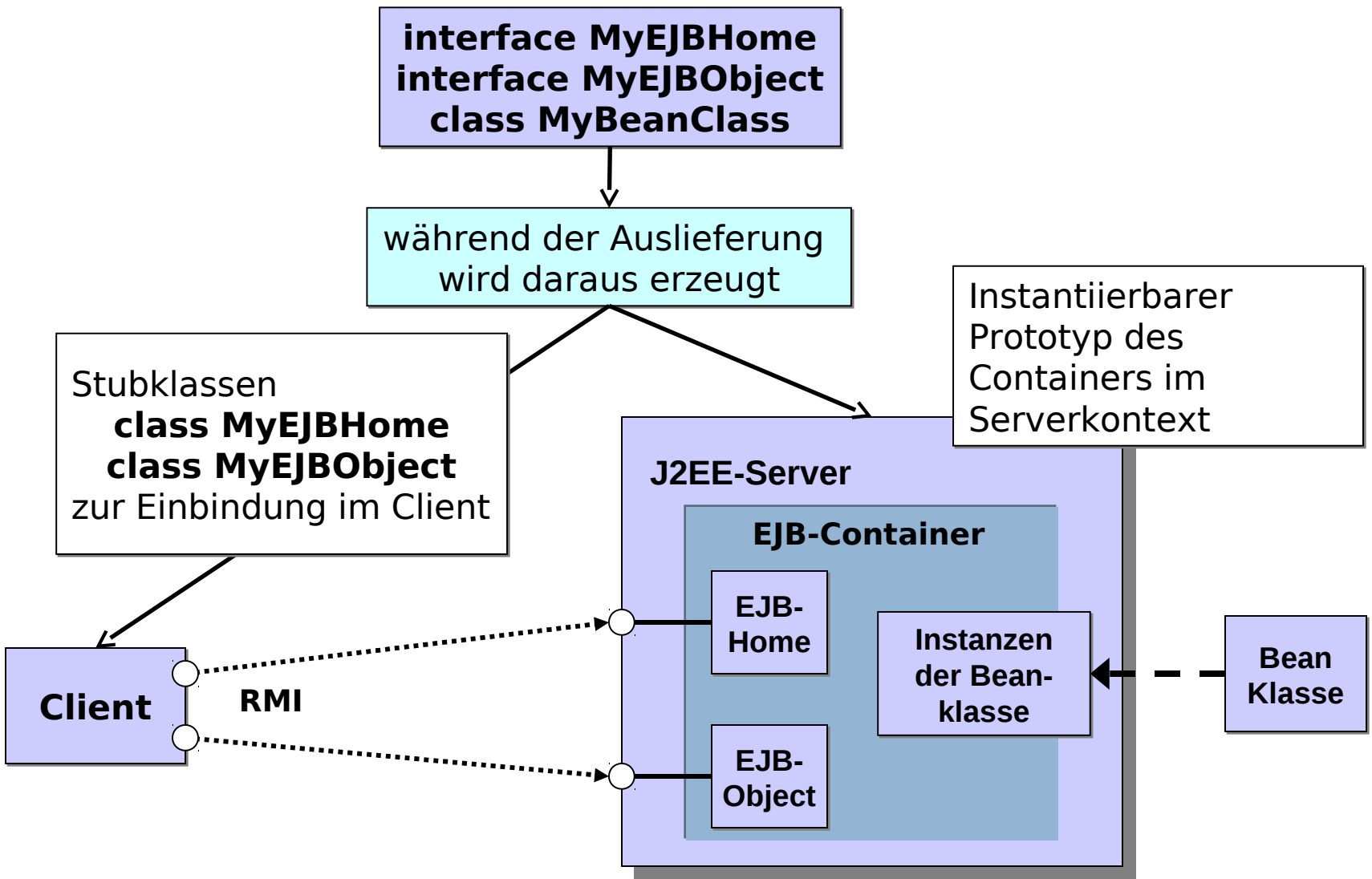


# **Vorlesung Software aus Komponenten**

## **3. Komponentenmodelle**

Prof. Dr. Hans-Gert Gräbe  
Wintersemester 2012/13



#### Bean-Schnittstelle

```
package SemOrg.Schnittstellen;  
public interface Buchung extends javax.ejb.EJBObject {  
    public void buchen(Kunde k, Seminartyp s)  
        throws java.rmi.RemoteException;  
}
```

#### Container-Schnittstelle

```
package SemOrg.Schnittstellen;  
public interface BuchungHome extends javax.ejb.EJBHome {  
    public Buchung create(int owner_id)  
        throws java.rmi.RemoteException, javax.ejb.CreateException;  
}
```

