Björn Persson

# Komponenter med J2EE

## Del 2

Komponentbaserad applikationsutveckling

december 2005

# Om denna sammanfattning

Detta är del 2 av sammanfattningen. Denna del behandlar EJB och relationer mellan EJB. Del 1 behandlar teknologier som bör förstås för att bättre kunna skapa applikationer med EJB.

Avsikten med denna sammanfattning är att ge en introduktion till Sun's Java 2, Enterprise Edition (J2EE) och hur man skapar komponenter (d.v.s. EJB) med J2EE. Sammanfattningen har skrivits av författaren för att lära sig, och just sammanfatta, hur dessa teknologier fungerar. Detta är **inte en ersättning till eventuell kurslitteratur**!

Kapitlen i denna sammanfattning har **inte** skrivits så att de är oberoende av varandra, d.v.s. de bör läsas från första till sista. Senare kapitel om relationer är dock inte tänkta att läsas från början till slut, utan skummas för att återkomma till när kunskapen i dem behövs.

Koden i denna sammanfattning har skapats med Java 2, Enterprise Edition (J2EE) SDK version 1.4.x på datorer med operativsystemen Windows XP/2003. För att köra EJB så har Resin v. 3.0.x använts (www.caucho.com – om en annan EJB-server används så kan exempel behöva justeras för att fungera). Som databashanterare har främst MySQL använts (www.mysql.com), men vissa exempel har även testats i Oracle (www.oracle.com). Övrig programvara som använts är Java-programmeringsmiljön JCreator (www.jcreator.com – för att skapa EJB och servlets), text-editor Crimson Editor (www.crimsoneditor.com – för att skapa *deployment descriptors*, webbsidor, JSP samt skärmdumpar av mappstrukturer) och Microsoft Visio 2003 för klass- och sekvensdiagram (samt "givetvis" Word 2003 för detta dokument samt CutePDF och Adobe Acrobat för att skapa PDF ☺).

## Konventioner i sammanfattningen

Vissa engelska begrepp saknar generellt accepterade översättningar och är därför skrivna på engelska för att kunna relatera till begreppen i engelsk litteratur. I de fall då översättning används så följs det översatta ordet första gången med det engelska inom parentes. Dessa ord skrivs i kursiv stil för att visa att de har "lånats", t.ex. *servlet* och Java-bönor (*Java beans*).[1]

Kod har skrivits med typsnitt av fast bredd (`Courier New`) för att göras mer lättläst samt längre exempel har inneslutits i en ram (se exempel nedan).

```
enVariabel = 1 + 2    //Kommentarer i kod skriv med fet stil
```

Jag är givetvis tacksam för alla konstruktiva synpunkter på sammanfattningens utformning och innehåll.

Eskilstuna, december 2005

Björn Persson, e-post: `bjorn.persson@mdh.se`
Ekonomihögskolan, Mälardalens högskola
Personlig hemsida: `http://www.eki.mdh.se/personal/bpn01/`

[1] Eftersom många termer i J2EE saknar bra översättning, t.ex. *servlets* och *deployment descriptor*, så skrivs dessa endast med kursiv stil i de avsnitt de introduceras i (och för att spara tid för mig vid redigering ☺).

# Innehållsförteckning



ATT GÖRA (?):

* Förklara EJB:ers olika livscykeltillstånd.

* Förklara relationer och local-gränssnitt.

* get-metoder i local- resp. remote-gränssnitt.

* Förklara relationer med sammansatt nyckel?

* Göra en sammanfattning om datorbokningssystem?

# 10 Enterprise JavaBeans (EJB)

*Enterprise JavaBeans* (EJB) är en del av Sun's *Java 2, Enterprise Edition* (J2EE). Detta är den ”mest” intressanta teknologin inom J2EE, i alla fall om vi talar om komponenter i distribuerade applikationer. ☺ Övriga teknologier i J2EE (och J2SE, t.ex. JNDI[2], JDBC[3] och RMI[4]) är stöd för EJB, d.v.s. teknologier som krävs för att EJB ska fungera (och som vi bör ha en grundläggande förståelse av).

EJB är komponenter som ska exekvera på en server och används därför av klienter över nätverk (eller via ett webbgränssnitt, servlets/JSP). Genom att placera komponenterna på en server så centraliseras affärslogiken, vilket gör det lättare att uppdatera applikationer. Centraliseringen gör även livet lättare för administratörer då endast ändringar i gränssnitt kräver uppdateringar av klientdatorer. Ytterligare en fördel med centralisering är att säkerheten kan ökas då den bara behöver upprätthållas på ett ställe.

En nackdel med centralisering är att om servern havererar så är inte applikationen tillgänglig för klienterna. Detta kan lösas genom att duplicera applikationen på flera servrar (spegling eller klustring).

## 10.1 Introduktion

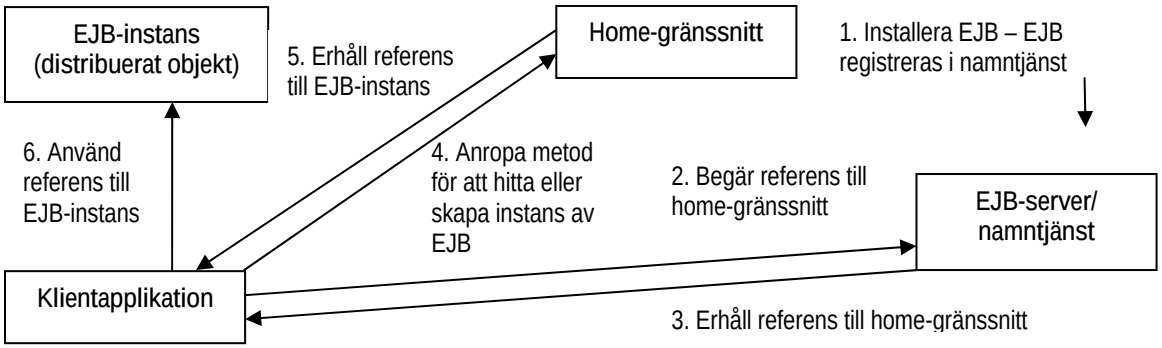
En EJB[5], d.v.s. komponenten, motsvarar bl.a. av en klass i de klassdiagram som tas fram i analysfasen vid utveckling av applikationer. Men en EJB implementeras med ett antal gränssnitt och klasser, d.v.s. inte bara en klass som i ”vanliga” objektorienterade applikationer. Dessa klasser och gränssnitt betraktas som en enhet, d.v.s. en EJB.

Ett av skälen till denna uppdelning av EJB i delar är för funktion. En del är, vad jag kallar, gränssnittet för ”livscykelhantering” (vars riktiga namn är *home*-gränssnitt) – ett gränssnitt vi kan fråga efter en existerande instans av EJB eller skapa en ny instans. Metoder i detta gränssnitt returnerar en referens till EJB:s, vad jag kallar, ”gränssnitt för affärslogik” (vars riktiga namn är *remote*-gränssnitt). När vi har en referens till EJB:s ”affärslogikgränssnitt”, så anropas metoder i gränssnittet som om EJB vore ett ”vanligt” objekt (precis som med RMI). Denna uppdelning, och antal nya termer, kan till en början göra att utveckling av EJB:er kan kännas svårt (komplext), men med några EJB ”under bältet” så brukar ett mönster framgå.[6]

Att använda en EJB påminner om hur man använder distribuerade objekt med RMI. Precis som med RMI så registreras objekt i en namntjänst – skillnaden är dock att vi registrerar *home*-gränssnittet istället för själva objektet. Registrering av *home*-gränssnittet sker då vi installerar EJB i EJB-server (oftast genom att vi kopierar EJB till en speciell mapp). För att kunna skapa eller hitta en instans av EJB så frågar vi sen EJB-server (eller snarare dess namntjänst) efter en referens till *home*-gränssnittet (via JNDI). Via *home*-gränssnittet så anropar vi metoder för att hitta eller skapa en instans av EJB, d.v.s. för att erhålla en referens till instans (dess *remote*-gränssnitt). Denna (andra) referens kan sen betraktas som ett ”vanligt” objekt, om än med något längre svarstid om instansen finns på en annan dator i nätverk.

---

[2] JNDI används för att erhålla referenser till instanser av EJB.
[3] JDBC används inte direkt – EJB-server gör (eller kan göra) jobbet åt oss. ☺
[4] RMI används när metoder i EJB anropas (utan att vi märker det, annat än prestandamässigt ☺) – dock används inte RMI-register, m.m..
[5] Förkortningen EJB använder jag för både en komponent, ”böna”, och för teknologin Enterprise JavaBeans. Sammanhanget avgör (eller bör avgöra) vilket som menas. ☺
[6] J2EE är, liksom de flesta Java-teknologierna, en relativt komplex teknologi men mycket genialisk. Detta gör ofta att Java-teknologier är aningen svårare att lära sig men också oftast värt mödan när man ser det genialiska.




För att lättare kunna distribuera en EJB:s gränssnitt och klasser så paketeras dessa i JAR-filer (ZIP-filer med .JAR som filändelse). För att koppla samman gränssnitt och klasser i EJB, samt beskriva för EJB-servern vilka tjänster som komponenten ska använda, så använder vi en XML-fil kallad *deployment descriptor*.

Det finns två typer av komponenter: ”datakomponenter” och ”arbetskomponenter”. En datakomponent, kallat ***entity beans*** i EJB, och motsvarar data som lagras i datakällor (främst relationsdatabaser i J2EE). Arbetskomponenter motsvarar fasadklasserna, d.v.s. klasserna som skapas för att hantera logiken i objektorienterade applikationer, och kallas ***session beans*** i EJB. (*Entity* och *session beans* behandlas mer ingående senare i kapitlet.)

I en J2EE-server så exekverar EJB i, vad som kallas, en **container** (eller EJB-container). Uppgiften för en container är att skapa en miljö för EJB:s att exekvera i samt erbjuda tjänster (t.ex. transaktionshantering) till EJB. Tjänsterna som containers erbjuder sparar utvecklare timmar med arbete då de slipper utveckla tjänsterna om och om igen (eller snarare anpassa efter att dom utvecklats första gången). Nackdelen med att använda tjänsterna är att det kostar i form av prestanda – de har gjorts generella för att kunna användas av alla typer av komponenter. (I de fall då prestanda är mycket viktigt så kan det alltså vara värt tiden att själv utveckla de tjänster som önskas.)

Ett skäl till att använda containers är för att skärma EJB från den faktiska implementationen av J2EE-serverar. Containers (eller snarare J2EE-servern) implementerar gränssnitten i J2EE som Sun definierat. (Allt detta leder till att redan existerande applikationsservrar kan utöka sin funktionalitet genom att, utöver redan existerande funktionalitet, implementera J2EE-gränssnitten och nå fler kunder.)

I Resin så exekverar EJB och servlets/JSP i samma container, d.v.s. servlets kan använda EJB:ers *local-* och *local home*-gränssnitt (se nedan för förklaring av begrepp). Av denna anledning kommer (för enkelhetens skull) alla EJB-exempel att använda servlets som klienter i denna sammanfattning.

## 10.2 EJB-komponenters uppbyggnad

En EJB-komponent består som sagt av flera delar: två (eller fyra) gränssnitt (*interface*), en eller två klasser samt en ”konfigurationsfil”.

- *Remote*-gränssnitt – definierar komponentens publika metoder mot klienten, d.v.s. affärslogiken. Metoder i detta gränssnitt motsvarar oftast de i klassdiagram som togs fram i analysfasen för applikation.
  Utökar gränssnittet `javax.ejb.EJBObject`
  Namnstandard: `<Komponentnamn>` Ex: `Hus`, `Kund`

- *Home*-gränssnitt – innehåller metoder för komponentens livscykel (skapande, sökande, m.m.).
  Utökar gränssnittet `javax.ejb.EJBHome`
  Namnstandard: `<Komponentnamn>Home` Ex: `HusHome`, `KundHome`

- *Bean*-klass – implementerar komponentens affärsmetoder, och några av home-gränssnittets metoder, men inte någon av ovanstående gränssnitt (mer om detta nedan).
  Implementerar gränssnittet `javax.ejb.EntityBean` eller `javax.ejb.SessionBean`
  Namnstandard: `<Komponentnamn>Bean`[7] Ex: `HusBean`, `KundBean`

- Primärnyckelsklass (*primary key class* – endast entity bean) – behöver endast skapas om primärnyckel i tabell är sammansatt, d.v.s. består av fler än ett värde (annars kan vi använda klasser som `String` eller någon av de primitiva typernas *wrapper*-klasser[8], t.ex. `Integer` för `int` och `Float` för `float`).[9]
  Implementerar gränssnittet `java.io.Serializable`
  Namnstandard: `<Komponentnamn>PK` Ex: `HusPK`, `KundPK`

- *Deployment descriptor* (en textfil med XML) – kopplar samman de två gränssnitten, *bean*-klassen och eventuell primärnyckelsklass.

I EJB versioner tidigare än 2.0 så måste alla klienter, d.v.s. även andra EJB i samma container, använda *remote*- och (*remote*-)*home*-gränssnitt. Detta innebar att alla klienter måste använda RMI för att nå en EJB. Om klienten finns på en annan dator så är RMI ett måste, men om klienten finns på samma dator så är RMI mer av ett ”hinder” (*overhead*). I EJB version 2.0 introducerades gränssnitten *local* och *local home*. Tanken var att klienter i samma container (d.v.s. andra EJB) skulle kunna ”hoppa förbi” RMI och gå direkt på EJB, d.v.s. göra lokalt användande mer effektivt. Därmed kan en EJB har två eller fyra gränssnitt (*remote* och *home* och/eller *local* och *local home*).

- *Local*-gränssnitt – definierar komponentens publika metoder mot klienten, d.v.s. affärslogiken. Detta motsvarar *remote*-gränssnittet, men kan endast användas av klienter i samma container som EJB.
  Utökar gränssnittet `javax.ejb.EJBLocalObject`
  Namnstandard: `<Komponentnamn>`[10] eller `<Komponentnamn>Local` Ex: `Hus`, `KundLocal`

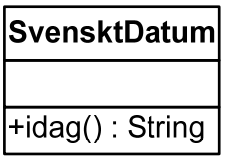
- *Local home*-gränssnitt – innehåller metoder för komponentens livscykel (skapande, sökande, m.m.). Detta gränssnitt motsvarar *home*-gränssnittet, men kan endast användas av klienter i samma container som EJB.
  Utökar gränssnittet `javax.ejb.EJBLocalHome`
  Namnstandard: `<Komponentnamn>Home`[11] eller `<Komponentnamn>LocalHome` Ex: `HusHome`, `KundLocalHome`

*Deployment descriptor* används även för att koppla samman local- och local home-gränssnitten med övriga gränssnitt och klasser.

---

[7] Vissa författare vill använda suffixet EJB, t.ex. KundEJB, istället för Bean.
[8] En *wrapper*-klass är en klass som innesluter ett annat värde (eller objekt). I Java-versioner tidigare än 1.5 så måste vi ”förpacka” värden av de primitiva typerna innan vi kan lagra dem i klasser som Vector (som tar ett argument av typen Object som inparameter till metoden `add()`).
[9] I Resin så fungerar det att använda de primitiva typerna som ”primärnyckelsklasser”, d.v.s. vi behöver inte använda *wrapper*-klasserna (vilket dock rekommenderas för att behålla kompabilitet).
[10] Att inte använda suffixet `Local` fungerar endast om vi inte har ett remote-gränssnitt också.
[11] Ibid.

### 10.2.1 Exempel på gränssnitt och klasser för en EJB-komponent




I detta första exempel (som borde[12] fungerar i alla version av EJB) skapas en väldigt enkel *stateless session bean* (utan implementation av metod) för att visa hur gränssnitt och bean-klass kan se ut. Namnet på EJB (liksom klass i klassdiagram till höger) är SvensktDatum och, för att följa namnstandard ovan, så ges remote-gränssnittet detta namn (d.v.s. `SvensktDatum`), home-gränssnittet `SvensktDatumHome` samt bean-klassen namnet `SvensktDatumBean`. Eftersom det är en session bean så behöver vi ingen primärnyckelsklass.

## Remote-gränssnitt

Remote-gränssnitt ska innehålla metoder som motsvarar affärslogik för komponenter, d.v.s. de metoder för klasser som tagits fram i applikationens analysfas (metoderna i klassdiagrammet). Detta är alltså EJB:s publika ”ansikte”. I detta exempel så innehåller klassen bara en metod `idag()` som returnerar en sträng med dagens datum i svenskt format (på formen åååå-mm-dd).

Eftersom metoder i en EJB är tänkta att användas från klienter på andra datorer (och att RMI är protokollet för kommunikation ☺) så måste alla metoder i remote-gränssnittet slänga undantaget `java.rmi.RemoteException`. Gränssnittet måste därför också utöka (ärva från) gränssnittet `java.rmi.Remote`. Detta gör vi genom att utöka gränssnittet `javax.ejb.EJBObject`, som i sin tur utökar gränssnittet `Remote`. Gränssnittet `EJBObject` innehåller även en del metoder som vi **inte** behöver implementera (de implementeras av EJB-server).

```
import java.rmi.RemoteException;

public interface SvensktDatum extends javax.ejb.EJBObject
{
  public String idag() throws RemoteException;
}  //interface SvensktDatum
```

## Home-gränssnitt

Home-gränssnitt innehåller metoder för att bl.a. kunna skapa (eller hitta existerande) EJB. I detta enkla exempel så behöver vi endast en metod för att kunna skapa en ny instans av EJB, d.v.s. metoden `create()` utan några parametrar. Detta är den enklaste formen av create-metoder, d.v.s. en create-metod kan finnas i flera former (d.v.s. med noll eller fler parametrar).

Precis som med remote-gränssnitt så måste home-gränssnitt utöka gränssnittet `java.rmi.Remote`, vilket man gör genom att utöka gränssnittet `javax.ejb.EJBHome` (som i sin tur utökar gränssnittet `Remote`). Och precis som metoder i remote-gränssnitt så måste även metoder i home-gränssnittet slänga `java.rmi.RemoteException`. Create-metoder måste även slänga `javax.ejb.CreateException`. (find-metoder, i entity beans, måste även slänga `javax.ejb.FinderException` – mer om detta när entity beans beskrivs).

```
import javax.ejb.CreateException;
```

[12] Med borde menar jag att jag testat att kompilera koden i JCreator samt installera i Resin. Men lägg **inte** ner tid på att själva skriva av (eller kopiera) koden för att testa. Senare exempel är bättre för det. Övriga exempel med entity beans i sammanfattning fungerar endast med EJB version 2.0 och senare

```
import java.rmi.RemoteException;

public interface SvensktDatumHome extends javax.ejb.EJBHome
{
  SvensktDatum create() throws RemoteException, CreateException;
}  //interface SvensktDatumHome
```

## Bean-klass

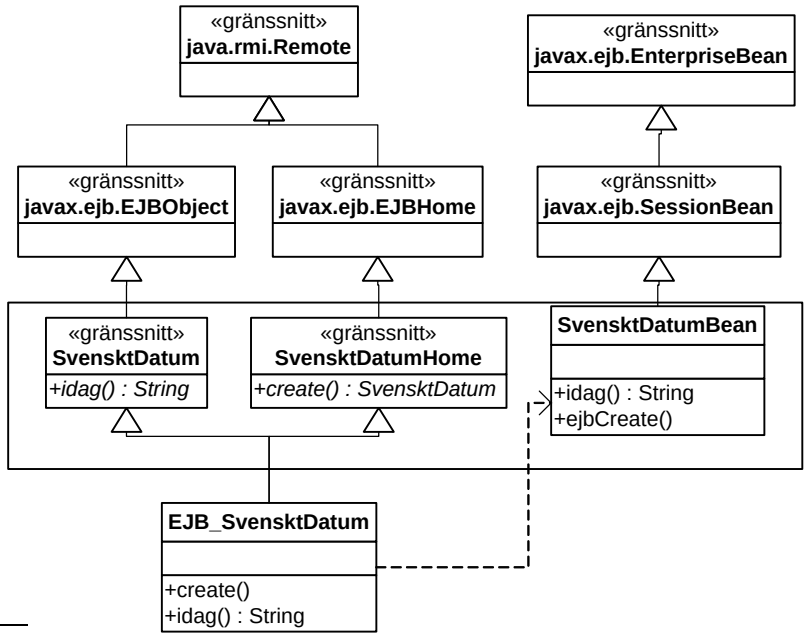
Komponentklassen (eller bean-klassen, *bean class*) innehåller implementationen av vår EJB-komponents publika metoder – metoden `idag()` i detta fall. Men observera att denna klass **inte** implementerar (ärver från) vårt remote-gränssnitt! Detta kan verka underligt till att börja med, men detta är en genialisk (om än svårförståelig) lösning som objektorientering ger möjlighet till (mer om detta nedan).

Eftersom detta är en session bean så måste vår bean-klass implementera gränssnittet `javax.ejb.SessionBean`, även om metoderna från gränssnittet inte behöver innehålla någon kod.

```
public class SvensktDatumBean implements javax.ejb.SessionBean
{
  public String idag()
  {
    String datumIdag = null;
    //Kod för att skapa en sträng enligt svenskt format (åååå-mm-dd)
    return datumIdag;
  }

  /*** Implementera metoder i gränssnittet SessionBean ************************/
  public void ejbCreate() { }
  public void setSessionContext(javax.ejb.SessionContext ctx) {}
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
}  //class SvensktDatumBean
```

## Exempel på en faktisk implementation av EJB

Metoderna i remote- och home-gränssnitten ovan implementeras inte av våra bean-klasser, utan av en (eller flera) klass(-er) som EJB-server genererar när vi installerar[13] våra EJB i EJB-container. Hur genereringen sker avgörs av tillverkaren av EJB-servern.




Den (eller de) av EJB-servern genererade klassen(-erna) skapar sen instans av bean-klass när klienten anropar en metod

[13] De genererade klasserna kan även genereras när EJB används första gången – avgörandet om när görs av tillverkaren av EJB-servern (d.v.s. har inte definierats av Sun).

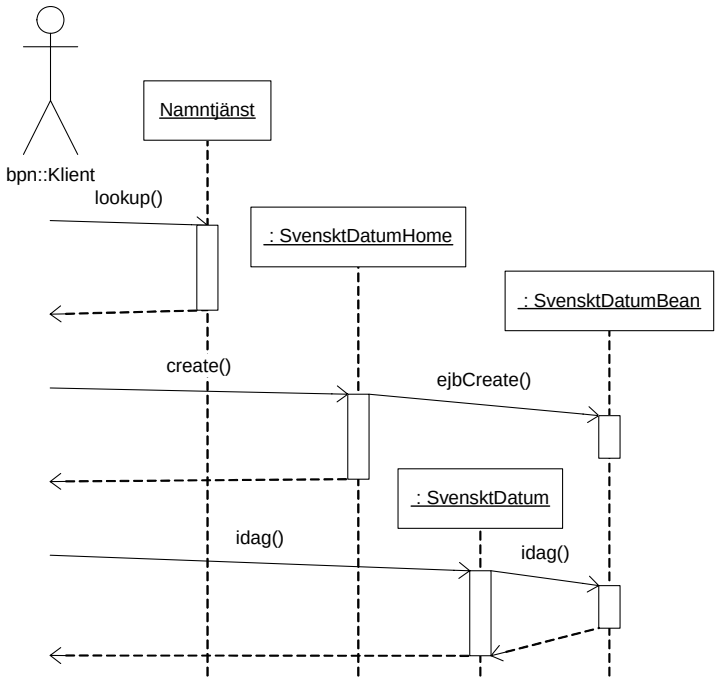
i remote-gränssnittet (ett anrop som hanteras av metod i genererad klass). Den genererade klassens metod kan sen anropar metod med samma namn i den nyskapade instansen av bean-klassen. Mellan metodanrop kan EJB-server förstöra instansen av bean-klassen för att spara minne (om den så önskar). Detta är inget som vi behöver bry oss om eftersom vi (och andra användare av våra EJB) endast kommer använda gränssnitten remote och home. Men förståelse av hur det skulle kunna lösas bör underlätta förståelse för hur EJB fungerar (och varför EJB-specifikationen kräver flera gränssnitt och klasser för en EJB). ☺

I klassdiagram visas hur gränssnitt i J2EE-specifikationen relaterar till gränssnitt och klasser som skapas för en EJB. Klassdiagrammet visar även ett exempel på hur de genererade klasserna är relaterade till en EJB:s klasser och gränssnitt.

Klasser och gränssnitt som vi behöver skapa har inneslutits i en ram.[14] Remote-gränssnittet (`SvensktDatum`) utökar (ärver) från `EJBObject` som i sin tur ärver från `Remote` och home-gränssnittet (`SvensktDatumHome`) utökar `EJBObject` som också, i sin tur, utökar `Remote`. Eftersom vår EJB är en session bean så implementerar bean-klassen (`SvensktDatumBean`) gränssnittet `SessionBean` som i sin tur utökar gränssnittet `EnterpriseBean`.

EJB-server skulle kunna generera en klass (`EJB_SvensktDatum`) som implementerar EJB:s gränssnitt samt använder sig av EJB:s bean-klass (`SvensktDatumBean`). Men EJB-server skulle, som sagt, även kunna generera flera klasser för att implementera gränssnitten. I Resin så genereras t.ex. en klass `SvensktDatumBean__EJB` med tre inre (nästlade) klasser (`Remote`, `Home` och `Bean`).

I sekvensdiagram[15] till höger visas hur metoder anropas för att skapa och fråga EJB SvensktDatum om aktuellt datum och tid. Gränssnitten `SvensktDatumHome` och `SvensktDatum` implementeras som sagt av en klass (eller klasser) som EJB-server genererar (`EJB_SvensktDatum` i klassdiagram ovan).




## Deployment descriptor

Uppgiften för en deployment descriptor är att koppla samma de två (eller fyra) gränssnitten med klasserna (bean- och primärnyckelklass) samt ange vilka av EJB-servers tjänster som EJB vill använda (men även för att koppla samman EJB med relationer – mer om detta senare).

Allt i en deployment descriptor innesluts av taggen `<ejb-jar>`[16] samt alla EJB:er i taggen `<enterprise-beans>`. Därefter använder vi en tagg `<session>` eller `<entity>` för att beskriva varje session respektive entity beans. För varje EJB krävs minst namnet som EJB ska

[14] Ramen är **inte** UML-standard. ☺
[15] Det finns en svag risk att diagram inte är korrekt… analys har aldrig varit min starka sida. ☺
[16] D.v.s. rottaggen för XML-dokumentet är `<ejb-jar>`. ☺

refereras till (JNDI-namn, `<ejb-name>`), fullständiga[17] namnen på gränssnitten remote (`<remote>`) och home (`<home>`) samt fullständigt namn på bean-klassen (`<ejb-class>`). För entity beans behövs ytterligare fler taggar (mer om detta senare).

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>SvensktDatum</ejb-name>
      <home>SvensktDatumHome</home>
      <remote>SvensktDatum</remote>
      <ejb-class>SvensktDatumBean</ejb-class>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Eftersom en deployment descriptor består av XML så är skiftläge (gemen eller versal) och mellanslag viktiga (liksom stavning ☺). D.v.s. texten i deployment descriptor **måste** skrivas 100% rätt!. M.a.o. – börja felsöka i deployment descriptor (speciellt om kod kopierats från t.ex. Word- och PowerPoint-dokument, som detta, i PDF-filer).

