Documentation Application VIF

Domainobject and Metamodel Framework

Benno Luthiger
benno.luthiger(+)id.ethz.ch

Version 1.7
Valid for VIF Framework Rel. 1.0.2
Zurich, August 6, 2007

History:

Version 1.7 (August 6, 2007):

Adjusted DTD for simple domain object to cover LDAP access.

Version 1.6 (November 1, 2006):

Added SQLRange for filtering of retrieved data.

Added the section about lightweight homes and models.

Added a sample explaining the use of the framework.

Version 1.5 (April 12, 2005):

Added join type 'NO_JOIN' to DTD of JoinedObjectDef.

Version 1.4 (November 15, 2004):

Extended DTD of Joined Domain Object.

Added subsections in chapter 8.

Table of Content

Abstract

This document describes the domain object framework used for the application VIF.

The first chapter explains the motivation for the framework and basic interactions between application and database. The next chapters describe definition objects and the meta model needed to provide the generic functionality. The framework is able to handle table joins which is explained in chapter 4. Chapter 5 gives a short introduction into domain object serialization and its use with regard to the servlet framework. The last chapters are basically cookbooks. Chapter 6 describes how to create new database adapters for that the framework is able to interact with a new database. Chapter 7 tells how to create concrete domain objects. Chapter 8 explains how to use the domain objects to create specific SQL statements.

1. Basic interaction with the database

The motivation for the domain object framework is to provide as much generic functionality as possible. The more functionality is handled in the framework, the less the programmer has to implement by herself.

The main purpose of the domain object framework is to link the application with the database. Each entry which has to make persistent or which is read from the database is accessible for the application as instance of a domain object. The framework provides all functionality to access the information in the database. There are no SQL statements needed in the application outside the domain object framework. Furthermore the framework includes functionality for easy serialization to XML. With that, the domain object framework integrates seamlessly to the servlet framework, which offers the possibility to display the serialized domain objects via XSL transformation.

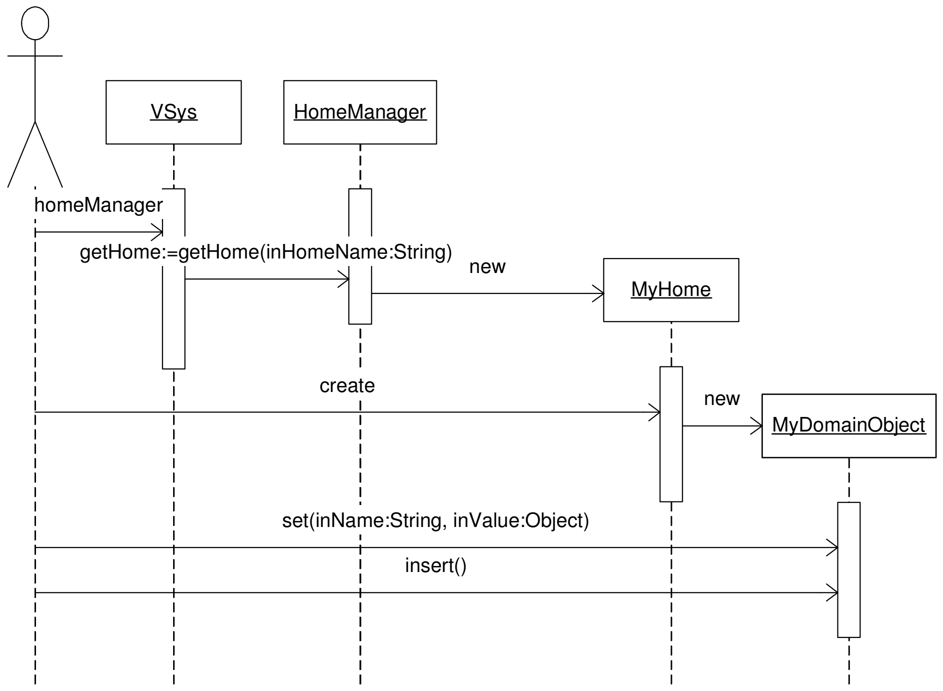

VSys
HomeManager
homeManager
getHome:=getHome(inHomeName:String)
new
MyHome
create
new
MyDomainObject
set(inName:String, inValue:Object)
insert()

Diagram 1: Sequence diagram of basic interaction: insert of new entry

Diagram 1 shows the interaction sequence to insert a new entry in the database. Each domain object has its home, which is both a fabric for the domain object and a manager of its instances. If the application wants to make information persistent, it first calls the appropriate domain object home. After the application has access to the home, it creates an instance of the domain object by calling `DomainObjectHome.create()`. After the application has access to a domain object's instance, it can set its values (by calling `DomainObject.set(inAttributeName, inAttributeValue)`) and then, by calling `DomainObject.insert()`, make the information stored in the domain object persistent.

Diagram 2 shows other basic functionality of data manipulation. An existing table entry has to be retrieved and edited or deleted. This is fulfilled by creating a KeyObject and setting its appropriate values.

After that, the Method `DomainObject.findByKey(KeyObject)` is sent. This returns an instance of the domain object filled with the values from the data base table. After the application has access to a domain object's instance, it can read it's information (by calling `DomainObject.get(inAttributeName)`) or modify it's information (by calling `DomainObject.set(inAttributeName, inAttributeValue)`). The modified information is stored to the database by calling `DomainObject.update().DomainObject.delete()` On the other hand removes the entry from the database.

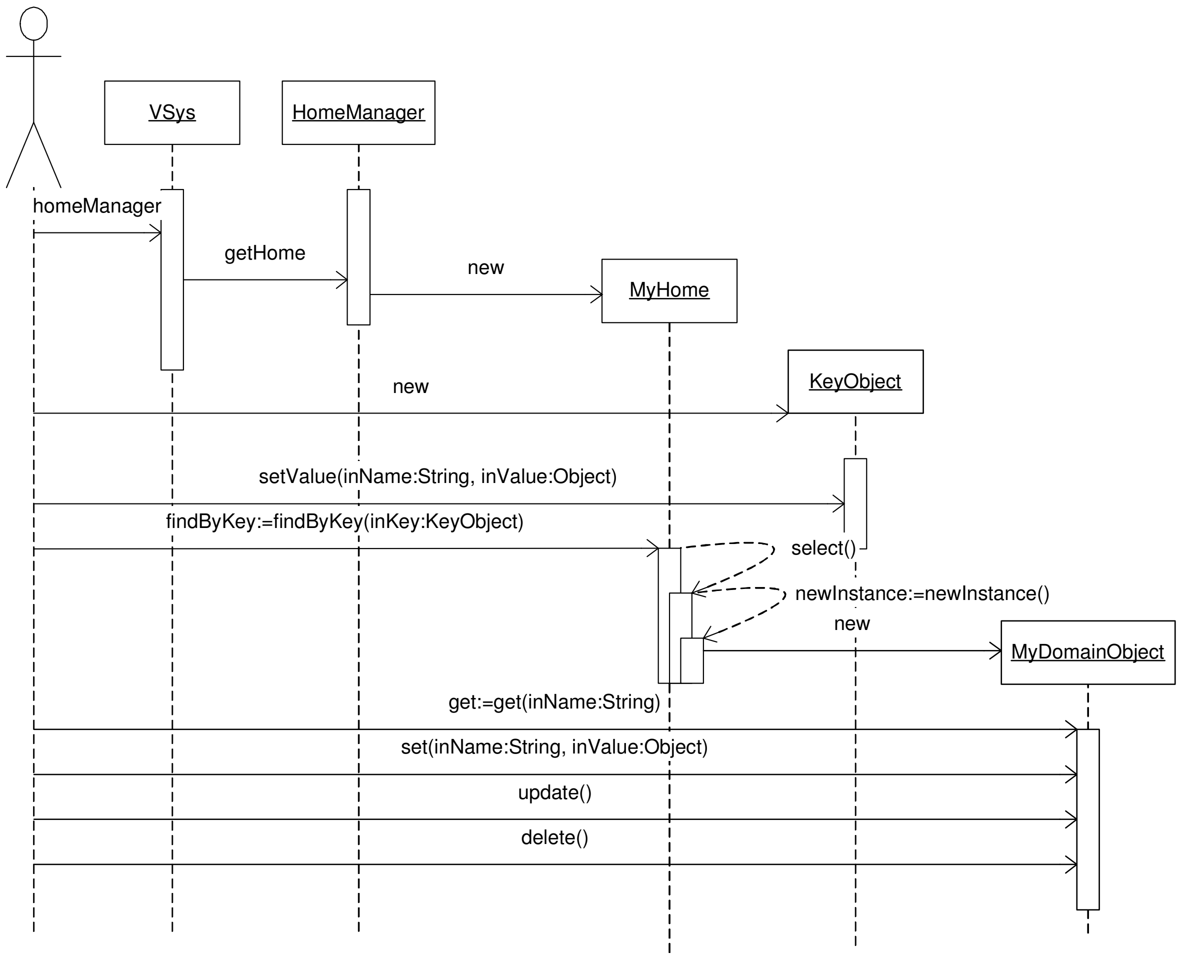

VSys
HomeManager
homeManager
getHome
new
MyHome
new
KeyObject
setValue(inName:String, inValue:Object)
findByKey:=findByKey(inKey:KeyObject)
select()
newInstance:=newInstance()
new
MyDomainObject
get:=get(inName:String)
set(inName:String, inValue:Object)
update()
delete()

Diagram 2: Sequence diagram of basic interaction: data retrieval, update and delete

The example Code 1 shows an update using the domain object functionality described.

```
KeyObject lKey = new KeyObjectImpl();
try {
    lKey.setValue(NAME, "VIF Application");
    MyHome lMyHome = (MyHome)VSys.homeManager.getHome(HOME_CLASS_NAME);
    MyDomainObject lMyDomainObject = (MyDomainObject)lMyHome.findByKey(lKey);
```

```
     lMyDomainObject.set(REMARK, “Hello World”);
     lMyDomainObject.update();
}
catch (BOMNotFoundException exc) {
     //handle exception
}
catch (BOMInvalidKeyException exc) {
     //handle exception
}
catch (Exception exc) {
     //handle exception
}
```

Code 1: Example of update

2. The Object Model definition

For that the domain object framework can access the information stored in a database, the framework needs information about the organization of the database, i.e. table and column names. This meta information is provided by an XML. Each domain object model has its object model definition, which is parsed the first time the home has to create a domain object. The result of the parsing process is an instance of the `ObjectDef` class, containing all the meta information needed to access the correct database tables and fields. In the actual version of the framework, the XML is provided as static string.