#### Bean-Klasse

```
package SemOrg.Server;
public class BuchungBean implements javax.ejb.SessionBean {

    public BuchungBean() {}

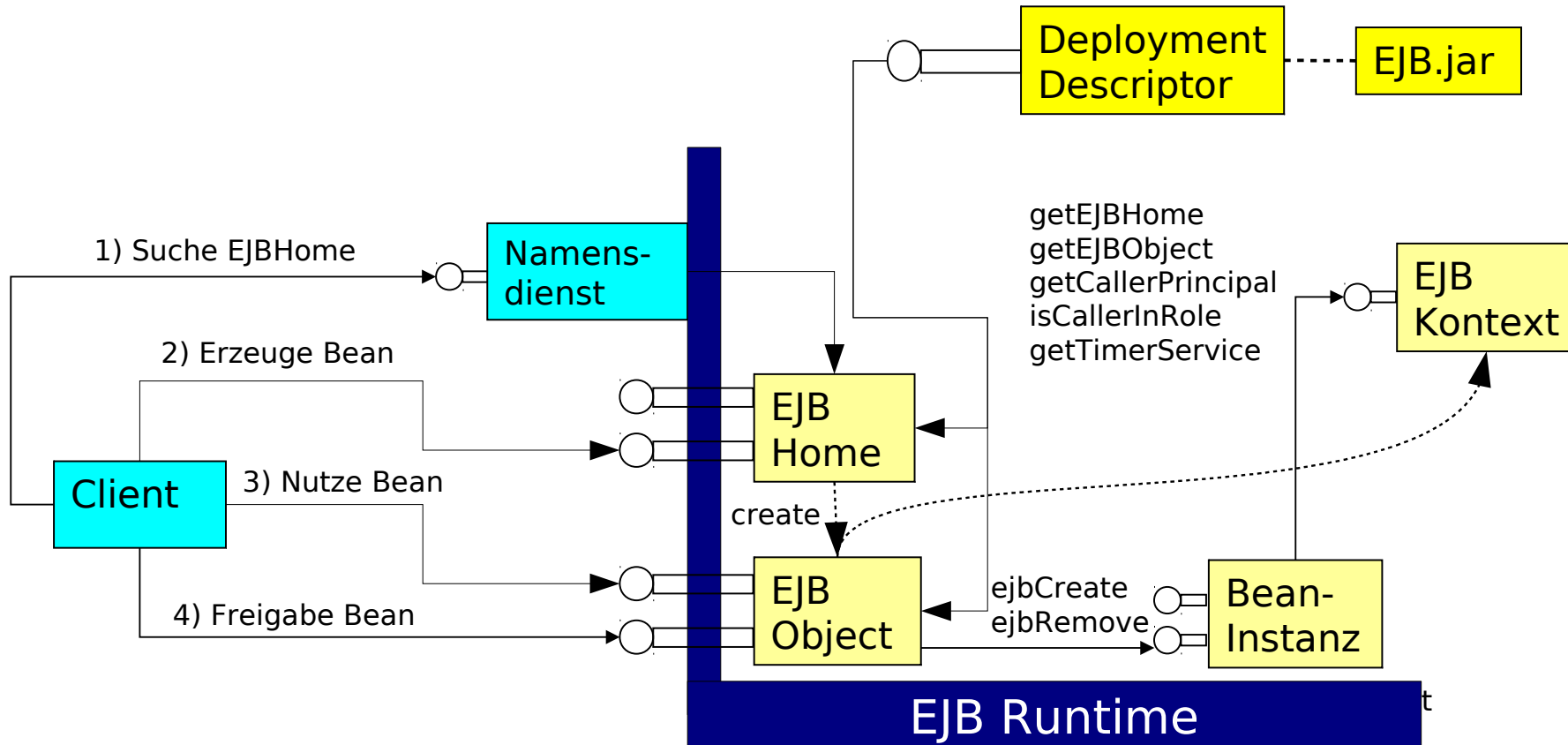
    // Operationen der Remote-Schnittstelle Buchung
    public void buchen(Kunde k, Seminartyp s)
        throws java.rmi.RemoteException {
        // Code zur Ausführung der Buchung
    }

    // Implementierung der Schnittstelle SessionBean
    private javax.ejb.SessionContext ctx;
    public void setSessionContext(javax.ejb.SessionContext sc) {
        this.ctx=sc; // Session-Context für Callback-Methoden
    }
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
}
```

#### Deployment-Deskriptor

```
<!-- Deployment descriptor -->
<enterprise-beans>
  <session>
    <ejb-name>SemOrg/Buchung</ejb-name>
    <home>SemOrg.Schnittstellen.BuchungHome</home>
    <remote>SemOrg.Schnittstellen.Buchung</remote>
    <ejb-class>SemOrg.Server.BuchungBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
  </session>
</enterprise-beans>
```

## Architektur von EJB 2



## Container

- Die EJB 2.x Architektur geht davon aus, dass der EJB Server Hersteller auch der Containerhersteller ist.
- Der Container wird im Rahmen des Deployments der Komponente automatisch vom Server bereitgestellt und die Komponente im Namensdienst registriert.
- 

### Componenten-Container-Modell

Um Komponenten von den Spezifika der unterliegenden Infrastruktur abzuschirmen, werden Container als standardisierte Laufzeitumgebungen eingeführt.