## Paketera EJB

När gränssnitt och klasser kompilerats (till .CLASS-filer), samt deployment descriptor skapas, så brukar man paketera EJB. Genom att paketera EJB så kan en EJB hanteras som en enhet/modul istället för 3-6 .CLASS-filer och en XML-fil (i en eller flera mappar). EJB paketeras i JAR-filer (*Java Archive*, ett ZIP-arkiv med filändelsen .JAR istället för .ZIP) med bibehållen mappstruktur (om EJB skapas i paket så ska .CLASS-filerna ligga i en mappstruktur som motsvarar paketen och dess delar).

Om en applikation ska innehålla flera EJB så brukar JAR-filer med EJB paketeras i ytterligare ett paket – en EAR-fil (*Enterprise Archive*). En EAR-fil är en JAR-fil med filändelsen .EAR (d.v.s. ytterligare ett ZIP-arkiv). Man packar alltså JAR-filerna i ytterligare en JAR-fil. Genom att packa flera EJB (d.v.s. JAR-filer) i en och samma fil så underlättar man för administratörer av applikationsservrar att installera en EJB-applikation.

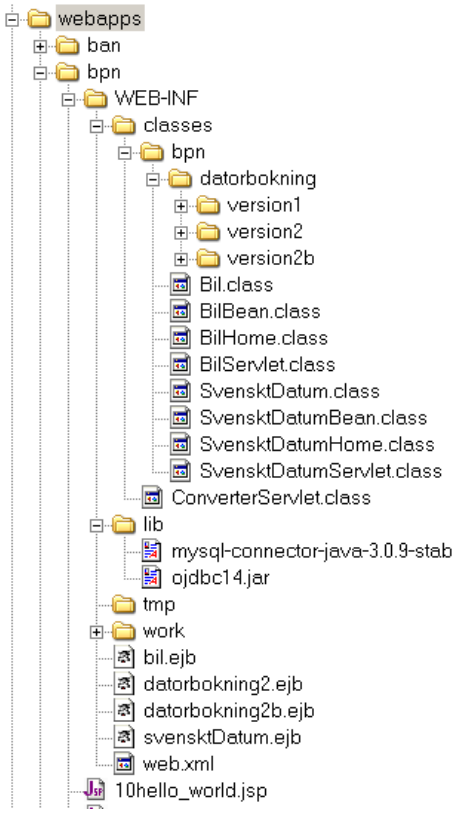
Mycket av denna paketering, men även skapande av deployment descriptor, brukar kunna göras med EJB-servrars installationsverktyg, ofta med ett grafiskt gränssnitt. Verktyget, liksom tjänster som EJB-server erbjuder, är det som gör att olika EJB-servrar konkurrerar med varandra om kunder.

(För att komplicera det ytterligare så kan man även skapa WAR-filer, *Web Archive*, som innehåller HTML-filer och andra webbresurser för installation i J2EE-servrar med webbcontainer. Återigen är det ett ZIP-arkiv fast med filändelsen .WAR.)

Om vi använder Resin så behöver vi **inte** packa filer i JAR-filer för att använda EJB (ytterligare ett själ till varför jag gärna använder Resin – jag kompilerar gränssnitt och klasser direkt till webbserver ☺), även om vi kan det. En nackdel med Resin (i alla fall gratisversionen) är att det inte finns ett grafiskt verktyg för att skapa JAR- och EAR-filer.

[17] Med fullständigt namn menas paketnamn och gränssnitts-/klassnamn, t.ex. `paket1.paket2.Klassnamn`.

## Installera EJB




När EJB paketerats är det dags att installera (*deploy*) EJB i EJB-container. Hur detta sker beror på vilken EJB-container man använder. Av detta skäl är de flesta böcker om EJB vaga på denna punkt (vilket inte är så roligt om man är nybörjare på EJB).

I Resin så finns det (bl.a.) två sätt att installera EJB på

- paketera i JAR-filer.
- kopiera in .CLASS-filer direkt.

Jag föredrar själv det andra alternativet då detta är lättast under tiden man utvecklar EJB, d.v.s. man slipper packa EJB varje gång men kompilerat en ny version av EJB. När sen EJB är buggfri (d.v.s. fungerar) så kan man paketera i JAR-filer.

Resin (och andra webbservrar) använder ett begrepp kallat webbapplikation. En **webbapplikation** är en mapp, alla undermappar (som inte i sig är webbapplikationer) och alla webbresurser i dessa mappar. **Webbresurser** är HTML-/XML-dokument, bildfiler (JPG/GIF), servlets, JSP, EJB, m.m.. I bilden till höger så finns en mapp `WEBAPPS` (som oftast är namnet på roten för webbservern i Resin) i vilken två webbapplikationer har skapats: `ban` och `bpn`. I `bpn` (och andra webbapplikationer) finns en mapp `WEB-INF`[18] där konfigurationsfil för webbapplikation (`WEB.XML`) och deployment descriptor (filer med filändelsen `.EJB`) placeras. Mappen `WEB-INF` innehåller sen en mapp `CLASSES` där vi placerar filer med filändelsen `.CLASS`, d.v.s. våra kompilerade gränssnitt och klasser för EJB (men även servlets). Observera att om våra gränssnitt och klasser ingår i paket så måste detta reflekteras i mappstrukturen under mappen `CLASSES`. I bilden ovan så finns t.ex. klassen `Sal` (i den icke expanderade mappen `VERSION1`) i ett paket `bpn.datorbokning.version1` och gränssnittet `SvensktDatum` finns i paketet `bpn`. Om vi använder JCreator för att skapa våra EJB:s så genererar JCreator mappstrukturen åt oss när vi kompilerar gränssnitt och klasser. D.v.s. vi kan kopiera mappstrukturen (eller kompilera direkt till mappen `CLASSES` i Resin-servern). (I mappen `LIB` placeras JAR-filer – i bild ovan så finns drivrutinerna för MySQL och Oracle där.)

I webbapplikationens rotmapp (t.ex. `bpn`), eller någon av dess undermappar (**utom** `WEB-INF`[19]), placeras HTML-dokument, bildfiler, JSP och andra webbresurser (**utom** servlets och EJB). Filen `10HELLO_WORLD.JSP` ligger t.ex. i webbapplikationens rotmapp.

I mappen `CLASSES` placeras alltså filer med extensionen `.CLASS`, i `WEB-INF` filer med extensionen `.EJB` samt, om vi skapat JAR-filer, filer med extensionerna `.JAR`, `.WAR` och `.EAR` i mappen `LIB`.

[18] Denna mapp bör vara namngiven med versaler – övriga mappar och filer i Resin kan variera i skiftläge (gemener eller versaler).
[19] Resin är konfigurerad så att klienter (d.v.s. webbläsare) inte kan läsa filerna i mappen `WEB-INF` och dess undermappar.

### Klient till EJB

Klienter använder JNDI för att erhålla en referens till EJB:s home-gränssnitt. Vilken typ av drivrutin som krävs liksom URL till namntjänst (*name server*) varierar mellan olika J2EE-servrar. Av detta skäl så är, åter igen, många böcker vaga på hur detta sker (vilket, åter igen, inte är så roligt om man är nybörjare på EJB).

Resin har sin egen namntjänst och därmed sina egna drivrutiner. Så för att underlätta exempel i fortsättningen kommer jag att använda servlets för att testa EJB då man kan använda den ”lokala” namntjänsten för att erhålla referenser till local home-gränssnitt. (Och därför, liksom att det är lättare, så använder exempel bara local- och local home-gränssnitt i fortsättningen.)

I detta exempel så undviker jag att (exakt) visa hur man använder JNDI för att erhålla en referens till home-gränssnitt då detta sker på lite olika sätt beroende på vilken EJB-container man använder. Men som exempel försöker visa så är det endast erhållandet av referens till home-gränssnittet som gör användandet av EJB, liksom användandet av create-metoder i home-gränssnitt i stället för det reserverade ordet `new`, det som gör användandet av EJB annorlunda från vanliga objekt.

Först deklareras variabler för både home- och remote-gränssnitt. I en större applikation så räcker det med en variabel för EJB:s home-gränssnittet medan vi kan deklarera en variabel av remote-gränssnittets typ för varje instans av EJB vi använder. I EJB måste vi, som sagt, först hämta en referens till EJB:s home-gränssnitt, vilket vi använder JNDI till. När vi har en referens till ett initialt kontext i namntjänst så kan vi fråga namntjänsten efter en referens till EJB:s home-gränssnitt. Här används JNDI-namn som vi anger i deployment descriptor. Nästa steg är att skapa en instans av EJB genom att anropa en create-metod som returnerar en referens till EJB:s remote-gränssnitt. Efter detta så kan vi, precis som med vanliga objekt, anropa metoder i remote-gränssnitt.

```
SvensktDatumHome home = null;  //Variabel för referens till home-gränssnit
SvensktDatum sd = null;        //Variabel för referens till remote-gränssnitt

//Hämta en referens till initialt kontext i namntjänst ("hypotetiskt")
Context initial = (Context) new InitialContext().lookup(...);

//Fråga namntjänst efter referens till EJB:s home-gränssnitt
home = (SvensktDatumHome) initial.lookup("SvensktDatum");

//Skapa en instans av EJB – returnerar en referens till EJB:s remote-gränssnitt
sd = home.create();

//Anropa metod i remote-gränssnitt och skriv ut resultatet
System.out.println("Datum idag på svenskt format" + sd.idag());
```

Dags att titta närmare på olika typer av EJB:er, d.v.s. vad *entity* och *session beans* är.

## 10.3 Entity, Session och Message-driven beans

En komponent kan vara en *entity bean*, en *session bean* eller en *message-driven bean*. Komponenter av typen ***entity bean*** motsvarar objekt som ska manipuleras av applikationen och har därför tillstånd. D.v.s. *entity bean* har egenskaper som beskriver objektet som komponenten motsvarar. Komponentens tillstånd (värde på egenskaperna) lagras i databas, vilket kan hanteras av komponenten själv (*bean managed persistence*, BMP) eller av EJB-servern (*container managed persistence*, CMP). I det senare fallet överlåter vi till EJB-servern att hantera transaktioner, d.v.s. risken för fel (buggar) minskar drastiskt (något som vi strävar efter ☺).

Komponenter av typen ***session bean*** sköter själva exekveringen av affärslogiken som applikationen ska hantera. En *session bean* kan vara en ren arbetskomponent (*stateless*) som inte behåller sitt tillstånd mellan metodanrop eller behålla sitt (eller del av sitt) tillstånd (*stateful*) mellan metodanropen (tillståndet lagras dock **inte** i en databas!).

En ***message-driven bean*** fungerar som dess namn visar på: den utför något när den erhåller ett meddelande (ifrån en JMS-server, *Java Message Service*). Denna typ av komponenter behandlas inte vidare i denna sammanfattning.

Utöver entity och session beans (liksom message-driven) olika funktion (data resp. arbete) så skiljer de sig även på andra punkter, bl.a. vilket gränssnitt som bean-klass ska ärva från och vilka typer av metoder (find- och create-metoder – se nedan) som måste finnas i deras home-gränssnitt.

### 10.3.1 Transaktioner, objektpoolning och samtidighet

En av de mest intressanta tjänster som en EJB-server kan erbjuda är antagligen transaktions-hantering. Att skriva korrekt kod för transaktioner är något som är viktigt, och att slippa behöva skriva koden – om och om igen för den delen... Detta är något som EJB-servrar kan hjälpa till med.

Exekvering brukar ske stötvis, d.v.s. oftast inte konstant. Därför brukar EJB-server använda något som kallas för objektpooler. Ett visst antal objekt av varje komponent skapas och placeras i en objektpool. När en klient anropar en metod i en komponent så tilldelar EJB-server ett objekt från poolen som kan exekvera koden åt klienten. Genom att endast skapa ett mindre antal med objekt så sparas resurser, främst minne. Allt detta är möjligt då klienten håller en referens till komponentens remote- (eller local-) gränssnitt och inte till bean-klassen (d.v.s. ett objekt) som exekverar koden.

Samtidighet (*concurrency*), eller synkronisering, är ytterligare ett område av programmering i fleranvändarsystem som är komplext och svårt att få rätt. Återigen är det något som en EJB-server kan hjälpa oss med.

### 10.3.2 Entity beans

En entity bean motsvarar alltså data som applikation ska behandla. De klasser i klassdiagram som innehåller, eller främst innehåller, tillstånd är våra entity beans, t.ex. Person, Dator och Bil. Affärslogiken i en entity beans består främst i accessmetoder (set- och get-metoder) för att läsa och sätta värden för attribut i EJB. Dessa affärsmetoder (accessmetoder) definieras i remote- och/eller local-gränssnitt.

Utöver metoder för affärslogiken så måste användare av EJB kunna skapa nya, eller hitta existerande, instanser av EJB. För detta används home-gränssnitt och för entity beans måste gränssnittet därför innehålla find- och create-metoder för att hitta respektive skapa instanser av entity bean (se mer nedan).

I de flesta fall vill vi använda oss av *container-managed persistence* (CMP), d.v.s. låta EJB-server hantera kommunikation med datakälla. Genom att använda CMP så slipper vi skriva SQL-satser för att läsa och uppdatera data i databaser samt vi slipper sköta transaktioner. Detta kan spara timmar med felsökning utöver den tid det tar att skriva koden för databaskommunikationen. I denna sammanfattning kommer alla EJB i exempel att använda CMP.

En entity beans bean-klass ska implementera (ärva från) gränssnittet `javax.ejb.EntityBean` (och en session beans bean-klass gränssnittet `SessionBean`). I detta gränssnitt finns metoder som EJB-servern anropar för att meddela EJB att något händer (främst att EJB:s tillstånd

ändras, t.ex. skapas eller förstörs).[20] Dessa metoder används främst vid *bean-managed persistence* (BMP) men beskrivs kort nedan.

## Metoder i gränssnittet EntityBean

Gränssnittet `javax.ejb.EntityBean` innehåller följande metoder:

- `ejbActivate()` – anropas innan objekt tas från objektpool.
- `ejbLoad()` – anropas innan objekts data läses från databas.
- `ejbPassivate()`– anropas innan objekt placeras i objektpool.
- `ejbRemove()` – anropas innan objekts raderas i både EJB-server och databas.
- `ejbStore()` – anropas innan objekts data skrivs till databas.
- `setEntityContext()` – anropas efter att objekt har skapats.
- `unsetEntityContext(EntityContext ctx)` – anropas innan objekt förstörs.

Ovanstående metoder **måste** implementeras av bean-klassen[21]! (Ett sätt att komma runt behovet av att implementera metoderna i varje EJB är att skapa en klass som ärver från EntityBean och implementerar metoderna samt sen låta alla EJB ärva från den nya klassen. Men om vi använder Resin så kan vi också använda Cauchos abstrakta klass `AbstractEntityBean` som gör just det. ☺) Och om BMP används så måste EJB-utvecklaren själv skriva kod för databasåtkomst, vilket bl.a. görs i metoderna `ejbLoad()` och `ejbStore()`.

## Find-metoder

Find-metoder används för att finna en eller flera instanser av entity beans. Varje entity bean **måste** innehålla en metod som heter `findByPrimaryKey()` med en parameter – typen på parametern är samma som primärnyckelklassen och typen på returvärdet ska vara EJB:s remote- (eller local-) gränssnitt. Denna metod används bl.a. av EJB-server för att hitta en viss instans av en entity bean, d.v.s. metoden returnerar alltid **en** instans av EJB. Metoden ska definieras i EJB:s home-gränssnitt, men implementeras i klass genererad av EJB-server. I EJB Person (med personnummer som primärnyckel, och därmed `String` som primärnyckelsklass) så skulle `findByPrimaryKey()` kunna definieras enligt följande:

```
Person findByPrimaryKey(String persnr);
```

Vill vi ha andra sätt att finna instanser av EJB så kan vi lägga till fler find-metoder. Dessa metoder måste ha prefixet `find` (men lämpligen `findBy`) och implementeras genom att lägga till EQL-satser[22] i deployment descriptor (mer om find-metoder och EQL i senare avsnitt). Dessa frivilliga find-metoder kan returnera en instans av EJB, eller en samling av EJB, som matchar ett visst kriterium – d.v.s. dessa metoder bör ha returtypen `Collection`. Vilket dessa kriterium är avgörs av vilka parametrar som skickas till metod, vi kan alltså ha en eller flera parametrar för metod. I EJB Person så kanske vi vill ha en metod för att hitta personer med ett visst efternamn samt ett visst för- och efternamn. Vi kan då definiera metoder med namn som `findByEfternamn()` och `findByForOchEfternamn()` med en respektive två strängar som parametrar till metoderna:

```
Collection findByEfternamn(String enamn);
```

[20] Denna typ av metoder kallas *callback*-metoder.
[21] Bean-klassen behöver endast innehålla tomma metoder, d.v.s. metoderna måste finnas men behöver **inte** innehålla någon kod.
[22] *Enterprise JavaBeans Query Language* – ett frågespråk som starkt påminner om SQL.

```
Collection findByForOchEfternamn(String fnamn, String enamn);
```

## Create-metoder

Om vi vill kunna skapa en ny instans av entity bean så måste vi även ha en eller flera create-metoder. Som parameter till create-metoder skickas värden för EJB:s attribut – minst en parameter för varje kolumn i tabell som har restriktionen `NOT NULL`.[23] Returtyp för create-metoder är (liksom `findByPrimaryKey()`) EJB:s remote-gränssnitt. Create-metoder definieras i EJB:s home-gränssnitt men implementeras av ejbCreate-metoder i EJB:s bean-klass (se nedan). Om vi i EJB Person vill kunna skapa en instans genom att skicka personnummer samt för- och efternamn (vilka bör var `NOT NULL` i databas ☺) så kan create-metod ha följande definition:

```
Person create(String persnr, String enamn, String fnamn);
```

För varje create-metod i home-gränssnittet så måste vi alltså definiera en ejbCreate-metod, men även ejbPostCreate-metod, i EJB:s bean-klass. Parametrar till ejbCreate- och ejbPostCreate-metoder måste vara av samma typ och antal som create-metod i home-gränssnitt. Men returtyp för ejbCreate-metoder ska vara primärnyckelsklassen och för ejbPostCreate-metoder `void`. I ejbCreate-metoder så anropar vi set-metoder för de attribut som metods parametrar motsvarar. ejbPostCreate-metoder innehåller främst implementation då EJB är relaterad till andra EJB (se kapitel *Relationer mellan EJB* för exempel på implementation av `ejbPostCreate()`).

I bean-klass för EJB Person så skulle vi kunna lägga till nedanstående ejbCreate- och ejbPostCreate-metoder för att matcha create-metod ovan:

```
String ejbCreate(String persnr, String enamn, String fnamn) {
  setPersonnummer(persnr);
  setEfternamn(enamn);
  setFornamn(fnamn);
}  //ejbCreate(String, String, String)

void ejbPostCreate(String persnr, String enamn, String fnamn) { //Ingen impl. }
```

För att kunna ta bort en instans av EJB så används (oftast[24]) metoden `remove()` i EJB:s remote-gränssnitt, d.v.s. vi behöver inte lägga till en egen remove-metod. **Observera** att när `remove()` anropas så tas även motsvarande post i databas bort!

## 10.3.3 Kompilera och testa EJB

Som nämnt tidigare så öka komplexiteten, både kodmässigt och miljön som koden ska exekvera i. När vi kompilerar kod så meddelar kompilator eventuella syntaxfel, men inte logiska fel i koden. Andra saker som kompilatorn ”missar” är avsaknad av en del metoder eller, enligt EJB-specifikationen (men inte enligt Java-syntax), felaktiga metoder. Dessa ”missar” är något som inte kompilatorn kan (eller ska?) fånga, utan de upptäcks först när EJB installeras i EJB-server eller instanser av EJB skapas och metoder i instans anropas. Ett

[23] Om en kolumn i tabell är av typen räknare så behövs ingen parameter för attributet i EJB – värdet genereras oftast när post i tabell skapas.

[24] Vi kan om vi vill även använda metoden `remove()` i EJB:s home-gränssnitt och bifoga instansens *handle* som parameter. En *handle* är ett unikt värde som EJB-server genererar för att kunna unikt identifiera instanser av EJB – EJB-server kan frågas om en *handle* för instans (för bl.a. detta syfte). Men eftersom vi antagligen oftast vill ta bort en instans av en EJB som vi har en referens till så är det lättare att anropa `remove()` i remote-gränssnittet. ☺

problem är att fel kan finnas i någon av de två/fyra gränssnitten, en/två klass/klasser eller deployment descriptor, d.v.s. det finns flera ställen att felsöka i.

### 10.3.4 Entity beans – exempel

Dags att titta på ett enkelt exempel på hur en implementation av en entity bean ser ut. Vi ska skapa en EJB Bil med attributen `regnr` (registreringsnummer) och `marke` (märke). För detta exempel krävs en tabell med följande utformning:

```
CREATE TABLE bilar(
  regnr char(6) PRIMARY KEY,
  marke varchar(25));
```

Koden i detta exempel bygger på att vi använder Resin (från Caucho) som EJB-server (men även servlet-server ☺). D.v.s. om en annan EJB-server används så behöver kanske en del justeringar göras för att exempel ska fungera. Resins konfigurationsfil `WEB.XML` måste (bl.a.) innehålla en datakälla (`jdbc/test` i exempel nedan) samt konfiguration för EJB-containers datakälla (`ejb` i exempel). Spara filen i mappen `WEB-INF` för webbapplikationen (eller lägg till i en redan existerande fil).

**Observera:** EJB-server behöver tillgång till databashanterarens drivrutiner. Dessa kan placeras i EJB-servers LIB-mapp (eller under webbapplikationens – `WEB-INF\lib`). Om flera jobbar i samma webbapplikation så måste ni ersätta paketnamnet i exempel (`bpn`) med t.ex. er egen användaridentitet i nätverket.

## Steg i exempel

1. Konfigurera datakälla (i t.ex. MySQL, punkt 1a, eller Oracle, punkt 1b). Denna punkt behöver endast göras en gång för varje webbapplikation.
2. Definiera remote-gränssnitt (eller local-gränssnitt eftersom vi ska testa EJB lokalt).
3. Definiera home-gränssnitt (eller local home-gränssnitt eftersom vi ska testa EJB lokalt).
4. Implementera bean-klass.
5. Skapa (eller ”hitta”) primärnyckelsklass.
6. Skapa *deployment descriptor*.
7. Skapa servlet-klass för att testa EJB.

## 1 (a). Konfigurera datakälla för MySQL

Att använda MySQL som databas är praktiskt då databashanteraren är gratis och relativt enkel att konfigurera, d.v.s. en databas ni kan ladda ner och installera hemma.[25] Nedan konfigureras en datakälla med JNDI-namnet `jdbc/test` (för just databasen `test` [26]i MySQL) samt datakälla för CMP med JNDI-namnet `ejb`.

```
<web-app>
  <!-- *** START: Konfiguration av databas ****************************** -->
  <database>
    <jndi-name>jdbc/test</jndi-name>
    <driver>
```

[25] Som klient mot MySQL kan man använda MySQL Front eller MySQL:s Administrator och Query Browser.
[26] När MySQL installeras så brukar just databasen `test` skapas.

```
        <type>org.gjt.mm.mysql.Driver</type>
        <url>jdbc:mysql://localhost:3306/test</url>
        <user>username</user>
        <password>password</password>
      </driver>
    </database>
    <!-- *** SLUT: Konfiguration av databas ******************************** -->

    <!-- *** START: Konfiguration databas for CMP ************************** -->
    <ejb-server jndi-name="ejb">
      <data-source>jdbc/test</data-source>
      <config-directory>WEB-INF</config-directory>
    </ejb-server>
    <!-- *** SLUT: Konfiguration databas for CMP ************************** -->
  </web-app>
```

## 1 (b). Konfigurera datakälla för Oracle

Oracle är en av de mest använda kommersiella databashanterare, därför är den val två som databas. Den enda skillnaden mellan konfiguration av datorkällor för MySQL och Oracle är drivrutinen (taggen `<type>`) och sökvägen (taggen `<url>`).

```
  <web-app>
    <!-- *** START: Konfiguration av databas ******************************** -->
    <database>
      <jndi-name>jdbc/test</jndi-name>
      <driver>
        <type>oracle.jdbc.driver.OracleDriver</type>
        <url>jdbc:oracle:thin:@julia:1521:orc1</url>
        <user>username</user>
        <password>password</password>
      </driver>
    </database>
    <!-- *** SLUT: Konfiguration av databas ******************************** -->

    <!-- *** START: Konfiguration databas for CMP ************************** -->
    <ejb-server jndi-name="ejb">
      <data-source>jdbc/test</data-source>
      <config-directory>WEB-INF</config-directory>
    </ejb-server>
    <!-- *** SLUT: Konfiguration databas for CMP ************************** -->
  </web-app>
```

## 2. Definiera remote-gränssnitt

Remote-gränssnittet (eller local-gränssnittet som i detta exempel) är gränssnittet med de publika (affärs-)metoderna i EJB. Detta gränssnitt motsvarar klasser i klassdiagram vi tar fram i analysfasen. Skillnaden mellan remote- och local-gränssnitt är vilket gränssnitt som vi utökar (`EJBObject` respektive `EJBLocalObject`) samt att metoder i local-gränssnitt **inte** behöver slänga `java.rmi.RemoteException`.

Eftersom detta är en entity bean så är våra affärsmetoder accessmetoder för EJB:s attribut, d.v.s. bilens registreringsnummer (`regnr`) och märke (`marke`). Eftersom vi inte vill att primärnyckeln för tabellen (`bilar` i detta fall) ska kunna ändras så definierar vi inte en set-metoden för primärnyckelattributet (d.v.s. metoden `setRegnr()`).

```
package bpn.j2eeexempel;          //Paketnamn - ersätt med egen användaridentitet

import javax.ejb.EJBLocalObject;

public interface Bil extends EJBLocalObject
{
  public String getRegnr();
  public String getMarke();
  public void setMarke(String marke);
```

```
} //interface Bil
```

## 3. Definiera home-gränssnitt

Home-gränssnittet använder vi för att styra en EJB:s livscykel, bl.a. skapande av nya instanser. För att skapa nya entity beans så använder vi create-metoder. Create-metoder används istället för det reserverade ordet `new`, och därmed konstruktorer, för att avgöra hur en entity bean ska skapas. (Vi får för övrigt **inte** ha några konstruktorer i bean-klassen!) Hur många create-metoder som entity bean ska innehålla samt vilka parametrar som create-metoder ska ha beror på hur vi vill kunna skapa instanser av entity bean. Det är viktigt att komma ihåg att när vi skapar en ny instans av entity bean med create-metoder, så skapas även en motsvarande post i tabell (i relationsdatabas).

Om vi vill använda redan existerande instanser av entity bean (eller snarare poster i tabell) så används find-metoder. Home-gränssnittet behöver innehålla en find-metod för varje sätt vi vill ”hitta” en entity bean. Parameter (eller parametrar) till find-metoder används i WHERE-klausul i EQL-satser (mer om dessa senare). En entity bean måste ha minst en find-metod, `findByPrimaryKey()`, som har primärnyckelklassen som parametertyp.

