung der Bean geladen werden
    - serialisiertes Objekt in `.ser`-Datei

Black-Box Sicht einer JavaBean



Design-Time und *Run-Time*-Beans

- *Run-Time*-Bean:
  - Sammlung von Klassen mit Programmlogik
  - plus ggf. serialisierte Objektzustände (`.ser`-Dateien)
- *Design-Time*-Bean:
  - zusätzliche Klassen für
    - *Customizer* bzw. *Property*-Editoren
    - Introspektion (*BeanInfo*)
  - Icons für die Bean
  - diese Klassen und Icons werden nur von graphischen Entwicklungsumgebungen benötigt
- Klassen (und `.ser`-Dateien) sind typischerweise in einem Java-Archiv abgelegt

JDK als Komponenten-Framework

- Für JavaBeans übernimmt JDK die Rolle des Komponenten-Frameworks
- Dienste für JavaBeans:
  - Graphisches Layout und Darstellung
    - für visuelle Komponenten: AWT bzw. Swing
    - visuelle Bean erbt von AWT `Component`;
      AWT `Container` dient als Komponenten-Container
  - Ereignismodell, Properties, Introspektion, Persistenz, *Customization*
    - Bereitstellung der benötigten Klassen / Interfaces

Beispiel: *Smiley*-Bean

```
public class SmileyBean extends Canvas {
  private boolean smile = true;
  ...
  public SmileyBean() {
    setSize(250,250);
  }
  public synchronized void toggleSmile() {
    smile = !smile;
    repaint();
  }
  public void paint(Graphics g) { ... }
}
```

- Vollständiger Code: Orfali / Harkey, Kap. 28 bzw. CD-ROM
- `SmileyBean` ist lediglich eine normale Java-Klasse

Beispiel: *Container* für *Smiley*-Bean

```
public class SmileyPlace extends Frame ... {
  SmileyPlace() {
    SmileyBean smiley = null;
    // instantiiere die Bean
    try {
      smiley = (SmileyBean)Beans.instantiate(null,
                                             "SmileyBean");
    }
    catch (Exception e) { ... }
    // füge Bean in Container ein
    add(smiley);
    ...
  }
  static public void main(String args[]) { ... }
  ...
```

Konventionen für Beans

- Beans dürfen nicht mit `new` erzeugt werden
  - stattdessen: `Beans.instantiate(classloader, name)`
- Beans müssen einen *Default*-Konstruktor (ohne Argumente) besitzen
- Visuelle Beans sollten von `java.awt.Component` (oder einer Unterklasse) abgeleitet werden
- Als Komponenten-Container sollte ein AWT Container dienen
  - Hinzufügen der Beans über die Methode `add()`
- *Cast*-Operationen oder `instanceof` sollten nicht verwendet werden
  - stattdessen: `Beans.getInstanceOf(obj, class)` bzw. `Beans.isInstanceOf(obj, class)`

5.3.1 Ereignisse

- JavaBeans können Ereignisse erzeugen bzw. Ereignisse behandeln
- Ereignismodell: *Publish-Subscribe-Modell*
  - identisch mit dem Ereignismodell des AWT
  - Beans geben (über *Introspection*) bekannt, welche Ereignisse sie auslösen können
  - interessierte Beans registrieren sich bei der Ereignisquelle
- Ereignisse sind Objekte, die von `EventObject` abgeleitet sind
  - beinhalten Ereignisquelle sowie beliebige andere Parameter des Ereignisses
  - Namenskonvention: Klassenname muß mit `Event` enden
    - z.B. `ButtonEvent`

Ereigniskonsumenten (*Event Listener*)

- Erhalten Ereignisbenachrichtigungen
- Müssen Schnittstelle implementieren, die von `java.util.EventListener` abgeleitet ist
  - Namenskonvention: Ereignisname mit Endung `Listener`
  - pro Ereignistyp eine Methode, ohne Namenskonvention
    - Methode erhält Ereignisobjekt als Parameter
- Beispiel (für Ereignis `ButtonEvent`):

```
public interface ButtonListener extends EventListener {
  public void onPress(ButtonEvent e);
  public void onRelease(ButtonEvent e);
}
```

Ereignisquellen (*Event Sources*)

- Sind Beans, die Ereignisse erzeugen
- Ereignisse werden für *Listener* über zwei Methoden verfügbar gemacht (Namenskonvention!):
  - `add`*`ListenerType`*`()`: Anmelden eines *Listeners*
  - `remove`*`ListenerType`*`()`: Abmelden eines *Listeners*
  - z.B. für `ButtonEvent`:

```
void addButtonListener(ButtonListener l)
void removeButtonListener(ButtonListener l)
```

- Ereignisquelle ist für Verwaltung der Listener und Aufruf der Methode zur Ereignismeldung (z.B. `onPress()`) verantwortlich
  - typisch mittels `Vector` und Schleife über alle *Listener*

Beispiel: *Smiley* und *Button*

- `Button` ist AWT-Element und auch JavaBean
  - stellt Ereignis `ActionEvent` zur Verfügung:

```
public interface ActionListener extends EventListener {
  public void actionPerformed(ActionEvent e);
}
```

- *Button*-Ereignis soll zum Umschalten des *Smiley* führen
- Einfachste Realisierung: „direkte Verdrahtung“
  - `SmileyBean` implementiert `ActionListener`-Interface

```
public class SmileyBean extends Canvas
       implements ActionListener {
  ...
  public void actionPerformed(ActionEvent e) {
    toggleSmile();
  }
```

Beispiel: *Container*

```
public class SmileyPlace extends Frame ... {
  SmileyPlace() {
    SmileyBean smiley = null;
    Button button = null;
    try { // instantiiere die Beans
      smiley = (SmileyBean)Beans.instantiate(null, "SmileyBean");
      button = (Button)Beans.instantiate(null, "java.awt.Button");
    }
    catch (Exception e) { ... }

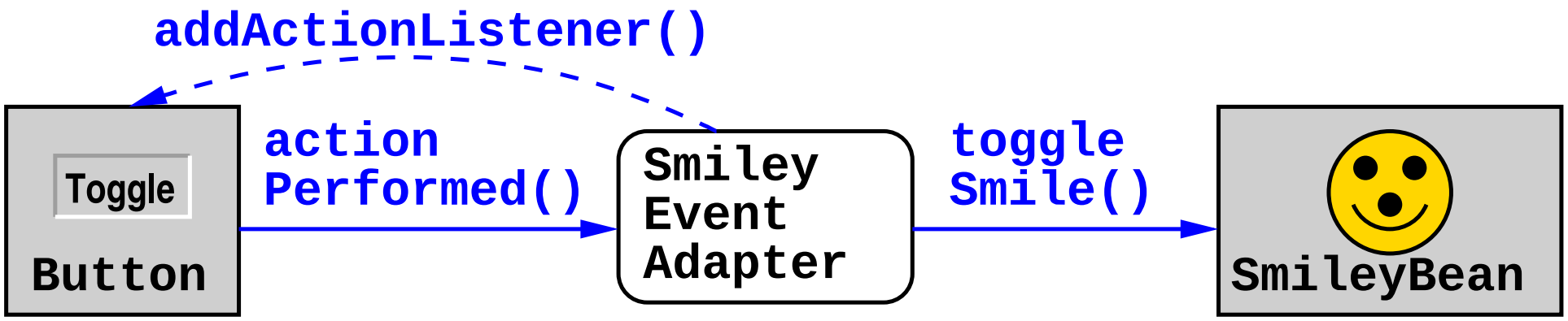
    add(smiley, "North"); // füge Beans in Container ein
    add(button, "South");

    button.addActionListener(smiley); // verbinde die Beans
    ...
  }
```

Ereignis-Adapter

- Nachteil der direkten Verdrahtung:
  - *Listener* muß passende Schnittstelle implementieren
  - widerspricht Idee, Beans über Werkzeuge zu „verdrahten“
    - Bean kann Ereignisse nicht a-priori kennen
    - Code der Bean kann nicht geändert werden
- Abhilfe: Verwendung eines Adapters (Indirektion)
  - Adapter implementiert passende *Listener*-Schnittstelle und ruft Methode in der Ziel-Bean auf
  - Vorteile (u.a.):
    - Beans bleiben vollständig unabhängig voneinander
    - Adapter kann Schnittstellen ggf. anpassen
    - Adapter kann Ereignis auch an *Remote*-Objekt leiten

Beispiel: *Smiley* und *Button*




```
public class SmileyEventAdapter implements ActionListener {
  private SmileyBean target;
  SmileyEventAdapter(SmileyBean t) {
      target = t;
  }
  public void actionPerformed(ActionEvent e) {
      target.toggleSmile();
  }
}
```

5.3.2 *Properties*

- Eigenschaften einer Komponente
  - können zur *Design*-Zeit mit Entwicklungswerkzeugen verändert werden
    - und werden dann persistent gespeichert
  - können zur Laufzeit abgefragt und/oder gesetzt werden
- Zugriff ausschließlich über Get- und Set-Methoden
  - d.h. Implementierung nicht unbedingt 1-zu-1 als Attribut
- Arten von *Properties*:
  - einfache und *Indexed Properties* (d.h. Arrays)
  - *Bound* und *Constrained Properties*
    - Benachrichtigungs-Ereignisse bei Änderungen

Namenskonventionen für Get- und Set-Methoden

- Einfache *Properties* (Beispiel: `int Age`)
  - `public int getAge();`
  - `public void setAge(int value);`
- Einfache Boole'sche *Properties*
  - `public boolean isMarried();`
  - `public void setMarried(boolean value);`
- *Indexed Properties* (Beispiel: `short[] TabStops`)
  - `public short[] getTabStops();`
  - `public short getTabStops(int index);`
  - `public void setTabStops(short[] value);`
  - `public void setTabStops(int index, short value);`
- Alle Methoden sind optional

Bound Properties

- Eine Bean kann Änderungen von *Properties* als Ereignisse an interessierte Beans weitermelden
- Dazu: Bean muß Ereignis `PropertyChangeEvent` unterstützen

```
public interface PropertyChangeListener extends EventListener {
  public void propertyChange(PropertyChangeEvent e);
}
```

- D.h. Bean muß folgende Methoden besitzen:

```
void addPropertyChangeListener(PropertyChangeListener l)
void removePropertyChangeListener(PropertyChangeListener l)
```

- JDK bietet Klasse `PropertyChangeSupport` zur Verwaltung und Benachrichtigung der *Listener*

Constrained Properties

- Wie *Bound Properties*, aber *Listener* haben die Möglichkeit, der Änderung zu widersprechen (Veto)
- Dazu: Ereignis `VetoableChangeEvent`

```
public interface VetoableChangeListener extends EventListener {
  public void vetoableChange(VetoableChangeEvent e);
}
```

- *Listener* kann in `vetoableChange()` eine `PropertyVeto-Exception` werfen, um der Änderung zu widersprechen
- Hilfsklasse `VetoableChangeSupport` zur Verwaltung und Benachrichtigung der *Listener*
  - bei Veto: Benachrichtigung abbrechen, alle *Listener* mit ursprünglichem Wert erneut benachrichtigen, liefert `PropertyVetoException` an Aufrufer

Client/Server-Programmierung

WS 2017/2018

10.11.2017

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 17. November 2017

5.3.3 Persistenz

- Beans können wie normale Objekte serialisiert und in Dateien gespeichert werden
  - Bean (bzw. Oberklasse) muß Schnittstelle `Serializable` implementieren
- Beispiel:

```
FileOutputStream fos = new FileOutputStream("Smiley.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(smiley);
```