#### Ein einfacher Client

```
public class Client {  
  
    public static void main(String[] args) {  
        Client einClient=new Client();  
        einClient.run();  
    }  
  
    public void run() {  
        Kunde einKunde=getKunde();  
        Seminar einSeminar=getSeminar();  
  
        try {  
            // Namenskontext des Servers finden  
            javax.naming.Context ctx = new javax.naming.InitialContext();  
            Object temp = ctx.lookup("java:comp/env/Buchung");  
        }  
    }  
}
```

## 3.5. Enterprise Java Beans

### Beispiel Seminarorganisation

```
// EJB-Container-Objekt vom Typ BuchungHome  
// auf dem Server finden und instanziiieren  
BuchungHome home= (BuchungHome) PortableRemoteObject.narrow(  
    temp, BuchungHome.class);  
  
// Bean-Instanz anfordern und Buchung auslösen  
Buchung bean = home.create();  
bean.buchen(einKunde, einSeminartyp);  
}  
catch(Exception e) { }  
}
```

## 3.5. Enterprise Java Beans

### Kontrakte

#### Deployment-Kontrakt

Komponente wird in einem speziellen Paketformat ausgeliefert, das spezifische Metainformationen im XML-Format enthält

- Dienst-Schnittstelle, Factory-Schnittstelle, Bean-Implementierung, Ressourcen-Zuordnung, Laufzeitbedingungen

#### Lebenszyklus-Kontrakt

Komponente implementiert spezielle Lebenszyklus-Events, die vom Container „automagisch“ aufgerufen werden können

- OnActivate() {...}
- OnPassivate() { ... }

#### Container-Service-Kontrakt

Der Container stellt der Komponente ein Kontext-Objekt zur Verfügung, über welches die Komponente transparent Dienste aus dem Kontext in Anspruch nehmen kann.

- WhoIsCaller() { ... }
- AccessDataBase() { ... }
- LocateComponent() { ... }

## 3.5. Enterprise Java Beans

### Kontrakte

#### **Umgebungs-Kontrakt**

Der Container sichert eine funktionierende Umgebung für die Komponente entsprechend der Deployment-Information.

#### **Erweiterungs-Kontrakt**

Der Container kann selbst erweiterbar sein und Verhaltensänderungen ausgerollter Komponenten unterstützen

- nutzergetriebene Unterbrechungen
- Unterstützung der Laufzeitkonfiguration von Einheiten
- Einbindung weiterer Dienste zur Laufzeit
- Versionsmanagement ausgerollter Komponenten
- Aspektorientiertes Verhalten

#### **Client-Container-Kontrakt**

Client nutzt Komponentendienste über den Container.

Framework bietet Dienste zum Auffinden der Komponente.

## 3.5. Enterprise Java Beans

### EJB 2 - Nachteile

#### Probleme des Standards

- Viele Schnittstellen zu implementieren (EJBHome, EJBObject, callback zum Kontext)
- Nur eine Geschäftsmethoden-Schnittstelle pro EJB Bean
- Weitere Konventionen sind zu beachten (RMI, spezielle Basis-Schnittstellen)
- Zusätzliche und andere Konventionen für EJB-Klassen im Vergleich zu normalen (plain old) Java-Klassen
- Konfiguration von Beans und Applikationen für Beans erfolgt durch riesige XML-basierte Deployment-Deskriptoren
- EJB-Laufzeit zu rudimentär spezifiziert
- Komplexität der Interaktion mit der Persistenzschicht
- Keine Schnittstellen für Logging, Tracing und andere Basisdienste, um Komponenten zu testen und in der Laufzeit zu verfolgen.
- Beans müssen vom Client manuell gesucht und angesprochen werden.