Eftersom vi ska använda ett local-gränssnitt (för affärslogiken) så måste vi skapa ett motsvarande local home-gränssnitt. Alla create-metoder måste även slänga `javax.ejb.CreateException` liksom find-metoder slänger `javax.ejb.FinderException`. Men eftersom det är ett local home-gränssnitt (och inte ett ”remote” home-gränssnitt) så behöver dessa metoder **inte** slänga `java.rmi.RemoteException`.

```
package bpn.j2eeexempel;          //Paketnamn - ersätt med egen användaridentitet

import javax.ejb.EJBLocalHome;
import javax.ejb.FinderException;
import javax.ejb.CreateException;

public interface BilHome extends EJBLocalHome
{
  Bil create() throws CreateException;
  Bil create(String regnr, String marke) throws CreateException;

  Bil findByPrimaryKey(String regnr) throws FinderException;
}  //interface BilHome
```

## 4. Implementera bean-klass

Bean-klassen “implementerar” metoderna i remote- och home-gränssnitten (eller local- och local home-gränssnitt). Varför jag satt implementera inom citattecken beror på att bean-klassen inte implementerar (*implements*) gränssnitten utan endast metoder med samma (eller liknande namn[27]) som i gränssnitten.

Tabellen innehåller två attribut – två attribut som vi vill använda – och därför behövs accessmetoder (en set- och en get-metod) för de två attributen.[28] I.o.m. EJB v.2.0 så ska dessa metoder vara abstrakta (och därmed även klassen).[29] Klassen måste innehålla både set- och get-metod (för att göra ett attribut endast läsbart för klient så definierar vi inte en set-metod i remote-gränssnittet, enligt ovan).

[27] Med liknande menar jag att home-gränssnitt innehåller create-metoder (t.ex. `create()`) medan bean-klassen innehåller ejbCreate-metoder (d.v.s. `ejbCreate()`). Därav liknande namn…
[28] Vi behöver inte använda alla attribut i en tabell i en EJB om vi inte vill.
[29] Metoderna implementeras av klasser som EJB-container genererar automatiskt.

Med två create-metoder i home-gränssnitt (`create()` och `create(String, String)`) så måste vi skapa två ejbCreate-metoder (med samma antal och typ på parametrar) i bean-klassen. Metodernas returtyp ska vara primärnyckelsklassen, d.v.s. `String` i detta exempel. I den första ejbCreate-metoden så returnerar vi bara `null`.[30] Men i den andra metoden anropar vi även set-metoderna för de två attributen och skickar vidare create-metodens parametrar.

För varje ejbCreate-metod så måste vi ha en motsvarande ejbPostCreate-metod. ejbPostCreate-metoderna ska ha samma (typ på och antal) parametrar som ejbCreate-metod samt ha returtyp `void`. Dessa metoder används främst när vi har relationer (se nästa kapitel) eller om vi använder bean managed persistence.

Vi behöver inte skapa några motsvarande metoder till find-metoder i home-gränssnittet.[31]

Sist av allt implementerar vi metoderna i gränssnittet `EntityBean`. Metoderna behöver inte innehålla någon kod men måste finnas i klassen eftersom de finns i gränssnittet EntityBean som klassen implementerar.

```
package bpn.j2eeexempel;          //Paketnamn - ersätt med egen användaridentitet

import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.CreateException;

public abstract class BilBean implements EntityBean
{
  /*** Accessmetoder för attribut i tabell *********************************/
  public abstract void setRegnr(String regnr);
  public abstract String getRegnr();
  public abstract void setMarke(String marke);
  public abstract String getMarke();

  /*** ejbCreate- och ejbPostCreate-metoder ********************************/
  public String ejbCreate() throws CreateException
  {
    return null;
  }
  public void ejbPostCreate() {}

  public String ejbCreate(String regnr, String marke) throws CreateException
  {
    setRegnr(regnr);
    setMarke(marke);
    return null;
  }
  public void ejbPostCreate(String regnr, String marke) {}

  /*** Implementera gränssnittet EntityBean ********************************/
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void ejbStore() {}
  public void ejbLoad() {}
  public void setEntityContext(EntityContext sc) {}
  public void unsetEntityContext() {}
}  //class BilBean
```

## 5. Skapa (eller ”hitta”) primärnyckelsklass

För detta exempel så är skapande av primärnyckelsklassen enkelt – klassen String finns redan. ☺

[30] För att vara kompatibel med EJB v. 1.x så ska Create-metoder returnera primärnyckelklassen. I.o.m. EJB v. 2.0 så behöver inte primärnyckeln returneras utan vi returnerar bara null (eftersom vi måste returnera något…).
[31] Vi använder EJB Query Language (EQL) i *deployment descriptor* för detta, utom för `findByPrimaryKey()` som EJB-server skapar åt oss automatiskt.

## 6. Skapa deployment descriptor

Sist av allt så binder vi samman de två gränssnitten, bean-klassen och vår primärnyckelsklass med en deployment descriptor. Vi kan använda samma deployment descriptor för flera EJB:er (något som vi bör göra när vi har relationer mellan EJB).

Eftersom vår EJB är en entity bean så innesluter vi beskrivningen av EJB med taggen `<entity>`. I beskrivningen av en entity bean behöver vi (bl.a.) ett namn (som vi kan referera till EJB med, d.v.s. JNDI-namn), namn på gränssnitt och klasser (bean- och primärnyckelsklass), namn på tabell samt attribut i tabell (utöver primärnyckel).

För att namnge EJB används taggen `<ejb-name>`. Sen använder vi taggarna `<local>`, `<local-home>` och `<ejb-class>` för att ange local- och local home-gränssnitt respektive bean-klass. För att använda CMP så sätter vi värdet Container i taggen `<persistence-type>` (om vi vill använda BMP så använder vi `Bean` som värde). (Taggen `<reentrant>` används för att tillåta EJB att anropa metoder i sig själv.) Taggarna `<sql-table>`, `<prim-key-class>` och `<primkey-field>` används för att ange tabell, klass på primärnyckeln samt namnet på primärnyckelattributet i EJB. Sist av allt använder vi taggarna `<cmp-field>` och `<field-name>` för att ange attribut i tabellen (som inte är en del av primärnyckeln) som vi vill använda.

Glöm **inte** att ersätta paketnamnet `bpn` (tre ställen) om ni ändrat i gränssnitt och klasser ovan!

```
<ejb-jar>
  <enterprise-beans>

    <entity>
      <ejb-name>Bil</ejb-name>

      <local>bpn.j2eeexempel.Bil</local>
      <local-home>bpn.j2eeexempel.BilHome</local-home>
      <ejb-class>bpn.j2eeexempel.BilBean</ejb-class>

      <persistence-type>Container</persistence-type>
      <reentrant>True</reentrant>
      <sql-table>bilar</sql-table>

      <prim-key-class>String</prim-key-class>
      <primkey-field>regnr</primkey-field>

      <cmp-field><field-name>marke</field-name></cmp-field>
    </entity>

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

## 7. Skapa servlet-klass

För att testa EJB i Resin så används lämpligen en servlet. Skälet till detta är att EJB- och servlet-container är den samma i Resin, d.v.s. vi får lokal åtkomst och kan använda lokala gränssnitt (*local* och *local home*)

```
package bpn.j2eeexempel; //Paket som klass ska finnas i (samma som EJB f lätthet)

import javax.servlet.*;       //Importera paket för servlets
import javax.servlet.http.*;
import java.io.*;             //Importera paket för PrintWriter + IOException
import javax.naming.*;        //Importera paket för JNDI
import javax.ejb.*;           //Importera paket för EJB

public class BilServlet extends HttpServlet
```

```
{
    /*** Instansvariabler ************************************************/
  private BilHome home = null;  //Det behövs endast en referens till home-gr.snitt


    /*** Överskugga metoder i HttpServlet ********************************/
    //Metod som anropas när servlet-klass laddas (dvs endast en gång)
  public void init() throws ServletException
  {
    try
    {
        //Hämta referens till initialt kontext - använd lokal namntjänst (Resins)
      Context initial =
        (Context) new InitialContext().lookup("java:comp/env/ejb");

        //Hämta referens till Bils home-gränssnitt
      home = (BilHome) initial.lookup("Bil");
    }
    catch(NamingException e)
    {
      throw new ServletException(e);  //Kapsla in fel i ServletExeption
    }
  }  //init()


    //Metod som anropas om servlets begärs med HTTP GET
  public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws IOException
  {
    PrintWriter out = res.getWriter();
    res.setContentType("text/html");

      //Skriv ut rubrik för hemsida
    out.println("<h1>Märke på bil med regnr abc123</h1>");

    try
    {
        //Skapa en instans av EJB och därmed post i tabell
        //Ta bort kommentar på rad nedan för att testa create-metod.
        // Om fel {Invalid argument value,  message from
        // server: "Duplicate entry 'ghi789' ... } visas så finns post redan.
      //Bil bil = home.create("ghi789", "BMW 525");

        //Hämta bil med regnr "abc123" och skriv ut märke
      Bil bil = home.findByPrimaryKey("abc123");
      out.println(bil.getMarke());
    }
    catch(Exception e)
    {
      out.println("<PRE>");
      e.printStackTrace(out);
      out.println("</PRE>");
    }
  }  //doGet()


    //Metod som anropas om servlets begärs med HTTP POST
  public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws IOException
  {
    doGet(req, res);  //Anropa metod doGet() ovan
  }  //doPost()

}  //class BilServlet
```

### 10.3.5 Entity beans – exempel med sammansatt primärnyckel

Entity beans som motsvarar poster i databas där primärnyckel är sammansatt, d.v.s. består av två eller fler kolumner, så måste vi skapa en egen primärnyckelsklass. Klassen används bl.a. av J2EE-server för att CMP ska fungera. I detta exempel behandlas en EJB Stereoartikel – delar i en komplett stereo (förstärkare, högtalare, CD-spelare, m.m.). Primärnyckeln består av märke på del (t.ex. Technics eller Sony) samt modell (t.ex. SLP-92).

För detta exempel krävs en tabell med följande utformning:

```
CREATE TABLE stereoartiklar(
  marke varchar(25),
  modell varchar(25),
  beskrivning varchar(150),
  CONSTRAINT PRIMARY KEY(marke, modell));
```

## 1. Konfigurera datakälla

Samma som för exempel ovan, d.v.s. om det redan är gjort så är detta steg överflödigt. ☺

## 2. Definiera local-gränssnitt

Eftersom EJB ska användas lokalt, d.v.s. av servlets i samma container, så skapar vi ett local-gränssnitt istället för remote-gränssnitt. Gränssnittet ska, som sagt, innehålla affärsmetoder, accessmetoder i detta fall eftersom entity bean. Eftersom inte primärnyckel inte får ändras så definieras endast get-metoder för attribut i den (`marke` och `modell`) samt både get- och set-metoder för övriga attribut (`beskrivning`).

```
package bpn.j2eeexempel;

import javax.ejb.EJBLocalObject;

public interface Stereoartikel extends EJBLocalObject
{
  public String getMarke();
  public String getModell();
  public String getBeskrivning();
  public void setBeskrivning(String beskrivning);
}  //interface Stereoartikel
```

Hade gränssnittet varit ett remote-gränssnitt (med t.ex. namnet StereoartikelRemote) så hade vi behövt importera undantaget `java.rmi.RemoteException` och klassen EJBObject (istället för EJBLocalObject). Gränssnittet skulle utöka EJBObject (istället) samt alla metoder måste slänga undantaget RemoteException.

## 3. Definiera (local) home-gränssnitt

I home-gränssnitt definieras två create-metoder och två find-metoder. Create-metoder har parameter för obligatoriska kolumner i tabell respektive alla kolumner i tabell. Här är det alltså inte någon skillnad (eller behöver inte vara ☺) om vi använder en sammansatt primärnyckel eller inte. Find-metoder är den obligatoriska `findByPrimaryKey()`, med primärnyckelsklass som parameter, och en `findAll()` för att kunna hämta alla instanser av EJB. Denna senare metod är främst för att kunna testa EJB, d.v.s. find-metoder bör ha parametrar för att hitta en begränsad mängd av instanser.

```
package bpn.j2eeexempel;

import java.util.Collection;
import javax.ejb.CreateException;
import javax.ejb.EJBLocalHome;
import javax.ejb.FinderException;

public interface StereoartikelHome extends EJBLocalHome
{
  /*** create-metoder ****************************************************/
  public Stereoartikel create(String marke, String modell)
```

```
      throws CreateException;
    public Stereoartikel create(String marke, String modell, String beskrivning)
      throws CreateException;

    /*** find-metoder *****************************************************/
    public Stereoartikel findByPrimaryKey(StereoartikelPK pk)
      throws FinderException;
    public Collection findAll() throws FinderException;
  }  //interface StereoartikelHome
```

*Observera att vi inte kan kompilera gränssnittet ovan förrän vi skapat primärnyckelsklassen nedan!*

Återigen, om gränssnittet hade varit ett remote home-gränssnitt (med t.ex. namnet StereoartikelHomeRemote) så hade vi behövt importera `java.rmi.RemoteException` och EJBHome (istället för EJBLocalHome). Gränssnittet skulle utöka EJBHome (istället) samt alla metoder slänga undantaget RemoteException. Alla create- och find-metoder skulle returnera remote-gränssnitt istället för local-gränssnitt.

## 4. Implementera bean-klass

Bean-klassen i exempel nedan ärver från klassen AbstractEntityBean som följer med Resin. Genom att ärva från denna klass så slipper vi implementera metoderna i gränssnittet EntityBean (enligt kommentar i kod nedan). Om exempel körs i en annan J2EE-server så måste bean-klassen implementera gränssnittet EntityBean istället.

```
package bpn.j2eeexempel;

import javax.ejb.CreateException;
import com.caucho.ejb.AbstractEntityBean;

public abstract class StereoartikelBean extends AbstractEntityBean
{
  /*** Accessmetoder för CMP-attribut ***********************************/
  public abstract String getMarke();
  public abstract void setMarke(String marke);
  public abstract String getModell();
  public abstract void setModell(String modell);
  public abstract String getBeskrivning();
  public abstract void setBeskrivning(String beskrivning);

  /*** ejbCreate- och ejbPostCreate-metoder *****************************/
  public StereoartikelPK ejbCreate(String marke, String modell)
    throws CreateException
  {
    setMarke(marke);
    setModell(modell);
    return null;
  }  //ejbCreate(String, String)

  public void ejbPostCreate(String marke, String modell) { }

  public StereoartikelPK ejbCreate(String marke, String modell,
    String beskrivning) throws CreateException
  {
    setMarke(marke);
    setModell(modell);
    setBeskrivning(beskrivning);
    return null;
  }  //ejbCreate(String, String, String)

  public void ejbPostCreate(String marke, String modell, String beskrivning) { }

  //Metoder i gränssnitt EntityBean implementeras av klassen AbstractEntityBean

}  //class StereoartikelBean
```

Även om vi använder remote- och (remote) home-gränssnitt så är bean-klass opåverkad.

## 5. Skapa (eller ”hitta”) primärnyckelsklass

Primärnyckelsklasser är mycket simpla klasser: attribut, konstruktorer och metoder. Klassen ska även implementera gränssnittet `java.io.Serializable` – ett gränssnitt som klasser implementerar för att t.ex. kunna skrivas till disk.[32] Primärnyckelsklasser ska innehålla ett attribut för varje kolumn i tabells primärnyckel. Attributen ska implementeras som publika instansvariabler, d.v.s. åtkomst ska inte ske med accessmetoder. Klasserna ska även innehålla en ”standardkonstruktor”, d.v.s. en konstruktor utan parametrar, och lämpligen även konstruktor med parametrar för respektive attribut i klassen.

En primärnyckelsklass bör även överskugga minst två metoder från klassen Object:

- `equals()` – för jämförelse mellan två instanser av klassen.
- `hashCode()` – returnerar ett värde som är unikt (*hash code*) för instans baserat på värden för primärnyckeln i tabell.

I `equals()` testar vi först att objekt i parameter (d.v.s. objektet vi ska jämföra med objektet som exekverar koden – ”aktuellt” objekt) är av samma typ som ”aktuellt” objekt, vilket görs med det reserverade ordet `instanceof`. Om så är fallet så kan vi konvertera parametern till rätt typ. Därefter kan vi jämföra om värdet på respektive objekts attribut är lika – ett resultat som returneras från metoden. *Hash code* är ett värde som är tänkt[33] att vara unikt och som beräknas med någon form av formel. I klassen nedan så XOR:as[34] (med operatorn `^`) *hash*-värdet från de två strängarna i attributen i metoden `hashCode()`.

```
package bpn.j2eeexempel;

import java.io.Serializable;

public class StereoartikelPK implements Serializable
{
  /*** Publika instansvariabler för attribut ********************************/
  public String marke;
  public String modell;

  /*** KOnstruktorer *******************************************************/
  public StereoartikelPK() { }         //Standard-konstruktor - obligatorisk

  public StereoartikelPK(String marke, String modell)
  {
    this.marke = marke;
    this.modell = modell;
  }  //StereoartikelPK()

  /*** Överskuggade metoder från Object ************************************/
  public boolean equals(Object obj)
  {
    if(!(obj instanceof StereoartikelPK)) return false;

    StereoartikelPK sa = (StereoartikelPK)obj;

    if(this.marke.equals(sa.marke) && this.modell.equals(sa.modell))
```

[32] *Serializing* innebär att något ”skrivs” seriellt, d.v.s. efter varandra. Detta behövs för att bl.a. skriva objekt till en fil eller för att skickas över nätverk.

[33] Med en bra formel så ska *hash*-värden vara unika, men ibland så händer det att *hash*-värden för två objekt ändå inte blir unika.

[34] Operationen XOR är en logisk operation som ”adderar” två värden baserat på bitar i värdena, d.v.s. dess 1:or och 0:or. Om båda bitarna på position x är 0 så blir ”summan” en 0:a, om båda bitarna är 1 så blir ”summan” en 0:a eller om en av bitarna är 1 så blir ”summan” en 1:a.

```
        return true;

      return false;
    }  //equals()


    public int hashCode()
    {
      return marke.hashCode() ^ modell.hashCode();
    }  //hashCode()

  }  //class StereoartikelPK
```

## 6. Skapa deployment descriptor

I deployment descriptor namnger vi EJB samt vilka gränssnitt och klasser EJB består av. Som primärnyckelklass anger vi egenskapade klass. Sen anger vi namn på tabell, dess logiska namn samt kolumner som inte ingår i primärnyckel. Sist av allt definierar vi EQL för metoden `findAll()`.

```
<ejb-jar>

  <!--  ***************** EJBer ******************** -->
  <enterprise-beans>

    <!--  ***************** Stereoartikel ***************** -->
    <entity>
      <ejb-name>Stereoartikel</ejb-name>
      <local-home>bpn.j2eeexempel.StereoartikelHome</local-home>
      <local>bpn.j2eeexempel.Stereoartikel</local>
      <ejb-class>bpn.j2eeexempel.StereoartikelBean</ejb-class>

      <prim-key-class>bpn.j2eeexempel.StereoartikelPK</prim-key-class>

      <persistence-type>Container</persistence-type>
      <reentrant>True</reentrant>

      <abstract-schema-name>stereoartiklar</abstract-schema-name>
      <sql-table>stereoartiklar</sql-table>

      <cmp-field><field-name>beskrivning</field-name></cmp-field>

      <query>
        <query-method>
          <method-name>findAll</method-name>
        </query-method>
        <ejb-ql>SELECT o FROM stereoartiklar o</ejb-ql>
      </query>
    </entity>

  </enterprise-beans>

</ejb-jar>
```

## 7. Skapa servlet-klass

Servlet placeras i ett paket (lämpligen ersätts namnet på paketet `bpn` med egen användaridentitet i nätverk) – samma som EJB för lätthetens skulle. Sen importeras klasser och bibliotek som används i servlet.

Först i klassen deklareras en variabel för att hålla referensen till EJB:s home-gränssnitt. Referensen till home-gränssnittet hämtas sen i init-metoden (precis som tidigare och kommande klienter).

I `doGet()` hämtas referens till PrintWriter (för utskrift) och webbsida ”påbörjas”. Variabeln artiklar deklareras för att hålla vektor med EJB som metoden `findAll()` bör returnera.[35] Sen testas metoderna `findByPrimaryKey()` (och dess EJB:s innehåll skrivs ut) samt `findAll()`. För att anropa metoden `findByPrimaryKey()` så måste vi först skapa en instans av primärnyckelklassen (StereoartikelPK), vars konstruktor vi bifogar värden för dess attribut. Resultatet från `findAll()` (d.v.s. vektorn) skrivs ut i nästa try-catch-block och sist ”avslutas” webbsida.

```
package bpn.j2eeexempel;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Collection;
import java.util.Iterator;
import javax.ejb.FinderException;
import javax.naming.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class StereoartikelServlet extends HttpServlet {
  /*** Instansvariabler ***************************************************/
  private StereoartikelHome home = null;

  /*** Överskuggade metoder ***********************************************/
  public void init() throws ServletException {
    try {
      Context context =
        (Context)new InitialContext().lookup("java:comp/env/ejb");

      home = (StereoartikelHome)context.lookup("Stereoartikel");
    }
    catch(NamingException ne) {
      throw new ServletException(ne);
    }
  } //init()

  public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    out.println("<html>\n<head>\n<title>Stereoartikel</title>\n</head>");
    out.println("<body><h1>Stereoartikel</h1>");

    Collection artiklar = null;

    try {
      StereoartikelPK artPK = new StereoartikelPK("Technics", "ML-540");
      Stereoartikel art = home.findByPrimaryKey(artPK);

      out.println("<p>Beskrivning för stereoartikel av märke "
        + artPK.marke + " och modell " + artPK.modell + ": "
        + art.getBeskrivning()+ ".</p>");

      artiklar = home.findAll();
    }
    catch(FinderException ce) {
      out.println("<pre>");
      ce.printStackTrace(out);
      out.println("</pre>");
    }

    out.println("<p>Alla stereoartiklar:<br>");

    Iterator iter = artiklar.iterator();

    while(iter.hasNext()) {
```

[35] Variabeln deklareras utanför try-catch-block då dess innehåll hämtas i ett block men används i ett annat.

```
        Stereoartikel art = (Stereoartikel)iter.next();
        out.println(art.getMarke() + " " + art.getModell() + " "
          + art.getBeskrivning() + "<br>");
      }

      out.println("<p><a href=\"../j2eeexempel.html\">Tillbaka</a></p>");
      out.println("</body>\n</html>");
    }  //doGet()
  }  //class StereoartikelServlet
```

## 10.3.6 Entity beans och EQL [ UTVECKLA ]

## Placering av EQL [ UTVECKLA ]

EQL placeras i entity-taggen i deployment descriptor. I sin enklaste form används nedanstående fyra taggar:

- `<query>` – innesluter alla övriga taggar för EQL.
- `<query-method>` – innehåller tagg för frågemetod (och eventuella parametrar – se *Villkor för att välja ut EJB* nedan).
- `<method-name>` – innehåller namn på fråga, måste vara samma som i home-gränssnitt.
- `<ejb-ql>` – innehåller själva EQL-satsen.

```
<entity>
  <ejb-name>Stereoartikel</ejb-name>
    ...
  <abstract-schema-name>stereoartiklar</abstract-schema-name>
    ...
  <query>
    <query-method>
      <method-name>findAll</method-name>
    </query-method>
    <ejb-ql>SELECT o FROM stereoartiklar o</ejb-ql>
  </query>
</entity>
```

EQL påminner (medvetet) om SQL, dock med några skillnader (lite annorlunda och några saker som saknas – mer om detta nedan). I EQL-satsen ovan hämta hela EJB:er (`SELECT o`) från den abstrakta ”tabellen” (*abstract schema name* – i taggen `<abstract-schema-name>`).

## Frågemetoder

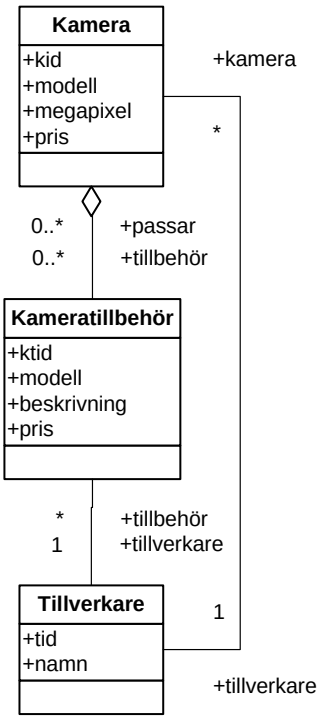
Entity beans måste som sagt innehålla en find-metod: `findByPrimaryKey()`. Utöver denna obligatoriska metod så bör en entity bean innehålla en find-metod för varje sätt vi vill kunna hitta instanser av EJB. Find-metoder definieras i home-gränssnitt, d.v.s. är publika, och kan användas (direkt) av klienter och ska retur. Returvärden från find-metoder kan vara remote-/local-gränssnitt, samlingar eller typer/klasser som kan ”beständigas” (*be serializable*).

Entity beans kan även innehålla select-metoder, metoder som till stor del påminner om find-metoder. Skillnaden mot find-metoder är dock att select-metoder definieras i bean-klass och endast kan användas från bean-klass. Det finns dock inget som säger att man inte kan definiera en affärsmetod (i remote-/local-gränssnitt) som anropar en select-metod och returnerar dess resultat. Returvärden från en select-metod kan vara samlingar av typen Collection eller Set, remote-/local-gränssnitt eller vilka typer/klasser som helst (metoder kan ju endast anropas från bean-klass ☺).

```
public class StereoartikelBean implementes javax.ejb.EntityBean {
  ...
  public abstract String ejbSelectXxxx(Xxxx xxx) throws FinderException;
  ...
}  //class StereoartikelBean
```

## Sökvägar och villkor

**Sökvägar** (*paths*) används för att skapa frågemetoder som använder data från andra EJB än EJB med frågemetod. Om vi t.ex. vill hämta alla kameror från en tillverkare med ett visst namn (se högra relationen i klassdiagram till höger) så skulle EQL-satsen kunna se ut något liknande den nedan.




```
<ejb-ql>SELECT k FROM kameror k
  WHERE k.tillverkare.namn = 'Canon'</ejb-ql>
```

EQL-satsen ovan (`k.tillverkare.namn`) betyder ”hämta alla kameror där kameras tillverkares namn är Canon”. I detta fall används relationen mellan Kamera och Tillverkare.

Ovanstående EQL-sats visar även hur vi kan använda villkor för att söka fram EJB:er baserat på ett villkor. Ett **villkor** begränsar vilka EJB som ska hämtas och för detta används en WHERE-klausul, ungefär som i SQL. Men för att göra EQL-satsen mer dynamisk, d.v.s. att inte hårdkoda värden att söka efter, kan vi även lägga till parametrar i EQL-satsen genom att lägga till ett frågetecken och nummer på parameter. Numret på parametern ska motsvara argumentet i frågemetodens signatur.[36]

För varje parameter lägger vi till en tagg `<method-param>`, inneslutna i taggen `<method-params>`, med typ på parameter. Återigen ska typen motsvara frågemetodens argument.

```
<query>
  <query-method>
    <method-name>findByTillverkare</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </query-method>
  <ejb-ql>SELECT k FROM kameror k WHERE k.tillverkare.namn = ?1</ejb-ql>
</query>
```

Sökvägar kan även användas för att hämta värden på en eller flera EJB:ers attribut, men då bara i select-metoder.

## Använda relationsattribut som returnerar samlingar [ UTVECKLA ]

I relationsmodellen (d.v.s. databaser ☺) så använder vi relationer och join-villkor för att sammanfoga poster från en tabell med relaterade poster i en annan tabell. Men med EJB så har vi ”bara” tillgång till EJB:er, deras abstrakta ”tabell” och deras relationer till andra EJB. För att kunna ”navigera” dessa relationer används det reserverade ordet `IN()` i FROM-klausulen (istället för ytterligare abstrakta ”tabeller”).

[36] Med början på 1 för första argumentet.