- `Beans.instantiate(classloader, name)` sucht `.ser`-Datei mit gegebenem Namen
  - falls erfolgreich: erzeuge Bean mit dem serialisierten Zustand aus der Datei
  - sonst: erzeuge „frische“ Bean der angegebenen Klasse

5.3.4 Introspektion

- Über Methode `java.beans.Introspector.getBeanInfo()` kann Information über eine Bean abgefragt werden:
  - Methoden, *Properties*, Ereignisse, ...
- Information wird in `BeanInfo`-Objekt zurückgeliefert
  - Bean kann Information explizit bereitstellen
    - über eine Klasse *BeanName*`BeanInfo` (Namenskonvention!), die `BeanInfo`-Schnittstelle implementiert
      - z.B. `SmileyBeanBeanInfo` für `SmileyBean`
  - sonst: `Introspector` erzeugt die Information dynamisch
    - setzt voraus, daß sich die Bean genau an die Namenskonventionen hält
- Erweiterung des Java *Reflection*-Mechanismus

Information in der `BeanInfo`

- `BeanDescriptor`: Bean-Klasse und *Customizer*-Klasse
  - *Customizer* ist optional; implementiert Bean-spezifische GUI (für Entwicklungswerkzeuge), um Eigenschaften der Bean anzupassen
- `PropertyDescriptor`: Beschreibung der *Properties*
  - Name, Typ (Klasse), Get- und Set-Methode, *Bound*/*Constrained* Property?, ...
  - *Property Editor*: optional; implementiert Bean-spezifische Editor-Komponente (für Entwicklungswerkzeuge) für eine *Property* (z.B. Farbe)

Information in der `BeanInfo` ...

- `EventSetDescriptor`: Beschreibung der Ereignisse
  - *Listener*-Klasse, *Listener*-Methoden, *Add* und *Remove*-Methoden für *Listener*, ...
- `MethodDescriptor`: Beschreibung der Methoden
  - Name, `Method`-Objekt (Java *Reflection*), Beschreibung der Parameter, ...
- Icon zur Darstellung der Bean in Entwicklungswerkzeugen
- *Default-Property* und *Default*-Ereignis
  - die am wahrscheinlichsten vom Benutzer eines Entwicklungswerkzeugs ausgewählt werden

5.3.5 Zusammenfassung

- Komponentenmodell für Java
  - vorwiegend für clientseitige, visuelle Komponenten
  - Ziel: werkzeugunterstützte Erstellung von Anwendungen
- Java Beans unterstützt:
  - Methoden
  - Eigenschaften (*Properties*)
    - auch mit Ereignismeldung/Veto bei Änderung
  - Ereignisse (über *Listener*-Interface)
  - Introspektion (für Werkzeuge)
  - Anpassung (*Customization*) (für Werkzeuge)
  - Persistenz (Speicherung der konfigurierten *Properties*)

5.4 Enterprise Java Beans (EJB 3)

- Was sind *Enterprise Java Beans*?
  - serverseitige Komponenten-Architektur für Entwicklung und *Deployment* von komponenten-basierten verteilten Geschäftsanwendungen
  - d.h.: Komponenten-Modell für Anwendungslogik einer verteilten Anwendung
- EJBs werden immer in speziellem EJB-Container ausgeführt
  - anwendungsunabhängiger Anwendungsserver
  - Implementierungen von verschiedenen Anbietern
    - z.B. JBoss, OpenEJB, IBM WebSphere, BEA WebLogic
  - Schnittstelle zu EJBs ist standardisiert
  - Zugriff auf EJBs erfolgt immer über EJB-Container

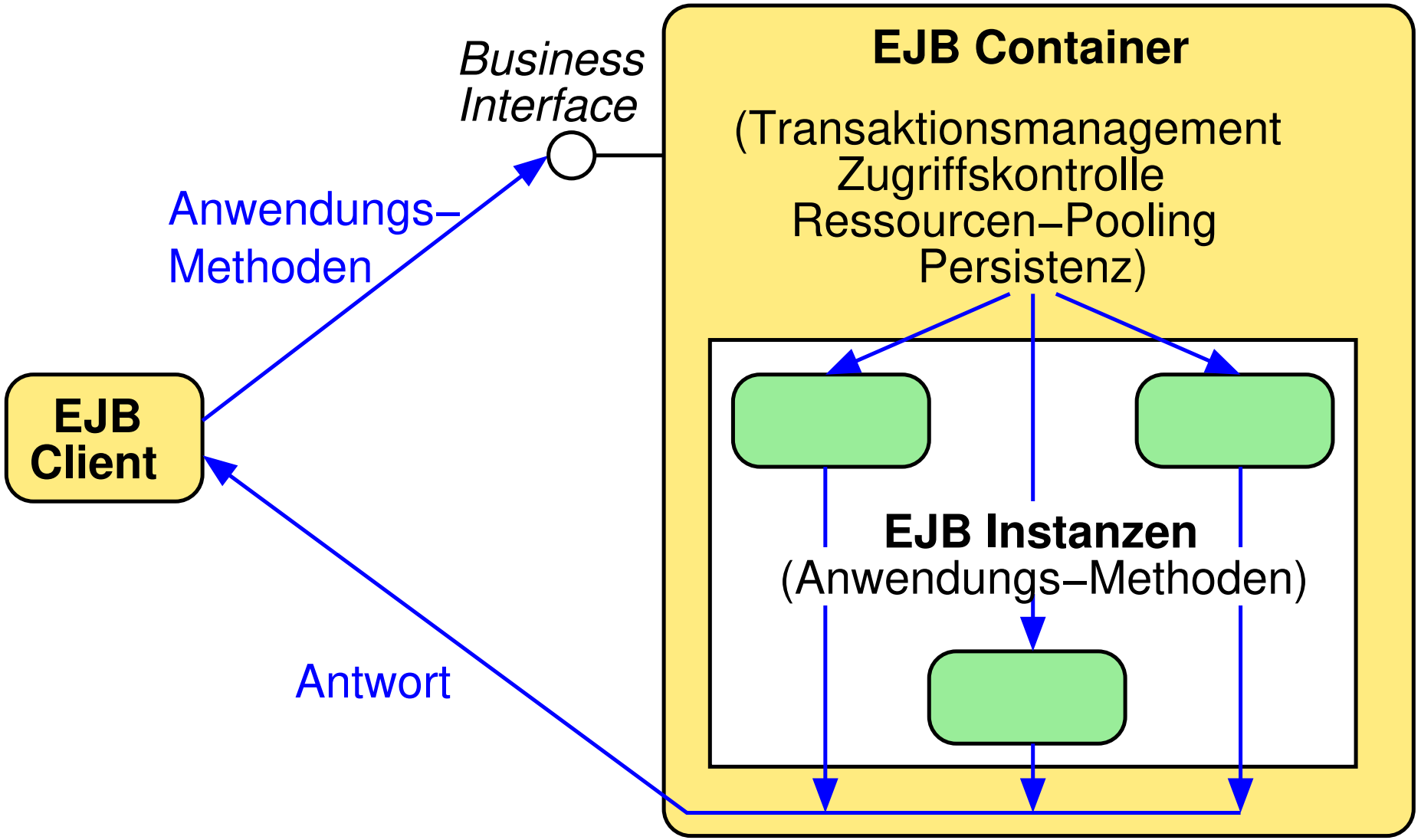
Ziel der EJB-Architektur

- Anwendungsprogrammierer erstellt nur Anwendungslogik
  - und zwar komponentenbasiert ...
- Alle wichtigen, anwendungsunabhängigen Dienste werden vom EJB-Container realisiert
  - Nutzung der Dienste ist **implizit**
    - wird nicht ausprogrammiert
    - Spezifikation und Konfiguration der Dienste bei Erstellung und/oder beim *Deployment* der EJB
  - EJB-Container übernimmt Rolle eines *Component Transaction Monitors*
    - realisiert Komponentenmodell, Transaktions- und Ressourcen-Management

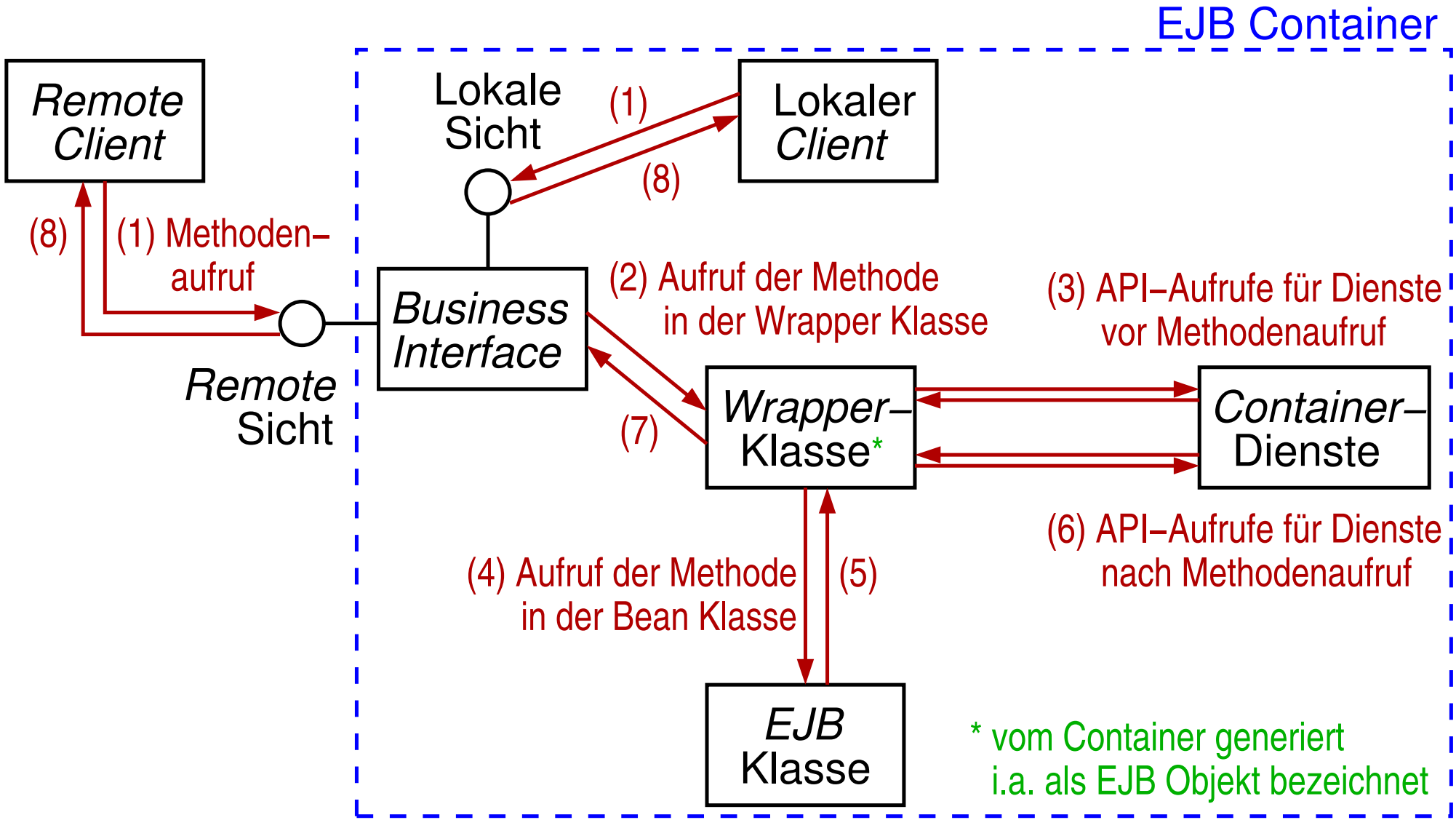
Component Transaction Monitor

- Vereinigung von Transaktionsmonitor und ORB
- Aufgabe: automatisches Management von
  - Transaktionen
  - Persistenz von Objekten
  - Sicherheit (insbes. Zugriffskontrolle)
  - Ressourcenverwaltung (z.B. *Pooling*)
  - Objektverteilung
  - Nebenläufigkeit
- Fokus: zuverlässige, transaktionsorientierte Anwendungen mit vielen Nutzern