### Ziele von EJB 3

- Genauere Spezifikation des Bean-Lebenszyklus durch mehr Erweiterungspunkte im Kontext
- Dependency Injection: Einzelne Attributwerte, Methoden- oder Klassendefinitionen werden zur Laufzeit aus dem Kontext importiert
- Verwendung von Meta-Daten
- Interzeptoren und aspektorientierte Ansätze für Infrastrukturdienste
- Vereinfachung der Persistenzbehandlung
- Reduktion der Zahl der Artefakte, die ein Bean-Entwickler bereitstellen muss
- Vereinfachung der EJB Typen durch Reduktion der Zahl der Schnittstellen
- Verbesserung der Testmöglichkeiten außerhalb des Containers

### Wie wird das umgesetzt?

- Einsatz von POJO's (plain old java objects) und POJI's (plain old java interfaces)
  - Dienstschnittstelle als Java-Schnittstelle
  - kein Home-Interface mehr
  - Annotation durch Metadaten für die Konfiguration der EJB-Typen, Local/Remote, Transaktionen, Sicherheit
- Dependency Injection
  - für Attribute, Eigenschaften, Ressourcennutzung
  - werden aus der Umgebung durch Annotationen in die Laufzeit der Beaninstanz „injiziert“
- Erweiterter Lebenszyklus-Unterstützung
  - nutzerdefinierte Callback-Methoden
- Interzeptoren
  - Unterbrechung der Geschäftsmethode in AOP-Manier

### Programmiermodell: Die Bean-Klasse

- Bean-Klasse als zentrales Artefakt, das bereitgestellt wird
- keine Home-Schnittstelle mehr, Laufzeitunterstützung kommt direkt aus dem Kontext
- Beantyp wird durch Annotation festgelegt
- Beanklasse kann Ausnahmen über Annotation vom Typ **@ApplicationException** werfen
- Beanklasse wirft keine **java.rmi.RemoteException** mehr

```
public interface Buchung {  
    public void buchen(Kunde k, Seminartyp s) ;  
}
```

```
@Stateful public class BuchungBean implements Buchung {  
    public void buchen(Kunde k, Seminartyp s) {  
        //Code zur Ausführung der Buchung  
    }  
}
```

### Programmiermodell: Ereignisse im Lebenszyklus

- Ereignisse im Lebenszyklus werden durch den Container über annotierte Callbacks an die Bean-Klasse weitergegeben
- Callbacks können **RuntimeException** auslösen, die Rollback von Transaktionen nach sich ziehen, aber keine **ApplicationException**.
- Letzteres kann durch annotierte **CallbackListener** implementiert werden

```
@Stateful public class BuchungBean implements Buchung {  
    @PreDestroy dispose() { ... }  
}
```

### Programmiermodell: Interzeptoren

- Interzeptoren unterbrechen die Abarbeitung einer Geschäftsmethode
- Interzeptormethoden können in der Beanklasse oder in eigenen Interzeptorklassen definiert sein
  - können **RuntimeException** oder **ApplicationException** werfen, wie in der Schnittstelle der Geschäftsmethode vereinbart
  - Abarbeitung im selben Transaktions- und Sicherheitskontext wie die Geschäftsmethode
  - Können JNDI, JDBC, JMS, andere Beans und den **EntityManager** (Schnittstelle zum Persistenzkontext) nutzen
  - verwendet Dependency Injection
- Für dieselbe Geschäftsmethode können mehrere Interzeptoren definiert sein
- Signatur einer Interzeptormethode:  
**public Object methodName(invocationContext) throws Exception**

## Programmiermodell: Interzeptoren (Fortsetzung)

- **InvocationContext** definiert als

```
public interface InvocationContext {  
    public Object getBean();  
    public Method getMethod();  
    public Object[] getParameters();  
    public void setParameters(Object[]);  
    public EJBContext get EJBContext();  
    public java.util.Map getContextData();  
    public Object proceed() throws Exception;  
}
```
- Kontext-Daten werden von allen Interzeptoren derselben Routine gemeinsam genutzt.
- Aufruf-Ende von **proceed()** innerhalb des Interzeptors - Rückkehr in die unterbrochene Routine

## Wichtige Annotationen und deren Defaultwerte

- **@Local** und **@Remote**: Annotation von Klassen und Schnittstellen
  - Default: @Local
  - ```
@Remote public interface Buchung {  
    public void buchen(Kunde k, Seminartyp s) ;  
}
```
- **@TransactionManagement**: Annotation der Beanklasse
  - Default `TransactionManagementType.CONTAINER`
- **@TransactionAttribute**: Annotation der Klasse oder einzelner Methoden
  - Beschreibt das Transaktionsverhalten
  - Default: `REQUIRED`
- **@Timeout**: Definition einer Timeout-Methode der Bean
- **@ApplicationException**
- Weitere Annotationen für Sicherheitsaspekte:
  - **@RolesReferences**, **@RolesAllowed**, **@RunAs**, **@SecurityRoles**