```
<ejb-ql>SELECT DISTINCT k FROM kameror k, IN( k.tillbehor) AS t
  WHERE t.tillverkare.namn = ?1</ejb-ql>
```

Exempel ovan visar även hur vi kan använda det reserverade ordet `DISTINCT`, som i SQL, för att sortera bort dubbletter.

## Använda jämförelseoperatorer i WHERE-klausul [ UTVECKLA ]

Om vi vill använda jämförelseoperatorer för större än (>) och mindre än (<) i WHERE-klausul så uppstår problem då XML-taggar bygger på dessa. Så när vi ska använda dem så måste vi innesluta EQL-satsen i ”`<![CDATA[`” och ”`]]>`”.

```
<ejb-ql><![CDATA[SELECT DISTINCT k FROM kameror k, IN( k.tillbehor) AS t
  WHERE t.pris <= ?1]]></ejb-ql>
```

## Restriktioner på EQL [ UTVECKLA ]

- Kommentarer är inte tillåtna.
- Tid och datum måste jämföras som typen Long (d.v.s. i millisekunder).
- Jämförelse av två olika typer av EJB fungerar inte.

### 10.3.7 Session beans

När vi använder EJB, och därmed n-skiktade lösningar, så är det viktigt att separera affärslogik från presentationslogik. Presentationslogik är den logik som krävs för att kunna presentera data för, och hämta data från, användaren av applikationen. Affärslogiken är den logik som är ”själva” exekveringen av applikationen. (Ibland är det dock svårt att skilja på dessa två typer av logik och ibland, bl.a. för prestanda i applikation, så behöver viss logik dubbleras i EJB och presentationslager, t.ex. verifiering av datas riktighet.) Affärslogiken bör placeras i EJB av typen session beans.

Namnet på denna typ av EJB är tänkt att visa på hur EJB ska användas: under en klients session mot EJB. D.v.s. så länge som klientapplikation (presentationslager) exekverar och är ansluten till EJB-server (”session” är aktiv) så ska session beans existera. Ett annat sätt att se på session beans är att de är en förlängning av klienten, d.v.s. utför exekvering åt klienten. I andra avseenden motsvarar session beans fasadobjekt i ”vanlig” objektorientering.

Det finns två typer av session beans: *stateful* och *stateless*. En ***stateful session bean*** behåller sitt (interna) tillstånd (d.v.s. värden på instansvariabler) mellan klients metodanrop. Ett exempel på denna typ av session bean är Resebyråanställd – en anställd utför en reservation åt kund och behöver hålla reda på kunden och vad kunden bokar. Och eftersom en reservation kan gälla flera saker (t.ex. passagerare, hytter och bilar) så måste denna EJB hålla reda på dessa saker (som antagligen läggs till en sak i taget, d.v.s. med flera metodanrop).

***Stateless session beans*** gör motsatsen, d.v.s. de behåller inte sitt tillstånd mellan metodanrop (och har därmed inget behov av instansvariabler). Denna typ av session bean brukar främst innehålla metoder som gör någon form av beräkningar och returnerar resultatet direkt.

## Create-metoder

En session bean måste ha en create-metod, dock utan parametrar. Och precis som med entity beans så ska returtypen från metoderna vara EJB:s remote-gränssnitt.

Även session beans har metoden `remove()` i remote-gränssnittet. Skillnaden mot entity beans är dock att inga data raderas i databas när metod anropas. ☺

## Find-metoder

Session beans ska **inte** ha någon find-metod då den inte behöver hittas, d.v.s. de skapas alltid.

## Metoder i gränssnittet SessionBean

Bean-klassen för en session bean implementerar gränssnittet `javax.ejb.SessionBean` som innehåller följande metoder (som måste implementeras av bean-klassen, om än utan kod i):

- `ejbActivate()` – anropas innan objekt aktiveras.
- `ejbPassivate()` – anropas innan objekt deaktiveras.
- `ejbRemove()` – anropas innan objekt förstörs.
- `setSessionContext(SessionContext ctx)` – anropas efter att objekt skapats.

## 10.3.8 Session beans – exempel med stateful session bean

Exempel nedan bygger på en enkel EJB Raknare som innehåller två metoder:

- `antal()` – returnera antal gånger som metod har anropats (om inte nedanstående metod har anropats).
- `nollstall()` – nollställer räknare.

*Detta exempel bygger på att vi använder Resin (från Caucho) som EJB-server (men även servlet-server ☺). D.v.s. om en annan EJB-server används så behöver antagligen en del justeringar göras för att exempel ska fungera.*

## Steg i exempel

Att skapa en session bean är ”lättare” än en entity bean då vi inte behöver kontakt med en datakälla och därmed inte behöver en primärnyckelklass heller.

1. Definiera remote-gränssnitt (eller local-gränssnitt eftersom vi ska testa EJB lokalt).
2. Definiera home-gränssnitt (eller local home-gränssnitt eftersom vi ska testa EJB lokalt).
3. Implementera bean-klass.
4. Skapa *deployment descriptor*.
5. Skapa servlet-klass för att testa EJB.

## 1. Definiera local-gränssnitt

Local-gränssnittet ska innehålla affärsmetoderna för EJB – `antal()` och `nollstall()` i detta fall. Gränssnittet ökar EJBLocalObject (precis som entity beans).

```
package bpn.j2eeexempel;
import javax.ejb.EJBLocalObject;

public interface Raknare extends EJBLocalObject {
  public int antal();       //Antal gånger metod anropats
```

```
    public void nollstall();  //Nollställ räknare
}  //interface Raknare
```

Om EJB ska användas utanför container (t.ex. av en Java-klient) så måste vi även (eller istället) definiera ett remote-gränssnitt. Gränssnittet ska då utöka EJBObject (precis som entity beans) och alla metoder måste slänga undantaget RemoteException.

## 2. Definiera home-gränssnitt

Home-gränssnittet för en session bean behöver endast innehålla create-metoder (och får inte innehålla några find-metoder). I detta exempel så finns endast en create-metod (även om vi skulle kunna använda en parameter för startvärde på räknare) och metodens returtyp är local-gränssnittet.

```
package bpn.j2eeexempel;
import javax.ejb.EJBLocalHome;
import javax.ejb.CreateException;

public interface RaknareHome extends EJBLocalHome {
       Raknare create() throws CreateException;
}  //interface RaknareHome
```

Om EJB definierar ett remote-gränssnitt så måste den även definiera ett remote home-gränssnitt. Gränssnittet utökar då EJBHome och metoder slänger även undantaget RemoteException (som entity beans).

## 3. Implementera bean-klass

Bean-klasser för session beans ska implementerar gränssnittet SessionBean (istället för EntityBean) och inte vara abstrakta (som entity beans). Klassen ska innehålla minst en ejbCreate-metod (men ingen `ejbPostCreate()`), vars returtyp är `void` (och inte primärnyckelsklass eftersom session beans inte har en sådan ☺).

```
package bpn.j2eeexempel;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.CreateException;

public class RaknareBean implements SessionBean {
  /*** Instansvariabler ****************************************************/
  private int raknare;

  /*** Create-metoder ******************************************************/
  public void ejbCreate() throws CreateException {
    raknare = 0;
  }

  /*** Implementation av metoder i local-gränssnitt ************************/
  public int antal() {
    return ++raknare;           //Öka på räknare och returnera dess värde
  }

  public void nollstall() {
    raknare = 0;               //Nollställ räknare
  }

  /*** Implementera gränssnittet SessionBean *******************************/
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void setSessionContext(SessionContext sc) {}
```

```
  }  //class RaknareBean
```

Även session beans bean-klass är opåverkad av om remote- och/eller local-gränssnitt används.

## 4. Skapa *deployment descriptor*

I deployment descriptor är taggen för en session bean mindre komplex än den för entity beans. Vi behöver endast taggar för EJB:s namn, gränssnitt (home och remote/local) samt bean-klass. Taggarna är de samma som för entity beans, men de innesluts inom taggen `<session>` istället för `<entity>`.

```
<ejb-jar>
  <!--  ***************** EJBer ******************** -->
  <enterprise-beans>

    <!--  ***************** Raknare ***************** -->
    <session>
      <ejb-name>Raknare</ejb-name>
      <local-home>bpn.j2eeexempel.RaknareHome</local-home>
      <local>bpn.j2eeexempel.Raknare</local>
      <ejb-class>bpn.j2eeexempel.RaknareBean</ejb-class>
    </session>

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

Även med session beans så används taggarna <remote> och <home> ifall remote-gränssnitt används.

## 5. Skapa servlet-klass

Här deklareras instansvariabler i servlet-klass för att hålla både home- och remote-gränssnitt, d.v.s. variabler som är ”globala” för alla användare av servlet. Skälet till detta är att vi endast behöver en instans av EJB.[37] Därför hämtas referenser till både home- och remote-gränssnitt i `init()`.

I `doGet()` hämtas referens till PrintWriter och webbsida påbörjas (som ”vanligt”). Därefter anropas metoden `antal()` i EJB och resultatet placeras i en variabel `antal`. `antal` används sen för att se om värdet är större än 5 – om så är fallet så anropas `nollstall()` i EJB för att nollställa EJB:s räknare och metoden `antal()` anropas igen.

```
package bpn.j2eeexempel;  //Paket som klass ska finnas i (samma som EJB f lätthet)

import javax.servlet.*;       //Importera paket för servlets
import javax.servlet.http.*;
import java.io.*;             //Importera paket för PrintWriter
import javax.naming.*;        //Importera paket för JNDI
import javax.ejb.*;           //Importera paket för EJB

public class RaknareServlet extends HttpServlet {
    /*** Instansvariabler ************************************************/
  private RaknareHome home = null;   //Det behövs endast en referens till Home
  private Raknare raknare = null;    //Det räcker med en referens till Remote

    /*** Överskugga metoder i HttpServlet ********************************/
    //Metod som anropas när servlet-klass laddas (dvs endast en gång)
  public void init() throws ServletException {
```

[37] Hmmm… Halva meningen med att använda EJB är att de kan användas av flera användare samtidigt. Och att endast skapa en instans av EJB är motsägelsefull. ☺ Exempel med EJB Räknare är m.a.o. inte ett av mina bättre.

```
      try {
          //Hämta referens till initialt kontext - använd lokal (Resins) namntjänst
        Context initial =
          (Context) new InitialContext().lookup("java:comp/env/ejb");

          //Hämta referens till Raknares Home-gränssnitt
        home = (RaknareHome) initial.lookup("Raknare");

        raknare = home.create();      //Skapa en instans av EJB
      }
      catch(Exception e) {
        throw new ServletException(e);
      }
    }  //init()

      //Metod som anropas om servlets begärs med HTTP GET
    public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws IOException {
      PrintWriter out = res.getWriter();
      res.setContentType("text/html");

      out.println("<html>\n<body>");  //Påbörja webbsida

        //Skriv ut rubrik för hemsida
      out.println("<h1>Antal gånger som servlet har laddats</h1>");
      out.println("<p>Räknare nollställs när den når över 5</p>");

      int antal = raknare.antal();    //Anropa första metoden i EJB

        //Om antal större än 5 - nollställ räknare
      if(antal > 5) {
        raknare.nollstall();          //Anropa andra metoden i EJB
        antal = raknare.antal();      //Anropa första metoden i EJB (igen)
      }

      out.println("<p>Antal gånger som servlet laddats: " + antal + "</p>");

      out.println("</body>\n</html>");  //Avsluta webbsida
    }  //doGet()
  }  //class RaknareServlet
```

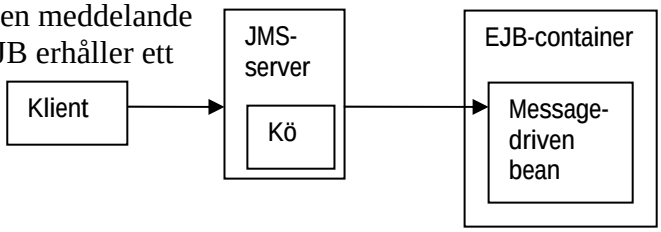
### 10.3.9 Session beans – stateless session bean

Den största skillnaden mellan stateless och stateful session beans är att stateless beans inte har några attribut, d.v.s. inga instansvariabler som behåller tillstånd mellan anrop av EJB:s metoder. Ytterligare en skillnad är att stateless session beans inte (eller behöver inte[38]) placeras i en objektpool, d.v.s. de använder inte tillståndet passiv (*passive*).

Ett annat sätt att se på stateless session beans är att de är objekt för att samla metoder (”funktionsbibliotek”).

### 10.3.10 Message-driven beans

Message-driven beans fungerar på ett mycket annorlunda sätt än entity och session beans. När en message-driven bean installeras i EJB-container så kopplas de till en kö i en JMS-server (*Java Message Service*). Klienter skickar sen meddelande till kön som skickas vidare till EJB. När EJB erhåller ett meddelande så utförs den enda metoden som finns i EJB.




[38] Det är upp till tillverkaren av J2EE-server om dom vill använda en objektpool för stateless session beans. Men för vårt välbefinnande (☺) så förutsätter vi att dom inte gör det.

# 11 Relationer mellan EJB

Liksom ”vanliga” objekt så kan även EJB har relationer (associationer) till andra EJB. Relationer kan kategoriseras efter:

- kardinallitet/multiplicitet (relationsgrad: en till en [1:1], en till många [1:M], många till en [M:1][39] eller många till många [M:N])
- och riktning (enkelriktad eller dubbelriktad)

Detta ger 7[40] olika typer av relationer:

- enkelriktad 1:1-relation
- dubbelriktad 1:1-relation
- enkelriktad 1:M-relation
- dubbelriktad 1:M-relation (och även specialfallet M:1-relation)
- enkelriktad M:1-relation
- enkelriktad M:N-relation
- dubbelriktad M:N-relation

För att ”skapa” relationer mellan EJB så använder vi deployment descriptor för att beskriva relationer. Observera att de flesta EJB-servrar kräver att bägge EJB som ingår i relationen har definierats i samma deployment descriptor som relationen.

Observera att introduktion av relationer mellan EJB ökar EJB:ers komplexitet, d.v.s. fler saker som kan bli/vara fel!

*Att beskriva de olika typerna av relationer tar en hel del plats, d.v.s. det är mycket kod (kod för 14 EJB:er tar plats). Detta kapitel kanske inte ska läsas i sin helhet, utan främst användas som referens (läs t.ex. de först 3-4 avsnitten och skumma resten). Men genom att visa exempel här så ges här (förhoppningsvis ☺) mallar att utgå från. Vissa typer av relationer ställer även till problem – ibland måste t.ex. en enkelriktad relation implementeras som en dubbelriktad.[41] Men genom att inte ha några publika accessmetoder (d.v.s. i remote- och/eller local-gränssnitt) så kan detta döljas för användare av EJB.[42]*

*Exemplens alla EJB:er och servlets placeras i samma paket (`bpn.relationer`) för enkelhetens skull (bl.a. kommer dom att placeras i samma mapp i filsystemet samt vi slipper behöva bry oss om säkerhet och synlighet, d.v.s. `public`, `protected` och `private`). Ersätt gärna namnet på paketet `bpn` med egen användaridentitet i (eventuellt) nätverk.*

*För att skapa kod i nedanstående exempel så har Resin använts som EJB-server och MySQL som databashanterare. Tabeller har placerats i databasen `test` (som skapas som standard när MySQL installeras) samt en användaridentitet `username`[43] med lösenordet `password` skapats och getts rättigheterna SELECT, INSERT, UPDATE samt DELETE i databasen. SQL-kod innehåller ingen kod för referensintegritet, d.v.s. främmande nycklar, då standardtypen för databaser i aktuell version av MySQL inte stödjer detta.[44] Konfigurationsfilen `WEB.XML` för webbapplikation har nedanstående utseende. Om*

---

[39] Ja, man skiljer på 1:M- och M:1-relationer. ☺
[40] Egentligen 8, men dubbelriktad M:1 är ett specialfall av dubbelriktad 1:M.
[41] Bl.a. beroende på i vilken tabell som främmande nyckel placeras.
[42] Detta kan vara något som endast gäller för EJB-servern Resin – andra servrar kanske har en bättre lösning.
[43] Om du är osäker på hur du skapar användare och ger rättigheter till databaser så kan du alltid använda användaridentiteten root (som du satte lösenord för när du installerade MySQL). Användaridentiteten root ska **inte** användas för detta i produktionsmiljö – då bör ett speciellt konto för applikationer skapas!
[44] Det är för övrigt inte något som behövs för exempel nedan heller… Kommande versioner av MySQL kommer (förhoppningsvis) som standard att stödja referensintegritet.

*andra servrar (databas och/eller EJB) används så kan exempel och konfigurationsfilen behöva anpassas för att fungera.*

```
<web-app>
  <servlet-mapping url-pattern="/servlet/*" servlet-name="invoker"/>

  <!-- *** START: Konfiguration av databas ******************************** -->
  <database>
    <jndi-name>jdbc/test</jndi-name>
    <driver>
      <type>org.gjt.mm.mysql.Driver</type>
      <url>jdbc:mysql://localhost:3306/test</url>
      <user>username</user>
      <password>password</password>
    </driver>
  </database>
  <!-- *** SLUT: Konfiguration av databas ********************************* -->

  <!-- *** START: Konfiguration databas for CMP ************************** -->
  <ejb-server jndi-name="ejb">
    <data-source>jdbc/test</data-source>
    <config-directory>WEB-INF</config-directory>
  </ejb-server>
  <!-- *** SLUT: Konfiguration databas for CMP *************************** -->
</web-app>
```

## 11.1 Grundläggande om relationer

För att definiera relationer så använder vi taggen `<relationships>` (på samma nivå som taggen `<enterprise-beans>`). En relation består av två ”roller” (*roles* – motsvarande två EJB:s involvering i relation) vilka beskrivs av taggen `<ejb-relationship-role>` (som alltså måste användas två gånger för varje relation – en för EJB på respektive ”sida” av relationen).

Nedan visas de ”huvudsakliga” taggarna i en deployment descriptor som beskriver session och entity beans samt relationen mellan två entity beans (session beans har alltså inga relationer ☺).

```
<ejb-jar>
  <enterprise-beans>
    <!-- Definition av EJBs -->
    <session>
      ...
    </session>

    <entity>
      ...
    </entity>

    <entity>
      ...
    </entity>
  </enterprise-beans>

  <relationships>
    <!-- Definition av relationer mellan EJBs -->
    <ejb-relation>
      <ejb-relationship-role>
        ...
      </ejb-relationship-role>
      <ejb-relationship-role>
        ...
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>

</ejb-jar>
```

I varje relationsroll så anger vi kardinalitet (multiplicitet) med taggen `<multiplicity>` för att ange EJB:s roll i relation (om EJB finns på en 1- eller M-sida av relation), vilket vi använder orden `One` eller `Many` för.

En relation som beskriv i en deployment descriptor är en relation som hanteras av container och kallas därför för *Container Managed Relationship* (CMR). Och om vi vill att en EJB ska kunna referera till (kunna frågas om) en EJB på andra sidan av relation, så använder vi kombinationen av taggarna `<cmr-field>` och `<cmr-field-name>` för att namnge CMR-fält (och därmed även namn på accessmetoder). Vi måste då lägga till en abstrakt metod med ”samma” namn i bean-klassen.[45] Och om EJB ingår i en 1:M-relation så måste metoden returnera en vektor av typen `Collection` [46]. Vill vi bara ha en enkelriktad relation så utelämnar vi alltså CMR-fält på ”målsidan” av relation. (Mer om detta nedan.)

I exempel nedan så definieras två EJB:er i taggarna `<entity>` (fullständig definition finns i första exempel nedan): `Person` och `Mantalsadress` (namn har markerats med grå bakgrundsfärg). Dessa ingår i en 1:1-relation (”multipliciteten” är `One` för båda) och den är enkelriktad då endast EJB Person har ett CMR-attribut (`mantalsadress` – även det markerat med grå bakgrundsfärg).

```
<ejb-jar>
  <enterprise-beans>
    <!-- Definition av EJBs -->
    <entity>
      <ejb-name>Person</ejb-name>
      ...
    </entity>

    <entity>
      <ejb-name>Mantalsadress</ejb-name>
      ...
    </entity>
  </enterprise-beans>

  <relationships>

    <!-- Definition av relationer mellan EJBs -->
    <ejb-relation>             <!-- EJB-relation -->

      <ejb-relationship-role>        <!-- EJB Persons roll i relation -->
        <relationship-role-source>
          <ejb-name>Person</ejb-name>
        </relationship-role-source>
        <multiplicity>One</multiplicity>
        <cmr-field>                        <!-- CMR-attribut -->
          <cmr-field-name>mantalsadress</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>

      <ejb-relationship-role>        <!-- EJB Mantalsadress roll i relation -->
        <relationship-role-source>
          <ejb-name>Mantalsadress</ejb-name>
        </relationship-role-source>
        <multiplicity>One</multiplicity>
      </ejb-relationship-role>

    </ejb-relation>

  </relationships>

</ejb-jar>
```

[45] Jag har satt ”samma” inom citattecken då metoderna i bean-klassen kommer ges prefixet ”get” och ”set” samt första bokstaven i CMR-fältet kommer ges en stor bokstav (t.ex. `getBokningar()` om CMR-fältet kallades `bokningar`).

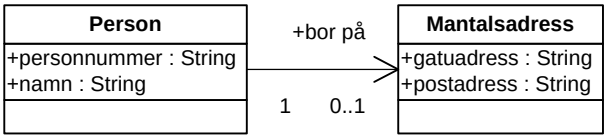
[46] `Collection` är ett gränssnitt (interface) som bl.a. implementeras av klasserna `ArrayList` och `Vector`.

## 11.2 Exempel på implementationer av EJB-relationer

I nedanstående 7 avsnitt (”under-under-kapitel”) ges exempel på de sju (åtta) typerna av relationer som kan förekomma mellan EJB:er. Koden för alla EJB:er är mycket liknande och när mönster uppfattas (d.v.s. att det känns som en upprepning) så kanske inte fortsatt läsning av kapitel behövs – återkom och använd som referens då typ av relation ska implementeras.

### 11.2.1 Enkelriktad 1:1-relation

En person kan endast vara mantalsskriven på en adress (mantalsadress), men vi kan inte fråga en adress (en bostad) om vem som bor där.




Underliggande tabeller har i stort sett samma utseende som klasser i klassdiagram. Men eftersom relationer i databaser måste lösas med en främmande nyckel så måste vi lägga till en sådan. Eftersom detta är en 1:1-relation även i databas så kan nyckeln placeras i antingen eller av tabellerna. Här väljer jag att placera den i tabellen `mantalsadresser`. Nedan visas SQL-satser för att skapa tabellerna samt lägga till testdata i tabellerna.

```
CREATE TABLE personer(
  personnummer VARCHAR (10) PRIMARY KEY,
  namn VARCHAR(50));

INSERT INTO personer VALUES("712345", "Oscar");
INSERT INTO personer VALUES("812345", "Sune");
INSERT INTO personer VALUES("912345", "Pelle");
INSERT INTO personer VALUES("012345", "Niklas");

CREATE TABLE mantalsadresser(
  personnummer VARCHAR(10) PRIMARY KEY,
  gatuadress VARCHAR(50),
  postadress VARCHAR(50));

INSERT INTO mantalsadresser VALUES("712345", "Nygatan 7", "12345 Nystad");
INSERT INTO mantalsadresser VALUES("812345", "Storgatan 8", "23456 Storstad");
INSERT INTO mantalsadresser VALUES("912345", "Storgatan 9", "23456 Storstad");
```

När vi utvecklar EJB:er så är det en bra idé att börja utan relationer (ju mindre komplext, ju lättare att felsöka). Först när EJB fungerar så bör man börja blanda in relationer. I detta första exempel (enkelriktad 1:1-relation) så visas hur vi skapar EJB:erna var för sig och sen hur gränssnitt, klass(-er) och deployment descriptor ändras (anpassas) för att lägga till relation. I efterkommande exempel visas endast fullständig kod (för slutgiltiga EJB med relationer).

### Implementation av EJB Person (version 1a)

EJB Person används i tre relationer – i detta avsnitt visas version 1 (version 1a innehåller inga relationer medan version 1b innehåller relation till EJB Mantalsadress). Version 2 och 3 används för att beskriva två andra typer av relationer mellan EJB:er.

#### Local-gränssnitt

Local-gränssnittet är som sagt det publika gränssnittet för EJB, d.v.s. det som innehåller affärsmetoder som användare av EJB kommer att använda. Eftersom detta är en entity bean så innehåller det främst accessmetoder (set- och get-metoder) för EJB:s attribut – CMP[47]- och

[47] CMP – *Container Managed Persistence*

CMR[48]-attribut (d.v.s. attribut motsvarande värden i tabells kolumner respektive relationer – mer om CMR-attribut när vi börjar implementera relationen mellan EJB:er). EJB Person innehåller två CMP-attribut: `personnummer`, som även är primärnyckel i tabell (d.v.s. får inte ändras), och `namn` (som får ändras ☺). Vi behöver alltså accessmetoder för dessa attribut (med undantag av set-metod för personnummer eftersom primärnyckel).

Local-gränssnitt ska utöka (*extend*), eller ärva, från gränssnittet `EJBLocalObject` (om remote-gränssnitt, så `EJBObject` istället), d.v.s. vi måste importera gränssnittet med en import-sats. Och eftersom vi ska placera EJB i ett paket så använder vi package-satsen för att namnge paket att placera gränssnitt i (`bpn.relationer` i exempel[49] – fullständigt namn för gränssnitt kommer, enligt kod nedan, att bli `bpn.relationer.Person`). Package-satsen kommer vara den samma för alla exempel och import-satserna kommer i stort sett vara de samma i alla exempel (med mindre variationer, d.v.s. de kommer utelämnas i senare exempel).

```
package bpn.relationer;

import javax.ejb.EJBLocalObject;

public interface Person extends EJBLocalObject
{
  /*** Publika accessmetoder ************************************************/
  public String getPersonnummer();
  public String getNamn();
  public void setNamn(String namn);
}  //interface Person
```

### Local home-gränssnitt

Local home-gränssnittet för entity beans används som sagt för att skapa nya eller hitta existerande instanser av EJB. Vi bör (måste) alltså definiera create- och find-metoder för de sätt vi vill kunna skapa och hitta instanserna. I detta exempel så definierar vi den enklaste create-metoden – d.v.s. create-metoden har parametrar för att kunna ange alla värden på kolumner i tabellen (`personnummer` och `namn` – värden som borde ha restriktionen, *constraint*, `NOT NULL` i tabellen då det är det som beskriver en person ☺).

Alla entity beans måste ha en metod `findByPrimaryKey()`, med primärnyckelklassen som typ på parameter, så en sådan definieras. Och för enkelhetens skull, d.v.s. för att kunna lista alla instanser av EJB, så lägger vi även till en metod `findAll()` som returnerar en vektor (av typen `Collection`) med alla instanser av EJB.[50] Vi behöver inte implementera någon av find-metoderna eftersom vi använder CMP. Men vi måste använda EQL (i deployment descriptor) för att tala om för EJB-server hur den ska implementera metoden `findAll()` för oss.

Local home-gränssnittet ska utöka (eller ärva ☺) från `EJBLocalHome` (`EJBHome` om remote-gränssnitt), så vi måste importera det. Och eftersom create- och find-metoder kan slänga undantag så måste vi importera dem också. (Vi skulle kunna skippa två rader om vi importerade hela paketet `javax.ejb` – det är ingen prestandaskillnad vid exekvering – men jag tycker bättre om att visa var jag hämtat vad från, d.v.s. vilka gränssnitt/klasser/undantag som finns i vilka paket.) Och sista import-satsen gäller gränssnittet `Collection` som returneras av metoden `findAll()`.