Grundlegende Elemente in einer EJB-Umgebung



Das EJB 3 Programmiermodell



5.4.1 Arten von EJBs

- *Entity Beans*
  - Daten-Objekte, die persistent (in Datenbank) gespeichert und von mehreren Clients genutzt werden
  - d.h. Objekte bzw. Beziehungen der Anwendungsmodellierung
    - z.B. Kunde, Konto, Aktie, ...
  - in EJB 3 nicht weiterentwickelt, aber weiterhin unterstützt
  - (im folgenden nicht mehr behandelt, siehe Skript WS07/08)
- *Entities* (ab EJB 3.0, Java Persistence API)
  - Ziele und Aufgaben wie bei *Entity Beans*, aber nur lokal zugreifbar, leichtgewichtiger und besser standardisiert
  - Persistenz wird automatisch durch Container realisiert
  - auch unabhängig von Java EE verwendbar

5.4.1 Arten von EJBs ...

- *Session Beans*
  - realisieren Aktionen, die mehrere Anwendungsobjekte (*Entities* bzw. *Entity Beans*) betreffen
    - d.h. die Geschäftslogik
  - zustandslos (*stateless*) oder zustandsbehaftet (*stateful*)
    - d.h. merkt sich die Bean Daten zwischen zwei Aufrufen eines Clients?
    - *stateful Session Beans* repräsentieren Client-Sitzungen
  - kein **persistenter** Zustand
- *Message Driven Beans* (ab EJB 2.0)
  - ähnlich wie *Session Beans*, aber asynchrone Schnittstelle
    - Operationen werden über JMS-Nachrichten aufgerufen statt über (entfernte) Methodenaufrufe
  - (im folgenden nicht mehr behandelt)

5.4.2 Anatomie einer EJB-Anwendung

- Eine EJB-Anwendung besteht aus:
  - *Session* (bzw. *Message Driven*) *Beans* für die Anwendungslogik
    - *Session Beans* sind normale Java-Klassen mit Annotationen
  - *Local* und *Remote Business Interfaces* der *Session Beans*
    - normale Java-Interfaces, mit Annotationen
    - i.d.R. durch zugehörige *Session Bean* implementiert
  - *Entities* (bzw. *Entity Beans*) für das Datenmodell
    - *Entities* sind normale Java-Klassen mit Annotationen
  - *Deployment*-Deskriptoren
    - XML-Dokumente
    - beschreiben Beans und vom Container benötigte Dienste

5.4.2 Anatomie einer EJB-Anwendung ...

- Mehrere *Beans* werden mit gemeinsamen Deployment-Deskriptoren in ein JAR-Archiv gepackt
  - Deployment-Deskriptoren im Verzeichnis `META-INF`
  - `META-INF/ejb-jar.xml` kennzeichnet Archiv als EJB Archiv
- EJB-Anwendung besteht aus ein oder mehreren JAR-Archiven
- Alle weiteren benötigten Klassen werden zur Laufzeit generiert
  - z.B.:
    - *Wrapper*-Klassen für EBJ-Klassen bzw. Implementierungsklassen der *Business Interfaces*
    - *Stub*-Klassen für die Clients
  - Basis: generische Proxy-Klasse `java.lang.reflect.Proxy`

Deployment-Deskriptoren

- `ejb-jar.xml`: für *Session Beans* (und andere)
  - kann Annotationen in den EJB-Klassen überschreiben bzw. ergänzen
    - z.B. Kennzeichnung als EJB, Art der EJB, Verwaltung der EJB zur Laufzeit (Transaktionen, Zugriffskontrolle), ...
  - Arbeitsaufteilung zwischen EJB-Entwickler (Annotationen) und Anwendungsentwickler (*Deployment*-Deskriptor)
- `persistency.xml`: für *Entities*
  - analog zu `ejb-jar.xml`
  - spezifiziert zusätzlich die Verbindung zur Datenbank
- Zusätzlich Container-spezifische Deskriptoren möglich

5.4.3 Einschub: Java Annotationen

- Eingeführt mit Java 5 (JDK 1.5)
- Erlauben die Spezifikation von beliebigen Metadaten für Programm-Elemente
  - z.B. für Typen, Methoden, Attribute, Parameter, Annotationen
- Annotationen werden vom Compiler übersetzt und sind getypt
  - Typdefinition ähnlich wie bei Schnittstellen
- Die Annotationen eines Elements können zur Laufzeit über das *Reflection API* abgefragt werden
  - bei entsprechender Definition der Annotation
- Neben Annotationen des Java-Standards können auch eigene Annotationen definiert werden

Beispiel

- Definition einer Annotation:

```
import java.lang.annotation.*;

// Meta-Annotation: Annotation soll auch zur Laufzeit verfügbar sein
@Retention(RetentionPolicy.RUNTIME)

// Meta-Annotation: Annotation ist auf Klassen und Interfaces anwendbar
@Target(ElementType.TYPE)

// Definition der Annotation
public @interface CodeCategory {
    boolean isTested();
    String[] tester() default {};
    boolean isReviewed() default false;
}
```

Beispiel ...

- Verwendung der Annotation:

```
@CodeCategory(isTested = true, tester = {"joe","max"})
public class MyClass {
    // ...
}
```

- Zugriff auf die Annotation zur Laufzeit:

```
MyClass obj = new MyClass();
CodeCategory cat =
    obj.getClass().getAnnotation(CodeCategory.class);
System.out.println("is tested: " + cat.isTested());
```

Anmerkungen zum Beispiel

- Bei Verwendung einer Annotation müssen die „Attribut“-Werte mit der Syntax *<Name:>* = *<Wert>* angegeben werden
  - die Reihenfolge ist dabei beliebig
  - Ausnahme: die Annotation hat nur ein (oder gar kein) Attribut
    - z.B. `@Retention` und `@Target`
- Attribute können auch mit Default-Werten definiert werden
  - z.B. `tester` und `isReviewed`
  - bei Verwendung der Annotation brauchen dann keine Werte angegeben zu werden
  - eine leere Parameterliste kann auch ganz weggelassen werden
- Attribute können auch Arrays sein
  - z.B. `tester`

5.4.4 Beispiel *Hello World*

Remote Business Interface

```
package org.Hello;
import javax.ejb.Remote;

// Java Annotation: markiert Schnittstelle als Remote EJB Interface
@Remote
public interface HelloRemote
{
  public String sayHello();
}
```

EJB-Klasse

```
package org.Hello;
import javax.ejb.Stateless;

// Java Annotation: markiert Klasse als Stateless Session Bean
@Stateless
public class HelloImpl implements HelloRemote
{
  public String sayHello()
  {
    return "Hallo? Jemand da?";
  }
}
```

5.4.4 Beispiel *Hello World* ...

Deployment Deskriptor (`ejb-jar.xml`)

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar/>
```

- „Leerer“ Deskriptor
- Zeigt lediglich an, daß es sich um eine EJB handelt

Client (OpenEJB)

```
import org.Hello.*;
import java.util.Properties;
import javax.naming.InitialContext;

public class HelloClient {
  public static void main(String args[]) {
    try {
      // Properties für JNDI (hier: Nutzung von OpenEJB)
      Properties p = new Properties();
      p.put("java.naming.factory.initial",
            "org.apache.openejb.client" +
            ".RemoteInitialContextFactory");
      p.put("java.naming.provider.url",
            "ejbd://127.0.0.1:4201");
```

Client (OpenEJB) ...

```
        // User und Passwort sind optional!
        p.put("java.naming.security.principal", "user");
        p.put("java.naming.security.credentials", "password");
        InitialContext ctx = new InitialContext(p);

        Object obj = ctx.lookup("HelloImplRemote");
        HelloRemote hello = (HelloRemote)obj;
        System.out.println(hello.sayHello());
        System.out.println(hello.sayHello());
      }
      catch (Exception e) { ... }
    }
}
```

Nutzung von OpenEJB im Labor H-A 4111

- Private Installation (i.w. Konfigurationsdateien):
  - Auf einem Rechner im Labor H-A 4111: Aufruf des Skripts `/opt/dist/tools/openejb_install.sh`
    - erzeugt private Verzeichnisse für *Deployment* der EJBs
    - editiert Konfigurationsdateien so, daß jeder Benutzer unterschiedliche Ports nutzt
  - Umgebungsvariable und Pfad setzen (in `$HOME/.profile`):