Code 2 shows an example of a model object definition.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<objectDef objectName='Permission' parent='org.hip.kernel.bom.DomainObject'
     version='1.0'>
     <keyDefs>
          <keyDef>
               <keyItemDef seq='0' keyPropertyName='ID'/>
          </keyDef>
     </keyDefs>
     <propertyDefs>
          <propertyDef propertyName='ID' valueType='Number'
                   propertyType='simple'>
               <mappingDef tableName='tblPermission'
                    columnName='PERMISSIONID'/>
          </propertyDef>
          <propertyDef propertyName='Label' valueType='String'
                   propertyType='simple'>
               <mappingDef tableName='tblPermission'
                    columnName='SPERMISSIONLABEL'/>
          </propertyDef>
          <propertyDef propertyName='Description' valueType='String'
                   propertyType='simple'>
               <mappingDef tableName='tblPermission'
                    columnName='SPERMISSIONDESCRIPTION'/>
          </propertyDef>
     </propertyDefs>
</objectDef>
```

Code 2: Example of XML containing meta information

Each object definition consists of a collection of property definitions. Each property definition is transformed to a domain object attribute once an instance of the domain object is created. The property definitions provide the methods to get and set the domain object values. Each property definition has a mapping definition. With this definition the property definition is able to map its attributes to a specified database field.

The task to create an `ObjectDef` instance is fulfilled by the `ObjectDefGenerator`. This class parses the XML containing the object description (for its DTD see appendix, Code 10) and creates the `ObjectDef` instance. The `ObjectDef` then is passed to the `DomainObjectHome` it belongs to. When this home has to create a new instance of `DomainObject`, it is able to initialize the `DomainObject`'s `PropertySet` with the information given by the `ObjectDef`.

A further important implication of the information contained in the object definition is that it makes it possible to create all needed SQL commands (SELECT, INSERT, UPDATE, DELETE) in a generic way. With that, there's no need of explicit SQL statements outside of the framework. Therefore, the data design is neatly separated from the proper application. If the data design changes, the application has to be adjusted at well defined positions, i.e. the XMLs containing the object model definitions.

3. Metamodel

The `PropertySet` is the core of each object in the model framework. Essentially the `PropertySet` wrapps a `Hashtable`, but adds the functionality to make the object serializable. (For further details see the class Diagram 4 in the appendix).

One basic task the framework has to fulfill is to initialize an object's `PropertySet`. The `DomainObject`'s `PropertySet` can be initialized with the help of the `ObjectDef`, i.e. with object definition provided in form of an XML and parsed by the `ObjectDefGenerator`. But the `ObjectDef` and its components (`PropertyDef`, `KeyDef` etc.) each contain their own instances of `PropertySet`. The question arises how these property sets have to be initialized. This is done by the meta model. Each model object (e.g. `ObjectDef`, `PropertyDef`, `KeyDef`) contains an associated meta model object (e.g. `ObjectDefDef`, `PropertyDefDef`, `KeyDefDef`) with the necessary information.

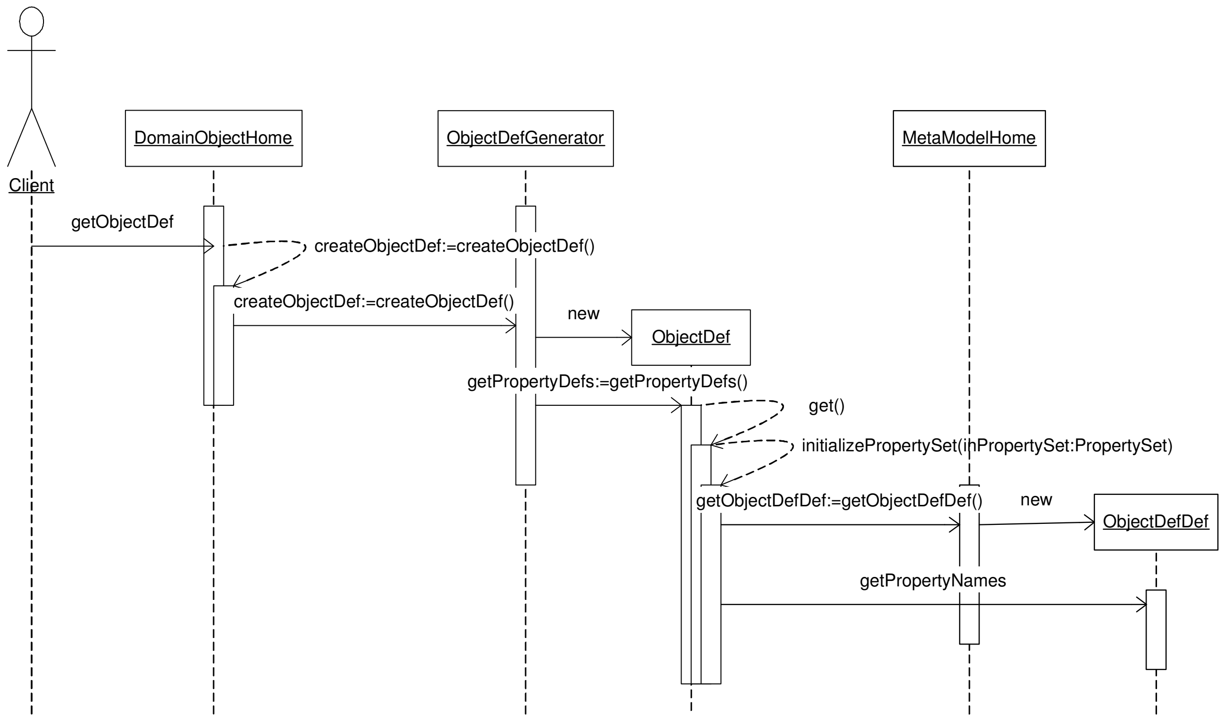

Client
DomainObjectHome
ObjectDefGenerator
MetaModelHome
getObjectDef
createObjectDef:=createObjectDef()
createObjectDef:=createObjectDef()
new
ObjectDef
getPropertyDefs:=getPropertyDefs()
get()
initializePropertySet(inPropertySet:PropertySet)
getObjectDefDef:=getObjectDefDef()
new
ObjectDefDef
getPropertyNames

Diagram 3: Interaction with the meta model

4. Table joins

The domain objects generated the way described above can interact with only one table in the database. For to read joined tables too, there exists an extension to the simple domain object framework.

Table joins are portrayed to the framework as classes deriving from `JoinedDomainObjectHome`. These homes have an object description (i.e. the class `JoinedObjectDef`) which is created by the `JoinedObjectDefGenerator`. The XML describing this object definition must meet the DTD shown at Code 11.

The `DomainObject`s created by homes derived from `JoinedDomainObjectHome` are read only.

5. Serialization

Serialization is the linkage between the servlet framework and the domain object framework. The servlet framework has the functionality to display XML via transformation with XSL. The domain object framework on the other side has the functionality needed by the domain objects to interact with the database. What's missing is the functionality to hand over the domain object's data to the servlet framework to be displayed.

This is done by serialization. The domain object framework allows easy serialization by use of the visitor pattern. Various serializers are feasible making it possible to generate data in various formats. The framework provides one serializer that generates XML.

```
   MyDomainObject lData = (MyDomainObject)lMyDomainObjectHome.findByKey(lKey);
   XMLSerializer lXML = new XMLSerializer();
   lData.accept(lXML);
   String lSerialized = lXML.toString();
```

Code 3: Serialization of a domain object

6. Database adapters

For that the framework can communicate with databases from different vendors, an adapter layer exists to decouple the domain objects from the database. Actually adapters exist for MySQL, Oralce and Derby databases. This chapter shows how to create new adapters for that the framework can interact with further databases.

The singleton class `DBAdapterSelector` knows the database actually configured for the application (see Diagram 5 in the appendix). This class, therefore, is able to select the appropriate adapters (`DBAdapterSelector.getSimpleDBAdapter(), DBAdapterSelector.getJoinDBAdapter()`) and column modifiers (e.g. `DBAdapterSelector.getColumnModifierUCase()`). An instance of an adapter class aggregates an instance of `ObjectDef` or `JoinedObjectDef` respectively. Therefore, the adapter has the information needed to create the SQL statements to interact with the database. Each `DomainObjectHome` (or `JoinedDomainObjectHome`) aggregates a `DBAdapter` and thus is able to interact with the database.

The class diagram shows that each adapter class inherits from `DefaultDBAdapterSimple` or `AbstractDBAdapterJoin` respectively. For to create a new adapter for a further database, you have to create a class specializing from `AbstractDBAdapterJoin` and implementing `DBAdapterJoin` interface, in case of a very exotic database, you have to spezialize from `DefaultDBAdapterSimple` (implementing the `DBAdapterSimple` interface) too. After creating this class, you have to modify `DBAdapterSelector` for that the selector is able to return the newly created adapter classes when it detects the new database installed on the system. The selector detects the installed database by analyzing the property `org.hip.vif.db.url` in *vif.properties*.

After creating the concrete adapter classes for the new database, you have to provide a full set of column modifier classes `DBAdapterSelector` can return if it detects the new database. Column modifier classes have to implement the `ColumnModifier` interface, i.e. they have to implement the `modifyColumn()` method.

7. How to create domain objects

For to use the domain object framework you have to do the following steps:

1. Create a concrete domain object home (e.g. `MyDomainObjectHomeImpl`) and its interface. The interface derives from `DomainObjectHome`, the class from `DomainObjectHomeImpl`.
2. Within your domain object home create an object description. This description has the form of an XML and has to meet the DTD shown in the appendix (Code 10). This XML is the return value of the domain object's protected method `getObjectDefString()`. With this object definition, the model attributes are mapped to the table fields.
3. Create a concrete domain object model (e.g. `MyDomainObjectImpl`) and its interface. The interface derives from `DomainObject`, the class from `DomainObjectImpl`.
4. Edit your domain object home so that its public method `getObjectClassName()` returns the fully qualified path to its domain object. Edit your domain object so that its public method `getHomeClassName()` returns the fully qualified path to its domain object home.

Developers using the *Eclipse* IDE can significantly reduce the effort needed to create the home and model using the *VIFDom Plug in* provided by the VIF project (see http://sourceforge.net/projects/vif).

8. How to create SQL statements

In this chapter we learn how to use the domain object framework to create the various SQL statements needed to interact with a database.

8.1. Simple domain objects

Imagine we have a member table and we want the framework to manipulate the content of this table. The domain object's object definition string is shown in Code 12 (see Appendix C: Example of object definitions).

8.1.1. Insert

The following code fragment inserts a new entry to the table:

```
DomainObject lMember = ((DomainObjectHome)
VSys.homeManager.getHome("org.hip.vif.bom.impl.MemberHomeImpl")).create();
lMember.set("UserID", "myUserID");
lMember.set("Name", "myName");
lMember.set("Firstname", "myFirstName");
lMember.set("Street", "myStreet");
lMember.set("ZIP", "myZIP");
lMember.set("City", "myCity");
lMember.set("Tel", "myTel");
lMember.set("Fax", "myFax");
lMember.set("Mail", "myMail");
lMember.set("Sex", new Integer(1));
lMember.set("Language", "myLanguage");
lMember.set("Password", "myPassword");
lMember.insert(true);
```

This code produces the following SQL statement:

```
INSERT INTO tblMember( SZIP, SFIRSTNAME, SLANGUAGE, SFAX, SMAIL, STEL,
  SSTREET, SPASSWORD, SCITY, SUSERID, DTMUTATION, BSEX, MEMBERID, SNAME )
  VALUES ('myZIP', 'myFirstName', 'myLanguage', 'myFax', 'myMail', 'myTel',
  'myStreet', 'myPassword', 'myCity', 'myUserID', NULL, 1, NULL, 'myName' )
```

8.1.2. Update

The following code fragment updates an existing table entry:

```
KeyObject lKey = new KeyObjectImpl();
lKey.setValue("UserID", "myUserID");
DomainObject lMember = ((DomainObjectHome)
   VSys.homeManager.getHome("org.hip.vif.bom.impl.MemberHomeImpl"
   )).findByKey(lKey);
lMember.set("Mail", "newMail");
lMember.update(true);
```

This code produces the following SQL statement:

```
UPDATE tblMember SET SMAIL = 'newMail' WHERE tblMember.MEMBERID = 217
```

8.1.3. Delete

The following code fragment deletes an existing table entry:

```
KeyObject lKey = new KeyObjectImpl();
lKey.setValue("UserID", "myUserID");
DomainObject lMember = ((DomainObjectHome)
    VSys.homeManager.getHome("org.hip.vif.bom.impl.MemberHomeImpl"
    )).findByKey(lKey);
lMember.delete(true);
```

This code produces the following SQL statement:

```
DELETE FROM tblMember WHERE tblMember.MEMBERID = 217
```

8.1.4. Select

The domain object framework offers the possibility to create a variety of SQL select statements.

SELECT

The following code fragment selects all entries in the table:

```
DomainObjectHome lHome = (DomainObjectHome)
  VSys.homeManager.getHome("org.hip.vif.bom.impl.MemberHomeImpl");
QueryResult lEntries = lHome.select();
```

This code produces the following SQL query string:

```
SELECT tblMember.SLANGUAGE, tblMember.SUSERID, tblMember.SFAX,
    tblMember.STEL, tblMember.SFIRSTNAME, tblMember.SNAME, tblMember.SSTREET,
    tblMember.SPASSWORD, tblMember.MEMBERID, tblMember.BSEX, tblMember.SZIP,
    tblMember.SCITY, tblMember.SMAIL, tblMember.DTMUTATION FROM tblMember
```

SELECT … WHERE

The following code fragment selects all entries matching the specified criterion:

```
  DomainObjectHome lHome = (DomainObjectHome)
    VSys.homeManager.getHome("org.hip.vif.bom.impl.MemberHomeImpl");
  KeyObject lKey = new KeyObjectImpl();
  lKey.setValue("Sex", new Integer(1));
  QueryResult lEntries = lHome.select(lKey);
```

This code produces the following SQL query string:

```
  SELECT tblMember.SLANGUAGE, tblMember.SUSERID, tblMember.SFAX,
    tblMember.STEL, tblMember.SFIRSTNAME, tblMember.SNAME, tblMember.SSTREET,
    tblMember.SPASSWORD, tblMember.MEMBERID, tblMember.BSEX, tblMember.SZIP,
    tblMember.SCITY, tblMember.SMAIL, tblMember.DTMUTATION
    FROM tblMember
    WHERE tblMember.BSEX = 1
```

SELECT … WHERE … ORDER BY …

The following code fragment selects all entries matching the specified criterion and returns the entries in the specified order:

```
DomainObjectHome lHome = (DomainObjectHome)
  VSys.homeManager.getHome("org.hip.vif.bom.impl.MemberHomeImpl");
KeyObject lKey = new KeyObjectImpl();
lKey.setValue("Sex", new Integer(1));
OrderObject lOrder = new OrderObjectImpl();
lOrder.setValue("Name", 1);
lOrder.setValue("Firstname", 2);
QueryResult lEntries = lHome.select(lKey, lOrder);
```

This code produces the following SQL query string:

```
SELECT tblMember.SLANGUAGE, tblMember.SUSERID, tblMember.SFAX,
  tblMember.STEL, tblMember.SFIRSTNAME, tblMember.SNAME, tblMember.SSTREET,
  tblMember.SPASSWORD, tblMember.MEMBERID, tblMember.BSEX, tblMember.SZIP,
  tblMember.SCITY, tblMember.SMAIL, tblMember.DTMUTATION
  FROM tblMember
  WHERE tblMember.BSEX = 1
  ORDER BY tblMember.SNAME, tblMember.SFIRSTNAME
```

SELECT … WHERE … LIKE …

The following code fragment uses the *LIKE* comparison operator in the key object:

```
DomainObjectHome lHome = (DomainObjectHome)
  VSys.homeManager.getHome("org.hip.vif.bom.impl.MemberHomeImpl");
KeyObject lKey = new KeyObjectImpl();
lKey.setValue("Name", "A%", "LIKE");
QueryResult lEntries = lHome.select(lKey);
```

This code produces the following SQL query string:

```
SELECT tblMember.SLANGUAGE, tblMember.SUSERID, tblMember.SFAX,
  tblMember.STEL, tblMember.SFIRSTNAME, tblMember.SNAME, tblMember.SSTREET,
  tblMember.SPASSWORD, tblMember.MEMBERID, tblMember.BSEX, tblMember.SZIP,
  tblMember.SCITY, tblMember.SMAIL, tblMember.DTMUTATION
  FROM tblMember
  WHERE tblMember.SNAME LIKE 'A%'
```

SELECT … WHERE … LIKE … OR. … LIKE

The following code fragment uses two criteria and links them with *OR*:

```
DomainObjectHome lHome = (DomainObjectHome)
  VSys.homeManager.getHome("org.hip.vif.bom.impl.MemberHomeImpl");
KeyObject lKey = new KeyObjectImpl();
lKey.setValue("Name", "A%", "LIKE");
lKey.setValue("Name", "a%", "LIKE", false);
QueryResult lEntries = lHome.select(lKey);
```

This code produces the following SQL query string:

```
SELECT tblMember.SLANGUAGE, tblMember.SUSERID, tblMember.SFAX,
  tblMember.STEL, tblMember.SFIRSTNAME, tblMember.SNAME, tblMember.SSTREET,
  tblMember.SPASSWORD, tblMember.MEMBERID, tblMember.BSEX, tblMember.SZIP,
  tblMember.SCITY, tblMember.SMAIL, tblMember.DTMUTATION
  FROM tblMember
  WHERE tblMember.SNAME LIKE 'A%' OR tblMember.SNAME LIKE 'a%'
```

SELECT … WHERE (… OR …) AND …

The following code fragment groups criteria using nesting of key objects. Each key object set to an outer key object is framed with brackets:

```
DomainObjectHome lHome = (DomainObjectHome)
  VSys.homeManager.getHome("org.hip.vif.bom.impl.MemberHomeImpl");
KeyObject lKey = new KeyObjectImpl();
lKey.setValue("Name", "A%", "LIKE");
lKey.setValue("Name", "a%", "LIKE", false);
KeyObject outKey = new KeyObjectImpl();
outKey.setValue(lKey);
outKey.setValue("Sex", new Integer(1));
QueryResult lEntries = lHome.select(outKey);
```

This code produces the following SQL query string:

```
SELECT tblMember.SLANGUAGE, tblMember.SUSERID, tblMember.SFAX,
  tblMember.STEL, tblMember.SFIRSTNAME, tblMember.SNAME, tblMember.SSTREET,
  tblMember.SPASSWORD, tblMember.MEMBERID, tblMember.BSEX, tblMember.SZIP,
  tblMember.SCITY, tblMember.SMAIL, tblMember.DTMUTATION
  FROM tblMember
  WHERE (tblMember.SNAME LIKE 'A%' OR tblMember.SNAME LIKE 'a%')
  AND tblMember.BSEX = 1
```

SELECT … WHERE UCASE(…) LIKE …

The following code fragment uses an SQL function to modify values in the criterion:

```
DomainObjectHome lHome = (DomainObjectHome)
  VSys.homeManager.getHome("org.hip.vif.bom.impl.MemberHomeImpl");
KeyObject lKey = new KeyObjectImpl();
lKey.setValue("Name", "A%", "LIKE", true,
  DBAdapterSelector.getInstance().getColumnModifierUCase());
QueryResult lEntries = lHome.select(lKey);
```

This code produces the following SQL query string:

```
SELECT tblMember.SLANGUAGE, tblMember.SUSERID, tblMember.SFAX,
  tblMember.STEL, tblMember.SFIRSTNAME, tblMember.SNAME, tblMember.SSTREET,
  tblMember.SPASSWORD, tblMember.MEMBERID, tblMember.BSEX, tblMember.SZIP,
  tblMember.SCITY, tblMember.SMAIL, tblMember.DTMUTATION
  FROM tblMember
  WHERE UCASE(tblMember.SNAME) LIKE 'A%'
```

SELECT … WHERE … BETWEEN … AND …

To retrieve entries that fall in a specified data range, you can use the `BetweenObject`. The following code fragment uses an SQL function to modify values in the criterion:

```
DomainObjectHome lHome = (DomainObjectHome)
  VSys.homeManager.getHome("org.hip.vif.bom.impl.MemberHomeImpl");
KeyObject lKey = new KeyObjectImpl();
Date lDate1 = Date.valueOf("1989-08-20");
Date lDate2 = Date.valueOf("1999-04-24");
SQLRange lBetween = new BetweenObjectImpl(lDate1, lDate2);
lKey.setValue("Mutation", lBetween);
QueryResult lEntries = lHome.select(lKey);
```

This code produces the following SQL query string:

```
SELECT tblMember.SLANGUAGE, tblMember.SUSERID, tblMember.SFAX,
  tblMember.STEL, tblMember.SFIRSTNAME, tblMember.SNAME, tblMember.SSTREET,
  tblMember.SPASSWORD, tblMember.MEMBERID, tblMember.BSEX, tblMember.SZIP,
  tblMember.SCITY, tblMember.SMAIL, tblMember.DTMUTATION FROM tblMember
  WHERE tblMember.DTMUTATION  BETWEEN DATE('1989-08-20') AND
  DATE('1999-04-24')
```

SELECT … WHERE … IN …

In a similar manner you can utilize the `InObject` range object to filter for entries whose filter values match a specified set of values.

```
DomainObjectHome lHome = (DomainObjectHome)
  VSys.homeManager.getHome("org.hip.vif.bom.impl.MemberHomeImpl");
KeyObject lKey = new KeyObjectImpl();
SQLRange lIn = new InObjectImpl(new Object[] {new Integer(18),
  new Integer(45), new Integer(65)});
lKey.setValue("Age", lIn);
QueryResult lEntries = lHome.select(lKey);
```

This code produces the following SQL query string:

```
SELECT tblMember.SLANGUAGE, tblMember.SUSERID, tblMember.SFAX,
  tblMember.STEL, tblMember.SFIRSTNAME, tblMember.SNAME, tblMember.SSTREET,
  tblMember.SPASSWORD, tblMember.MEMBERID, tblMember.NAGE, tblMember.SZIP,
  tblMember.SCITY, tblMember.SMAIL, tblMember.DTMUTATION FROM tblMember
  WHERE tblMember.NAGE  IN (18, 45, 65)
```

8.2. Joined domain objects

Joined domain objects can be used to retrieve data from table joins. Imagine a situation where a member can register to various groups. The data can be modeled as depicted in figure. We may be interested in the data of all groups a specified member is participating in, i.e. which are associated to the specified member. The joined domain object's object definition string is shown in Code 13 (see Appendix C: Example of object definitions).

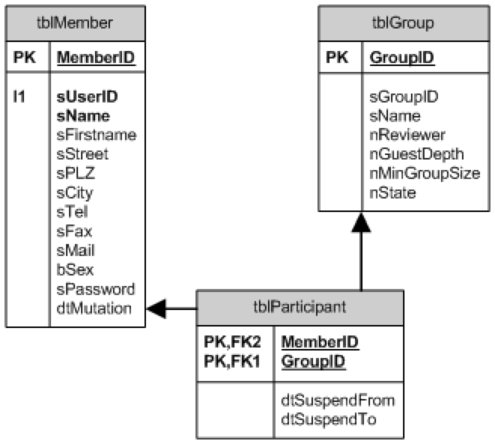


Table joins don't have insert, update or delete functionality. Therefore, we only show how to select.

The following code fragment selects all entries matching the specified criterion, i.e. it selects all groups associated with the specified member:

```
JoinedDomainObjectHome lHome = ((JoinedDomainObjectHome)
    VSys.homeManager.getHome("org.hip.vif.bom.impl.JoinMemberToGroupHome"));
KeyObject lKey = new KeyObjectImpl();
lKey.setValue("MemberID", new BigDecimal(217));
QueryResult lEntries = lHome.select(lKey);
```

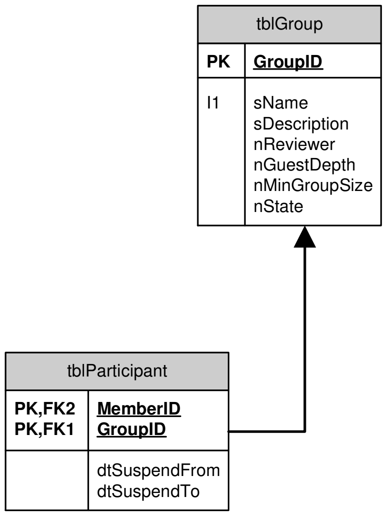
This code produces the following SQL query string (in the case of a MySQL database, see Chapt. 6):

```
SELECT tblParticipant.MEMBERID, tblGroup.GROUPID, tblGroup.SGROUPID,
    tblGroup.SNAME, tblGroup.NREVIEWER, tblGroup.NGUESTDEPTH,
    tblGroup.NMINGROUPSIZE, tblGroup.NSTATE
    FROM tblParticipant INNER JOIN tblGroup
    ON tblParticipant.GROUPID = tblGroup.GROUPID
    WHERE tblParticipant.MEMBERID = 75
```

The WHERE clause of the select statement can be further specified, as in the case of simple domain objects.

8.3. Nested queries

SQL allows the construction of nested queries to retrieve calculated values of grouped fields. The use of nested queries is supported by the Domain Object framework too.



Imagine a situation where we have different groups and various persons that can register to one or more groups. We then want to retrieve not only the group values, but also the number of participants each group has at a certain moment. With a SQL statement like the following we can retrieve such a query:

```
SELECT tblGroup.SNAME, tblGroup.GROUPID, tblGroup.SDESCRIPTION,
   tblGroup.NMINGROUPSIZE, tblGroup.NSTATE, tblGroup.NREVIEWER,
   tblGroup.NGUESTDEPTH, count.Registered FROM tblGroup INNER JOIN
   (SELECT tblParticipant.GROUPID, COUNT(tblParticipant.MEMBERID)
   AS Registered FROM tblParticipant GROUP BY tblParticipant.GROUPID)
   count ON tblGroup.GROUPID = count.GROUPID WHERE tblGroup.NSTATE = '2'
   ORDER BY tblGroup.SNAME, tblGroup.GROUPID
```

Obviously, this is not a simple statement. However, with an adjusted object definition, you can create `DomainObjectHomes` which are able to create such queries. You can find the object definition for this SQL statement in Code 14. Because nested domain objects are an extended version of joined domain objects, nested queries can be created using `JoinedDomainObjectHomes`:

```
JoinedDomainObjectHome lHome = ((JoinedDomainObjectHome)
   VSys.homeManager.getHome("org.hip.vif.bom.impl.NestedGroupHome"));
KeyObject lKey = new KeyObjectImpl();
lKey.setValue("State", new BigDecimal(3));
QueryResult lEntries = lHome.select(lKey);
```

8.4. Nested queries with placeholders

Having a complex data model as backend, you frequently need complex queries to retrieve data in a useful form. Sometimes you might need nested queries containing a filter condition with a value you don't know in advance but results from conditions at runtime.

Thus you can't create such queries using the nested queries depicted in 8.3. Instead, you have to work with placeholders that are filled at runtime. You define placeholders in a similar way you define nested objects in the object description of a Joined Object Model:

```
 <?xml version='1.0' encoding='ISO-8859-1'?>
 <joinedObjectDef objectName='NestedParticipantsOfGroup'
parent='org.hip.kernel.bom.ReadOnlyDomainObject' version='1.0'>
     <columnDefs>
          <columnDef columnName='GroupID'
               domainObject='org.hip.vif.bom.impl.ParticipantImpl'/>
 …
          <columnDef columnName='MemberID' as='GroupAdminID'
               nestedObject='NestedSelect'
               domainObject='org.hip.vif.bom.impl.GroupAdminImpl'/>
 …
     </columnDefs>
     <joinDef joinType='EQUI_JOIN'>
          <objectDesc objectClassName='org.hip.vif.bom.impl.ParticipantImpl'/>
          <objectPlaceholder name='NestedSelect' />
          <joinCondition>
               <columnDef columnName='MemberID'
                    domainObject='org.hip.vif.bom.impl.ParticipantImpl'/>
               <columnDef columnName='MemberID' nestedObject='NestedSelect'
                    domainObject='org.hip.vif.bom.impl.GroupAdminImpl'/>
          </joinCondition>
     </joinDef>
 </joinedObjectDef>
```

Code 4: A joined object definition defining a placeholder

The placeholder is defined using the `objectPlaceholder` element. You have to provide a name (resulting in a `AS name` piece in the SQL command) for that the placeholder can be referenced at other places of the SQL command.

Having defined such a nested object model home, you can fill the home at runtime with any domain object home you like to create the SQL query you need. To do that, you create an instance of `PlacefillerCollection` and add the home and key to create the SQL SELECT query you want to fill in and the name for referencing this query:

```
 NestedHomeWithPlaceholder lPlaceholderHome =
     (NestedHomeWithPlaceholder)VSys.homeManager.getHome(HOME1_CLASS_NAME);

 SimpleHome lPlacefillerHome =
     (SimpleHome)VSys.homeManager.getHome(HOME2_CLASS_NAME);

 KeyObject lKey = new KeyObjectImpl();
 lKey.setValue(NAME, "VIF Application");
```

```
PlacefillerCollection lPlacefillers = new PlacefillerCollection();
lPlacefillers.add(lPlacefillerHome, lKey, "NestedSelect");

lKey = new KeyObjectImpl();
lKey.setValue(NAME, "Framework");

QueryResult lResult = lParticipantsHome.select(lKey,
   new OrderObjedImp("Date"), lPlacefillers);
```

Code 5: Example working with placeholders

This `PlacefillerCollection` instance is passed to the `select()` method to create the final query.

8.5. Using AS and ALIAS

You can use the `alias` attribute to rename a certain attribute in the domain object model but leave the corresponding table column unchanged. Using the `as` attribute you rename both the domain object model attribute and the underlying table column (by adding `AS` *`renamed`* to the SQL SELECT command).