Och precis som med local-gränssnittet så använder vi package-satsen för att namnge paketet som gränssnitt ska placeras i (samma som EJB:s alla ”delar”).

---

[48] CMR – *Container Managed Relation*
[49] Ersätt som sagt gärna namnet på paketet `bpn` med egen användaridentitet i nätverk.
[50] Denna metod kanske inte är så lämplig om det finns 1000 eller fler instanser av en EJB. ☺

```
package bpn.relationer;

import java.util.Collection;
import javax.ejb.CreateException;
import javax.ejb.EJBLocalHome;
import javax.ejb.FinderException;

public interface PersonHome extends EJBLocalHome
{
  /*** create-metoder *******************************************************/
  public Person create(String namn, String personnr) throws CreateException;

  /*** find-metoder *********************************************************/
  public Person findByPrimaryKey(String personnr) throws FinderException;
  public Collection findAll() throws FinderException;
}  //interface PersonHome
```

### Bean-klass

Bean-klassen skiljer sig på "en"[51] detalj från en EJB utan relation: EJB:er med relationer har accessmetoder för relationer (CMR-attribut), d.v.s. för att kunna hämta referens till EJB (eller EJB:er) i andra änden av en relation. Eftersom vi börjar med att implementera EJB utan relation så markeras endast plats för accessmetoder med en kommentar.

Eftersom Resin används som EJB-server så ärvs från Cauchos klass `AbstractEntityBean` för att slippa behöva implementera metoderna i J2EE-gränssnittet `EntityBean`. Denna klass måste importeras liksom undantaget som ejbCreate-metoder slänger. Namnet på paket som klass ska placeras i är det samma som gränssnitten ovan.

```
package bpn.relationer;

import javax.ejb.CreateException;
import com.caucho.ejb.AbstractEntityBean;

public abstract class PersonBean extends AbstractEntityBean
{
  /*** Accessmetoder för CMP-attribut ***************************************/
  public abstract String getPersonnummer();
  public abstract void setPersonnummer(String personnr);
  public abstract String getNamn();
  public abstract void setNamn(String namn);

  /*** Accessmetoder för CMR-attribut ***************************************/
    //Här kommer accessmetoder för relation placeras...

  /*** ejbCreate- och ejbPostCreate-metoder *********************************/
  public String ejbCreate(String personnr, String namn) throws CreateException
  {
    setPersonnummer(personnr);
    setNamn(namn);
    return null;
  }  //ejbCreate()

  public void ejbPostCreate(String personnr, String namn) { }

  //Metoder i gränssnittet EntityBean implementeras av AbstractEntityBean

}  //class PersonBean
```

Eftersom primärnyckel (`personnummer`) för underliggande tabell (`personer`) inte är sammansatt så behöver vi inte skapa en primärnyckelsklass – vi kan använda klassen `String`. D.v.s. gränssnitt och klasser är färdiga för att kompileras till CLASS-filer. Glöm **inte** att

[51] En större detalj i alla fall. ☺

endast ”syntax” kontrolleras, d.v.s. vi måste göra en deployment descriptor och klient för att kontrollera om gränssnitt/klasser är buggfria! Kopiera CLASS-filerna till rätt[52] mapp under `WEB-INF\classes\` (eller ställ in programmeringsmiljön, IDE:n[53], så att den kompilerar direkt till `WEB-INF\classes\`) samt fortsätt med att skapa deployment descriptor och klient.

### Deployment descriptor

Deployment descriptor används som sagt för att koppla samma gränssnitt och klasser samt tala om för EJB-server vilka tjänster som ska användas. Vi namnger därför EJB (`Person`), så att vi kan hitta den med JNDI, samt anger vilka gränssnitt och klasser som EJB består av.[54] Eftersom vi ska använda CMP så anger vi beständighetstyp som `Container`.

För att EJB-server ska hitta rätt tabell och veta vilka attribut som den ska sparas (d.v.s. CMP-attributen) så lägger vi till dessa i deployment descriptor. Sist av allt så lägger vi till definitionen av metoden `findAll()` m.h.a. EQL.

Eftersom EJB först ska testas utan relationer så lämnas taggen `<relationships>` tom (eller utelämnas helt ☺). Spara filen som `bpn_relationer.ejb` (ersätt gärna `bpn` med egen användaridentitet för att undvika konflikter) i mappen `WEB-INF`. Nästa steg är att skapa klienten.

```
<ejb-jar>

  <!--  ***************** EJBer ******************* -->
  <enterprise-beans>

    <!--  ***************** Person ***************** -->
    <entity>
      <ejb-name>Person</ejb-name>
      <local-home>bpn.relationer.PersonHome</local-home>
      <local>bpn.relationer.Person</local>
      <ejb-class>bpn.relationer.PersonBean</ejb-class>

      <prim-key-class>String</prim-key-class>
      <primkey-field>personnummer</primkey-field>

      <persistence-type>Container</persistence-type>
      <reentrant>True</reentrant>

      <abstract-schema-name>personer</abstract-schema-name>
      <sql-table>personer</sql-table>

      <cmp-field><field-name>personnummer</field-name></cmp-field>
      <cmp-field><field-name>namn</field-name></cmp-field>

      <query>
        <query-method>
          <method-name>findAll</method-name>
        </query-method>
          <ejb-ql>SELECT o FROM personer o</ejb-ql>
      </query>
    </entity>

  </enterprise-beans>

  <!--  ***************** Relationer ***************** -->
  <relationships>
    <!--  Har kommer relationer placeras -->
  </relationships>

</ejb-jar>
```

[52] Om klasserna ligger i paket så måste mappstrukturen var den samma som paketens.
[53] IDE – *Integrated Development Enviroment*, program som JCreator och Visual Studio.NET.
[54] Glöm inte att ändra namnet på paketet `bpn` om ni ändrade i gränssnitt och bean-klass ovan!

### Klient

Som klient används en servlet med namnet `Person111Servlet` (första 1:an = enkelriktad, andra och tredje 1:an = 1:1-relation) – senare exempel med EJB Person kommer testas med servlets med liknande standard på namn.[55]

Eftersom EJB ska testas utan relation så ser koden i stort sett likadan ut som tidigare (och kommande) exempel i sammanfattning. En instansvariabel av local home-gränssnittet deklareras och JNDI används i metoden `init()` för att hämta en referens till EJB:s local home-gränssnitt.

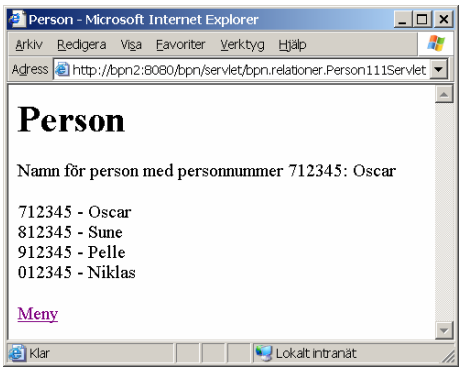
I `doGet()` ”påbörjas HTML-dokument” och en variabel `personer` (av typen `Collection`) deklareras för att hålla vektor med alla instanser av EJB. (Variabeln deklareras utanför try-catch-block då den kommer användas i blocket för if-satsen – variabler måste deklareras i samma block som de ska användas.) I första try-catch-blocket testas metoden `findByPrimaryKey()`, vars resultat skrivs ut med en gång, och metoden `findAll()`. Om den senare metoden fungerade så bör variabeln `personer` innehålla en vektor med alla instanser av EJB. Det är därmed möjligt att loopa över vektor och skriva ut värden på respektive EJB:s attribut. I loopen kommer så småningom även relationen till EJB Mantalsadress att testas, men för nu så markeras endast var i kod med en kommentar.

*När jag testar saker så brukar jag göra en enkel webbsida med länkar till servlets – det är lättare att skriva HTML-koden (och därmed URL:er till servlets) en gång och senare klicka sig fram. (Gör det också lättare att i framtiden återvända för att testa servlets igen när man glömt exakt vilket namn man gav servlet. ☺) Webbsida för dessa exempel har jag kallat för `relationer.html`. För att göra det enklare att navigera så skapas sist i servlet en länk tillbaka till denna webbsida med länkar.*

```
package bpn.relationer;

import java.io.PrintWriter;
import java.util.Collection;
import java.util.Iterator;
import javax.naming.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.ejb.*;

public class Person111Servlet extends HttpServlet
{
  /*** Instansvariabler ****************************************************/
  private PersonHome home = null;

  /*** Överskugga metoder i klassen HttpServlet ****************************/
  public void init() throws ServletException
  {
    try
    {
      Context initial =
        (Context)new InitialContext().lookup("java:comp/env/ejb");

      home = (PersonHome)initial.lookup("Person");
    }
    catch(NamingException ne)
    {
      throw new ServletException(ne);
    }
  } //init()


  public void doGet(HttpServletRequest req, HttpServletResponse res)
```

[55] Servlets som kommer vara klienter mot EJB Person i relationerna enkelriktad 1:M och dubbelriktad 1:M kommer heta `Person11MServlet` resp. `Person21MServlet`. Dessa tre servlets har gjorts för att testa respektive relation var för sig och därmed göra klienter i exempel så korta som möjligt.

```
      throws java.io.IOException
    {
      PrintWriter out = null;
      res.setContentType("text/html");
      out = res.getWriter();

      //Påbörja HTML-dokument och skriv ut rubrik
      out.println("<HTML>\n<HEAD>\n<TITLE>Person</TITLE>\n</HEAD>\n<BODY>");
      out.println("<H1>Person</H1>");

      Collection personer = null;       //Variabel för att testa findAll() nedan

      /*** Testa metoden findByPrimaryKey() och findAll() ******************/
      try
      {
        Person person = home.findByPrimaryKey("712345");
        out.println("<P>Namn för person med personnummer 712345: "
          + person.getNamn() + "</P>");

        personer = home.findAll();
      }
      catch(FinderException fe)
      {
        out.println("<PRE>");
        fe.printStackTrace(out);
        out.println("</PRE>");
      }

      //Om personer kunde hämtas, d.v.s. metoden findAll() "fungerade"
      if(personer != null)
      {
        Iterator iter = personer.iterator();

        while(iter.hasNext() )
        {
          Person person = (Person)iter.next();
          out.println(person.getPersonnummer() + " - " + person.getNamn()
            + "<BR>");

          /*** Testa relation till Mantalsadress ***********************/
            //Här kommer kod läggas till när vi ska testa relationen
          /************************************************************/
        }
      }
      else
        out.println("Metoden findAll() fungerade inte...");

      //Skriv ut länk tillbaka till meny och avsluta HTML-dokument
      out.println("<P><A HREF=\"../relationer.html\">Meny</A></P>");
      out.println("</HTML>\n</HTML>");
    }  //doGet()

  }  //class Person111Servlet
```




Kompilera servlet till CLASS-fil och kopiera till rätt mapp under `WEB-INF\classes\`. Öppna sen URL till servlet – låt någon eller några minuter gå så att Resin kan generera och kompilera sina klasser för EJB – och kontrollera att all kod är korrekt. När EJB Person är buggfri är det dags att skapa EJB Mantalsadress (utan relationer).

## Implementation av EJB Mantalsadress

Implementationen av EJB Mantalsadress (utan relationer) ser i stort sett likadan ut som den för EJB Person. Namn på gränssnitt, klasser och metoder ändras för att motsvara EJB Mantalsadress och dess attribut. Vi kan alltså, om vi vill, kopiera koden från EJB Person och endast ändra där det behövs. Om kod kopieras från EJB Person – tänk då på att det är lätt att

missa att ändra en metod här och en bokstav där. Kompilator fångar några av felen medan andra fel inte upptäcks förrän vi testar EJB med klienten.

*Och om kod kopieras från denna sammanfattning (d.v.s. PDF-filen) så har Word en tendens att ändra en del tecken (t.ex. dubbla citattecken, d.v.s. ") till ”finare” versioner (t.ex. ”) – tecken som har annan betydelse (eller ingen alls) i Java.*

### Local-gränssnitt

EJB Mantalsadress har tre (CMP-)attribut, `personnummer` (primärnyckel för tabell), `gatuadress` och `postadress`, vilka endast de två sista kräver publika accessmetoder. D.v.s. personnummer behöver inte ändras då detta görs genom att koppla en instans av EJB Mantalsadress till EJB Person. I övrigt ser gränssnittet ut som det för EJB Person (utom namn i gränssnittet ☺).

```
package bpn.relationer;

import javax.ejb.EJBLocalObject;

public interface Mantalsadress extends EJBLocalObject
{
  /*** Publika accessmetoder ************************************************/
  public String getGatuadress();
  public void setGatuadress(String adress);
  public String getPostadress();
  public void setPostadress(String adress);
}  //interface Mantalsadress
```

### Local home-gränssnitt

Även local home-gränssnittet ser i stort sett likadant ut som det för EJB Person. Skillnaden är, förutom namnet ☺, create-metoden och att typen på returvärdet från metoden `findByPrimaryKey()` är `Mantalsadress` (istället för `Person` som i EJB Person).

```
package bpn.relationer;

import java.util.Collection;
import javax.ejb.CreateException;
import javax.ejb.EJBLocalHome;
import javax.ejb.FinderException;

public interface MantalsadressHome extends EJBLocalHome
{
  /*** create-metoder *******************************************************/
  public Mantalsadress create(String personnr, String gadress, String padress)
    throws CreateException;

  /*** find-metoder *********************************************************/
  public Mantalsadress findByPrimaryKey(String personnr) throws FinderException;
  public Collection findAll() throws FinderException;
}  //interface MantalsadressHome
```

### Bean-klass

Även bean-klassen påminner om den för EJB Person (och andra EJB utan relationer). I bean-klassen för EJB Mantalsadress måste vi ha accessmetoder för alla attribut i EJB, d.v.s. även personnummer måste alltså ha accessmetoder.

```
package bpn.relationer;

import javax.ejb.CreateException;
import com.caucho.ejb.AbstractEntityBean;
```

```
public abstract class MantalsadressBean extends AbstractEntityBean
{
  /*** Accessmetoder för CMP-attribut ************************************/
  public abstract String getPersonnummer();
  public abstract void setPersonnummer(String personnr);
  public abstract String getGatuadress();
  public abstract void setGatuadress(String adress);
  public abstract String getPostadress();
  public abstract void setPostadress(String adress);

  /*** Accessmetoder för CMR-attribut ************************************/
    //Relation är enkelriktad, därav inga accessmetoder för relation här

  /*** ejbCreate- och ejbPostCreate-metoder ******************************/
  public String ejbCreate(String personnr, String gadress, String padress)
    throws CreateException
  {
    setPersonnummer(personnr);
    setGatuadress(gadress);
    setPostadress(padress);
    return null;
  }  //ejbCreate()

  public void ejbPostCreate(String personnr, String gadress, String padress) { }

  //Metoder i gränssnittet EntityBean implementeras av AbstractEntityBean

}  //class MantalsadressBean
```

## Deployment descriptor

I deployment descriptor läggs en tag `<entity>` till för EJB Mantalsadress.

```
<ejb-jar>

  <!--  ***************** EJBer ******************* -->
  <enterprise-beans>

    <!--  ***************** Person ***************** -->
    <entity>
      <ejb-name>Person</ejb-name>
      ...
    </entity>

    <!--  ***************** Mantalsadress ***************** -->
    <entity>
      <ejb-name>Mantalsadress</ejb-name>
      <local-home>bpn.relationer.MantalsadressHome</local-home>
      <local>bpn.relationer.Mantalsadress</local>
      <ejb-class>bpn.relationer.MantalsadressBean</ejb-class>

      <prim-key-class>String</prim-key-class>
      <primkey-field>personnummer</primkey-field>

      <persistence-type>Container</persistence-type>
      <reentrant>True</reentrant>

      <abstract-schema-name>mantalsadresser</abstract-schema-name>
      <sql-table>mantalsadresser</sql-table>

      <cmp-field><field-name>personnummer</field-name></cmp-field>
      <cmp-field><field-name>gatuadress</field-name></cmp-field>
      <cmp-field><field-name>postadress</field-name></cmp-field>

      <query>
        <query-method>
          <method-name>findAll</method-name>
        </query-method>
          <ejb-ql>SELECT o FROM mantalsadresser o</ejb-ql>
      </query>
    </entity>

  </enterprise-beans>

  <!--  ***************** Relationer ***************** -->
```

```
    <relationships>
      <!--  Har kommer relationer placeras -->
    </relationships>

  </ejb-jar>
```

## Klient

Även klient-servlet för EJB Mantalsadress är i mångt och mycket en upprepning av tidigare klienter.

```
  package bpn.relationer;

  import java.io.PrintWriter;
  import java.util.Collection;
  import java.util.Iterator;
  import javax.naming.*;
  import javax.servlet.*;
  import javax.servlet.http.*;
  import javax.ejb.*;

  public class MantalsServlet extends HttpServlet
  {
    /*** Instansvariabler ****************************************************/
    private MantalsadressHome home = null;

    /*** Överskugga metoder i klassen HttpServlet ****************************/
    public void init() throws ServletException
    {
      try
      {
        Context initial =
          (Context)new InitialContext().lookup("java:comp/env/ejb");

        home = (MantalsadressHome)initial.lookup("Mantalsadress");
      }
      catch(NamingException ne)
      {
        throw new ServletException(ne);
      }
    }  //init()


    public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws java.io.IOException
    {
      PrintWriter out = null;
      res.setContentType("text/html");
      out = res.getWriter();

      //Påbörja HTML-dokument och skriv ut rubrik
      out.println("<HTML>\n<HEAD>\n<TITLE>Mantalsadress</TITLE>\n</HEAD>\n<BODY>");
      out.println("<H1>Mantalsadress</H1>");

      Collection adresser = null;

      /*** Testa metoden findByPrimaryKey() och findAll() ******************/
      try
      {
        Mantalsadress adress = home.findByPrimaryKey("712345");
        out.println("<P>Gatu- och postadress för mantalsadress för person "
          + "med personnummer 712345: " + adress.getPostadress()
          + ", " + adress.getPostadress() + "</P>");

        adresser = home.findAll();
      }
      catch(FinderException fe)
      {
        out.println("<PRE>");
        fe.printStackTrace(out);
        out.println("</PRE>");
      }
```

```
      //Om adresser kunde hämtas, d.v.s. metoden findAll() "fungerade"
      if(adresser != null)
      {
        Iterator iter = adresser.iterator();

        while(iter.hasNext() )
        {
          Mantalsadress adress = (Mantalsadress)iter.next();
          out.println(adress.getGatuadress() + ", "
            + adress.getPostadress() + "<BR>");
        }
      }
      else
        out.println("Metoden findAll() fungerade inte...");

      //Skriv ut länk tillbaka till meny och avsluta HTML-dokument
      out.println("<P><A HREF=\"../relationer.html\">Meny</A></P>");
      out.println("</HTML>\n</HTML>");
    }  //doGet()

  }  //class Person111Servlet
```

## Ändringar för relation

Relationer mellan EJB:er beskrivs (bl.a.) i deployment descriptor. Jag brukar börja i den änden för att sen göra (eventuella) ändringar i bean-klass och eventuellt även remote- (och/eller local-)gränssnitt. Home- (eller local home-)gränssnitt och primärnyckelsklassen är oftast opåverkade av relationer mellan EJB – ändringar behöver främst göras om vi lägger till fler create-metoder och/eller accessmetoder för CMR-attribut.

### Ändringar i deployment descriptor

För varje relation (enkel- eller dubbelriktad[56]) så beskriver vi relation i deployment descriptor genom att lägga till en tagg `<ejb-relation>`. I denna tagg beskriver vi de två EJB:ernas roll (`<ejb-relationship-role>`) i relationen, bl.a. multiplicitet (`<multiplicity>` – en, `One`, eller många, `Many`) och CMR-attribut (*Container Managed Relationship field*, om något) för objekt i andra änden (`<cmr-field>`).

I denna enkelriktade 1:1-relation mellan Person och Mantalsadress så ska alltså endast EJB Person ha ett CMR-attribut. Och precis som med CMP-attribut så måste vi lägga till accessmetoder för CMR-attribut med prefixen set och get (d.v.s. attributet `mantalsadress` motsvaras av accessmetoderna `getMantalsadress()` och `setMantalsadress()`). I de flesta fall behöver vi endast ange namnet på CMR-attributet (`<cmr-field-name>`) och EJB-container löser hur EJB:er ska läsas från databasen. Men i detta fall finns den främmande nyckeln i tabellen `mantalsadresser` i 1:1-relationen, så därför måste även namnet på kolumn som är främmande nyckel anges, vilket taggen `<sql-column>` används för.

```
<ejb-jar>

  <!--  ***************** EJBer ******************* -->
  <enterprise-beans>

    <!--  ***************** Person ***************** -->
    <entity>
      <ejb-name>Person</ejb-name>
      ...
    </entity>

    <!--  ***************** Mantalsadress ***************** -->
    <entity>
```

[56] Vi behöver alltså inte beskriva relation en gång från bägge håll, d.v.s. **en** beskrivning för respektive EJB:s roll i relation räcker.

```
        <ejb-name>Mantalsadress</ejb-name>
        ...
      </entity>

    </enterprise-beans>

    <!--  ***************** Relationer ***************** -->
    <relationships>

      <!--  ***************** Person -> Mantalsadress ***************** -->
      <ejb-relation>
        <ejb-relationship-role>
          <relationship-role-source>
            <ejb-name>Person</ejb-name>
          </relationship-role-source>
          <multiplicity>One</multiplicity>
          <cmr-field>
            <cmr-field-name>mantalsadress</cmr-field-name>
            <sql-column>personnummer</sql-column>
          </cmr-field>
        </ejb-relationship-role>

        <ejb-relationship-role>
          <relationship-role-source>
            <ejb-name>Mantalsadress</ejb-name>
          </relationship-role-source>
          <multiplicity>One</multiplicity>
        </ejb-relationship-role>
      </ejb-relation>

    </relationships>

  </ejb-jar>
```

### Ändringar i EJB Person (version 1b)

I bean-klassen lägger vi till accessmetoder för CMR-attributet, i övrigt så är klassen oförändrad.

```
  public abstract class PersonBean extends AbstractEntityBean
  {
    /*** Accessmetoder för CMP-attribut ***************************************/
    public abstract String getPersonnummer();
    public abstract void setPersonnummer(String personnr);
    public abstract String getNamn();
    public abstract void setNamn(String namn);

    /*** Accessmetoder för CMR-attribut ***************************************/
    public abstract Mantalsadress getMantalsadress();
    public abstract void setMantalsadress(Mantalsadress adress);

    /*** ejbCreate- och ejbPostCreate-metoder *********************************/
    public String ejbCreate(String personnr, String namn) throws CreateException
    { ...  //Enligt tidigare version (1a) }
    public void ejbPostCreate(String personnr, String namn) { }
  }  //class PersonBean
```

Om vi vill låta klienten (till EJB) kunna använda accessmetoderna för CMR-attribut så måste vi lägga till accessmetoderna i local-gränssnittet.

```
  public interface Person extends EJBLocalObject
  {
    /*** Publika accessmetoder ****************************************************/
    public String getPersonnummer();
    public String getNamn();
    public void setNamn(String namn);

    public Mantalsadress getMantalsadress();
    public void setMantalsadress(Mantalsadress adress);
```

```
  }  //interface Person
```

### Ändringar i EJB Mantalsadress

Eftersom relation är enkelriktad (från Person till Mantalsadress) så är EJB Mantalsadress opåverkad av relation.

### Ny kod i klient till EJB Person

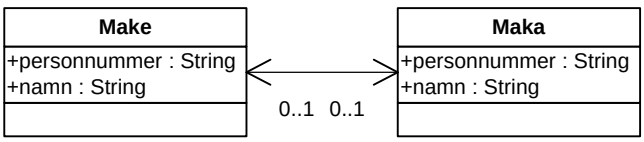
För att testa relationen så kan vi hämta alla personer (som i version av klient ovan) samt fråga respektive person om mantalsadress. Vi lägger därför till nedanstående kod mellan kommentarerna (som finns i version av klient ovan).


Först frågas person efter mantalsadress som vi i sin tur kan fråga om gatu- och postadress. Observera att om en person **inte** har en mantalsadress så slängs `ObjectNotFoundException` – ett undantag som vi kan fånga och meddela användaren om att personen inte har en adress. (I catch-sats fångas undantag av typen `Exception` och inte `ObjectNotFoundException`. Skälet till detta är för att koden inte ska bli för lång, d.v.s. lättare att läsa, och fungera i de flesta EJB-servrar. Resin slänger ett undantag av typen `EJBExceptionWrapper`, i paketet `com.caucho.ejb`, d.v.s. vi måste annars fånga detta fel och hämta det ursprungliga felet, med metoden `getRootCause()`, om vi vill kontrollera om ”rätt” fel).

```
/*** Testa relation till Mantalsadress ***********************/
try
{
  Mantalsadress adress = person.getMantalsadress();

    //Nedanstående rad kan generera ObjectNotFoundException,
    // dvs det finns inget relaterat objekt
  out.println(" - " + adress.getGatuadress() + ", "
    + adress.getPostadress() + "<BR>");
}
catch(Exception e)
{
  out.println("- (adress saknas för person...)<BR>");
}
/************************************************************/
```

## 11.2.2 Dubbelriktad 1:1-relation

En make är gift med en maka och en maka är gift med en make (förhoppningsvis den man som är gift med henne ☺).




I denna relation så känner båda EJB till varandra. Även tabellerna (eller posterna i tabellerna) känner till varandra, d.v.s. det finns en främmande nyckel (`partner`) i respektive tabell. Nedan visas SQL-kod för att skapa tabellerna och lägga till testdata i tabellerna. (Datamässigt så bör make och maka kunna placeras i samma tabell eftersom tabellernas kolumner är identiska. Men för att göra det lättare att implementera EJB:erna så har de placerats i olika tabeller. Och eftersom plural för både make och maka är det samma – makar – så har, mot rekommendationer för tabellnamn, singularformen av bägge använts som namn på tabellerna. Tabellerna borde hetat `makar` och `makar`, d.v.s. pluralformen.)

```
CREATE TABLE make(
  personnummer VARCHAR(10) PRIMARY KEY,
  namn VARCHAR(50),
  partner VARCHAR(10));

INSERT INTO make VALUES("734567", "Oscar", 754321);
INSERT INTO make VALUES("834567", "Sune", 854321);
INSERT INTO make VALUES("934567", "Pelle", NULL);
INSERT INTO make VALUES("034567", "Niklas", NULL);

CREATE TABLE maka(
  personnummer VARCHAR(10) PRIMARY KEY,
  namn VARCHAR(50),
  partner VARCHAR(10));

INSERT INTO maka VALUES("754321", "Olivera", 734567);
INSERT INTO maka VALUES("854321", "Sandra",  834567);
INSERT INTO maka VALUES("954321", "Petronella", NULL);
```

## Implementation av EJB Make

Implementationen av EJB Make (liksom Maka) är i stort sett den samma som för EJB Person i exempel ovan. Skillnaden är att vi har ett relationsattribut, maka (resp. make), för att komma åt EJB i andra änden av relationen.

I detta exempel visas gränssnitt, klasser och deployment descriptor med relationer med en gång (för att spara plats – se föregående exempel för skillnad mellan EJB med och utan relationer).

