Genova810DokDomenemodell

Domenemodellering og JGrape

Krav som stilles av persistensrammeverket

Hibernate har følgende krav/anbefalinger til klassene i domenemodellen:

1. Persistente klasser skal ha en default-constructor.
2. Alle klasser bør ha en "id"-property. Dette kan være et enkelt attributt eller en sammensatt nøkkel, mer om dette senere. Det anbefales å bruke en ikke-primitiv type siden man da kan benytte "null" som et flagg for at forekomsten ennå ikke er lagret i basen.
3. Hvis det benyttes sammensatte nøkler uten egne nøkkel-klasser må domeneklassen imlementere equals/hashCode samt Serializable.
4. Klassene bør ikke være final, siden dette hindrer Hibernate i å opprette proxyklasser for "lazy loading".
5. Det anbefales å ha "JavaBeans style properties", dvs. med get/set-metoder for aksess.
6. Man bør implementere equals/hashCode for domeneklassene. Disse metodene bør baseres på verdier fra naturlige kandidatnøkler og ikke id-attributt som tilordnes verdi ved første gangs lagring.
7. Hvis domeneobjekter skal kunne sendes til/fra en applikasjonsserver må klassene implementere Serializable.

Valg av nøkkelstrategi

Det er to hovedkategorier av nøkler: naturlige nøkler og surrogatnøkler. Naturlige nøkler har en mening innenfor domenet som klassen skal representere, mens surrogatnøkler er "dumme" identifikatorer som kun benyttes for å kunne skille mellom ulike forekomster av samme klasse.

Mens det tidligere har vært vanlig å benytte naturlige nøkler som primærnøkler på databasetabeller er det nå mer vanlig å inkludere en egen id-kolonne for en surrogatnøkkel. Dette forenkler bla. håndtering av fremmednøkler, samt at man unngår situasjoner der attributter fra refererte objekter inngår som en del av nøkkelen. Hibernate anbefaler at man bruker naturlige nøkler kun i de tilfeller hvor man jobber mot eksisterende databaser ("legacy"-databaser).

Det er fullt mulig å lage en modell hvor noen klasser benytter naturlige nøkler mens andre har surrogatnøkler.

NB! Primærnøkler bør være immutable!

Surrogatnøkler

Det er vanlig å benytte en heltalls-type for surrogatnøkkelkolonna, f.eks. en 32-bits integer. Siden nøkkelverdien ikke har noen forretningsmessig betydning er det vanlig å overlate generering av nøkkelverdi på nye forekomster til persistensrammeverket eller databasen, avhengig av hva som er støttet i databasen. Noen databasesystemer, f.eks. MySQL, Sybase og SQL Server, har støtte for en "identity"-type som tilordner unike nøkkelverdier ved "insert"-operasjoner. Andre, som f.eks. Oracle, har støtte for tilsvarende mekanismer med sin "sequence".

Som kolonne-navn for surrogatnøkler er det vanlig å benytte "id" eller "<tabellnavn>_id", og en tilsvarende navning på attributtet i domeneklassen.

Surrogatnøkler modelleres som f.eks. "int" eller "Integer" i UML-modellen. Valg av databasetype gjøres i databasemappingen.

Naturlige nøkler

Ved å bruke en naturlig nøkkel som primærnøkkel unngår man å bruke ekstra plass i databasetabellene og i Javaobjektene til å holde verdier som ikke har forretningsmessig mening. Typiske eksempler på naturlige nøkler kan være "ISBN-nummer", "navn", "kontonummer", "fødselsnummer" osv. Kravet er at det kun skal kunne finnes en forekomst med en gitt verdi at nøkkelen. Hvis dette ikke kan oppfylles for et enkelt attributt er det vanlig å lage sammensatte nøkler. Eksempel: klassen "OrdreLinje" kan ha en nøkkel som består av "ordrenummer" og "linjenummer", hvor de to verdiene til sammen alltid identifiserer en unik forekomst.

Store modeller med sammensatte naturlig nøkler kan ofte bli kompliserte å jobbe med siden nøkkelelementer har en tendens til å bli "trukket ned" over mange nivåer. Eksempel: i en modell som innholder domeneklassene "Konsern", "Firma", "Avdeling" og "Team" kan det lett bli slik at "konsern_navn" inngår som nøkkelelement i alle de andre klassene, mens "firma_navn" inngår som nøkkelelement i "Avdeling" og "Team" osv.

Hvis man velger å bruke sammensatte naturlige nøkler vil man pga. Hibernate måtte spesialhåndtere tilfeller der det finnes nøkkelelementer som er "trukket

ned" fra en assosiert klasse. Det vil f.eks. i eksempelet over ikke være mulig å hente opp forekomster av "Firma" uten at man har et sted å gjøre av "konsern_navn"-verdiene. Dette kan løses på ulike måter, Hibernate anbefaler at man oppretter egne klasser for de sammensatte nøklene. Dette er nærmere beskrevet i referansemanualen til Hibernate.