```
…
<columnDefs>
   <columnDef columnName='ID' alias='MyGroupID'
        domainObject='org.vif.bom.impl.GroupImpl/>
   <columnDef columnName='sName' as='MyGroupName'
        domainObject='org.vif.bom.impl.GroupImpl/>
…
<columnDefs>
…
```

Code 6: Joined object definition example with ALIAS and AS

With the domain object framework you separate the data use in an application from the underlying data structure. The model attributes exposed to the application are completely separated from the field names in the database tables.

Combining the data from various tables using Joinded Domain Objects, this might lead to difficulties. In the layer of the SQL commands, each column retrieved is easily identified by the unique combination `table_name.field_name`. However, joining different domain objects can lead to a problem if different domain objects use similar names for their model attributes. Using the `alias` attribute in the specification of your Joined domain objects model you can work around such problems.

Using the `as` attribute you can work around a different problem. The idea behind Joined Domain Objects is to reuse the domain object model specifications, i.e. the mapping of domain object model attributes to the underlying table structure. Using Joined Domain Objects Models you don't know the field names of the underlying tables you're joining, you even don't want to know that.

In some situations, however, you want to control the access to the underlying structure in a traceless way. By using the `as` attribute the domain object framework creates SQL code like `(table_name.field_name) AS myColumn`. Thus, you have control over the access of the retrieved column by defining its virtual name for a specific query without altering the underlying data structure.

8.6. Column modifiers and templates

You can create virtual columns containing calculated data based upon values in other table columns using a `modifier` or `template` attribute in the object definition of your Joined Object Model home:

```
<columnDefs>
…
   <columnDef columnName='GroupID' modifier='COUNT' valueType='Number'
        domainObject='org.hip.vif.bom.impl.GroupImpl'/>
   <columnDef columnName='dtRegistered' template='NOW() &gt;= {0}'
        as='RegisteredBefore' valueType='Number'
        domainObject='org.hip.vif.bom.impl.GroupImpl'/>
…
</columnDefs>
```

Code 7: Object definition with *modifier* and *template* attribute

You can use a modifier from the following list: COUNT, AVG, MIN, MAX, SUM, VARIANCE, STDDEV (see DTD of Joined Domain Object definition, Code 11). Using for example COUNT as modifier will create the a piece of SQL command like `COUNT(table_name.field_name)`.

Using the template attribute you're not restricted to the list of modifiers. You only have to provide a template representing valid SQL code and containing a placeholder according to the rules of Java's `MessageFormat` class. For example the template `NOW() &gt;= {0}` will create the SQL code: `NOW() >= table_name.field_name`. Note that you have to escape the characters < or > in the templates with `&lt;` and `&gt;` respectively.

8.7. Creating Set operations

SQL offers the feature of Set operations to combine the retrieved rows of two (or more) SELECT statements. Such Set operations have the form of:

```
(SELECT … FROM … WHERE …) UNION (SELECT … FROM … WHERE …)
```

Possible Set operations are: UNION, UNION ALL, INTERSECT, MINUS. The set of columns must correspond in the data retrieved by the various SELECT statements.

The domain object framework supports this feature. Here's an example of a UNION Set operation using the Domain Object framework:

```
KeyObject lKey1 = new KeyObjectImpl();
lKey1.setValue(NAME, "VIF Application");
KeyObject lKey2 = new KeyObjectImpl();
lKey2.setValue(NAME, "Framework");

MyHomeVers1 lMyHome1 =
   (MyHomeVers1)VSys.homeManager.getHome(HOME1_CLASS_NAME);
MyHomeVers2 lMyHome2 =
   (MyHomeVers2)VSys.homeManager.getHome(HOME2_CLASS_NAME);
SetOperatorHome lUnionHome = new SetOperatorHomeImpl(SetOperatorHome.UNION);
lUnionHome.addSet(lMyHome1, lKey1);
lUnionHome.addSet(lMyHome2, lKey2);
```

```
QueryResult lResult = lUnionHome.select(new OrderObjectImpl(DATE));
```

Code 8: Example of a Set operation

First the two homes are created whose retrieved data should be unified. Then an instance of the `SetOperatorHome` is created. The type of the Set operation is passed to its constructor. Then this set operator home is initialized with the homes and keys. Having prepared such a home, the combined data can be retrieved by calling its `select()` method.

8.8. Hidden columns

Hidden columns are columns that do not show up in the columns list of a SQL SELECT command. They are needed in combination with Set operations. Hidden columns are defined as child elements of a `columnDefs` element, e.g.:

```
…
<columnDefs>
   <columnDef columnName='ID' domainObject='org.vif.bom.impl.GroupImpl/>
   <columnDef columnName='sName' domainObject='org.vif.bom.impl.GroupImpl/>
…
   <hidden columnName='dtDate' domainObject='org.vif.bom.impl.GroupImpl/>
…
</columnDefs>
…
```

Code 9: Defining a hidden column in a joinedObjectDef

With hidden columns you can pass the information to the domain object model how to map a certain attribute of the model to the table column in the database. The domain object model needs such information not only to retrieve data in the column of a SQL SELECT command, but to create the SQL filter expression (`WHERE key = value`) too. Usually, you pass such information via the `columnDef` element. In combination with the Set operations, however, you're no longer free to define SQL columns. Instead, the two sets you retrieve and want to combine (using the Set operation) have to correspond on the columns returned. Thus you need to provide mapping information without the need to create a new column in the SELECT command. This can be done with hidden columns.

9. Lightweight Homes and Models

The strength of the VIF Domainobject Framework is its comprehensive XML support through its serialization features. In this view, a single domain object as well as a set of domain objects are XML and, thus, can be processed and transformed with XSLT. XSLT transformation is a good choice for thin client applications where the input is processed on a server and the data is sent to the users web browser.

In a rich client application, however, XSLT is not of much use. In such a context, the models created by the VIF Domainobject Framework are too heavy and too powerful. Especially in cases where large data sets have to be retrieved and filled in tables or lists, these models are too heave to be efficient. For such cases, the VIF Domainobject Framework offers the possibility of lightweight domain object creation through the use of `AlternativeModelFactory` and `AlternativeModel` interfaces.

A model class implementing the `AlternativeModel` interface can be a lightweight object that holds the model's data in public attributes and offers only limited functionality:

```
public class LightWeightItem extends AbstractLightWeight
      implements AlternativeModel {
  public long id;
  public String title;
  public String text;
  public Timestamp created;
  public Timestamp modified;

  public LightWeightItem (long inID, String inTitle, String inText,
      Timestamp inCreated, Timestamp inModified) {
      super();
      id = inID;
      title = inTitle;
      text = inText;
      created = inCreated;
      modified = inModified;
  }

  public String toString() {
      return title;
  }
}
```

In the factory class implementing the `AlternativeModelFactory` interface, the object relational mapping takes place:

```
private class LWItemFactory implements AlternativeModelFactory {
  public AlternativeModel createModel(ResultSet inResultSet)
                          throws SQLException {
      return new LightWeightItem(inResultSet.getLong("TermID"),
                 inResultSet.getString("sTitle"),
                 inResultSet.getString("sText"),
                 inResultSet.getTimestamp("dtCreation"),
                 inResultSet.getTimestamp("dtMutation"));
  }
}
```

What do you have to do to create collections of `AlternativeModels` using the VIF Domainobject Framework? The first thing you have to implement is a CollectableItemHome that derives from your normal domain object's home:

```
public class CollectableItemHome extends TermHome {
 public CollectableItemHome() {
      super();
 }

 /**
  * Overrides to create an AlternativeQueryStatement.
  *
  * @return org.hip.kernel.bom.QueryStatement
  */
 public QueryStatement createQueryStatement() {
      return new AlternativeQueryStatement(this);
 }

 /**
  * Overrides to create an AlternativeQueryResult.
  *
  * @param inStatement QueryStatement
  * @return QueryResult
  * @throws SQLException
  */
 public QueryResult select(QueryStatement inStatement)
          throws SQLException {
      if (VSys.assertNotNull(this, "select(QueryStatement)",
                              inStatement) == Assert.FAILURE)
          return new AlternativeQueryResult(null, null, null);

      QueryResult outResult = inStatement.executeQuery();
      return outResult;
 }
}
```

The important points here are that the home overrides the `createQueryStatement()` method to create an instance of `AlternativeQueryStatement` as well as the `select()` method. This `AlternativeQueryStatement` will create an `AlternativeQueryResult` afterward that will be returned when the collectable home's `select()` method is called. With these preparations, we're able to fit the parts together:

```
DomainObjectHome lHome = new CollectableItemHome();
QueryResult lResult = lHome.select();
Collection lItems = lResult.load(new LWItemFactory());
```

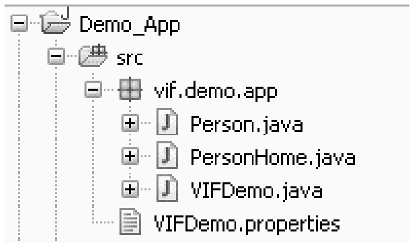
First, we create an instance of `CollectableItemHome`. With this home, we create an instance of `AlternativeQueryResult` using the home's `select()` method. This query result object knows what to do when we call its `load()` method and pass and instance of `AlternativeModelFactory`. In our case, we pass an instance of `LWItemFactory` that creates a `LightWeightItem` for each entry retrieved and adds it to the collection returned. So we end up with a collection of lightweight items retrieved from our data store using the VIF Domainobject Framework.

10. Sample application

Let's look how we can store, retrieve and modify data in a database using the framework. Assume we have a Derby database *derbyTest* with a table `tblPerson` created with the following SQL command:

```
CREATE TABLE tblPerson (
  PersonID      BIGINT generated always as identity,
  sName         VARCHAR(99) not null,
  sFirstname    VARCHAR(50),
  sText         CLOB,
  nYear         INTEGER,
  dtCreation    timestamp NOT NULL,
  PRIMARY KEY (PersonID)
);
```

The simple demo application consists of three class files and one properties file:



The entries in this table are represented by instances of the `Person` class, managed by the `PersonHome`. (See chapter 7 for home and model creation). `VIFDemo` is the sample application.

The properties file specifies how to connect to the database:

```
org.hip.vif.db.driver=org.apache.derby.jdbc.EmbeddedDriver
org.hip.vif.db.url=jdbc:derby:/file/system/path/to/derbyTest
org.hip.vif.db.userId=app
org.hip.vif.db.password=
```

The model's home class contains the information to map the model's attribute to the table fields in the database:

```
package vif.demo.app;

import java.sql.*;
import java.util.Vector;

import org.hip.kernel.bom.DomainObject;
import org.hip.kernel.bom.impl.DomainObjectHomeImpl;
import org.hip.kernel.exc.VException;

public class PersonHome extends DomainObjectHomeImpl {
  public final static String KEY_ID = "ID";
  public final static String KEY_NAME = "Name";
  public final static String KEY_FIRSTNAME = "Firstname";
```

```
    public final static String KEY_TEXT = "Text";
    public final static String KEY_YEAR = "Year";
    public final static String KEY_CREATION = "Creation";

    private final static String OBJECT_CLASS_NAME = "vif.demo.app.Person";

    private final static String XML_OBJECT_DEF =
        "<?xml version='1.0' encoding='ISO-8859-1'?> " +
        "<objectDef objectName='PersonImpl'
            parent='org.hip.kernel.bom.DomainObject' version='1.0'>   " +
        "    <keyDefs> " +
        "        <keyDef>    " +
        "            <keyItemDef seq='0' keyPropertyName='" + KEY_ID +
                                                              "'/>  " +
        "        </keyDef>  " +
        "    </keyDefs>     " +
        "    <propertyDefs>  " +
        "        <propertyDef propertyName='" + KEY_ID + "'
                     valueType='Number' propertyType='simple'> " +
        "            <mappingDef tableName='tblPerson'
                     columnName='PersonID'/> " +
        "        </propertyDef>   " +
        "        <propertyDef propertyName='" + KEY_NAME + "'
                     valueType='String' propertyType='simple'> " +
        "            <mappingDef tableName='tblPerson'
                     columnName='sName'/>" +
        "        </propertyDef>   " +
        "        <propertyDef propertyName='" + KEY_FIRSTNAME + "'
                     valueType='String' propertyType='simple'> " +
        "            <mappingDef tableName='tblPerson'
                     columnName='sFirstname'/>     " +
        "        </propertyDef>   " +
        "        <propertyDef propertyName='" + KEY_TEXT + "'
                     valueType='String' propertyType='simple'> " +
        "            <mappingDef tableName='tblPerson'
                               columnName='sText'/>    " +
        "        </propertyDef>   " +
        "        <propertyDef propertyName='" + KEY_YEAR + "'
                     valueType='Number' propertyType='simple'> " +
        "            <mappingDef tableName='tblPerson'
                     columnName='nYear'/>    " +
        "        </propertyDef>   " +
        "        <propertyDef propertyName='" + KEY_CREATION + "'
                valueType='Timestamp' propertyType='simple'>    " +
        "            <mappingDef tableName='tblPerson'
                     columnName='dtCreation'/>     " +
        "        </propertyDef>   " +
        "    </propertyDefs> " +
        "</objectDef>";

    public PersonHome() {
        super();
    }

    public String getObjectClassName() {
        return OBJECT_CLASS_NAME;
    }

    protected String getObjectDefString() {
        return XML_OBJECT_DEF;
    }

    protected Vector createTestObjects() {
        return null;
    }

    /**
     * Adds a new entry to the person table.
     *
     * @return Long The unique id of the new entry.
     */
    public Long newPerson(String inName, String inFirstName,
            String inNote, int inYear) throws VException, SQLException {
        DomainObject lPerson = create();
        lPerson.set(PersonHome.KEY_NAME, inName);
        lPerson.set(PersonHome.KEY_FIRSTNAME, inFirstName);
        lPerson.set(PersonHome.KEY_TEXT, inNote);
        lPerson.set(PersonHome.KEY_YEAR, new Integer(inYear));
        lPerson.set(PersonHome.KEY_CREATION,
                    new Timestamp(System.currentTimeMillis()));
        return lPerson.insert(true);
    }
}
```

The mapping is done in the `XML_OBJECT_DEF`; which is the key element of each home class. Developers using the *Eclipse IDE* can use the *VIFDom Plug in* that simplifies greatly the creation of this object def.

The person home above provides the helper method `newPerson()` to insert new entries into the table. Such an entry is represented in the application by a `Person` model:

```
package vif.demo.app;

import java.sql.SQLException;

import org.hip.kernel.bom.impl.DomainObjectImpl;
import org.hip.kernel.exc.VException;

public class Person extends DomainObjectImpl {
    public final static String HOME_CLASS_NAME = "vif.demo.app.PersonHome";

    public Person() {
        super();
    }

    public String getHomeClassName() {
        return HOME_CLASS_NAME;
    }
    /**
     * Updates this person entry.
     */
    public void update(String inName, String inFirstName,
              String inNote, int inYear) throws VException, SQLException {
         set(PersonHome.KEY_NAME, inName);
         set(PersonHome.KEY_FIRSTNAME, inFirstName);
         set(PersonHome.KEY_TEXT, inNote);
         set(PersonHome.KEY_YEAR, new Integer(inYear));
         update(true);
    }
   }
```

The person model above provides the method `update()` the application can use to modify table entries. With these preparations we're ready to implement the application:

```
package vif.demo.app;

import java.sql.*;
import java.util.*;

import org.hip.kernel.bom.*;
import org.hip.kernel.exc.VException;
import org.hip.kernel.sys.VSys;

/**
 * Demo project demonstrating the use of the VIF Framework.
 */
public class VIFDemo {
 private final static String KEY_DRIVER
                          = "org.hip.vif.db.driver";
 private final static String KEY_URL        = "org.hip.vif.db.url";
 private final static String KEY_USER_ID    = "org.hip.vif.db.userId";
 private final static String KEY_USER_PWRD  = "org.hip.vif.db.password";

 private PersonHome personHome;

 public static void main(String[] args) {
      VIFDemo lDemo = new VIFDemo();
      Long lID;

      try {
           lID = lDemo.createNew();
           lDemo.showData();
           lDemo.update(lID);
           lDemo.showData();
      }
      catch (Exception exc) {
           exc.printStackTrace();
      }
 }

 public VIFDemo() {
      super();
      initDBSettings();
      personHome =
           (PersonHome)VSys.homeManager.getHome(Person.HOME_CLASS_NAME);
  }

  private void initDBSettings() {
      ResourceBundle lBundle =
                PropertyResourceBundle.getBundle("VIFDemo");
      Properties lProperties = new Properties();
      lProperties.setProperty(KEY_DRIVER, lBundle.getString(KEY_DRIVER));
      lProperties.setProperty(KEY_URL, lBundle.getString(KEY_URL));
      lProperties.setProperty(KEY_USER_ID,
                lBundle.getString(KEY_USER_ID));
      lProperties.setProperty(KEY_USER_PWRD,
                lBundle.getString(KEY_USER_PWRD));
      VSys.setVSysProperties(lProperties);
  }

  public Long createNew() throws VException, SQLException {
      return personHome.newPerson("Doe", "John", "First entry", 1995);
  }

  public void update(Long inID) throws VException, SQLException {
      KeyObject lKey = new KeyObjectImpl();
      lKey.setValue(PersonHome.KEY_ID, inID);

      Person lPerson = (Person)personHome.findByKey(lKey);
      lPerson.update("Doe", "Jane", "Changed entry", 1995);
  }

  public void showData() throws VException, SQLException {
      SQLRange lRange = new BetweenObjectImpl(
                         Timestamp.valueOf("2006-01-01 00:00:00"),
                         Timestamp.valueOf("2020-10-30 00:00:00"));
      KeyObject lKey = new KeyObjectImpl();
      lKey.setValue(PersonHome.KEY_CREATION, lRange);

      QueryResult lResult = personHome.select(lKey);

      while (lResult.hasMoreElements()) {
           Person lPerson = (Person)lResult.next();
           String lName = lPerson.get(PersonHome.KEY_NAME).toString();
           String lFirstName =
                    lPerson.get(PersonHome.KEY_FIRSTNAME).toString();
           String lNote = lPerson.get(PersonHome.KEY_TEXT).toString();
           System.out.println(lFirstName + " " + lName + ": " + lNote);
      }
  }

}
```

The sample application creates first an instance of `VIFDemo`. In the class' constructor, the framework settings are initialized (see the method i`nitDBSettings()`), i.e. the application's properties file is evaluated. Having this instance created, it's possible to create a new entry in the table (method `createNew()`). To demonstrate the proper creation, the table content is written to the console (method `showData()`). Next, the entry is retrieved and modified (method `update()`). To demonstrate that the modifications are stored, the actual table content is again written to the console.

The i`nitDBSettings()` method retrieves the properties concerning the database connectivity and passes them to the framework's `VSys` object. With this information, the framework can set up the connection manager needed to provide database connectivity. The `createNew()` method is very simple. It takes the model's home and calls the method `newPerson()` on it, providing the data that has to be stored in the entry. This method returns the new entry's unique id back to the application. This id, then, can be used to retrieve the item and edit it. An example of how to do this is shown in method `update()`. This method takes the entry's unique id to creates a key object. With this key object, the entry can be retrieved using the home's `findByKey()` method. Having retrieved the model, we can change it's content using the `update()` method we've implemented in the `Person` class. The `showData()` method is an example of a database query using an `SQLRange` object. A key object is created with the range set and passed to the home's `select()` method. This method returns a `QueryResult` that is evaluated in the subsequent loop.

This little application writes the following data to the console:

```
John Doe: First entry
Jane Doe: Changed entry
```

Appendix A: Class diagram framework

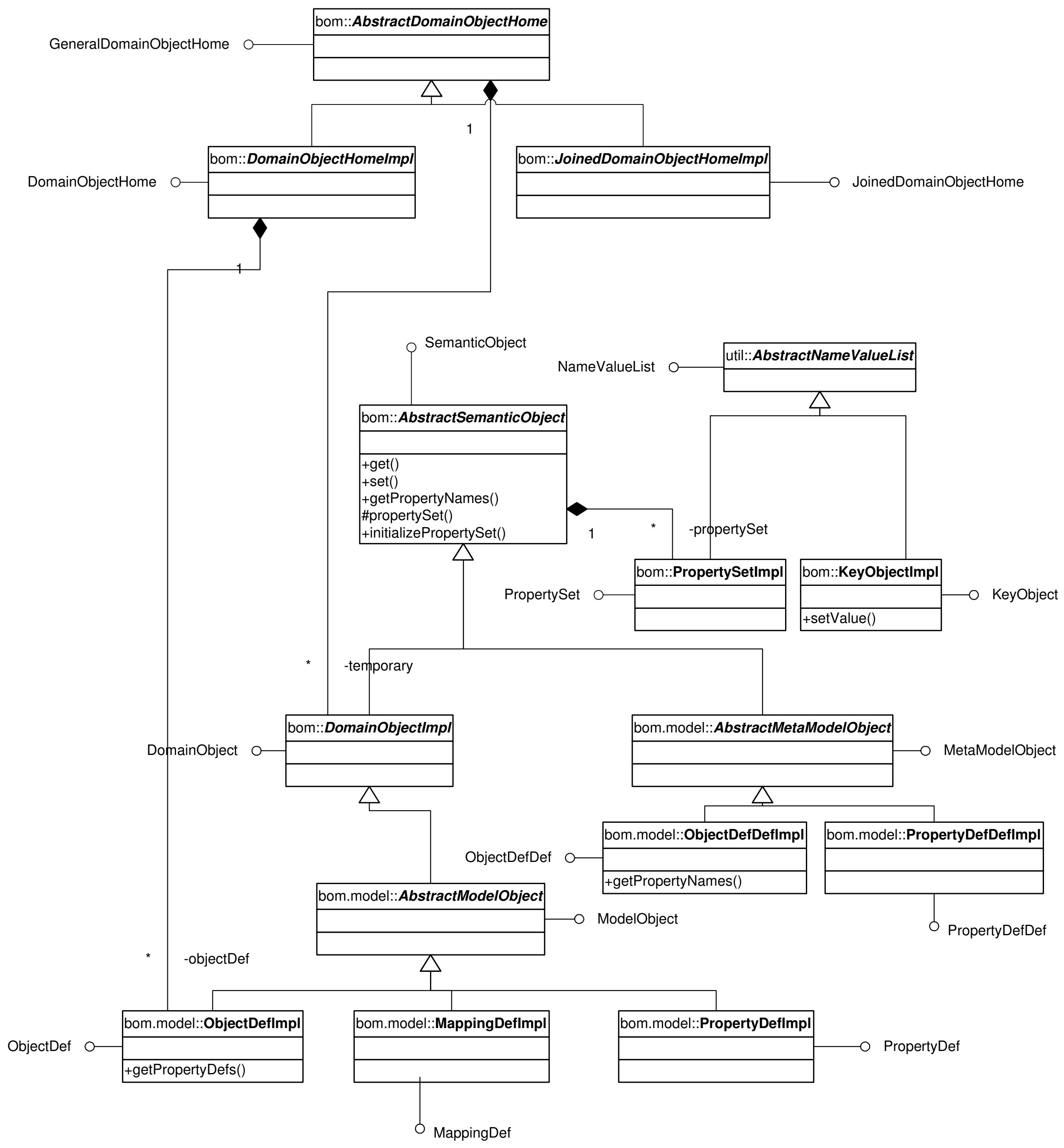


Diagram 4: Class diagram of the domain object framework

Appendix B: Document Type Definitions (DTD)

```
<!ELEMENT objectDef (keyDefs, propertyDefs)>
<!ATTLIST objectDef
     objectName CDATA #REQUIRED
     parent CDATA #REQUIRED
     version CDATA #REQUIRED
     baseDir CDATA #IMPLIED
>
<!ELEMENT keyDefs (keyDef)>
<!ELEMENT keyDef (keyItemDef+)>
<!ELEMENT keyItemDef EMPTY>
<!ATTLIST keyItemDef
     seq (0 | 1) #REQUIRED
     keyPropertyName IDREF #REQUIRED
>
<!ELEMENT propertyDefs (propertyDef+)>
<!ELEMENT propertyDef (mappingDef)>
<!ATTLIST propertyDef
     propertyName ID #REQUIRED
     valueType (Number | String | Date | Timestamp) #REQUIRED
     propertyType (simple | composite | objectRef) #REQUIRED>
<!ELEMENT mappingDef EMPTY>
<!ATTLIST mappingDef
     tableName CDATA #IMPLIED
     columnName CDATA #REQUIRED
>
```

Code 10: DTD defining a domain object description

```
<!ELEMENT joinedObjectDef (columnDefs+, joinDef+)>
<!ATTLIST joinedObjectDef
     objectName CDATA #REQUIRED
     parent CDATA #REQUIRED
     version CDATA #REQUIRED
>
<!ELEMENT columnDefs (columnDef+, hidden+)>
<!ELEMENT columnDef EMPTY>
<!ATTLIST columnDef
     columnName CDATA #REQUIRED
     alias CDATA #IMPLIED
     as CDATA #IMPLIED
     domainObject CDATA #IMPLIED
     nestedObject CDATA #IMPLIED
     valueType (Number | String | Timestamp) #IMPLIED
     modifier (COUNT | AVG | MIN | MAX | SUM | VARIANCE | STDDEV) #IMPLIED
     template CDATA #IMPLIED
>
<!ELEMENT hidden EMPTY>
<!ATTLIST hidden
     columnName CDATA #REQUIRED
     domainObject CDATA #REQUIRED
>
<!ELEMENT joinDef (objectDesc+, objectNested*, joinCondition+, joinDef*,
objectPlaceholder*)>
<!ATTLIST joinDef
     joinType (EQUI_JOIN | LEFT_JOIN | LEFT_JOIN_OUTER | RIGHT_JOIN |
RIGHT_JOIN_OUTER | NO_JOIN) #REQUIRED
>
<!ELEMENT objectDesc EMPTY>
<!ATTLIST objectDesc
     objectClassName CDATA #REQUIRED
>
<!ELEMENT joinCondition (columnDef+)>
<!ELEMENT objectNested (columnDef+, resultGrouping*)>
<!ATTLIST objectNested
     name CDATA #REQUIRED
>
<!ELEMENT objectPlaceholder EMPTY>
<!ATTLIST objectPlaceholder
     name CDATA #REQUIRED
>
<!ELEMENT resultGrouping (columnDef+)>
<!ATTLIST resultGrouping
     modifier (GROUP | ORDER) #REQUIRED
>
```

Code 11: DTD defining a description of a joined domain object

Appendix C: Example of object definitions

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<objectDef objectName='Member' parent='org.hip.kernel.bom.DomainObject'
         version='1.0'>
    <keyDefs>
         <keyDef>
              <keyItemDef seq='0' keyPropertyName='ID'/>
         </keyDef>
    </keyDefs>
    <propertyDefs>
         <propertyDef propertyName='ID' valueType='Number'
                   propertyType='simple'>
              <mappingDef tableName='tblMember' columnName='MEMBERID'/>
         </propertyDef>
         <propertyDef propertyName='UserID' valueType='String'
                   propertyType='simple'>
              <mappingDef tableName='tblMember' columnName='SUSERID'/>
         </propertyDef>
         <propertyDef propertyName='Name' valueType='String'
                   propertyType='simple'>
              <mappingDef tableName='tblMember' columnName='SNAME'/>
         </propertyDef>
         <propertyDef propertyName='Firstname' valueType='String'
                   propertyType='simple'>
              <mappingDef tableName='tblMember' columnName='SFIRSTNAME'/>
         </propertyDef>
         <propertyDef propertyName='Street' valueType='String'
                   propertyType='simple'>
              <mappingDef tableName='tblMember' columnName='SSTREET'/>
         </propertyDef>
         <propertyDef propertyName='ZIP' valueType='String'
                   propertyType='simple'>
              <mappingDef tableName='tblMember' columnName='SZIP'/>
         </propertyDef>
         <propertyDef propertyName='City' valueType='String'
                   propertyType='simple'>
              <mappingDef tableName='tblMember' columnName='SCITY'/>
         </propertyDef>
         <propertyDef propertyName='Tel' valueType='String'
                   propertyType='simple'>
              <mappingDef tableName='tblMember' columnName='STEL'/>
         </propertyDef>
         <propertyDef propertyName='Fax' valueType='String'
                   propertyType='simple'>
              <mappingDef tableName='tblMember' columnName='SFAX'/>
         </propertyDef>
         <propertyDef propertyName='Mail' valueType='String'
                   propertyType='simple'>
              <mappingDef tableName='tblMember' columnName='SMAIL'/>
         </propertyDef>
         <propertyDef propertyName='Sex' valueType='Number'
                   propertyType='simple'>
              <mappingDef tableName='tblMember' columnName='BSEX'/>
         </propertyDef>
```

```
          <propertyDef propertyName='Language' valueType='String'
                    propertyType='simple'>
               <mappingDef tableName='tblMember' columnName='SLANGUAGE'/>
          </propertyDef>
          <propertyDef propertyName='Password' valueType='String'
                    propertyType='simple'>
               <mappingDef tableName='tblMember' columnName='SPASSWORD'/>
          </propertyDef>
          <propertyDef propertyName='Mutation' valueType='Timestamp'
                    propertyType='simple'>
               <mappingDef tableName='tblMember' columnName='DTMUTATION'/>
          </propertyDef>
     </propertyDefs>
</objectDef>
```