### Local-gränssnitt

Eftersom EJB är en entity bean, precis som alla EJB i detta kapitel, så innehåller local-gränssnittet accessmetoder för CMP- och CMR-attribut då detta är affärslogiken för EJB. CMP-attributet `personnummer` är primärnyckel i underliggande tabell, vilket gör att vi endast definierar get-metod för attributet, medan vi definierar båda accessmetoder (set och get) för CMP-attributet `namn` och CMR-attributet `maka`.

```
package bpn.relationer;

import javax.ejb.EJBLocalObject;

public interface Make extends EJBLocalObject
{
  /*** Publika accessmetoder ***********************************************/
  public String getPersonnummer();
  public String getNamn();
  public void setNamn(String namn);

  public abstract Maka getMaka();
  public abstract void setMaka(Maka maka);
}  //interface Make
```

### Local home-gränssnitt

Local home-gränssnittet innehåller create-metod samt de två find-metoderna `findByPrimaryKey()` och `findAll()`. Den första find-metoden måste vi definiera, eftersom entity bean, medan den andra metoden är, åter igen, för att lättare testa vår EJB.

```
package bpn.relationer;

import java.util.Collection;
import javax.ejb.CreateException;
```

```
import javax.ejb.EJBLocalHome;
import javax.ejb.FinderException;

public interface MakeHome extends EJBLocalHome
{
  /*** create-metoder *******************************************************/
  public Make create(String namn, String personnr) throws CreateException;

  /*** find-metoder *********************************************************/
  public Make findByPrimaryKey(String personnr) throws FinderException;
  public Collection findAll() throws FinderException;
}  //interface MakeHome
```

### Bean-klass

I denna, liksom alla EJB i kapitel, så definierar vi de abstrakta accessmetoderna för CMP- och CMR-attribut liksom `ejbCreate()` (och motsvarande `ejbPostCreate()`). Bean-klasser, i motsatts till remote- och/eller local-gränssnitt, innehåller både accessmetoder för både CMP- och CMR-attribut för att CMP ska fungera.

```
package bpn.relationer;

import javax.ejb.CreateException;
import com.caucho.ejb.AbstractEntityBean;

public abstract class MakeBean extends AbstractEntityBean
{
  /*** Accessmetoder för CMP-attribut ***************************************/
  public abstract String getPersonnummer();
  public abstract void setPersonnummer(String personnr);
  public abstract String getNamn();
  public abstract void setNamn(String namn);

  /*** Accessmetoder för CMR-attribut ***************************************/
  public abstract Maka getMaka();
  public abstract void setMaka(Maka maka);

  /*** ejbCreate- och ejbPostCreate-metoder *********************************/
  public String ejbCreate(String personnr, String namn) throws CreateException
  {
    setPersonnummer(personnr);
    setNamn(namn);
    return null;
  }  //ejbCreate()

  public void ejbPostCreate(String personnr, String namn) { }

  //Metoder i gränssnittet EntityBean implementeras av AbstractEntityBean

}  //class MakeBean
```

## Implementation av EJB Maka

Implementationen av EJB Maka är som sagt i stort sett den samma som för EJB Make, därför visas endast koden utan kommentarer.

### Local-gränssnitt

```
package bpn.relationer;

import javax.ejb.EJBLocalObject;

public interface Maka extends EJBLocalObject
{
  /*** Publika accessmetoder ************************************************/
  public String getPersonnummer();
  public String getNamn();
  public void setNamn(String namn);

  public abstract Make getMake();
```

```
  public abstract void setMake(Make make);
}  //interface Maka
```

### Local home-gränssnitt

```
package bpn.relationer;

import java.util.Collection;
import javax.ejb.CreateException;
import javax.ejb.EJBLocalHome;
import javax.ejb.FinderException;

public interface MakaHome extends EJBLocalHome
{
  /*** create-metoder *******************************************************/
  public Maka create(String namn, String personnr) throws CreateException;

  /*** find-metoder *********************************************************/
  public Maka findByPrimaryKey(String personnr) throws FinderException;
  public Collection findAll() throws FinderException;
}  //interface MakaHome
```

### Bean-klass

```
package bpn.relationer;

import javax.ejb.CreateException;
import com.caucho.ejb.AbstractEntityBean;

public abstract class MakaBean extends AbstractEntityBean
{
  /*** Accessmetoder för CMP-attribut ***************************************/
  public abstract String getPersonnummer();
  public abstract void setPersonnummer(String personnr);
  public abstract String getNamn();
  public abstract void setNamn(String namn);

  /*** Accessmetoder för CMR-attribut ***************************************/
  public abstract Make getMake();
  public abstract void setMake(Make make);

  /*** ejbCreate- och ejbPostCreate-metoder *********************************/
  public String ejbCreate(String personnr, String namn) throws CreateException
  {
    setPersonnummer(personnr);
    setNamn(namn);
    return null;
  }  //ejbCreate()

  public void ejbPostCreate(String personnr, String namn) { }

  //Metoder i gränssnittet EntityBean implementeras av AbstractEntityBean

}  //class MakaBean
```

## Deployment descriptor för EJB:er Make och Maka samt relation

I deployment descriptor använder vi en entity-tagg för respektive EJB (Make och Maka) samt en tagg `<ejb-relation>` för att beskriva relationen mellan EJB. I detta fall så är relationen dubbelriktad och vi definierar därför CMR-attribut i bägge taggar `<ejb-relationship-role>`. Kolumner i tabell heter inte som CMR-attribut – därför måste vi ange namn på dem med `<sql-column>`.

```
<ejb-jar>

  <!--  ***************** EJBer ******************** -->
  <enterprise-beans>

    <!--  ***************** Make ***************** -->
    <entity>
```

```
        <ejb-name>Make</ejb-name>
        <local-home>bpn.relationer.MakeHome</local-home>
        <local>bpn.relationer.Make</local>
        <ejb-class>bpn.relationer.MakeBean</ejb-class>

        <prim-key-class>String</prim-key-class>
        <primkey-field>personnummer</primkey-field>

        <persistence-type>Container</persistence-type>
        <reentrant>True</reentrant>

        <abstract-schema-name>make</abstract-schema-name>
        <sql-table>make</sql-table>

        <cmp-field><field-name>personnummer</field-name></cmp-field>
        <cmp-field><field-name>namn</field-name></cmp-field>

        <query>
          <query-method>
            <method-name>findAll</method-name>
          </query-method>
            <ejb-ql>SELECT o FROM make o</ejb-ql>
        </query>
      </entity>

      <!--  ***************** Maka ***************** -->
      <entity>
        <ejb-name>Maka</ejb-name>
        <local-home>bpn.relationer.MakaHome</local-home>
        <local>bpn.relationer.Maka</local>
        <ejb-class>bpn.relationer.MakaBean</ejb-class>

        <prim-key-class>String</prim-key-class>
        <primkey-field>personnummer</primkey-field>

        <persistence-type>Container</persistence-type>
        <reentrant>True</reentrant>

        <abstract-schema-name>maka</abstract-schema-name>
        <sql-table>maka</sql-table>

        <cmp-field><field-name>personnummer</field-name></cmp-field>
        <cmp-field><field-name>namn</field-name></cmp-field>

        <query>
          <query-method>
            <method-name>findAll</method-name>
          </query-method>
            <ejb-ql>SELECT o FROM maka o</ejb-ql>
        </query>
      </entity>

    </enterprise-beans>

    <!--  ***************** Relationer ***************** -->
    <relationships>

      <!--  ***************** Make <-> Maka ***************** -->
      <ejb-relation>
        <ejb-relationship-role>
          <relationship-role-source>
            <ejb-name>Make</ejb-name>
          </relationship-role-source>
          <multiplicity>One</multiplicity>
          <cmr-field>
            <cmr-field-name>maka</cmr-field-name>
            <sql-column>partner</sql-column>
          </cmr-field>
        </ejb-relationship-role>

        <ejb-relationship-role>
          <relationship-role-source>
            <ejb-name>Maka</ejb-name>
          </relationship-role-source>
          <multiplicity>One</multiplicity>
          <cmr-field>
            <cmr-field-name>make</cmr-field-name>
```

```
            <sql-column>partner</sql-column>
          </cmr-field>
        </ejb-relationship-role>
      </ejb-relation>
    </relationships>

  </ejb-jar>
```

### Klient för EJB Make

Klienten för EJB Make är i stort sett identisk med den för EJB Person i exempel ovan. (D.v.s. den finns med här främst för att visa på likheterna – i fortsättningen kommer endast relevant kod i klienter att visas då koden i kommande klienter kommer se i stort sett identisk med denna och förra klienten.)

```
  package bpn.relationer;

  import java.io.PrintWriter;
  import java.util.Collection;
  import java.util.Iterator;
  import javax.naming.*;
  import javax.servlet.*;
  import javax.servlet.http.*;
  import javax.ejb.*;

  public class MakeServlet extends HttpServlet
  {
    /*** Instansvariabler ****************************************************/
    private MakeHome home = null;

    /*** Överskugga metoder i klassen HttpServlet ****************************/
    public void init() throws ServletException
    {
      try
      {
        Context initial =
          (Context)new InitialContext().lookup("java:comp/env/ejb");

        home = (MakeHome)initial.lookup("Make");
      }
      catch(NamingException fe)
      {
        throw new ServletException(fe);
      }
    }  //init()


    public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws java.io.IOException
    {
      PrintWriter out = null;
      res.setContentType("text/html");
      out = res.getWriter();

      //Påbörja HTML-dokument och skriv ut rubrik
      out.println("<HTML>\n<HEAD>\n<TITLE>Make</TITLE>\n</HEAD>\n<BODY>");
      out.println("<H1>Make</H1>");

      Collection makar = null;

      /*** Testa metoden findByPrimaryKey() och findAll() ******************/
      try
      {
        Make make = home.findByPrimaryKey("734567");
        out.println("<P>Namn för make med personnummer 734567: "
          + make.getNamn() + "</P>");

        makar = home.findAll();
      }
      catch(FinderException fe)
      {
        out.println("<PRE>");
```

```
        fe.printStackTrace(out);
        out.println("</PRE>");
      }

      //Om personer kunde hämtas, d.v.s. metode findAll() "fungerade"
      if(makar != null)
      {
        Iterator iter = makar.iterator();

        while(iter.hasNext() )
        {
          Make make = (Make)iter.next();
          out.println(make.getPersonnummer() + " - " + make.getNamn()
            + "<BR>");

          /*** Testa relation till Maka ********************************/
          Maka maka = make.getMaka();

          if(maka != null)
            out.println(" - " + maka.getPersonnummer() + ", "
              + maka.getNamn() + "<BR>");
          else
            out.println("- (maka saknas för person...)<br>");
          /************************************************************/
        }  //while
      }  //if(makar != null)
      else
        out.println("Kunde inte hitta någon EJB Make...");

      //Skriv ut länk tillbaka till meny och avsluta HTML-dokument
      out.println("<P><A HREF=\"../relationer.html\">Meny</A></P>");
      out.println("</HTML>\n</HTML>");
    }  //doGet()

  }  //class MakeServlet
```

Klienten för EJB Maka är i stort sett den samma som för EJB Make – ersätt Make och MakeHome med Maka och MakaHome samt Maka och MakaHome med Make och MakeHome (relationen är ju omvänd ☺). Därför visas inte klienten för EJB Maka.

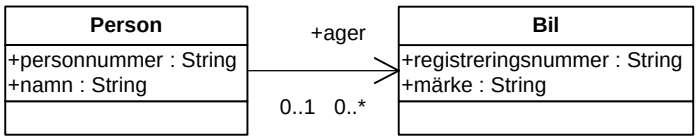
## Slutsatser

Om man jämför exempel med enkel- och dubbelriktad 1:1-relation så ser man många likheter (bl.a. beroende på att klasser i klassdiagram och tabeller i databas för EJB:er Person, Make och Maka är så pass lika). Koden i dessa exempel kan med andra ord användas som mallar (t.ex. genom att kopiera och klistra in) och smärre ändringar göras.[57]

I fortsättningen kommer inte klienter visas i sin helhet (eftersom det ”bara” är metoderna `findByPrimaryKey()` och `findAll()` som anropas) samt namn på paket som gränssnitt/klasser ingår i (`bpn.relationer`) och importerade paket utelämnas för att spara plats. Kompilator kommer meddela att gränssnitt, klass eller undantag inte känns igen, d.v.s. att paket och/eller klasser inte importerats.

## 11.2.3 Enkelriktad 1:M-relation (en till många)

En person kan äga flera bilar, men man kan inte fråga en bil om vem som äger den (däremot ett registreringsbevis – se nästa avsnitt).




[57] Till att börja med så behöver vi inte lära oss **exakt** hur koden ser ut (det gör man när man skrivit/kopierat samma kod ett antal gånger ☺) utan vilken kod som krävs.

I detta exempel tittar vi endast på hur vi kan hämta objekt i M-änden på en 1:M-relation. Men i nästa exempel tittar vi även på hur vi kan lägga till objekt i M-änden av en 1:M-relation.

Nedan visas SQL-kod för att skapa tabellerna och lägga till testdata i tabellerna. Om du redan skapat tabellen `personer` så behöver du inte göra det igen och tabellen med bilar heter `bilar2` för att undvika eventuell konflikt med tidigare exempel med EJB Bil i denna sammanfattning.

```
CREATE TABLE personer(
  personnummer VARCHAR(10) PRIMARY KEY,
  namn VARCHAR(50));

INSERT INTO personer VALUES("712345", "Oscar");
INSERT INTO personer VALUES("812345", "Sune");
INSERT INTO personer VALUES("912345", "Pelle");
INSERT INTO personer VALUES("012345", "Niklas");

CREATE TABLE bilar2(
  registreringnummer VARCHAR(6) PRIMARY KEY,
  marke VARCHAR(20),
  agare VARCHAR(10));

INSERT INTO bilar2 VALUES('ABC123', 'BMW 525 iX', '712345');
INSERT INTO bilar2 VALUES('DEF456', 'Audi 100 E', '712345');
INSERT INTO bilar2 VALUES('GHI789', 'Opel Kadett', '812345');
```

I nedanstående kod så har namn på paket (d.v.s. raden `package bpn.relationer;`) utelämnats liksom import-satser. Se tidigare exempel för vad som behöver (kan behöva) importeras – kompilator kommer meddela vilka gränssnitt eller klasser som saknas.

## Implementation av EJB Person (version 2)

För enkelhetens skull så visas endast kod som behövs för relation mellan Person och Bil. Om du redan gjort en EJB Person enligt tidigare exempel räcker det med att du lägger till kod som saknas.

### Local-gränssnitt

Local-gränssnittet ser i stort sett ut som den i version 1 av EJB Person. Utöver accessmetoder för EJB Persons CMP-attribut så lägger vi till en accessmetod (`getBilar()`) för att returnera alla bilar som person äger, d.v.s. CMR-attributet `bilar`. Metoden bör[58] ha `Collection` som returtyp då metoden kan returnera flera EJB:er. I detta exempel så kan vi inte lägga till fler bilar via EJB Person (vi måste använda EJB Bil för detta). Men vi skulle kunna lägga till en metod `addBil()`, en metod som vi själva måste implementera i bean-klassen, för detta (d.v.s. CMP klarar ännu inte av att automatiskt hantera kopplande av EJB på många-sidan av en relationer).

```
public interface Person extends EJBLocalObject
{
  /*** Publika accessmetoder ************************************************/
  public String getPersonnummer();
  public String getNamn();
  public void setNamn(String namn);

  public Collection getBilar();
```

[58] Metoden kan även ha någon klass som implementerar gränssnittet `Collection` som returtyp, men eftersom `Collection` är ett gränssnitt så är det praktiskt att använda detta som returtyp.

```
  }  //interface Person
```

### Local home-gränssnitt

Home-gränssnittet är det samma som i exempel med relation mellan EJB Person och EJB Mantalsadress.

```
public interface PersonHome extends EJBLocalHome
{
  /*** create-metoder *******************************************************/
  public Person create(String namn, String personnr) throws CreateException;

  /*** find-metoder *********************************************************/
  public Person findByPrimaryKey(String personnr) throws FinderException;
  public Collection findAll() throws FinderException;
}  //interface PersonHome
```

### Bean-klass

Bean-klassen är också den nästan identisk med version 1b av EJB Person. Men returtypen för accessmetoden (`getBilar()`) för CMR-attribut är, liksom i local-gränssnittet, av typen `Collection` då en person kan ha flera bilar.

```
public abstract class PersonBean extends AbstractEntityBean
{
  /*** Accessmetoder för CMP-attribut ***************************************/
  public abstract String getPersonnummer();
  public abstract void setPersonnummer(String personnr);
  public abstract String getNamn();
  public abstract void setNamn(String namn);

  /*** Accessmetoder för CMR-attribut ***************************************/
  public abstract Collection getBilar();

  /*** ejbCreate- och ejbPostCreate-metoder *********************************/
  public String ejbCreate(String personnr, String namn) throws CreateException
  {
    setPersonnummer(personnr);
    setNamn(namn);
    return null;
  }  //ejbCreate()

  public void ejbPostCreate(String personnr, String namn) { }

  //Metoder i gränssnittet EntityBean implementeras av AbstractEntityBean

}  //class PersonBean
```

## Implementation av EJB Bil (version 2)

Tidigare exempel i sammanfattning har behandlat EJB Bil. Det är alltså möjligt att ha samma namn på gränssnitt/klasser i samma EJB-container så länge som de ligger i olika paket och ges olika JNDI-namn. I detta exempel så ligger detta exempels gränssnitt och klasser i paketet `bpn.relationer` och JNDI-namnet kommer bli `Bil2` (för version 2).

### Local-gränssnitt

Local-gränssnittet är mycket enkelt – accessmetoder för EJB:s CMP-attribut. Eftersom relationen är enkelriktad (från Person till Bil) så finns inga accessmetoder för CMR-attribut i local-gränssnittet. På detta sätt kan inte klienten till EJB koppla samman en EJB Bil med en EJB Person. (Man kan dock använda create-metoden i local home-gränssnittet med en instans av EJB Person för ägaren för att koppla samman en bil med en ägare/person).

```
public interface Bil extends EJBLocalObject
{
  /*** Publika accessmetoder ************************************************/
  public String getRegistreringsnummer();
  public String getMarke();
  public void setMarke(String marke);
}  //interface Bil
```

### Local home-gränssnitt

En nyhet i local home-gränssnittet är att create-metod har en parameter av typen local-gränssnitt (för EJB Person) – till create-metoden bifogas värden för alla CMP-attribut och CMR-attribut. I övrigt innehåller gränssnittet, liksom tidigare exempel, find-metoder för att hitta m.h.a. primärnyckel och för att finna alla bilar i databas.

```
public interface BilHome extends EJBLocalHome
{
  /*** create-metoder *******************************************************/
  public Bil create(String regnr, String marke, Person agare)
    throws CreateException;

  /*** find-metoder *********************************************************/
  public Bil findByPrimaryKey(String regnr) throws FinderException;
  public Collection findAll() throws FinderException;
}  //interface BilHome
```

### Bean-klass

Om relation är enkelriktad så brukar vi endast behöva CMR-attribut i EJB på ena sidan av relationen. Men ibland behöver vi CMR-attribut på båda sidor om relation för att CMP ska fungera (tekniskt sätt), d.v.s. relationen blir dubbelriktad.[59] Men för att göra relation enkelriktad för klienter av EJB så definieras endast accessmetoder i bean-klass och inte remote- och/eller local-gränssnitt.

Underliggande tabell innehåller tre kolumner (`registreringsnummer`, `marke` och `agare`), men för den tredje kolumnen definieras inget CMP-attribut – detta löses m.h.a. CMR-attributet `agare`.

I create-metod skickar vi local-gränssnitt, d.v.s. en instans, av EJB Person som parameter till metod och därmed även till `ejbCreate()` och `ejbPostCreate()`. På detta sätt så kan vi koppla samman instanser av EJB – denna koppling **måste** ske i `ejbPostCreate()`.

```
public abstract class BilBean extends AbstractEntityBean
{
  /*** Accessmetoder för CMP-attribut ***************************************/
  public abstract String getRegistreringsnummer();
  public abstract void setRegistreringsnummer(String regnr);
  public abstract String getMarke();
  public abstract void setMarke(String marke);

  /*** Accessmetoder för CMR-attribut ***************************************/
  //Relation är enkelriktad, därav endast accessmetoder för relation i bean-klass
  // och inte i remote-/local-gränssnitt
  public abstract Person getAgare();
  public abstract void setAgare(Person agare);

  /*** ejbCreate- och ejbPostCreate-metoder *********************************/
  public String ejbCreate(String regnr, String marke, Person agare)
    throws CreateException
```

[59] Detta kan vara något som är relaterat till CMP i Resin – i andra EJB-servrar så kanske detta inte krävs.

```
    {
      setRegistreringsnummer(regnr);
      setMarke(marke);
        //Parametern agare används i ejbPostCreate() för att koppla samma EJB
      return null;
    }  //ejbCreate()

    public void ejbPostCreate(String regnr, String marke, Person agare)
    {
      setAgare(agare);  //Koppla samma EJB med varandra
    }

    //Metoder i gränssnittet EntityBean implementeras av AbstractEntityBean

  }  //class BilBean
```

## Deployment descriptor för EJB:er samt relation

Eftersom ett tidigare exempel med EJB Bil har gjorts så ges EJB Bil i detta exempel JNDI-namnet `Bil2` för att undvika eventuell konflikt mellan EJB:er. Tabellen och schemanamnet för bilar heter `bilar2` (och har ett attribut – `agare` – som saknas i tabellen `bilar` i förra exemplet).

EJB Person har ett CMP-attribut för varje kolumn i underliggande tabell men, som sagt tidigare, så har EJB Bil endast CMP-attribut för två av de tre kolumnerna i underliggande tabell (`registreringsnummer` och `marke`). Den tredje kolumnen i tabellen `bilar2`, d.v.s. `agare`, är en främmande nyckel och motsvaras av CMR-attributet `agare`.

```
<ejb-jar>

  <!--  ***************** EJBer ******************* -->
  <enterprise-beans>

    <!--  ***************** Person ***************** -->
    <entity>
      <ejb-name>Person</ejb-name>
      <local-home>bpn.relationer.PersonHome</local-home>
      <local>bpn.relationer.Person</local>
      <ejb-class>bpn.relationer.PersonBean</ejb-class>

      <prim-key-class>String</prim-key-class>
      <primkey-field>personnummer</primkey-field>

      <persistence-type>Container</persistence-type>
      <reentrant>True</reentrant>

      <abstract-schema-name>personer</abstract-schema-name>
      <sql-table>personer</sql-table>

      <cmp-field><field-name>personnummer</field-name></cmp-field>
      <cmp-field><field-name>namn</field-name></cmp-field>

      <query>
        <query-method>
          <method-name>findAll</method-name>
        </query-method>
          <ejb-ql>SELECT o FROM personer o</ejb-ql>
      </query>
    </entity>

    <!--  ***************** Bil ***************** -->
    <entity>
      <ejb-name>Bil2</ejb-name>
      <local-home>bpn.relationer.BilHome</local-home>
      <local>bpn.relationer.Bil</local>
      <ejb-class>bpn.relationer.BilBean</ejb-class>

      <prim-key-class>String</prim-key-class>
      <primkey-field>registreringsnummer</primkey-field>
```

```
        <persistence-type>Container</persistence-type>
        <reentrant>True</reentrant>

        <abstract-schema-name>bilar2</abstract-schema-name>
        <sql-table>bilar2</sql-table>

        <cmp-field><field-name>registreringsnummer</field-name></cmp-field>
        <cmp-field><field-name>marke</field-name></cmp-field>

        <query>
          <query-method>
            <method-name>findAll</method-name>
          </query-method>
            <ejb-ql>SELECT o FROM bilar2 o</ejb-ql>
        </query>
      </entity>

    </enterprise-beans>

    <!--  ***************** Relationer ***************** -->
    <relationships>

      <!--  ***************** Person <-> Bil ***************** -->
      <ejb-relation>
        <ejb-relationship-role>
          <relationship-role-source>
            <ejb-name>Person</ejb-name>
          </relationship-role-source>
          <multiplicity>One</multiplicity>
          <cmr-field>
            <cmr-field-name>bilar</cmr-field-name>
          </cmr-field>
        </ejb-relationship-role>

        <ejb-relationship-role>
          <relationship-role-source>
            <ejb-name>Bil2</ejb-name>
          </relationship-role-source>
          <multiplicity>Many</multiplicity>
          <cmr-field>
            <cmr-field-name>agare</cmr-field-name>
          </cmr-field>
        </ejb-relationship-role>
      </ejb-relation>
    </relationships>

  </ejb-jar>
```

## Klient för EJB Person

Klienten till version 2 av EJB Person ser i stort sett likadan ut som den för version 1b. Skillnaden är att vi använder EJB Bil istället för EJB Mantalsadress samt att returtyp för accessmetod för CMR-attribut är `Collection` (en person kan äga flera bilar). Koden nedan har förkortats en del då den är den samma som tidigare klienter (servlets).

```
  public class Person11MServlet extends HttpServlet {
    /*** Instansvariabler ****************************************************/
    private PersonHome home = null;

    /*** Överskugga metoder i klassen HttpServlet ****************************/
    public void init() throws ServletException {
      //Hämta referens till home-gränssnitt – se tidigare exempel
    }  //init()


    public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws java.io.IOException {
      //Hämta PrintWriter, m.m. samt påbörja HTML-dokument – se tidigare exempel

      Collection personer = null;

      /*** Testa metoden findByPrimaryKey() och findAll() ******************/
```

```
      try {
        Person person = home.findByPrimaryKey("712345");
        out.println("<P>Namn för person med personnummer 712345: "
          + person.getNamn() + "</P>");

        personer = home.findAll();
      }
      catch(FinderException fe) {
        fe.printStackTrace(out);
      }

      Iterator iter = personer.iterator();

      while(iter.hasNext() ) {
        Person person = (Person)iter.next();
        out.println(person.getPersonnummer() + " - " + person.getNamn()
          + "<BR>");

        /*** Testa relation till Bil *********************************/
        try {
          Collection bilar = person.getBilar();

          if(bilar != null) {
            Iterator iter2 = bilar.iterator();

            while(iter2.hasNext()) {
              Bil bil = (Bil)iter2.next();
              out.println(" - " + bil.getRegistreringsnummer() + ", "
                + bil.getMarke() + "<BR>");
            }
          }
        }
        catch(Exception e) {
          e.printStackTrace(out);
        }

        /*************************************************************/
      }  //while

      //Skriv ut länk tillbaka till meny och avsluta HTML-dokument – se tidigare ex.
    }  //doGet()

  }  //class Person11MServlet
```

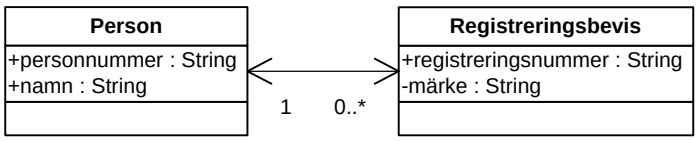
## Klient för EJB Bil

Vi kan inte fråga en EJB Bil om ägare eftersom EJB Bils local-gränssnitt inte innehåller någon publik metod att fråga om EJB i andra änden av relation, d.v.s. accessmetod för CMR-attribut. (Se även exempel med dubbelriktad 1:M-relation mellan EJB:er Person och Registreringsbevis.)

```
  public class BilServlet extends HttpServlet {
    /*** Instansvariabler ****************************************************/
    private BilHome home = null;

    /*** Överskugga metoder i klassen HttpServlet ****************************/
    public void init() throws ServletException {
      //Hämta referens till home-gränssnitt – se tidigare exempel
    }  //init()


    public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws java.io.IOException {
      //Hämta PrintWriter, m.m. samt påbörja HTML-dokument – se tidigare exempel

      Collection bilar = null;

      /*** Testa metoden findByPrimaryKey() och findAll() ******************/
      try {
        Bil bil = home.findByPrimaryKey("ABC123");
        out.println("<P>Marke för bil med registreringsnummer ABC123: "
          + bil.getMarke() + "</P>");
```

```
        bilar = home.findAll();
      }
      catch(FinderException fe) {
        fe.printStackTrace(out);
      }

      Iterator iter = bilar.iterator();

      while(iter.hasNext() ) {
        Bil bil = (Bil)iter.next();
        out.println(bil.getRegistreringsnummer() + " - "
          + bil.getMarke() + "<BR>");
      }  //while

      //Skriv ut länk tillbaka till meny och avsluta HTML-dokument – se tidigare ex.
    }  //doGet()

  }  //class BilServlet
```

### 11.2.4 Dubbelriktad 1:M-relation (och även M:1-relation)




En person kan äga flera bilar, vilket dokumenteras med registreringsbevis (1:M). Ett registreringsbevis anger vilken person som äger bilen. (Detta förutsätter att en bil endast kan ägas av en person.) Om man vänder på relationen (M:1) så är det samma sak, d.v.s. ett specialfall.