Attributt-typer

Verdier som ikke har egen identitet ("value objects") modelleres gjerne som attributter på domeneklassene.

Datatyper som er støttet:

- Primitive typer samt deres wrappertyper
- String
- Date
- Numeric
- Subklasser av GenovaEnumerator

+ andre typer som man "mappes" til Genovas interne typer, se oppsett av "Type Mapping From Modeling Types" i Genovas setup-database.

Optimistisk låsing

For å sikre seg mot feilaktige oppdateringer i databasen i flerbrukersammenheng er det vanlig å benytte optimistisk låsing. Dette innebærer at hver rad i databasen har en versjonskolonne som endres for hver oppdatering som gjøres. Gitt at man tar med versjons-verdien som utvalgskriterium ved "update" sikrer man seg mot å oppdatere rader som i mellomtiden har blitt oppdatert av andre brukere. Dette håndteres automatisk av de fleste persistensrammeverk inklkudert Hibernate.

Hvilken datatype som benyttes på versjonskolonna avhenger av hvilket databasesystem man bruker. Noen har innebygd støtte for dette som f.eks. "timestamp"-datatypen i Sybase og SQL Server, mens andre benytter en heltalls-teller.

Valg for optimistisk låsing gjøres i Genovas "Database Designer" (i setup-databasen).

Siden alle persistente objekter må ta vare på sin versjons-verdi gjøres dette enklest ved at domeneklassene arver fra rammeverksklassen PersistentClass.

Implementasjon av equals og hashCode

Gode eksempler på implementasjon av disse finner man bl.a i "Effective Java".

Hibernate anbefaler at man benytter naturlige nøkler for equals og hashCode.

Eksempel: klassen Customer med en sammensatt nøkkel bestående av {custNo, regDate, version}:

```
    public boolean equals(final Object other) {
        if ( other==null ) { return false; }
        if ( this==other ) { return true; }
        if (!( other instanceof Customer )) { return false; }
        final Customer that= (Customer) other;
        try { if ( this.getCustNo() != that.getCustNo() )
            { return false; } }
            catch (final Throwable wasted) { /*EMPTY*/ }
        try {
            final boolean b01= (this.getRegDate()==null);
            final boolean b02= (that.getRegDate()==null);
            if (b01!=b02) return false;
            if (!b01 && !( this.getRegDate().equals(that.getRegDate())
))
            { return false; } }
            catch (final Throwable wasted) { /*EMPTY*/ }
        try { if ( this.getDeklsekv() != that.getDeklsekv() )
            { return false; } }
            catch (final Throwable wasted) { /*EMPTY*/ }
        try { if ( this.getVersion() != that.getVersion() )
            { return false; } }
            catch (final Throwable wasted) { /*EMPTY*/ }
        return true;
    }

    public int hashCode() {
        int ii;
        int result=37;
        result= (31*result) + this.getClass().hashCode();
        try {
            ii= getCustNo();
        } catch (final Throwable wasted) { ii= 0; }
        result= (31*result) + ii;
```

```
        try {
            ii= (getRegDate()==null)? 0
                : getRegDate().hashCode();
        } catch (final Throwable wasted) { ii= 0; }
        result= (31*result) + ii;
        try {
            ii= getDeklsekv();
        } catch (final Throwable wasted) { ii= 0; }
        result= (31*result) + ii;
        try {
            ii= getVersion();
        } catch (final Throwable wasted) { ii= 0; }
        result= (31*result) + ii;
        return result;
    }
```

Transiente domeneklasser

Klasser for domeneobjekter som ikke skal lagres modelleres på sammen måte som andre domeneklasser, men de markeres som ikke persistente i modellen. De vil dermed ikke bli med i database mappingene i Genova, men er fullt tilgjengelige for bruk i objektseleksjoner og dialogmodeller.

Transiente attributter

Det finnes 2 ulike typer transiente attributter: Java-transiente og Genova-transiente.

Java-transiente attributter markeres som "Transient=True" på "Java-fliken" i Rose. Dette medfører at attributtet genereres med modifikatoren "transient" i Javakoden, og dermed ikke tas med ved f.eks. serialisering.

Genova-transiente attributter markeres som "Persistent=False" på "Genova DB-fliken" i Rose. Dette medfører ingen endring i generert Javakode fra Rose, men attributtet tas ikke med som kolonne i databasemappingen for klassens tabell.

Enumeratorer og grupper

Modellering av enumeratorer og grupper er beskrevet i Genova User Guide, under kapittelet for Genova Rose Add-ins. Videre er bruk av enumerator-klasser beskrevet i dokumentasjonen for klientgeneratoren (for target Java/JFC).

UML pakkestruktur

I Rose modelleres domeneklassene i "Logical View". I tillegg opprettes der en Java-komponent for hver klasse i "Component View". Hvilket view som benyttes som grunnlag for pakkestruktur i Genovas workspace er bestemt av en parameter i setup-databasen. Det anbefales å benytte "Component View" siden det er denne pakkestrukturen som benyttes av Rose ved generering av Java-domeneklassene.