Code 12: Simple domain object: member table

For an explanation of a simple domain object's definition, see Code 2.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<joinedObjectDef objectName='JoinMemberToGroup'
          parent='org.hip.kernel.bom.ReadOnlyDomainObject' version='1.0'>
     <columnDefs>
          <columnDef columnName='ID'
               domainObject='org.hip.vif.bom.impl.GroupImpl'/>
          <columnDef columnName='GroupID'
               domainObject='org.hip.vif.bom.impl.GroupImpl'/>
          <columnDef columnName='Name'
               domainObject='org.hip.vif.bom.impl.GroupImpl'/>
          <columnDef columnName='Reviewers'
               domainObject='org.hip.vif.bom.impl.GroupImpl'/>
          <columnDef columnName='GuestDepth'
               domainObject='org.hip.vif.bom.impl.GroupImpl'/>
          <columnDef columnName='MinGroupSize'
               domainObject='org.hip.vif.bom.impl.GroupImpl'/>
          <columnDef columnName='State'
               domainObject='org.hip.vif.bom.impl.GroupImpl'/>
          <columnDef columnName='MemberID'
               domainObject='org.hip.vif.bom.impl.ParticipantImpl'/>
     </columnDefs>
     <joinDef joinType='EQUI_JOIN'>
          <objectDesc objectClassName='org.hip.vif.bom.impl.ParticipantImpl'/>
          <objectDesc objectClassName='org.hip.vif.bom.impl.GroupImpl'/>
          <joinCondition>
               <columnDef columnName='GroupID'
                    domainObject='org.hip.vif.bom.impl.ParticipantImpl'/>
               <columnDef columnName='ID'
                    domainObject='org.hip.vif.bom.impl.GroupImpl'/>
          </joinCondition>
     </joinDef>
</joinedObjectDef>
```

Code 13: Joined Domain Object: groups associated to a member

The object definition of a joined domain object is made up of two parts. The `<columnDefs>` define the fields that are retrieved by the query. The `<joinDef>` defines how the joining is done. Prerequisite of an object definition of a joined domain object are some simple domain objects. Therefore, the joined object definition can reference the simple domain objects in a way. This is done with the `<columnDef>` entries. Each such entry references a field or attribute in the specified domain object. When you write the `<joinDef>`, you have to specify the type of the join. The values you can use are defined in the DTD of the joined domain object (see Code 11). With the `<objectDesc>` elements you specify which domain objects are joined. In the `<joinCondition>` you define with which field the join is executed.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<joinedObjectDef objectName='NestedGroup'
        parent='org.hip.kernel.bom.ReadOnlyDomainObject' version='1.0'>
    <columnDefs>
        <columnDef columnName='Name'
             domainObject='org.hip.vif.bom.impl.GroupImpl'/>
        <columnDef columnName='ID' alias='" + KEY_GROUP_ID + "'
             domainObject='org.hip.vif.bom.impl.GroupImpl'/>
        <columnDef columnName='Description'
             domainObject='org.hip.vif.bom.impl.GroupImpl'/>
        <columnDef columnName='MinGroupSize'
             domainObject='org.hip.vif.bom.impl.GroupImpl'/>
        <columnDef columnName='State'
             domainObject='org.hip.vif.bom.impl.GroupImpl'/>
        <columnDef columnName='Reviewers'
             domainObject='org.hip.vif.bom.impl.GroupImpl'/>
        <columnDef columnName='GuestDepth'
             domainObject='org.hip.vif.bom.impl.GroupImpl'/>
        <columnDef columnName='Registered' nestedObject='count'
             valueType='Number'/>
    </columnDefs>
    <joinDef joinType='EQUI_JOIN'>
        <objectDesc objectClassName='org.hip.vif.bom.impl.GroupImpl'/>
        <objectNested name='count'>
             <columnDef columnName='GroupID'
                  domainObject='org.hip.vif.bom.impl.ParticipantImpl'/>
             <columnDef columnName='MemberID' alias='Registered'
                  modifier='COUNT'
                  domainObject='org.hip.vif.bom.impl.ParticipantImpl'/>
             <resultGrouping modifier='GROUP'>
                  <columnDef columnName='GroupID'
                       domainObject='org.hip.vif.bom.impl.ParticipantImpl'/>
             </resultGrouping>
        </objectNested>
        <joinCondition>
             <columnDef columnName='ID'
                  domainObject='org.hip.vif.bom.impl.GroupImpl'/>
             <columnDef columnName='GroupID' nestedObject='count'/>
        </joinCondition>
    </joinDef>
</joinedObjectDef>
```

Code 14: Nested Domain Object: groups with number of registered participants

The object definition of a nested domain object again has a `<columnDefs>` part to define the fields that are retrieved by the query. However, the last `<columnDef>` doesn't reference a domain object. Instead the `nestedObject` attribute indicates that this time a nested table is referenced. For the `<joinDef>` you have to specify the `joinType` and the two tables that are joined. However, the second table is a virtual table, which is defined in the `<objectNested>` part. You have to name the nested object for that it can be referenced. The second `<columnDef>` in this nested object is modified, indicated by the `modifier` attribute, telling that the field values are summed up for all records in the group defined by the `<resultGrouping>` element. For that the join can be fulfilled, you have to state the fields joining the two tables by the `<joinCondition>`.

Appendix D: Class diagram database adapters

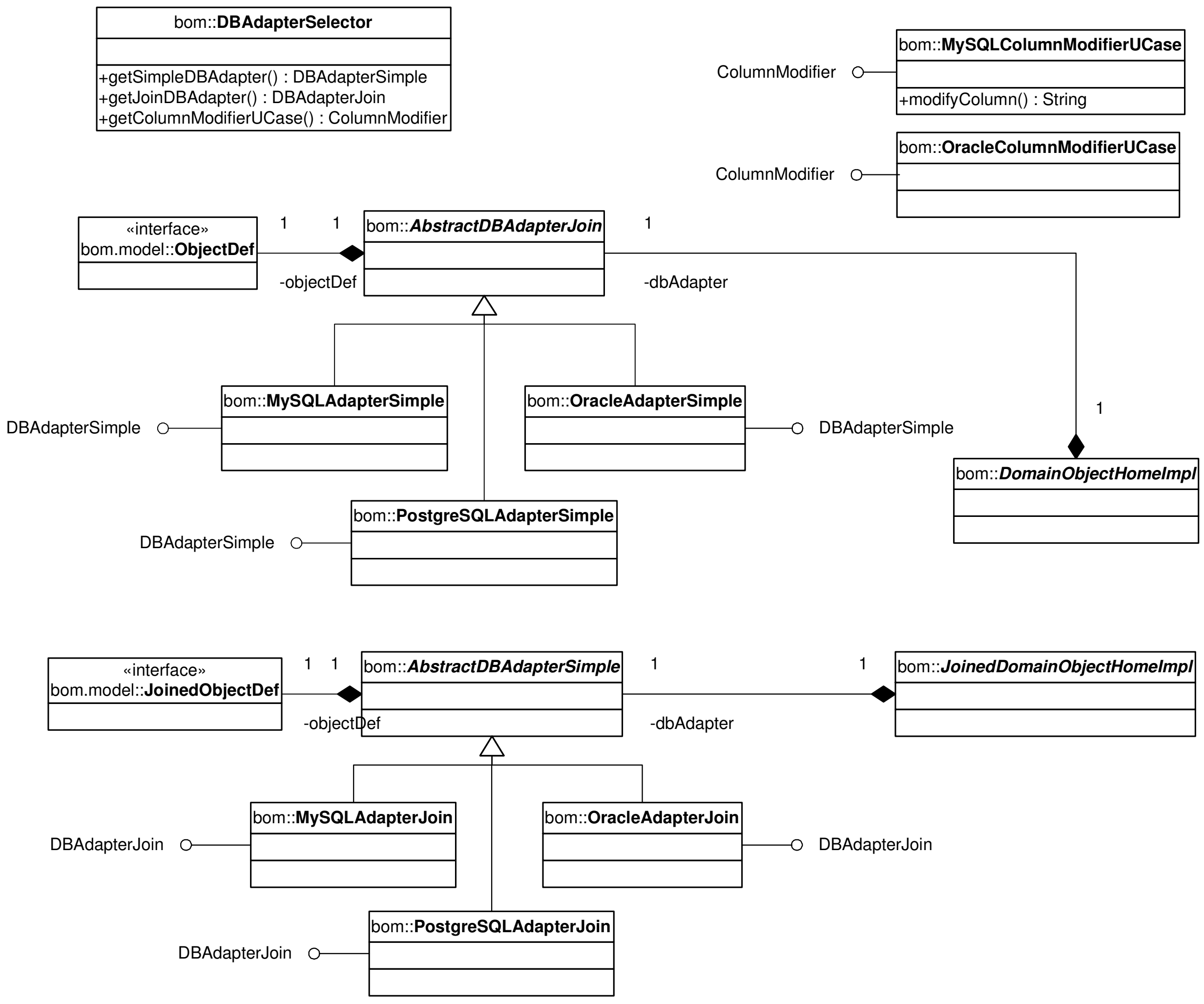

bom::DBAdapterSelector
+getSimpleDBAdapter() : DBAdapterSimple
+getJoinDBAdapter() : DBAdapterJoin
+getColumnModifierUCase() : ColumnModifier
ColumnModifier
bom::MySQLColumnModifierUCase
+modifyColumn() : String
ColumnModifier
bom::OracleColumnModifierUCase
«interface»
bom.model::ObjectDef
1
1
-objectDef
bom::AbstractDBAdapterJoin
1
-dbAdapter
1
bom::DomainObjectHomeImpl
DBAdapterSimple
bom::MySQLAdapterSimple
bom::OracleAdapterSimple
DBAdapterSimple
DBAdapterSimple
bom::PostgreSQLAdapterSimple
«interface»
bom.model::JoinedObjectDef
1
1
-objectDef
bom::AbstractDBAdapterSimple
1
-dbAdapter
1
bom::JoinedDomainObjectHomeImpl
DBAdapterJoin
bom::MySQLAdapterJoin
bom::OracleAdapterJoin
DBAdapterJoin
DBAdapterJoin
bom::PostgreSQLAdapterJoin

Diagram 5: Class diagram of database adapters