Nedan visas SQL-kod för att skapa tabellerna och lägga till testdata i tabellerna. Om du redan skapat tabellen `personer` så behöver du inte göra det igen. Tabellen `registreringsbevis` ser identisk ut med `bilar2`, d.v.s. implementationen av EJB Registreringsbevis kommer vara väldigt lik EJB Bil (version 2).

```
CREATE TABLE personer(
  personnummer VARCHAR(10) PRIMARY KEY,
  namn VARCHAR(50));

INSERT INTO personer VALUES("712345", "Oscar");
INSERT INTO personer VALUES("812345", "Sune");
INSERT INTO personer VALUES("912345", "Pelle");
INSERT INTO personer VALUES("012345", "Niklas");

CREATE TABLE registreringsbevis(
  registreringsnummer VARCHAR(6) PRIMARY KEY,
  marke VARCHAR(20),
  agare VARCHAR(10));

INSERT INTO registreringsbevis VALUES('ABC123', 'BMW 525 iX', '712345');
INSERT INTO registreringsbevis VALUES('DEF456', 'Audi 100 E', '712345');
INSERT INTO registreringsbevis VALUES('GHI789', 'Opel Kadett', '812345');
```

I nedanstående kod så har namn på paket (d.v.s. raden `package bpn.relationer;`) utelämnats liksom import-satser. Se tidigare exempel för vad som behöver (kan behöva) importeras.

## Implementation av EJB Person (version 3)

För enkelhetens skull så visas endast kod som behövs för relation mellan Person och Registreringsbevis. Om du redan gjort en EJB Person enligt tidigare exempel räcker det med att du lägger till kod som saknas. (Eventuellt kan en ZIP-fil med komplett kod för EJB Person laddas ner från samma webbsida som du fann denna sammanfattning.)

### Local-gränssnitt

Local-gränssnittet ser nästan likadant ut som version 1 och 2 (bortsätt från namn på metoder) men har även fått en metod `addRegistreringsbevis()` för att kunna lägga till registreringsbevis för en person. Typen på metods parameter är local-gränssnitt för EJB Registreringsbevis. (En annan lösning vore att ha två strängar som parametrar – registreringsnummer och märke – för att kunna hitta eller skapa ett ny EJB Registreringsbevis. Detta gör bl.a. att klient inte behöver ha någon kännedom om hur man skapar ett registreringsbevis.)

```
public interface Person extends EJBLocalObject
{
  /*** Publika accessmetoder ************************************************/
  public String getPersonnummer();
  public String getNamn();
  public void setNamn(String namn);

  public Collection getRegistreringsbevis();
  public void addRegistreringsbevis(Registreringsbevis regbevis);
}  //interface Person
```

### Local home-gränssnitt

Local home-gränssnittet är helt oförändrat jämfört med version 1 och 2 av EJB Person. Man skulle dock kunna lägga till en eller två create-metoder till med sträng för registreringsnummer eller EJB Registreringsbevis som ytterligare parameter till metoder. På detta sätt skulle vi kunna hitta/skapa en instans av EJB Registreringsbevis samtidigt som vi skapar en instans av EJB Person. Hur många create-metoder vi definierar, eller parametrar till create-metoder som behövs, beror på hur vi vill kunna skapa instanser av EJB:er. Viktigast är dock att vi bifogar värden på kolumner med restriktionen (*constraint*) `NOT NULL` i tabell.[60] (Se även *Implementation av EJB Registreringsbevis* för mer om create-metoder och relationer.)

```
public interface PersonHome extends EJBLocalHome
{
  /*** create-metoder *******************************************************/
  public Person create(String namn, String personnr) throws CreateException;

  /*** find-metoder *********************************************************/
  public Person findByPrimaryKey(String personnr) throws FinderException;
  public Collection findAll() throws FinderException;
}  //interface PersonHome
```

### Bean-klass

Bean-klassen ser också den i stort sett likadant som i version 1 och 2 av EJB Person. Men eftersom en person kan äga flera bilar så kan personen ha flera registreringsbevis, och vi lägger till metoden `addRegistreringsbevis()`. Detta är en metod som vi själva måste implementera (i motsatts till set- och get-metoder som EJB-server implementerar åt oss). I metoden frågar vi EJB efter lista (vektor) med registreringsbevis och lägger till det nya registreringsbeviset i vektorn (med metoden `add()`).

```
public abstract class PersonBean extends AbstractEntityBean
{
  /*** Accessmetoder för CMP-attribut ***************************************/
  public abstract String getPersonnummer();
  public abstract void setPersonnummer(String personnr);
```

[60] Om en kolumn är av typen ”räknare” (automatiskt ges ett värde) så behöver vi inte bifoga värde för kolumnen.

```
    public abstract String getNamn();
    public abstract void setNamn(String namn);

    /*** Accessmetoder för CMR-attribut ************************************/
    public abstract Collection getRegistreringsbevis();
    public void addRegistreringsbevis(Registreringsbevis regbevis)
    {
      Collection regbevisColl = this.getRegistreringsbevis(); //Hämta vektor
      regbevisColl.add(regbevis);                             //Lägg till i vektor
    }  //addRegistreringsbevis()

    /*** ejbCreate- och ejbPostCreate-metoder *******************************/
    public String ejbCreate(String personnr, String namn) throws CreateException
    {
      setPersonnummer(personnr);
      setNamn(namn);
      return null;
    }  //ejbCreate()

    public void ejbPostCreate(String personnr, String namn) { }

    //Metoder i gränssnittet EntityBean implementeras av AbstractEntityBean

  }  //class PersonBean
```

## Implementation av EJB Registreringsbevis

I denna EJB kommer det (något överflödigt?) finnas tre sätt att skapa en instans av EJB, d.v.s. tre create-metoder:

- Enbart skapa instans av EJB Registreringsbevis (`create(String, String)`). (En intressant fråga är om det kan existera ett registreringsbevis utan en ägare… D.v.s. denna metod kanske inte ska existera. ☺)
- Skapa instans av EJB Registreringsbevis, hitta instans av EJB Person (för personnummer i tredje parametern) samt använda CMR-attributet `agare` för att koppla instans av EJB Registreringsbevis med EJB Person (`create(String, String, String)`). Det förutsätts att det finns en EJB Person med personnummer som skickas som tredje parameter till create-metod (annars slängs ett undantag).
- Skapa instans av EJB Registreringsbevis samt använda CMR-attributet `agare` för att koppla instans med EJB Person (`create(String, String, Person)`).

### Local-gränssnitt

I local-gränssnittet finns accessmetoder för CMP-attribut och CMR-attributet `agare`.

```
  public interface Registreringsbevis extends EJBLocalObject
  {
    /*** Publika accessmetoder ***************************************************/
    public String getRegistreringsnummer();
    public String getMarke();
    public void setMarke(String marke);

    public Person getAgare ();
    public void setAgare(Person agare);
  }  //interface Registreringsbevis
```

### Local home-gränssnitt

Local home-gränssnittet innehåller create-metoder, enligt tidigare beskrivning, samt find-metoder enligt tidigare exempel.

```
  public interface RegistreringsbevisHome extends EJBLocalHome
```

```
{
  /*** create-metoder *******************************************************/
  public Registreringsbevis create(String regnr, String marke)
    throws CreateException;
  public Registreringsbevis create(String regnr, String marke, String agare)
    throws CreateException;
  public Registreringsbevis create(String regnr, String marke, Person agare)
    throws CreateException;

  /*** find-metoder *********************************************************/
  public Registreringsbevis findByPrimaryKey(String regnr) throws FinderException;
  public Collection findAll() throws FinderException;
}  //interface RegistreringsbevisHome
```

### Bean-klass

Bean-klassen blir något längre då vi har tre create-metoder, d.v.s. vi måste även ha tre ejbCreate-metoder liksom ejbPostCreate-metoder. I de två sista ejbPostCreate-metoden så visas även hur vi kopplar samma (”relaterar”) två EJB. Observera att vi **inte** får koppla samman två EJB:er i ejbCreate-metoder – det **måste** ske i ejbPostCreate-metoder för att EJB-container ska få en möjlighet att skapa en instans av primärnyckelsklassen![61]

I create-metod nummer 2 måste vi leta upp instans av EJB Person som motsvaras av personnumret som skickades som parameter till metod. Här visas också hur vi kan fånga ett undantag och slänga vidare det med en `EJBException` samt en (kanske) mer förklarande text.

```
public abstract class RegistreringsbevisBean extends AbstractEntityBean
{
  /*** Accessmetoder för CMP-attribut ***************************************/
  public abstract String getRegistreringsnummer();
  public abstract void setRegistreringsnummer(String regnr);
  public abstract String getMarke();
  public abstract void setMarke(String marke);

  /*** Accessmetoder för CMR-attribut ***************************************/
  public abstract Person getAgare();
  public abstract void setAgare(Person agare);

  /*** ejbCreate- och ejbPostCreate-metoder *********************************/
  //create-metod 1 – koppla inte samman med EJB Person
  public String ejbCreate(String regnr, String marke)
    throws CreateException
  {
    setRegistreringsnummer(regnr);
    setMarke(marke);
    return null;
  }  //ejbCreate(String, String)

  public void ejbPostCreate(String regnr, String marke) { }

  //create-metod 2 – koppla samman med EJB Person m.h.a. FK i tabell
  public String ejbCreate(String regnr, String marke, String agare)
    throws CreateException
  {
    Person person = null;

    try
    {
      PersonHome home = null;
      Context initial =
        (Context)new InitialContext().lookup("java:comp/env/ejb");

      home = (PersonHome)initial.lookup("Person");

      person = home.findByPrimaryKey(agare);
    }
```

[61] Detta är mest relevant då vi använder en ”räknare” (d.v.s. att värde på genereras av databashanterare) för primärnyckel i tabell.

```
      catch(NamingException ne)
      {
        throw new EJBException("Kunde inte hitta local home för EJB Person", ne);
      }
      catch(FinderException fe)
      {
        throw new EJBException("Kunde inte hitta instans av EJB Person", fe);
      }

      setRegistreringsnummer(regnr);
      setMarke(marke);
      return null;
    }  //ejbCreate(String, String, String)

    public void ejbPostCreate(String regnr, String marke, String agare)
    {
      Person person = null;

      try
      {
        PersonHome home = null;
        Context initial =
          (Context)new InitialContext().lookup("java:comp/env/ejb");

        home = (PersonHome)initial.lookup("Person");

        person = home.findByPrimaryKey(agare);
      }
      catch(NamingException ne)
      {
        throw new EJBException("Kunde inte hitta local home för EJB Person", ne);
      }
      catch(FinderException fe)
      {
        throw new EJBException("Kunde inte hitta instans av EJB Person", fe);
      }

      setAgare(person);  //Sätt värde för främmande nyckel i tabell
    }  //ejbPostCreate(String, String, String)

    //create-metod 3 – koppla samman med EJB Person m.h.a. EJB Person
    public String ejbCreate(String regnr, String marke, Person agare)
      throws CreateException
    {
      setRegistreringsnummer(regnr);
      setMarke(marke);
      //Vi måste använda ejbPostCreate() för att sätta relation med EJB Person!
      return null;
    }  //ejbCreate(String, String, Person)

    public void ejbPostCreate(String regnr, String marke, Person agare)
    {
      setAgaren(agare);  //Sätt relation efter att EJB skapats
    }  //ejbPostCreate(String, String, Person)

    //Metoder i gränssnittet EntityBean implementeras av AbstractEntityBean

  }  //class RegistreringsbevisBean
```

## Deployment descriptor för EJB:er samt relation

I deployment descriptor beskrivs de två EJB:erna samt relationen mellan dem. Eftersom relation är dubbelriktad så lägger vi till CMR-attribut på båda sidor i relation.

```
  <ejb-jar>

    <!--  ***************** EJBer ******************* -->
    <enterprise-beans>

      <!--  ***************** Person ***************** -->
      <entity>
        <ejb-name>Person</ejb-name>
        <local-home>bpn.relationer.PersonHome</local-home>
```

```
        <local>bpn.relationer.Person</local>
        <ejb-class>bpn.relationer.PersonBean</ejb-class>

        <prim-key-class>String</prim-key-class>
        <primkey-field>personnummer</primkey-field>

        <persistence-type>Container</persistence-type>
        <reentrant>True</reentrant>

        <abstract-schema-name>personer</abstract-schema-name>
        <sql-table>personer</sql-table>

        <cmp-field><field-name>personnummer</field-name></cmp-field>
        <cmp-field><field-name>namn</field-name></cmp-field>

        <query>
          <query-method>
            <method-name>findAll</method-name>
          </query-method>
            <ejb-ql>SELECT o FROM personer o</ejb-ql>
        </query>
      </entity>

      <!--  ***************** Registreringsbevis ***************** -->
      <entity>
        <ejb-name>Registreringsbevis</ejb-name>
        <local-home>bpn.relationer.RegistreringsbevisHome</local-home>
        <local>bpn.relationer.Registreringsbevis</local>
        <ejb-class>bpn.relationer.RegistreringsbevisBean</ejb-class>

        <prim-key-class>String</prim-key-class>
        <primkey-field>registreringsnummer</primkey-field>

        <persistence-type>Container</persistence-type>
        <reentrant>True</reentrant>

        <abstract-schema-name>registreringsbevis</abstract-schema-name>
        <sql-table>registreringsbevis</sql-table>

        <cmp-field><field-name>registreringsnummer</field-name></cmp-field>
        <cmp-field><field-name>marke</field-name></cmp-field>

        <query>
          <query-method>
            <method-name>findAll</method-name>
          </query-method>
            <ejb-ql>SELECT o FROM registreringsbevis o</ejb-ql>
        </query>
      </entity>

    </enterprise-beans>

    <!--  ***************** Relationer ***************** -->
    <relationships>

      <!--  ***************** Person <-> Registreringsbevis ***************** -->
      <ejb-relation>
        <ejb-relationship-role>
          <relationship-role-source>
            <ejb-name>Person</ejb-name>
          </relationship-role-source>
          <multiplicity>One</multiplicity>
          <cmr-field>
            <cmr-field-name>registreringsbevis</cmr-field-name>
          </cmr-field>
        </ejb-relationship-role>

        <ejb-relationship-role>
          <relationship-role-source>
            <ejb-name>Registreringsbevis</ejb-name>
          </relationship-role-source>
          <multiplicity>Many</multiplicity>
          <cmr-field>
            <cmr-field-name>agare</cmr-field-name>
          </cmr-field>
        </ejb-relationship-role>
      </ejb-relation>
```

```
    </relationships>

  </ejb-jar>
```

## Klient för EJB Person

Klienten för EJB Person ser i stort sett likadan ut som den i förra relationen (med EJB Bil). Därför visas endast kod för hur man kan relatera den till EJB Registreringsbevis med metoden `addRegistreringsbevis()`.

I nedanstående kod visas hur vi kan lägga till ett registreringsbevis till en person. Variabeln `person` förutsätts referera till en instans av EJB Person. Sist av allt tas registreringsbevis bort from EJB-server (och databas) och därmed även relationen mellan personen och registreringsbeviset.

```
  out.println("<p>Skapar nytt registreringsbevis för person:</p>");
  Context initial = (Context)new InitialContext().lookup("java:comp/env/ejb");

  RegistreringsbevisHome regHome =
    (RegistreringsbevisHome)initial.lookup("Registreringsbevis");

  Registreringsbevis regbevis = regHome.create("LMN123", "Saab 9-5");
  person.addRegistreringsbevis(regbevis);
  out.println("<p> - Nytt registreringsbevis för person skapat.</p>");
  regbevis.remove();
```

## Klient för EJB Registreringsbevis

I denna klient kommer vi även använda EJB Person, d.v.s. vi deklarerar instansvariabler för local home-gränssnitt för båda EJB:er. I init-metoden hämtar vi referenser till båda local home-gränssnitt. I klient testas alla tre create-metoder – registreringsbevis som tas bort i slutet på `doGet()`.

```
  public class RegistreringServlet extends HttpServlet {
    /*** Instansvariabler ****************************************************/
    private RegistreringsbevisHome home = null;
    private PersonHome phome = null;

    /*** Överskugga metoder i klassen HttpServlet ****************************/
    public void init() throws ServletException {
      try {
        Context initial =
          (Context)new InitialContext().lookup("java:comp/env/ejb");

        home = (RegistreringsbevisHome)initial.lookup("Registreringsbevis");
        phome = (PersonHome)initial.lookup("Person");
      }
      catch(NamingException ne) {
        throw new ServletException(ne);
      }
    }  //init()

    public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws java.io.IOException {
      PrintWriter out = null;
      res.setContentType("text/html");
      out = res.getWriter();

      //Påbörja HTML-dokument och skriv ut rubrik
      out.println(
        "<HTML>\n<HEAD>\n<TITLE>Registreringsbevis</TITLE>\n</HEAD>\n<BODY>");
      out.println("<H1>Registreringsbevis</H1>");

      Registreringsbevis regbevis = null, regbevis1 = null, regbevis2 = null,
        regbevis3 = null;
```

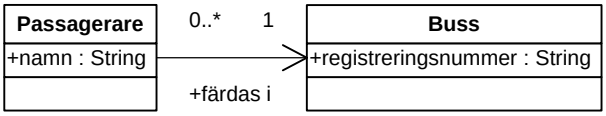
```
      Collection registreringsbevisColl = null;

      /*** Testa create- och find-metoder **********************************/
      try {
        Person agare = phome.findByPrimaryKey("712345");
        regbevis1 = home.create("IJK123", "Volvo V70");
        regbevis2 = home.create("LMN123", "Saab 9-5", "712345");
        regbevis3 = home.create("OPQ123", "Toyota Camry", agare);

        regbevis = home.findByPrimaryKey("ABC123");
        out.println(
          "<P>Marke för registreringsbevis med registreringsnummer ABC123: "
          + regbevis.getMarke() + "</P>");

        registreringsbevisColl = home.findAll();
      }
      catch(FinderException fe) {
        fe.printStackTrace(out);
      }
      catch(CreateException ce) {
        ce.printStackTrace(out);
      }

      out.println("<P>Registreringsbevis och ägare till fordon:<BR>");

      //Om registreringsbevis kunde hämtas, d.v.s. metoden findAll() "fungerade"
      if(registreringsbevisColl != null) {
        Iterator iter = registreringsbevisColl.iterator();

        while(iter.hasNext() ) {
          regbevis = (Registreringsbevis)iter.next();
          out.println(regbevis.getRegistreringsnummer() + " - "
            + regbevis.getMarke());
          Person agare = regbevis.getAgare();
          if(agare != null)
            out.println(" ägs av " + agare.getPersonnummer() + " "
              + agare.getNamn());
          else
            out.println("- ägare saknas för registreringsbevis");

          out.println("<BR>");
        }  //while
      }  //if(registreringsbevisColl != null)
      else
        out.println("Metoden findAll() fungerade inte...");

      /*** Ta bort instanser av EJB som skapats i servlet ******************/
      try {
        regbevis1.remove(); regbevis2.remove(); regbevis3.remove();
      }
      catch(RemoveException re) {
        out.println("<p><b>Fel:</b> Kunde inte ta bort instanser av EJB!</p>");
      }
      catch(NullPointerException npe) {
        out.println("<p><b>Fel:</b> Instanser av EJB har inte skapats!</p>");
      }

      //Skriv ut länk tillbaka till meny och avsluta HTML-dokument
      out.println("<P><A HREF=\"../relationer.html\">Meny</A></P>");
      out.println("</HTML>\n</HTML>");
    }  //doGet()

  }  //class RegistreringServlet
```

Koden ser komplex ut men resultatet är mindre så – resultatet visas i bild till höger.

### 11.2.5 Enkelriktad M:1-relation (många till en)

Sett ur datamodelleringens synvinkel så kan denna relation kännas udda, d.v.s. denna typ av relation kan kännas som samma som en 1:M-relation. Men i OO-modellering så kan ett objekt (klass) på en ”1-sida” vara relaterat med många objekt (på ”M-sidan”) utan att själv känna till objekten.

Flera passagerare kan färdas i en (stads-)buss[62], men bussen känner inte till vilka passagerarna är. Passagerare på samma buss känner inte (eller har oftast inte behov av att känna) andra passagerare på bussen (inte alla i de flesta fall i.a.f. ☺).




Nedan visas SQL-kod för att skapa tabellerna och lägga till testdata.

```
CREATE TABLE bussar(
  registreringsnummer CHAR(6) PRIMARY KEY);

INSERT INTO bussar VALUES('ABC123');
INSERT INTO bussar VALUES('DEF456');
INSERT INTO bussar VALUES('GHI789');

CREATE TABLE passagerare(
  namn VARCHAR(20) PRIMARY KEY,
  buss CHAR(6));

INSERT INTO passagerare VALUES('Sune', 'ABC123');
INSERT INTO passagerare VALUES('Oscar', 'ABC123');
INSERT INTO passagerare VALUES('Nisse', 'ABC123');
INSERT INTO passagerare VALUES('Kalle', 'DEF456');
```

Observera att tabellen `bussar` måste skapas innan tabellen `passagerare` om referensintegritet mellan tabeller används – kolumnen `buss` i tabellen `passagerare` är en främmande nyckel!

## Implementation av EJB Passagerare

EJB Passagerare är en mycket enkel EJB – endast ett (CMP-)attribut: `namn` (som också är primärnyckel i underliggande tabell). Detta kanske inte är en realistisk lösning då flera passagerare kan heta samma sak, men detta är ett exempel. ☺ Eftersom passagerare känner till bussen dom åker på så finns accessmetod för CMR-attributet `buss`.

### Local-gränssnitt

```
public interface Passagerare extends EJBLocalObject
{
   /*** Publika accessmetoder ************************************************/
   public String getNamn();

   public Buss getBuss();
   public void setBuss(Buss buss);
}  //interface Passagerare
```

### Local home-gränssnitt

Local home-gränssnittet är i stort sett en upprepning av tidigare exempel.

```
public interface PassagerareHome extends EJBLocalHome
{
   /*** create-metoder *******************************************************/
   public Passagerare create(String namn) throws CreateException;

   /*** find-metoder *********************************************************/
   public Passagerare findByPrimaryKey(String namn) throws FinderException;
```

[62] Om det varit en abonnerad buss så kanske man haft en passagerarlista, d.v.s. bussen känt till passagerarna.

```
      public Collection findAll() throws FinderException;
   }  //interface PassagerareHome
```

### Bean-klass

Inte heller bean-klassen är så annorlunda mot tidigare exempel.

```
   public abstract class PassagerareBean extends AbstractEntityBean
   {
      /*** Accessmetoder för CMP-attribut ***************************************/
      public abstract String getNamn();
      public abstract void setNamn(String namn);

      /*** Accessmetoder för CMR-attribut ***************************************/
      public abstract Buss getBuss();
      public abstract void setBuss(Buss buss);

      /*** ejbCreate- och ejbPostCreate-metoder *********************************/
      public String ejbCreate(String namn) throws CreateException
      {
         setNamn(namn);
         return null;
      }  //ejbCreate()

      public void ejbPostCreate(String namn) { }

      //Metoder i gränssnittet EntityBean implementeras av AbstractEntityBean

   }  //class PassagerareBean
```

## Implementation av EJB Buss

Även EJB Buss är mycket simpel – ett (CMP-)attribut: `registreringsnummer`. Eftersom relationen är enkelriktad så finns inga CMR-attribut.