```
export OPENEJB_HOME=$HOME/Soft/openejb-7.0.2
export PATH=$OPENEJB_HOME/bin:$PATH
```

- Start des Containers: `openejb start`
- Stoppen des Containers: `^C` oder `kill`

Übersetzung / *Deployment* des Servers

- Übersetzung
  - `javac -cp $OPENEJB_HOME/lib/javaee-api-7.0-1.jar:. org/Hello/*.java`
  - `CLASSPATH` muß nur gesetzt werden, wenn Java EE nicht installiert ist (wegen EJB-Klassen)
- Erzeugung der JAR-Datei für die EJB
  - `jar cvf myHelloEjb.jar org/Hello/*.class META-INF`
  - Verzeichnis `META-INF` muß *Deployment*-Deskriptor `ejb-jar.xml` enthalten

Übersetzung / *Deployment* des Servers ...

- *Deployment* der EJB
  - `openejb deploy -s ejbd://localhost:4201 myHelloEjb.jar`
  - OpenEJB-Container muß bereits laufen
    - ggf. vorher starten: `openejb start`
  - Optionen:
    - `-s`: Angabe der URL des OpenEJB-Daemons
      - z.Zt. nur `localhost` erlaubt
      - kann entfallen, wenn Standard-Port 4201 benutzt wird
    - `-u`: *Undeploy* und anschließendes *Deploy*
    - `-o`: *offline*-Modus (wenn OpenEJB-Container nicht läuft)

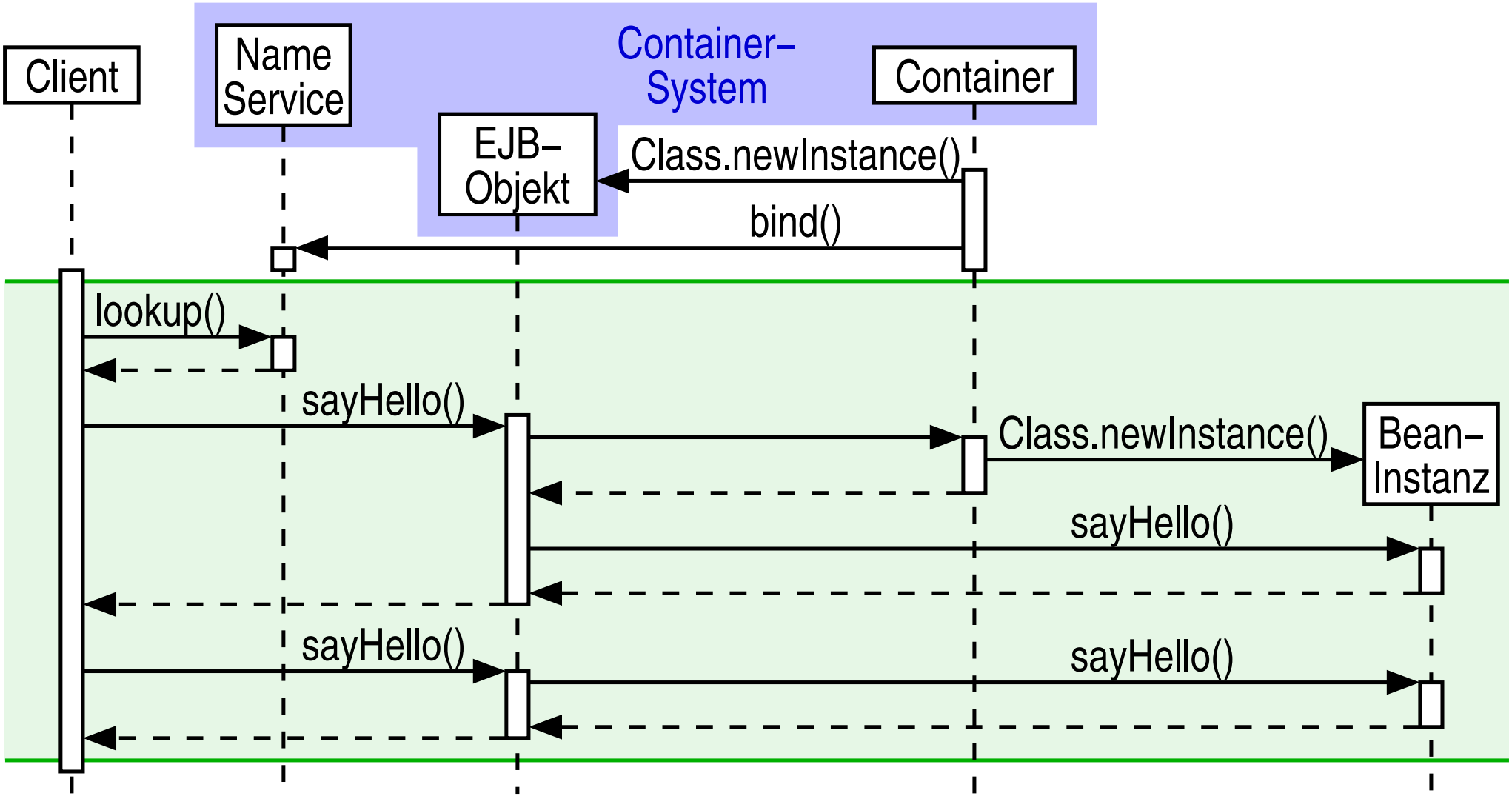
Übersetzung / *Deployment* des Servers ...

- Was passiert beim *Deployment*?
  - JAR-Datei mit Bean wird in `apps`-Verzeichnis von OpenEJB kopiert (damit Container sie findet)
  - *Business-Interface* der Bean wird beim Namensdienst registriert
    - genauer: Objekt einer Klasse, die *Interface* implementiert
  - *Stub*- und Implementierungsklassen werden dynamisch zur Laufzeit erzeugt
- *Undeployment*
  - `openejb undeploy -s ejbd://localhost:4201 $OPENEJB_HOME/apps/myHelloEjb.jar`
  - bzw. falls Container nicht läuft:
    `rm $OPENEJB_HOME/apps/myHelloEjb.jar`

Übersetzung / Start des Clients

- Vorbereitung
  - Java-Datei für *Business-Interface* muß beim Client vorhanden sein!
- Übersetzung
  - `javac -cp $OPENEJB_HOME/lib/javaee-api-7.0-1.jar:. HelloClient.java`
- Start
  - `java -cp $OPENEJB_HOME/lib/javaee-api-7.0-1.jar: $OPENEJB_HOME/lib/openejb-client-7.0.2.jar:. HelloClient`

Ablauf des Programms



Weitere Informationen

- OpenEJB *Home Page*
  - http://openejb.apache.org/
- Beispiele zu OpenEJB
  - http://openejb.apache.org/examples/
- Dokumentation zu OpenJPA
  - http://openjpa.apache.org/builds/2.4.1/apache-openjpa/docs/
  - OpenJPA ist der Standard *Persistence Provider* von OpenEJB
    - für *Entities*
- EJB 3.0 Spezifikationen
  - http://jcp.org/aboutJava/communityprocess/final/jsr220/

5.4.6 Dienste des EJB-Containers

- Ressourcen-Management
  - *Pooling*: Container hält einen Pool von Bean-Instanzen vor
    - z.B. bei *stateless Session Beans*: Aufrufe gehen an beliebige Instanz im Pool
    - Ziel u.a.: vermeide teuren Auf- und Abbau von Datenbank-Verbindungen
  - Passivierung und Aktivierung von *stateful Session Beans*
    - Container kann *Session Beans* temporär passivieren
      - z.B. wenn zuviele Bean-Instanzen vorhanden sind
      - bei Passivierung wird Zustand der Bean auf Festplatte gesichert und Bean-Instanz gelöscht
      - bei nächster Client-Anfrage: neue Bean-Instanz wird erzeugt und initialisiert

5.4.6 Dienste des EJB-Containers ...

- Namensdienst: Zugriff über JNDI-Schnittstelle
- Nebenläufigkeit
  - *Session Beans*: immer nur von einem Client genutzt
  - *Entities*: optimistisches Locking
    - setzt Versions-Attribut in der *Entity* voraus
    - Alternative: explizite *Read* und *Write-Locks*
- Persistenz (☞ **5.4.9**)
  - Datenfelder einer *Entity* werden automatisch mit dem Inhalt einer Datenbank synchronisiert
- Transaktionen (☞ **5.4.10**)
  - Methoden von Session Beans können automatisch als Transaktionen ausgeführt werden

5.4.6 Dienste des EJB-Containers ...

- Sicherheit
  - Authentifizierung
    - Vorgehen abhängig von EJB-Container-Implementierung
    - oft: Benutzername / Paßwort als *Properties* über JNDI übergeben
  - Autorisierung
    - Festlegung, wer welche Methoden aufrufen darf
      - über *Deployment*-Deskriptor (☞ **5.4.7**) oder Annotationen
    - rollenbasierte Zugriffskontrolle
  - sichere Kommunikation
    - abhängig von EJB-Container-Implementierung
    - meist: Nutzung von TLS/SSL

Client/Server-Programmierung

WS 2017/2018

17.11.17

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 17. November 2017

5.4.7 Der *Deployment*-Deskriptor `ejb-jar.xml`

- Beschreibt
  - aus welchen Klassen die EJB besteht
  - wie die EJB zur Laufzeit verwaltet werden soll, z.B.:
    - Art der EJB
    - Transaktionsmanagement
    - Zugriffskontrolle
- Ab EJB 1.1 ist der *Deployment*-Deskriptor in XML codiert
  - wird als `META-INF/ejb-jar.xml` in JAR-Datei der EJB abgelegt
  - wird beim Deployment vom Container gelesen
- Ab EJB 3.0 kann *Deployment*-Deskriptor auch leer sein
  - Metadaten auch als Annotationen im Java-Code möglich

Aufbau des *Deployment*-Deskriptors

- Prinzipieller Aufbau:

```
<?xml version="1.0" encoding="UTF-8"?>

<ejb-jar>
  <enterprise-beans>
    ...
  </enterprise-beans>

  <assembly-descriptor>
    ...
  </assembly-descriptor>
</ejb-jar>
```

Elemente innerhalb von `<ejb-jar>`

- `<enterprise-beans>`
  - Beschreibung der einzelnen EJBs innerhalb der JAR-Datei
  - ab EJB 3.0 besser durch Annotationen
- `<assembly-descriptor>`
  - Konfiguration der zusammengestellten Beans für eine Anwendung
  - z.B. Transaktionsverhalten, Zugriffskontrolle
- `<description>`
  - Beschreibung der Beans-Sammlung in der JAR-Datei
- und noch einige weitere ...

Elemente innerhalb von `<assembly-descriptor>`

- `<container-transaction>`
  - Transaktions-Attribute für einzelne Methoden (☞ **5.4.10**)
  - ab EJB 3.0 auch besser durch Annotationen
- `<security-role>`
  - Definiert Rollennamen für rollenbasierte Zugriffskontrolle
  - Unterelemente: `<description>` und `<role-name>`
- `<method-permission>`
  - legt fest, welche Rollen welche Methoden aufrufen dürfen
  - Unterelemente: `<description>`, `<role-name>`, `<method>`
  - statt `<role-name>` auch Element `<unchecked>` möglich: keine Zugriffskontrolle

Zur Zugriffskontrolle bei EJBs

- *Deployment*-Deskriptor (bzw. Annotationen) legt nur Abbildung zwischen Rollennamen und aufrufbaren Methoden fest
- Verwaltung von Benutzern und Paßworten sowie Abbildung von Benutzern auf Rollen muß durch EJB-Container erfolgen
  - abhängig von jeweiliger Implementierung
  - in OpenEJB:
    - `conf/users.properties`: Benutzer und Paßwort
      - z.B.: `roland=myPassWd`
    - `conf/groups.properties`: Zuordnung Benutzer zu Rollen
      - z.B.: `Admin=roland`

Beispiel

- *Session Bean* zur Vorlesungsverwaltung
  - jeder darf die Daten lesen, nur Administrator darf ändern
- *Deployment*-Deskriptor:

```
<ejb-jar>
  <assembly-descriptor>
    <security-role>
      <description>Rolle fuer Administratoren</description>
      <role-name>Admin</role-name>
    </security-role>
    <method-permission>
      <role-name>Admin</role-name>
      <method>
        <ejb-name>VorlesungEJB</ejb-name>
        <method-name>*</method-name>
```

```
      </method>
    </method-permission>
    <method-permission>
      <unchecked/>
      <method>
        <ejb-name>VorlesungEJB</ejb-name>
        <method-name>getInfos</method-name>
      </method>
    </method-permission>
  </assembly-descriptor>
</ejb-jar>
```

- Anmerkungen:
  - der *Deployment*-Deskriptor wird i.d.R. über ein Konfigurationswerkzeug erstellt
  - Rollen und Zugriffsrechte können auch über Annotationen festgelegt werden

5.4.8 *Session Beans*: Details

Stateless Session Beans

- Container verwaltet Pool identischer *Bean*-Instanzen
  - Erzeugung / Löschung nur durch Container
- Aufrufe von Anwendungs-methoden werden an beliebige *Bean*-Instanz im Pool geleitet
- keine Passivierung / Aktivierung

Stateful Session Beans

- Erzeugung / Löschung (indirekt) auf Veranlassung des Clients
- *Bean*-Instanzen werden Client-Sitzungen (und damit EJB-Objekten) fest zugeordnet
  - Aufrufe eines Clients gehen an dieselbe *Bean*-Instanz
- bei Bedarf: Passivierung / Aktivierung durch Container

Annotationen für Schnittstellen und Bean-Implementierungen

- `@Local` und `@Remote` (aus `javax.ejb`)
  - markieren ein Java-Interface als lokale bzw. *remote* Schnittstelle einer EJB
  - lokale Schnittstelle kann nur innerhalb des Containers (von anderen EJBs) genutzt werden
  - Zuordnung zur *Bean*-Klasse über `implements`-Beziehung
- `@Stateless` und `@Stateful` (aus `javax.ejb`)
  - markieren eine Java-Klasse als entsprechende *Session Bean*
- `@Remove` (aus `javax.ejb`)
  - markiert eine Methode in einer (*stateful*) *Session Bean*-Klasse, nach deren Ausführung die *Bean*-Instanz gelöscht werden soll

Beispiel

- RemIf.java:

```
@Remote
public interface RemIf {
  public String sayHello();
  public void bye();
}
```

- LocIf.java:

```
@Local
public interface LocIf {
  public String sayHello();
  public void doIt();
}
```

- MyBean.java:

```
@Stateful
public class MyBean implements LocIf, RemIf {
  public String sayHello() { ... }
  @Remove public void bye() { ... }
  public void doit() { ... }
}
```

Lebenszyklus-*Callbacks*

- `@PostConstruct` bzw. `@PreDestroy` (aus `javax.annotation`)
  - markiert Methode in einer *Bean*-Klasse, die nach Erzeugung bzw. vor Löschung einer *Bean*-Instanz aufgerufen werden soll
  - aber: keine Garantie, daß *Bean*-Instanz jemals gelöscht wird
- `@PrePassivate` bzw. `@PostActivate` (aus `javax.ejb`)
  - markiert Methode, die vor Passivierung bzw. nach Aktivierung einer *Bean*-Instanz aufgerufen werden soll
- Alle Callback-Methoden sollten wie folgt deklariert werden:
  - `public void` *name* `() { ... }`
- Auch möglich: Definition einer eigenen Klasse für die Callbacks

Interceptor-Methoden

- `@AroundInvoke` (aus `javax.interceptor`)
  - markiert Methode, die alle Methodenaufrufe einer *Bean* abfängt
  - Deklaration der Methode:
    - `public Object` *`name`*`(InvocationContext c) throws Exception { ... }`
  - `InvocationContext` erlaubt u.a.:
    - Abfrage von Zielobjekt, Methodenname und Parameter
    - Ausführung des abgefangenen Methodenaufrufs
- Einsatz z.B. Protokollierung, Zugriffskontrolle, ...
- Auch möglich: Definition einer eigenen Klasse für den *Interceptor*

5.4.8 *Session Beans*: Details ...

Dependency Injection

- Aufgabe: Beschaffung von Referenzen auf Ressourcen, die der Container bereitstellt
- Lösung: passendes Attribut wird mit einer Annotation versehen
  - Attribut wird dann automatisch vom Container initialisiert
- `@Resource` (aus `javax.annotation`)
  - für beliebige Resourcen des Containers
  - z.B. `SessionContext`: erlaubt Zugriff auf Benutzer-Name, aktuelle Transaktion, ...
- `@EJB` (aus `javax.ejb`)
  - speziell, um Referenzen auf EJBs zu holen
  - (Alternative: explizite Nutzung von JNDI)

Beispiel

- MyBeanLocal.java:

```
@Local public interface MyBeanLocal {
  public String getName();
}
```

- MyBean.java:

```
@Stateful
public class MyBean implements MyBeanLocal {
  // Attribut wird vom Container initialisiert!
  @Resource private SessionContext context;
  public String getName() {
    return context.getCallerPrincipal().getName();
  }
}
```

Beispiel ...

- HelloRemote.java:

```
@Remote public interface HelloRemote {
  public String sayHello();
  public String sayGoodBye();
}
```

- HelloImpl.java:

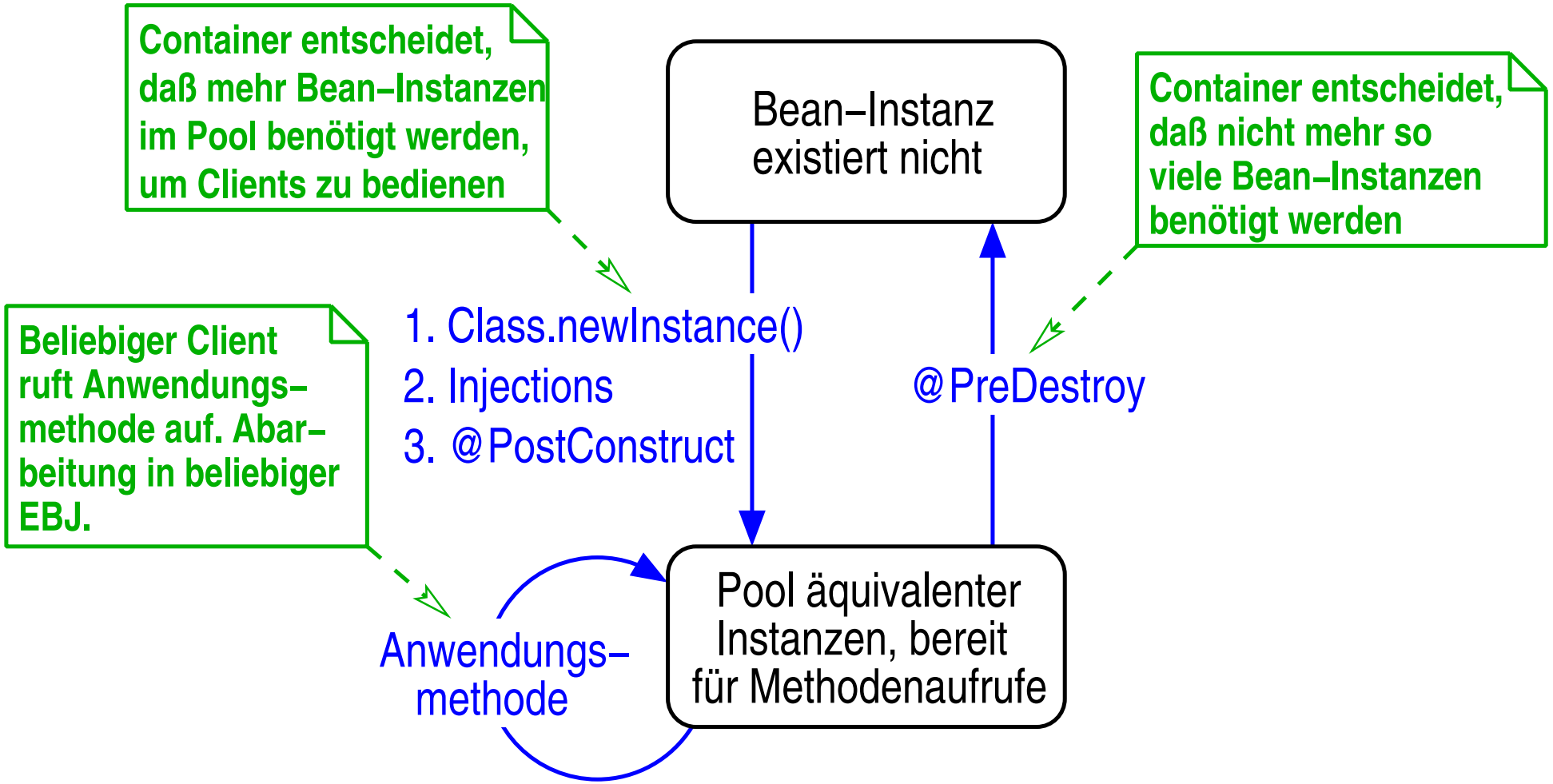
```
@Stateful
public class HelloImpl implements HelloRemote {
  // Referenz auf MyBean wird vom Container eingetragen!
  @EJB private MyBeanLocal myBean;
  public String sayHello() {
    return "Hallo " + myBean.getName();
  }
  @Remove public String sayGoodBye() { ... }
```

Beispiel ...

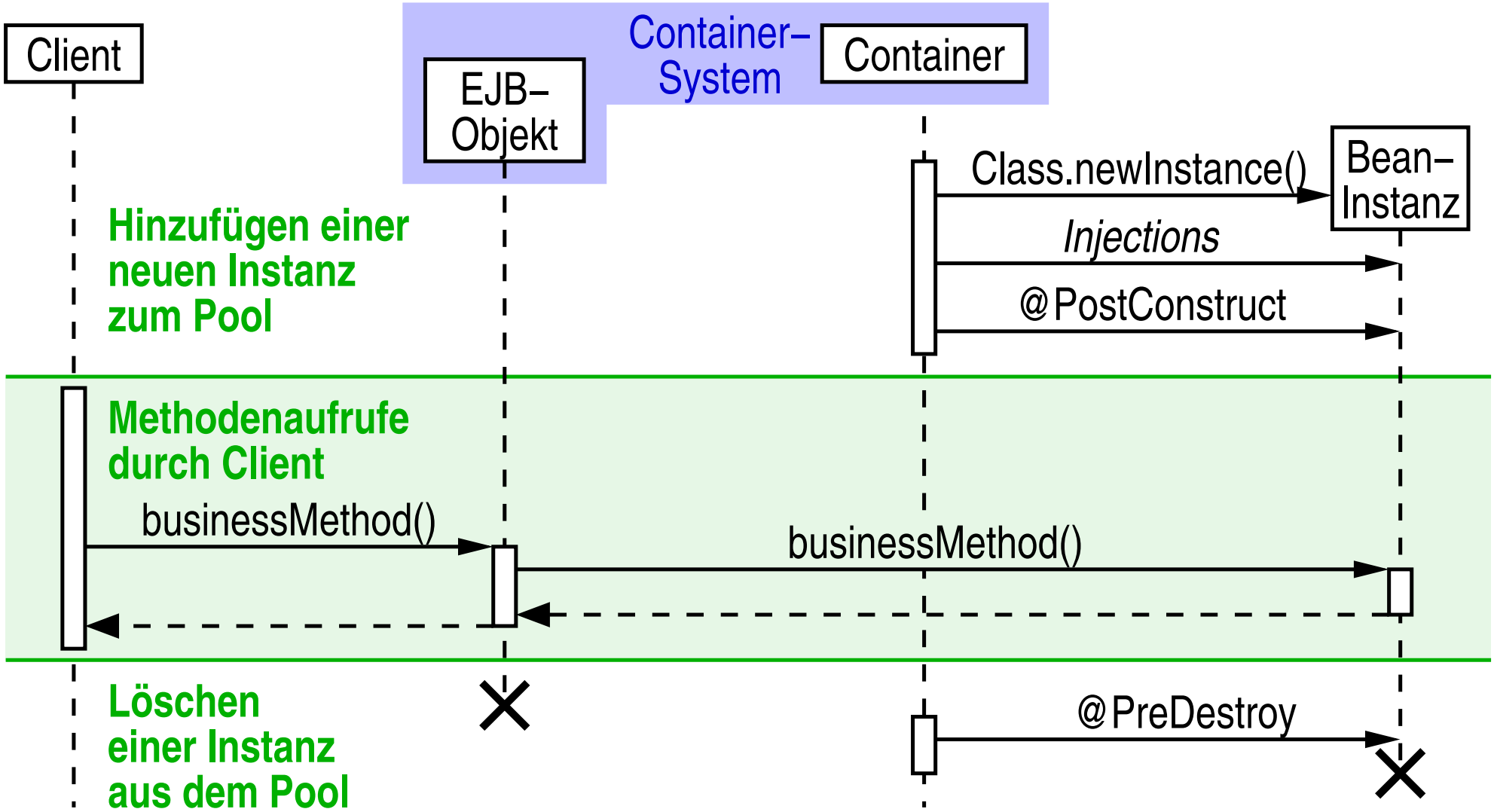
➥ HelloImpl.java ...:

```
@PostConstruct public void start() {
  System.out.println("@PostConstruct HelloImpl");
}
@PreDestroy public void stop() {
  System.out.println("@PreDestroy HelloImpl");
}
@AroundInvoke
public Object inv(InvocationContext c)
                  throws Exception {
  System.out.println("HelloImpl: Calling "
                     + c.getMethod().getName());
  return c.proceed();
}
}
```

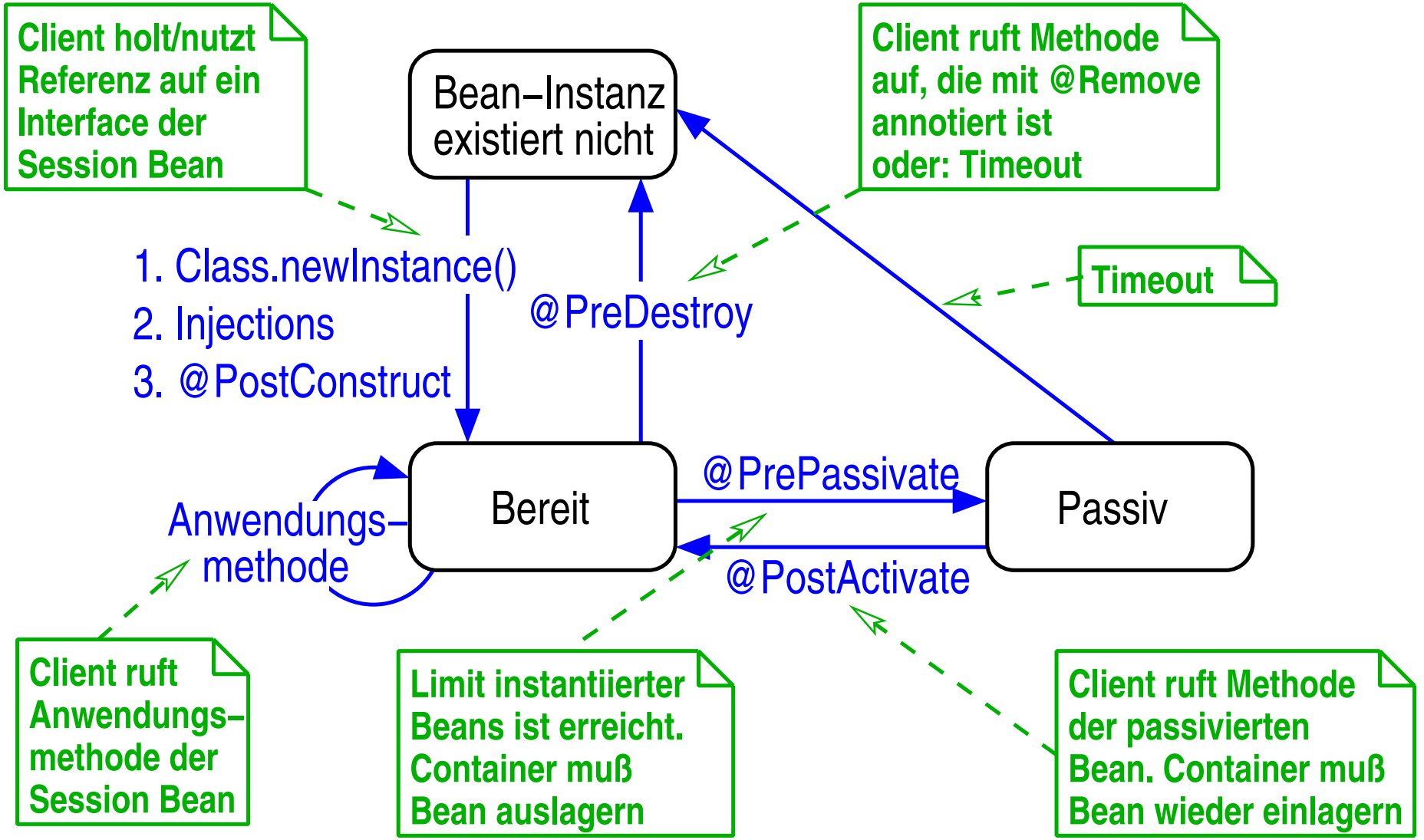
Lebenszyklus einer *Stateless Session Bean*



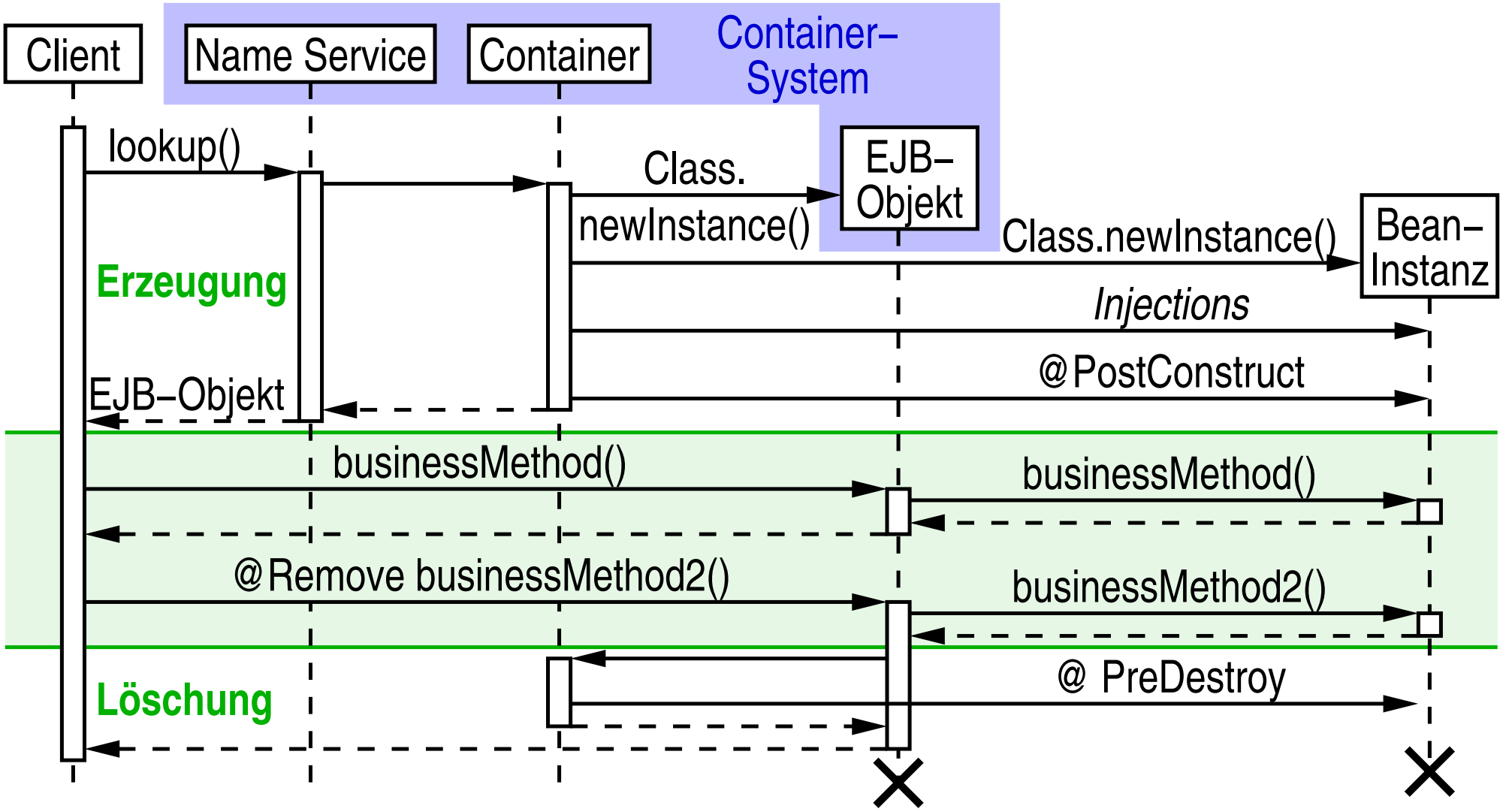
Typische Abläufe bei *Stateless Session Beans*



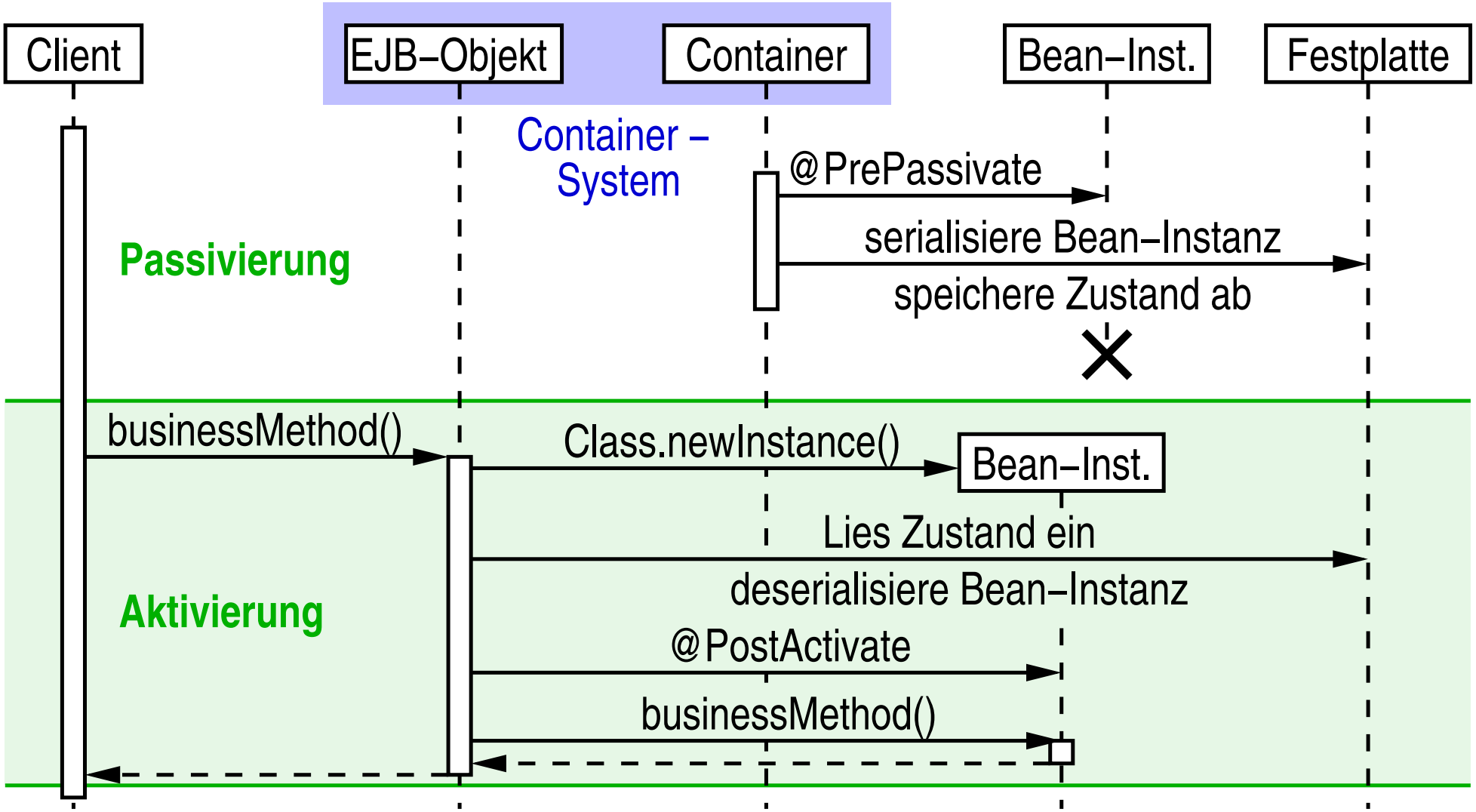
Lebenszyklus einer *Stateful Session Bean*



Erzeugung/Löschung einer *Stateful Session Bean*



Passivierung/Aktivierung einer *Stateful Session Bean*



5.4.9 *Entities*: Details

- *Entities* realisieren persistente Datenobjekte einer Anwendung
- Basis: *Java Persistence API* (JPA)
  - unabhängig von EJB und Java EE nutzbar
- Eigenschaften (Abgrenzung zu *Session Beans*):
  - für den Client sichtbare, persistente Identität (Primärschlüssel)
    - unabhängig von Objektreferenz
  - persistenter, für Client sichtbarer Zustand
  - nicht entfernt zugreifbar
  - Lebensdauer völlig unabhängig von der der Anwendung
- Persistenz der *Entities* wird automatisch durch *Persistence Provider* gesichert

Beispiel

- *Entity* `Account.java`:

```
import javax.persistence.*;

@Entity  // Markiert Klasse als Entity
public class Account implements java.io.Serializable {

   @Id  // Markiert Attribut als Primärschlüssel
   private int accountNo;
   private String name;

   public int getAccountNo() { return accountNo; }
   public String getName() { return name; }
   public void setName(String nm) { name = nm; }
   public Account(int no, String nm) {
      accountNo = no; name = nm;
   }
}
```

Beispiel ...

➥ *Deployment*-Deskriptor `META-INF/persistence.xml`:

```
<persistence
 xmlns="http://java.sun.com/xml/ns/persistence"
 version="1.0">
 <persistence-unit name="intro">
  <jta-data-source>My DataSource</jta-data-source>
  <non-jta-data-source>My Unmanaged DataSource
  </non-jta-data-source>
  <class>org.Hello.Account</class>
  <properties>
   <property name="openjpa.jdbc.SynchronizeMappings"
             value="buildSchema(ForeignKeys=true)"/>
  </properties>
 </persistence-unit>
</persistence>
```

Beispiel ...

- OpenEJB Konfigurationsdatei `conf/openejb.xml`:

```
<Resource id="My DataSource" type="DataSource">
  JdbcDriver org.hsqldb.jdbcDriver
  JdbcUrl jdbc:hsqldb:file:data/hsqldb/hsqldb
  UserName sa
  Password
  JtaManaged true
</Resource>

<Resource id="My Unmanaged DataSource" type="DataSource">
  JdbcDriver org.hsqldb.jdbcDriver
  JdbcUrl jdbc:hsqldb:file:data/hsqldb/hsqldb
  UserName sa
  Password
  JtaManaged false
</Resource>
```

Anmerkungen zum Beispiel

- Eine *Entity*-Klasse muss `Serializable` nicht implementieren
  - falls Sie es tut, können Objekte auch als Parameter / Ergebnis von *Remote*-Methoden einer *Session Bean* auftreten
    - übergeben wird dabei eine Kopie, die **nicht** mit der Datenbank synchronisiert wird
- Abbildung von Klasse auf Datenbank-Tabelle und von Attributen auf Spalten wird vom *Persistence Provider* vorgenommen
  - kann durch Annotationen genau gesteuert werden
- *Entity*-Klasse muß ein Primärschlüssel-Attribut deklarieren (`@Id`)
  - Primärschlüssel kann auch eigene Klasse sein
- *Entity*-Klasse darf auch Geschäftsmethoden besitzen

Anmerkungen zum Beispiel ...

- Beispiel verwendet *Field Access*
  - *Persistence Provider* greift direkt auf die Attribute zu
  - *Mapping*-Annotationen (hier: `@Id`) bei den Attributen
- Alternative: *Property Access*
  - *Persistence Provider* greift auf den Zustand nur über *get*- und *set*-Methoden zu
  - *Mapping*-Annotationen bei den *get*-Methoden
  - Achtung: es müssen immer *get*- **und** *set*-Methoden implementiert werden
- Pro *Entity* ist nur eine der Alternativen erlaubt

Anmerkungen zum Beispiel ...

- *Deployment*-Deskriptor legt fest:
  - Name der *Persistence Unit* (zum Zugriff durch *Session Bean*)
    - *Persistence Unit*: Einheit für Kapselung und *Deployment* von *Entities*
  - Namen der Datenquellen mit bzw. ohne Support für *Java Transaction API* (JTA, ☞ **5.4.10**, verteilte Transaktionen)
  - *Properties* für den *Persistence Provider*
    - hier: OpenJPA passt Datenbank-Schema zur Laufzeit an
- OpenEJB-Konfiguration legt fest:
  - JDBC Treiber und URL der Datenbank (☞ **2**)
  - Login-Name und Passwort
  - Unterstützung für JTA

Client/Server-Programmierung

WS 2017/2018

24.11.2017

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 17. November 2017

Beispiel zur Nutzung der *Entity* in einer *Session Bean*

- *Remote*-Schnittstelle `BankRemote.java`:

```
import javax.ejb.Remote;

@Remote
public interface BankRemote
{
  public Account create(int n, String name);
  public String getName(int n);
  public void close(int n);
}
```

Beispiel zur Nutzung der *Entity* in einer *Session Bean* ...

➥ *Bean*-Implementierung `BankImpl.java`:

```
import javax.ejb.*;
import javax.persistence.*;

@Stateless
public class BankImpl implements BankRemote {

  // Dependency Injection: Entity Manager für Persistenz—Einheit 'intro'
  @PersistenceContext(unitName="intro")
  private EntityManager manager;

  public Account create(int n, String name) {
    Account acc = new Account(n, name);
    // Objekt ab jetzt durch Persistence Provider verwalten
    manager.persist(acc);
```

Beispiel zur Nutzung der *Entity* in einer *Session Bean* ...

➥ *Bean*-Implementierung `BankImpl.java`:

```
        // Rückgabewert ist eine losgelöste Kopie des Objekts!
        return acc;
    }
    public String getName(int n) {
        // Findet Objekt mit gegebenem Primärschlüssel
        Account acc = manager.find(Account.class, n);
        return acc.getName();
    }
    public void close(int n) {
        Account acc = manager.find(Account.class, n);
        // Datenbank-Eintrag löschen
        manager.remove(acc);
    }
}
```

Beispiel zur Nutzung der *Entity* in einer *Session Bean* ...

- Client `BankClient.java`:

```
...
Object obj = ctx.lookup("BankImplRemote");
BankRemote bank = (BankRemote)obj;

// Erzeugt neue Entity (und Datenbankeintrag)
Account acc = bank.create(n, args[1]);

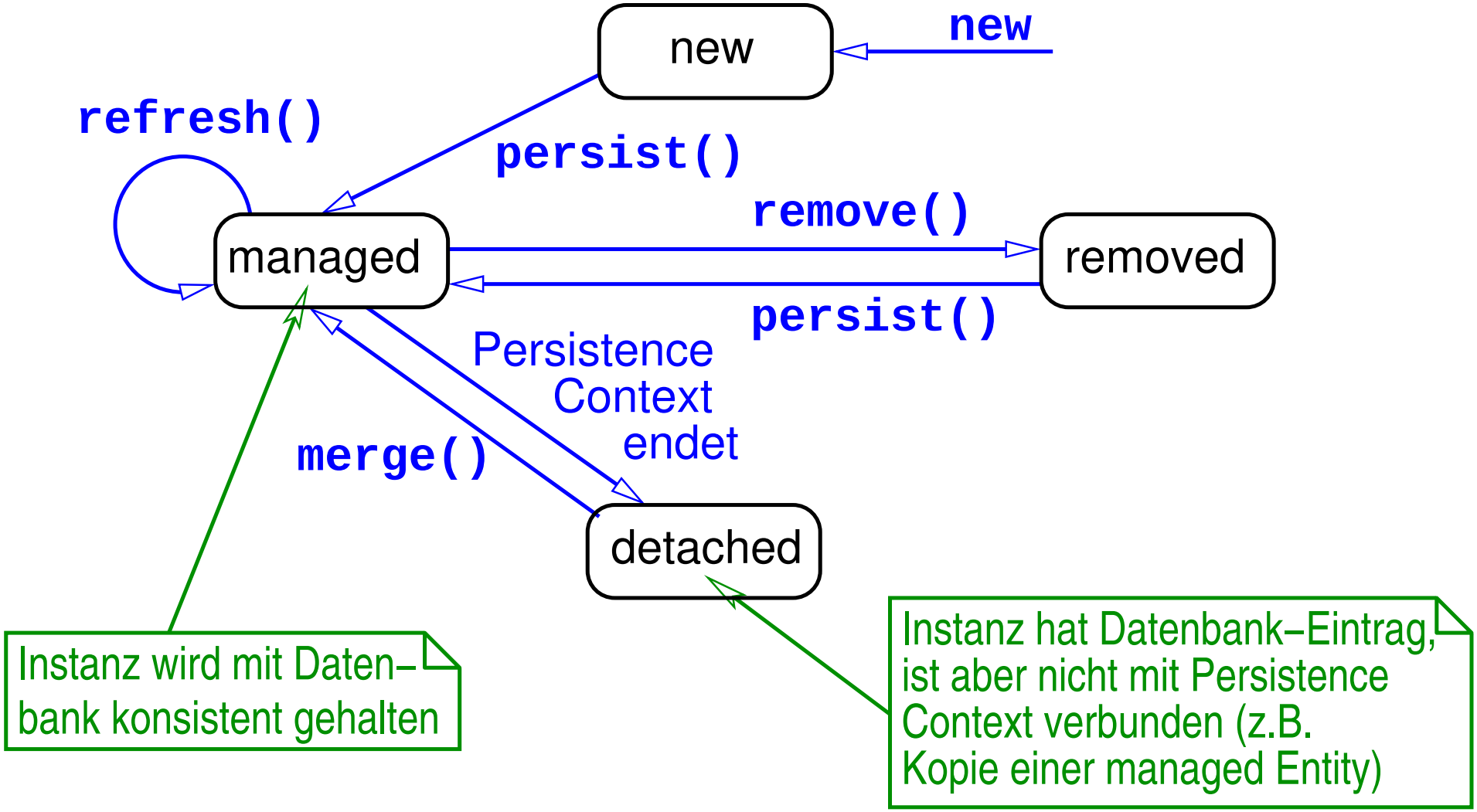
// acc ist eine Kopie des Eintrags, lokale Methodenaufrufe
System.out.println(acc.getName());
acc.setName("Niemand");