### Local-gränssnitt

```
   public interface Buss extends EJBLocalObject
   {
      /*** Publika accessmetoder ************************************************/
      public String getRegistreringsnummer();
   }  //interface Buss
```

### Local home-gränssnitt

```
   public interface BussHome extends EJBLocalHome
   {
      /*** create-metoder *******************************************************/
      public Buss create(String regnr) throws CreateException;

      /*** find-metoder *********************************************************/
      public Buss findByPrimaryKey(String regnr) throws FinderException;
      public Collection findAll() throws FinderException;
   }  //interface BussHome
```

### Bean-klass

Eftersom främmande nyckel finns i EJB som ska ha CMR-attribut så behöver vi inte något CMR-attribut, och därmed accessmetoder för CMR-attribut, i denna EJB.

```
   public abstract class BussBean extends AbstractEntityBean
   {
      /*** Accessmetoder för CMP-attribut ***************************************/
      public abstract String getRegistreringsnummer();
      public abstract void setRegistreringsnummer(String regnr);

      /*** Accessmetoder för CMR-attribut ***************************************/
      //Relation är enkelriktad, därav inga accessmetoder för relation här

      /*** ejbCreate- och ejbPostCreate-metoder *********************************/
      public String ejbCreate(String regnr) throws CreateException
      {
```

```
        setRegistreringsnummer(regnr);
        return null;
    }  //ejbCreate()

    public void ejbPostCreate(String regnr) { }

    //Metoder i gränssnittet EntityBean implementeras av AbstractEntityBean

}  //class BussBean
```

# Deployment descriptor för EJB:er samt relation

```
<ejb-jar>

  <!--  ***************** EJBer ******************* -->
  <enterprise-beans>

    <!--  ***************** Passagerare ***************** -->
    <entity>
      <ejb-name>Passagerare</ejb-name>
      <local-home>bpn.relationer.PassagerareHome</local-home>
      <local>bpn.relationer.Passagerare</local>
      <ejb-class>bpn.relationer.PassagerareBean</ejb-class>

      <prim-key-class>String</prim-key-class>
      <primkey-field>namn</primkey-field>

      <persistence-type>Container</persistence-type>
      <reentrant>True</reentrant>

      <abstract-schema-name>passagerare</abstract-schema-name>
      <sql-table>passagerare</sql-table>

      <cmp-field><field-name>namn</field-name></cmp-field>

      <query>
        <query-method>
          <method-name>findAll</method-name>
        </query-method>
          <ejb-ql>SELECT o FROM passagerare o</ejb-ql>
      </query>
    </entity>

    <!--  ***************** Buss ***************** -->
    <entity>
      <ejb-name>Buss</ejb-name>
      <local-home>bpn.relationer.BussHome</local-home>
      <local>bpn.relationer.Buss</local>
      <ejb-class>bpn.relationer.BussBean</ejb-class>

      <prim-key-class>String</prim-key-class>
      <primkey-field>registreringsnummer</primkey-field>

      <persistence-type>Container</persistence-type>
      <reentrant>True</reentrant>

      <abstract-schema-name>bussar</abstract-schema-name>
      <sql-table>bussar</sql-table>

      <cmp-field><field-name>registreringsnummer</field-name></cmp-field>

      <query>
        <query-method>
          <method-name>findAll</method-name>
        </query-method>
          <ejb-ql>SELECT o FROM bussar o</ejb-ql>
      </query>
    </entity>

  </enterprise-beans>

  <!--  ***************** Relationer ***************** -->
  <relationships>

    <!--  ***************** Passagerare -> Buss ***************** -->
    <ejb-relation>
```

```
        <ejb-relationship-role>
          <relationship-role-source>
            <ejb-name>Passagerare</ejb-name>
          </relationship-role-source>
          <multiplicity>Many</multiplicity>
          <cmr-field>
            <cmr-field-name>buss</cmr-field-name>
          </cmr-field>
        </ejb-relationship-role>

        <ejb-relationship-role>
          <relationship-role-source>
            <ejb-name>Buss</ejb-name>
          </relationship-role-source>
          <multiplicity>One</multiplicity>
        </ejb-relationship-role>
      </ejb-relation>

    </relationships>

  </ejb-jar>
```

## Klient för EJB Passagerare

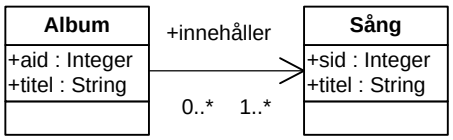
Eftersom klienten för EJB Buss i stort sett ser likadan ut som övriga klienter för andra EJB så redovisas endast klient för EJB Passagerare då den innehåller CMR-attributet.

```
  public class PassagerareServlet extends HttpServlet
  {
     /*** Instansvariabler ****************************************************/
     private PassagerareHome home = null;

     /*** Överskugga metoder i klassen HttpServlet ****************************/
     public void init() throws ServletException
     {
        try
        {
           Context initial =
              (Context)new InitialContext().lookup("java:comp/env/ejb");

           home = (PassagerareHome)initial.lookup("Passagerare");
        }
        catch(NamingException ne)
        {
           throw new ServletException(ne);
        }
     }  //init()


     public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws java.io.IOException
     {
        PrintWriter out = null;
        res.setContentType("text/html");
        out = res.getWriter();

        //Påbörja HTML-dokument och skriv ut rubrik
        out.println("<HTML>\n<HEAD>\n<TITLE>Passagerare</TITLE>\n</HEAD>\n<BODY>");
        out.println("<H1>Passagerare</H1>");

        Collection passagerarColl = null;

        /*** Testa metoden findByPrimaryKey() och findAll() ******************/
        try
        {
           Passagerare passagerare = home.findByPrimaryKey("Sune");
           out.println("<P>Namn för person med namn Sune: "
              + passagerare.getNamn() + "</P>");

           passagerarColl = home.findAll();
        }
        catch(FinderException fe)
        {
```

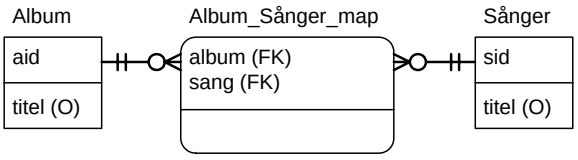
```
            out.println("<PRE>");
            fe.printStackTrace(out);
            out.println("</PRE>");
         }

         //Om passagerare kunde hämtas, d.v.s. metoden findAll() "fungerade"
         if(passagerarColl != null)
         {
            Iterator iter = passagerarColl.iterator();

            while(iter.hasNext() )
            {
               Passagerare passagerare = (Passagerare)iter.next();
               out.println(passagerare.getNamn() + "<BR>");

               /*** Testa relation till Buss ************************/
               Buss buss = passagerare.getBuss();
               out.println(" - " + buss.getRegistreringsnummer() + "<BR>");
               /*************************************************************/
            }  //while
         }  //if(passagerarColl != null)
         else
            out.println("Metoden findAll() fungerade inte...");

         //Skriv ut länk tillbaka till meny och avsluta HTML-dokument
         out.println("<P><A HREF=\"../relationer.html\">Meny</A></P>");
         out.println("</HTML>\n</HTML>");
      }  //doGet()

   }  //class PassagerareServlet
```

## 11.2.6 Enkelriktad M:N-relation (många till många)

Ett album innehåller flera sånger och en sång kan finnas på flera album, men en sång vet inte vilka album den finns på.




I många till många relationer så skiljer sig OO-modeller oftast från datamodeller. Klassdiagrammet ovan, d.v.s. relationen, brukar implementeras med en kopplingstabell i datamodeller, d.v.s. vi kan skapa tabellerna `album` och `sanger` samt kopplingstabellen `album_sanger_map`[63]. Nedan visas SQL-koden för att skapa tabellerna samt infoga testdata i tabellerna. (Det reserverade ordet `auto_increment` innebär räknare i MySQL, vilket används för primärnyckel på båda tabeller.)




```
CREATE TABLE album(
  aid int auto_increment PRIMARY KEY,
  titel varchar(60));

INSERT INTO album(titel) VALUES('Brothers in arms');
INSERT INTO album(titel) VALUES('Bigger, better, faster, more!');
INSERT INTO album(titel) VALUES('The great rock''n''roll swindle');
INSERT INTO album(titel) VALUES('Never mind the bullocks');

CREATE TABLE sanger(
  sid int auto_increment PRIMARY KEY,
  titel varchar(60));

INSERT INTO sanger(titel) VALUES('Money for nothing');
INSERT INTO sanger(titel) VALUES('What''s up');
INSERT INTO sanger(titel) VALUES('God save the Queen');
INSERT INTO sanger(titel) VALUES('Walk of life');
```

[63] Suffixet ”`_map`” används för att följa namnstandarden i Resins exempel.

```
INSERT INTO sanger(titel) VALUES('Spaceman');
INSERT INTO sanger(titel) VALUES('Road runner');
INSERT INTO sanger(titel) VALUES('So far away');

CREATE TABLE album_sanger_map(
  album int NOT NULL,
  sang int NOT NULL,
  CONSTRAINT PRIMARY KEY(album, sang));

INSERT INTO album_sanger_map VALUES(1, 1);
INSERT INTO album_sanger_map VALUES(1, 4);
INSERT INTO album_sanger_map VALUES(1, 7);
INSERT INTO album_sanger_map VALUES(2, 2);
INSERT INTO album_sanger_map VALUES(2, 5);
INSERT INTO album_sanger_map VALUES(3, 3);
INSERT INTO album_sanger_map VALUES(3, 6);
INSERT INTO album_sanger_map VALUES(4, 3);
```

## Implementation av EJB Album

Ett album bör innehålla namnet på artisten/-erna, eller som i nästa relationstyp, åtminstone relaterad till en EJB Artist.

### Local-gränssnitt

För att identifiera album används ett index (`aid`). I tabellen har kolumnen typen int (eller motsvarande om annan databashanterare än MySQL), men i EJB använder vi typen (klassen) Integer som primärnyckelsklass. Detta kräver (oftast[64]) en del konvertering när klienter använder EJB.

```
public interface Album extends EJBLocalObject
{
  /*** Publika accessmetoder ************************************************/
  public Integer getAid();
  public String getTitel();
  public void setTitel(String titel);

  public Collection getSanger();
}  //interface Album
```

### Local home-gränssnitt

Som parameter till create-metod finns endast sångens titel då primärnyckeln i underliggande tabell är en räknare. Och eftersom primärnyckeln (räknaren) i underliggande tabell är av typen `int` så måste vi använda *wrapper*-klassen `Integer` som primärnyckelsklass och därmed i metoden `findByPrimaryKey()`.

```
public interface AlbumHome extends EJBLocalHome
{
  /*** create-metoder *******************************************************/
  public Album create(String titel) throws CreateException;

  /*** find-metoder *********************************************************/
  public Album findByPrimaryKey(Integer id) throws FinderException;
  public Collection findAll() throws FinderException;
}  //interface AlbumHome
```

### Bean-klass

Bean-klassen är en "upprepning" av tidigare exempel, d.v.s. inget nytt. Men eftersom albumindex är en räknare så bör (kan) vi avstå från en set-metod för attributet.

```
public abstract class AlbumBean extends AbstractEntityBean
{
  /*** Accessmetoder för CMP-attribut ***************************************/
```

[64] I.o.m. Java version 1.5 så introducerades begreppen *boxing* och *unboxing* för konvertering mellan enkla typer (som int) och motsvarande wrapper-klass (som Integer). Jag har inte testat exakt hur det fungerar…

```
    public abstract Integer getAid();
    public abstract String getTitel();
    public abstract void setTitel(String titel);

    /*** Accessmetoder för CMR-attribut ************************************/
    public abstract Collection getSanger();

    /*** ejbCreate- och ejbPostCreate-metoder ******************************/
    public Integer ejbCreate(String titel) throws CreateException
    {
      setTitel(titel);
      return null;
    }  //ejbCreate()

    public void ejbPostCreate(String titel) { }

    //Metoder i gränssnittet EntityBean implementeras av AbstractEntityBean

  }  //class AlbumBean
```

# Implementation av EJB Sang

## Local-gränssnitt

Local-gränssnittet är en "upprepning" av tidigare exempel, d.v.s. inget nytt. Och även här används ett index (`sid`) för att identifiera en sång.

```
  public interface Sang extends EJBLocalObject
  {
    /*** Publika accessmetoder ***************************************************/
    public Integer getSid();
    public String getTitel();
    public void setTitel(String titel);
  }  //interface Sang
```

## Local home-gränssnitt

Även local-home är en "upprepning" av tidigare exempel, d.v.s. inget nytt.

```
  public interface SangHome extends EJBLocalHome
  {
    /*** create-metoder *********************************************************/
    public Sang create(String titel) throws CreateException;

    /*** find-metoder ***********************************************************/
    public Sang findByPrimaryKey(Integer id) throws FinderException;
    public Collection findAll() throws FinderException;
  }  //interface SangHome
```

## Bean-klass

Och även bean-klassen är en "upprepning" av tidigare exempel, d.v.s. inget nytt.

```
  public abstract class SangBean extends AbstractEntityBean
  {
    /*** Accessmetoder för CMP-attribut ************************************/
    public abstract Integer getSid();
    public abstract String getTitel();
    public abstract void setTitel(String titel);

    /*** Accessmetoder för CMR-attribut ************************************/
    public abstract Collection getAlbum();

    /*** ejbCreate- och ejbPostCreate-metoder ******************************/
    public Integer ejbCreate(String titel) throws CreateException
    {
      setTitel(titel);
      return null;
    }  //ejbCreate()

    public void ejbPostCreate(String titel) { }

    //Metoder i gränssnittet EntityBean implementeras av AbstractEntityBean
```

```
} //class SangBean
```

## Deployment descriptor för EJB:er samt relation

I beskrivningen av relationen använder vi taggen `<ejb-relation-name>` för att namnge kopplingstabell (`album_sanger_map`) mellan tabellerna `album` och `sanger`.[65] Och precis som med 1:M-relationen så måste vi ha ett CMR-attribut på båda sidor om relationen (men vi definierar inte en accessmetod till CMR-attribut i EJB Sångs local-gränssnitt). En skillnad är dock att CMR-attributets namn inte stämmer med SQL-kolumnen i deployment descriptor nedan. Detta kommer sig av att vi har en kopplingstabell mellan tabellerna.

Detta exempel visar även hur vi kan ange returtyp för CMR-attributs accessmetoder m.h.a. taggen `<cmr-field-type>`.

```
<ejb-jar>

  <!--  ***************** EJBer ******************* -->
  <enterprise-beans>

    <!--  ***************** Album ***************** -->
    <entity>
      <ejb-name>Album</ejb-name>
      <local-home>bpn.relationer.AlbumHome</local-home>
      <local>bpn.relationer.Album</local>
      <ejb-class>bpn.relationer.AlbumBean</ejb-class>

      <prim-key-class>Integer</prim-key-class>
      <primkey-field>aid</primkey-field>

      <persistence-type>Container</persistence-type>
      <reentrant>True</reentrant>

      <abstract-schema-name>album</abstract-schema-name>
      <sql-table>album</sql-table>

      <cmp-field><field-name>titel</field-name></cmp-field>

      <query>
        <query-method>
          <method-name>findAll</method-name>
        </query-method>
          <ejb-ql>SELECT o FROM album o</ejb-ql>
      </query>
    </entity>

    <!--  ***************** Sang ***************** -->
    <entity>
      <ejb-name>Sang</ejb-name>
      <local-home>bpn.relationer.SangHome</local-home>
      <local>bpn.relationer.Sang</local>
      <ejb-class>bpn.relationer.SangBean</ejb-class>

      <prim-key-class>Integer</prim-key-class>
      <primkey-field>sid</primkey-field>

      <persistence-type>Container</persistence-type>
      <reentrant>True</reentrant>

      <abstract-schema-name>sanger</abstract-schema-name>
      <sql-table>sanger</sql-table>

      <cmp-field><field-name>titel</field-name></cmp-field>

      <query>
```

[65] Detta är något som gäller Resin, d.v.s. andra EJB-servrar kan ha ett annat sätt att ange namnet på kopplingstabellen.

```
          <query-method>
            <method-name>findAll</method-name>
          </query-method>
            <ejb-ql>SELECT o FROM sanger o</ejb-ql>
        </query>
      </entity>

    </enterprise-beans>

    <!--  ***************** Relationer ***************** -->
    <relationships>

      <!--  ***************** Album <- Sang ***************** -->
      <ejb-relation>
        <ejb-relation-name>album_sanger_map</ejb-relation-name>

        <ejb-relationship-role>
          <relationship-role-source>
            <ejb-name>Album</ejb-name>
          </relationship-role-source>
          <multiplicity>Many</multiplicity>
          <cmr-field>
            <cmr-field-name>sanger</cmr-field-name>
            <cmr-field-type>java.util.Collection</cmr-field-type>
            <sql-column>album</sql-column>
          </cmr-field>
        </ejb-relationship-role>

        <ejb-relationship-role>
          <relationship-role-source>
            <ejb-name>Sang</ejb-name>
          </relationship-role-source>
          <multiplicity>Many</multiplicity>
          <cmr-field>
            <cmr-field-name>album</cmr-field-name>
            <cmr-field-type>java.util.Collection</cmr-field-type>
            <sql-column>sang</sql-column>
          </cmr-field>
        </ejb-relationship-role>
      </ejb-relation>

    </relationships>

  </ejb-jar>
```

## Klient för EJB Album

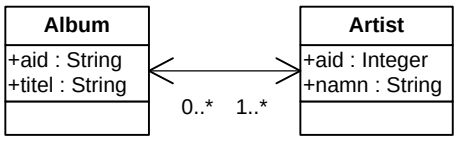
Klienten för EJB Sång är en ”upprepning” av tidigare klienter och visas inte för att spara plats.

```
  public class Album1MNServlet extends HttpServlet {
    /*** Instansvariabler ****************************************************/
    private AlbumHome home = null;

    /*** Överskugga metoder i klassen HttpServlet ****************************/
    public void init() throws ServletException {
      try {
        Context initial =
          (Context)new InitialContext().lookup("java:comp/env/ejb");

        home = (AlbumHome)initial.lookup("Album");
      }
      catch(NamingException ne) {
        throw new ServletException(ne);
      }
    }  //init()


    public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws java.io.IOException {
      PrintWriter out = null;
      res.setContentType("text/html");
      out = res.getWriter();

      //Påbörja HTML-dokument och skriv ut rubrik
```

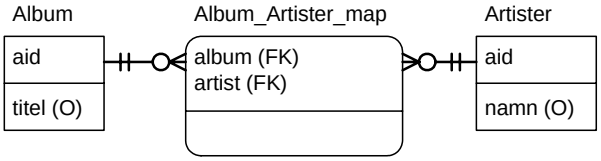
```
      out.println("<HTML>\n<HEAD>\n<TITLE>Album</TITLE>\n</HEAD>\n<BODY>");
      out.println("<H1>Album</H1>");

      Collection albumColl = null;

      /*** Testa metoden findByPrimaryKey() och findAll() ******************/
      try {
        Album album = home.findByPrimaryKey(new Integer(1));
        out.println("<P>Titel för album med id 1: "
          + album.getTitel() + "</P>");

        albumColl = home.findAll();
      }
      catch(FinderException fe) {
        fe.printStackTrace(out);
      }

      out.println("<p>Album och ev. sånger:<BR>");

      //Om album kunde hämtas, d.v.s. metoden findAll() "fungerade"
      if(albumColl != null) {
        Iterator iter = albumColl.iterator();

        while(iter.hasNext() ) {
          Album album = (Album)iter.next();
          out.println(album.getAid() + " - "
            + album.getTitel() + "<BR>");

          /*** Test av relation till EJB Sang **************************/
          Collection sanger = album.getSanger();

          if(sanger != null) {
            Iterator iter2 = sanger.iterator();

            while(iter2.hasNext()) {
              Sang sang = (Sang)iter2.next();
              out.println("- " + sang.getTitel() + " ("
                + sang.getSid() + ")<BR>");
            }
          }
          /************************************************************/
        }  //while
      }  //if(albumColl != null)

      //Skriv ut länk tillbaka till meny och avsluta HTML-dokument
      out.println("<P><A HREF=\"../relationer.html\">Meny</A></P>");
      out.println("</HTML>\n</HTML>");
    }  //doGet()

  }  //class Album1MNServlet
```

## 11.2.7 Dubbelriktad M:N-relation (många till många)

Ett album spelas in av flera artister och en artist kan spela in flera album.




Eftersom relationen mellan EJB är många till många så måste vi även här ha en kopplingstabell mellan tabellerna i datamodellen. Och för att göra den ”enkelt” så kan vi använda en räknare som primärnyckel (`aid`) i tabellen artister. Observera att de två tabellerna `album` och `artist` har samma namn på primärnyckeln – ett ”problem” som löses med namnen på de främmande nycklarna i kopplingstabellen (`album_artister_map`). Om tabellen album redan skapats så behöver endast de andra två tabellerna skapas. Nedan visas SQL-kod för att skapa tabellerna och lägga till lite testdata.




```
  CREATE TABLE album(
```

```
    aid int auto_increment PRIMARY KEY,
    titel varchar(60));

  INSERT INTO album(titel) VALUES('Brothers in arms');
  INSERT INTO album(titel) VALUES('Bigger, better, faster, more!');
  INSERT INTO album(titel) VALUES('The great rock''n''roll swindle');
  INSERT INTO album(titel) VALUES('Never mind the bullocks');

  CREATE TABLE artister(
    aid int auto_increment PRIMARY KEY,
    namn varchar(50));

  INSERT INTO artister(namn) VALUES('Dire Straits');
  INSERT INTO artister(namn) VALUES('4 non blondes');
  INSERT INTO artister(namn) VALUES('Sex Pistols');

  CREATE TABLE album_artister_map(
    album int NOT NULL,
    artist int NOT NULL,
    CONSTRAINT PRIMARY KEY(album, artist));

  INSERT INTO album_artister_map VALUES(1, 1);
  INSERT INTO album_artister_map VALUES(2, 2);
  INSERT INTO album_artister_map VALUES(3, 3);
  INSERT INTO album_artister_map VALUES(4, 3);
```

## Implementation av EJB Album

Implementationen av EJB Album ser i stort sett likadan ut som i förra exemplet. Skillnaden är att vi ersätter sångrelaterade namn med artistrelaterade namn. Precis som med EJB Person så visas endast kod relaterad till relation i detta exempel.

### Local-gränssnitt

```
  public interface Album extends EJBLocalObject
  {
    /*** Publika accessmetoder ************************************************/
    public Integer getAid();
    public String getTitel();
    public void setTitel(String titel);

    public Collection getArtister();
  }  //interface Album
```

### Local home-gränssnitt

```
  public interface AlbumHome extends EJBLocalHome
  {
    /*** create-metoder *******************************************************/
    public Album create(String titel) throws CreateException;

    /*** find-metoder *********************************************************/
    public Album findByPrimaryKey(Integer id) throws FinderException;
    public Collection findAll() throws FinderException;
  }  //interface AlbumHome
```

### Bean-klass

```
  public abstract class AlbumBean extends AbstractEntityBean
  {
    /*** Accessmetoder för CMP-attribut ***************************************/
    public abstract Integer getAid();
    public abstract String getTitel();
    public abstract void setTitel(String titel);

    /*** Accessmetoder för CMR-attribut ***************************************/
    public abstract Collection getArtister();

    /*** ejbCreate- och ejbPostCreate-metoder *********************************/
    public Integer ejbCreate(String titel) throws CreateException
    {
      setTitel(titel);
      return null;
```

```
    }  //ejbCreate()

    public void ejbPostCreate(String titel) { }

    //Metoder i gränssnittet EntityBean implementeras av AbstractEntityBean

  }  //class AlbumBean
```

# Implementation av EJB Artist

Även EJB Artist är en ”upprepning” av tidigare exempel och förklaringar utelämnas.

## Local-gränssnitt

```
  public interface Artist extends EJBLocalObject
  {
    /*** Publika accessmetoder ************************************************/
    public Integer getAid();
    public String getNamn();
    public void setNamn(String namn);

    public Collection getAlbum();
  }  //interface Artist
```

## Local home-gränssnitt

```
  public interface ArtistHome extends EJBLocalHome
  {
    /*** create-metoder *******************************************************/
    public Artist create(String namn) throws CreateException;

    /*** find-metoder *********************************************************/
    public Artist findByPrimaryKey(Integer id) throws FinderException;
    public Collection findAll() throws FinderException;
  }  //interface ArtistHome
```

## Bean-klass

```
  public abstract class ArtistBean extends AbstractEntityBean
  {
    /*** Accessmetoder för CMP-attribut ***************************************/
    public abstract Integer getAid();
    public abstract String getNamn();
    public abstract void setNamn(String namn);

    /*** Accessmetoder för CMR-attribut ***************************************/
    public abstract Collection getAlbum();

    /*** ejbCreate- och ejbPostCreate-metoder *********************************/
    public Integer ejbCreate(String namn) throws CreateException
    {
      setNamn(namn);
      return null;
    }  //ejbCreate()

    public void ejbPostCreate(String titel) { }

    //Metoder i gränssnittet EntityBean implementeras av AbstractEntityBean

  }  //class ArtistBean
```

# Deployment descriptor för EJB:er samt relation

```
  <ejb-jar>

    <!--  ***************** EJBer ******************** -->
    <enterprise-beans>

      <!--  ***************** Album ***************** -->
      <entity>
        <ejb-name>Album</ejb-name>
        <local-home>bpn.relationer.AlbumHome</local-home>
        <local>bpn.relationer.Album</local>
```

```
        <ejb-class>bpn.relationer.AlbumBean</ejb-class>

        <prim-key-class>Integer</prim-key-class>
        <primkey-field>aid</primkey-field>

        <persistence-type>Container</persistence-type>
        <reentrant>True</reentrant>

        <abstract-schema-name>album</abstract-schema-name>
        <sql-table>album</sql-table>

        <cmp-field><field-name>titel</field-name></cmp-field>

        <query>
          <query-method>
            <method-name>findAll</method-name>
          </query-method>
            <ejb-ql>SELECT o FROM album o</ejb-ql>
        </query>
      </entity>

      <!--  ***************** Artist ***************** -->
      <entity>
        <ejb-name>Artist</ejb-name>
        <local-home>bpn.relationer.ArtistHome</local-home>
        <local>bpn.relationer.Artist</local>
        <ejb-class>bpn.relationer.ArtistBean</ejb-class>

        <prim-key-class>Integer</prim-key-class>
        <primkey-field>aid</primkey-field>

        <persistence-type>Container</persistence-type>
        <reentrant>True</reentrant>

        <abstract-schema-name>artister</abstract-schema-name>
        <sql-table>artister</sql-table>

        <cmp-field><field-name>namn</field-name></cmp-field>

        <query>
          <query-method>
            <method-name>findAll</method-name>
          </query-method>
            <ejb-ql>SELECT o FROM artister o</ejb-ql>
        </query>
      </entity>

    </enterprise-beans>

    <!--  ***************** Relationer ***************** -->
    <relationships>

      <!--  ***************** Album <-> Artist ***************** -->
      <ejb-relation>
        <ejb-relation-name>album_artister_map</ejb-relation-name>

        <ejb-relationship-role>
          <relationship-role-source>
            <ejb-name>Album</ejb-name>
          </relationship-role-source>
          <multiplicity>Many</multiplicity>
          <cmr-field>
            <cmr-field-name>artister</cmr-field-name>
            <cmr-field-type>java.util.Collection</cmr-field-type>
            <sql-column>album</sql-column>
          </cmr-field>
        </ejb-relationship-role>

        <ejb-relationship-role>
          <relationship-role-source>
            <ejb-name>Artist</ejb-name>
          </relationship-role-source>
          <multiplicity>Many</multiplicity>
          <cmr-field>
            <cmr-field-name>album</cmr-field-name>
            <cmr-field-type>java.util.Collection</cmr-field-type>
            <sql-column>artist</sql-column>
```

```
          </cmr-field>
        </ejb-relationship-role>
      </ejb-relation>

    </relationships>

  </ejb-jar>
```

### Klient för EJB Album och EJB Artist

Klienter för EJB Album och EJB Artist är i stort sett den samma som tidigare exempel och visas därför inte.

## 11.3 Slutsats

Att skapa en enkel EJB med bara CMP-attribut är enkelt (i alla fall efter att ha skapat ett antal ☺). Men när vi introducerar relationer, och därmed CMR-attribut, så ökar komplexiteten och därmed antalet möjligheter till fel (buggar). Börja därför med att testa EJB utan relationer och verifiera att de fungerar felfritt så. Lägg först då till relationer mellan EJB.

Ovanstående 7 exempel bör också kunna användas som mallar för att skapa relationer mellan EJB (d.v.s. hjulet behöver inte uppfinnas helt varje gång ☺).

# 12 Litteraturförteckning


- Allamaraju, S., et al, **Professional Java Server Programming – J2EE Edition**, Wrox Press, 2000. ISBN: 1-861004-65-6.
- Horton, Ivor, **Beginning Java 2**, Wrox Press, 1999. ISBN: 1-861002-23-8.
- Monson-Haefel, Richard, **Enterprise JavaBeans, 3rd Ed.**, O'Reilly, 2001. ISBN: 0-596-00226-2.
- Orfali, Robert, et al, **Instant CORBA**, Wiley Computer Publishing, 1997. ISBN: 0-471-18333-4.
- Sasbury, Stephen & Scott R. Weiner, **Developing Enterprise Java Applications, 2nd Edition**, Wiley Computer Publishing, 2001. ISBN: 0-471-40593-0.
- Siegel, Jon, **CORBA 3 – Fundamentals and Programming, 2nd Edition**, Wiley Computer Publishing, 2000. ISBN: 0-471-29518-3.
- Tulachan, P.V., **Developing EJB 2.0 Components**, Prentice-Hall, 2002. ISBN: 0-13-034863-5.


*Några av dessa böcker kan finnas i nyare upplagor!*

## 12.1 Webbadresser till förlag

- O'Reilly & Associates
  `http://www.ora.com/`
- Wiley (och Wiley Computer Publishing)
  `http://www.wiley.com/` (`http://www.wiley.com/compbooks/`)
- Wrox Press
  `http://www.wrox.com/`

## 12.2 Webbadresser

- Object Management Group (OMG) och CORBA
  `http://www.omg.org/`
- Sun och Java
  `http://java.sun.com/`