// Remote-Aufrufe der Entity Bean
System.out.println(bank.getName(n));
bank.close(n);
```

Persistence Context

- Verbindung zwischen Instanzen im Speicher und der Datenbank
- Methoden der Schnittstelle `EntityManager` u.a.:
  - `void persist(Object entity)`
    - Instanz verwalten und persistent machen
  - `<T> T find(Class<T> entityClass, Object primaryKey)`
    - Instanz zu gegebenem Primärschlüssel suchen
  - `void remove(Object entity)`
    - Instanz aus der Datenbank löschen
  - `void refresh(Object entity)`
    - Instanz aus Datenbank neu laden
  - `<T> T merge(T entity)`
    - Zustand der Instanz in *Persistence Context* hineinmischen

Lebenszyklus einer *Entity*



Lebenszyklus einer *Entity* ...

- *Persistence Context* endet per Voreinstellung mit dem Ende der aktuellen Transaktion
  - Einstellung über Attribut `type` von `@PersistenceContext`
- Synchronisation mit Datenbank i.a. am Ende jeder Transaktion
  - einstellbar über `setFlushMode()` Methode von `EntityManager`
  - ggf. auch explizite Synchronisation durch Methode `flush()`
- JPA verwendet standardmäßig ein optimistisches Sperrprotokoll
  - Datensätze werden nicht gesperrt
  - bei gleichzeitigen Änderungen durch zwei Transaktionen wird eine davon zurückgesetzt
  - dazu notwendig: Versionsattribut (Annotation `@Version`)
- Lebenszyklus-Callbacks analog zu *Session Beans* möglich

Finden von *Entities*

- `EntityManager` erlaubt das Finden von Datensätzen über *Queries* in SQL und EJB-QL
  - EJB-QL ist SQL-ähnlich, aber portabel
- Beispiel:

```
Query query
  = manager.createQuery("SELECT a FROM Account a");
List<Account> result = query.getResultList();
```

- *Queries* können auch mit Namen vordefiniert werden
  - über Annotation `@NamedQuery` der *Entity*
  - als `named-query` Element im Deskriptor `META-INF/orm.xml`
  - Nutzung: `query = manager.createNamedQuery("myQuery");`

Abbildung zwischen Objekten und Relationen (*OR-Mapping*)

- *Default-Verhalten*:
  - jede *Entity* bekommt eine Tabelle
  - jedes Attribut bekommt eine Tabellen-Spalte
  - Namen werden in Großbuchstaben umgewandelt
- Verhalten anpaßbar über Annotationen (bzw. XML-Deskriptor), z.B.:
  - `@Table(name="...")`: Tabellenname für *Entity*
  - `@Column(name="...")`: Spaltenname für Attribut
  - `@Transient`: Attribut wird nicht persistent gemacht
- Auch Abbildung von Assoziationen zwischen Klassen durch entsprechende Annotationen möglich
  - u.a. `@OneToMany`, `@ManyToOne`, `@JoinColumn`, siehe Beispiel

Beispiel zum OR-Mapping

- Datenbank-Tabellen für Konto und Zahlungen:

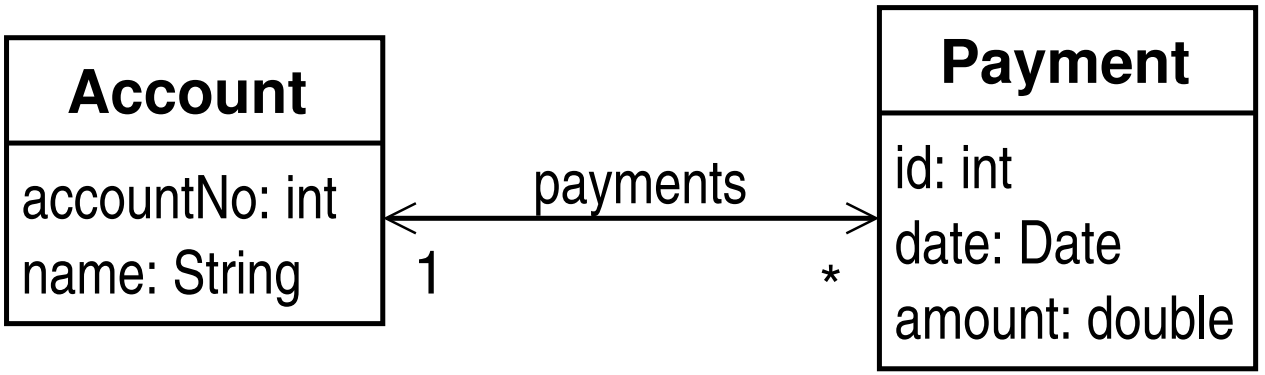
KONTO

KONTO_NR: INTEGER	NAME: VARCHAR(255)

BUCHUNG

ID: INTEGER	BETRAG: NUMERIC	DATUM: TIMESTAMP	KONTO: INTEGER

- Klassendiagramm für die *Entities*:



Beispiel zum OR-Mapping ...

➥ *Entity*-Klasse `Account.java`

```
@Entity
@Table(name="KONTO")
public class Account {
  @Id
  @Column(name="KONTO_NR")
  private int accountNo;
  private String name;
  @OneToMany(fetch=FetchType.LAZY, mappedBy="account")
  private Set<Payment> payments;

  public int getAccountNo() { return accountNo; }
  public Set<Payment> getPayments() { return payments; }
  public void addPayment(Payment p) { payments.add(p); }
  ...
```

Beispiel zum OR-Mapping ...

➥ *Entity*-Klasse `Payment.java`

```
@Entity
@Table(name="BUCHUNG")
public class Payment {
  @Id
  private int id;
  @ManyToOne
  @JoinColumn(name="KONTO", nullable=false)
  private Account account;
  @Column(name="DATUM")
  private Date date;
  @Column(name="BETRAG")
  private double amount;
  ...
```

Anmerkungen zum Beispiel

- Bedeutung des Parameters `type` bei `@OneToMany`:
  - `EAGER`: `Payments` werden beim Laden eines `Account`-Objekts sofort mitgeladen
  - `LAZY`: `Payments` werden erst beim Zugriff geladen
    - Zugriff muß über `get`-Methode (`getPayments()`) erfolgen!
- `mappedBy`-Parameter von `@OneToMany` realisiert eine bidirektionale Assoziation
- `@JoinColumn` definiert, welche Spalte der Tabelle `BUCHUNG` den Primärschlüssel des zugehörigen Eintrags in `KONTO` enthält
  - `nullable` gibt an, ob der Spalteneintrag leer sein darf
- Deskriptor `META-INF/persistence.xml` muss `<class>`-Tags für beide *Entities* enthalten

5.4.10 Transaktionen

- EJB-Container bieten Unterstützung für **flache** (evtl. verteilte) Transaktionen
- EJB-Container kann
  - automatisch Transaktionen um Client-Anfragen generieren
  - vom Client oder einer EJB explizit definierte Transaktions-Grenzen erkennen und an EJBs weitergeben
- EJB-Container regelt Weitergabe von Transaktionen bei Methodenaufrufen
  - z.B. wenn innerhalb einer Transaktion eine Methode gerufen wird, die neue Transaktion definiert
    - verschachtelte Transaktionen sind nicht erlaubt

Arten des Transaktions-Managements

- Festlegung durch Annotation der *Session Bean*
  - `@TransactionManagement` (aus `javax.ejb`)
  - Argument: `TransactionManagementType.BEAN` bzw. `CONTAINER`
- *Bean Managed Transactions*
  - *Session Bean* legt Transaktionsgrenzen selbst fest
- *Container Managed Transactions*
  - Container legt Grenzen und Weitergabe von Transaktionen fest
  - Verhalten durch Annotation der Methoden spezifiziert
    - `@TransactionAttribute` (aus `javax.ejb`)
- *Client Initiated Transactions*
  - Transaktionsgrenzen werden vom Client bestimmt

Bean Managed und *Client Initiated Transactions* mit JTA

- Verwendung des *Java Transaction API* (JTA)
  - Klasse `javax.transaction.UserTransaction`
  - Methoden `begin()`, `commit()`, `rollback()`
- Erzeugung eines `UserTransaction`-Objekts
  - Im Client (oder einer *Session Bean*): über JNDI
    - ```
      ut = (UserTransaction)ctx.lookup(
              "java:comp/env/UserTransaction");
      ```
  - In einer *Session Bean*:
    - über `SessionContext`: Methode `getUserTransaction()`
    - oder direkt über *Dependency Injection*:
      - `@Resource private UserTransaction ut;`
      ```

Container Managed Transactions: Beispiel Ticket-Buchung

```
import javax.ejb.*;

@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class TravelAgentBean implements TravelAgentRemote {
  ...
  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public Ticket bookPassage(CreditCard card, double price) {
    try {
      Reservation res = new Reservation(customer, cruise, price);
      entityManager.persist(res);

      pay.byCredit(customer, card, price);

      return new Ticket(customer, cruise, price);
    }
    catch (Exception e) { throw new EJBException(e); }
  }
```

Transaktionsverhalten des Beispiels

- Abarbeitung von `bookPassage()` soll immer innerhalb einer Transaktion erfolgen
  - Transaktionsattribut `Required`
  - wenn Aufruf nicht in einer Transaktion erfolgt, wird automatisch neue Transaktion erzeugt
- Transaktion wird an die Methoden der genutzten EJBs weitergegeben
  - gemäß Transaktionsattribut der gerufenen Methode
- Falls `bookPassage()` eine System-Exception wirft: *Rollback*, sonst: *Commit* am Ende der Methode
  - alle Unterklassen von `RuntimeException` sind System-Exceptions, insbes. `EJBException`

Container Managed Transactions: Transaktionsattribute

- `NOT_SUPPORTED`
  - Methode unterstützt keine Transaktionen
  - ggf. bereits existierende Transaktion wird bei Aufruf suspendiert
- `SUPPORTS`
  - Methode unterstützt Transaktionen, erzeugt aber keine eigene Transaktion
  - ggf. bei Aufruf existierende Transaktion wird übernommen
- `REQUIRED`
  - Methode muß innerhalb einer Transaktion ausgeführt werden
  - ggf. wird beim Aufruf eine neue Transaktion erzeugt

Container Managed Transactions: Transaktionsattribute ...

- `REQUIRES_NEW`
  - beim Methodenaufruf wird immer eine neue Transaktion erzeugt
  - ggf. bereits existierende Transaktion wird suspendiert
- `MANDATORY`
  - Methode muß innerhalb einer Transaktion ausgeführt werden
  - ansonsten wird Exception geworfen
- `NEVER`
  - Methode darf nicht innerhalb einer Transaktion ausgeführt werden
  - ansonsten wird Exception geworfen

Container Managed Transactions: Transaktionsattribute ...

Transaktions–Attribut	Transaktion des Aufrufers	Transaktion der gerufenen Methode
NOT_SUPPORTED	keine T1	keine keine
SUPPORTS	keine T1	keine T1
REQUIRED	keine T1	T2 T1
REQUIRES_NEW	keine T1	T2 T2
MANDATORY	keine T1	Exception T1
NEVER	keine T1	keine Exception

5.4.11 Zusammenfassung

- EJBs: Komponentenmodell für Anwendungsserver
- *Entities* für Datenmodell
  - Persistenz und Transaktionen durch Container verwaltet
- *Session Beans* für Client-Sitzungen
  - Transaktionen durch Client, Bean oder Container verwaltet
- Container realisiert daneben u.a. Ressourcenmanagement, Namens- und Sicherheitsdienste
- Nicht besprochen: *Entity Beans*, *Message Driven Beans*, und vieles andere mehr ...
  - *Web Services* kommt später noch