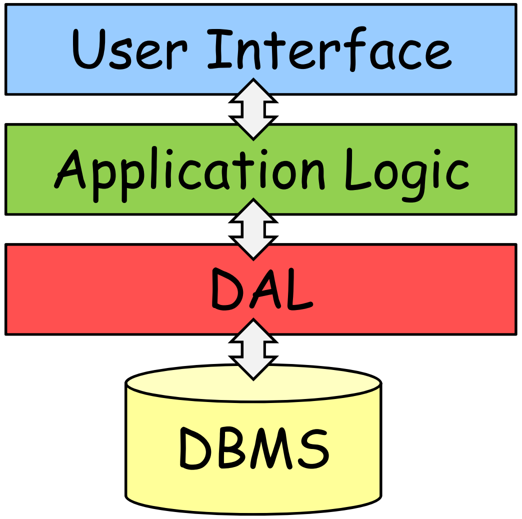
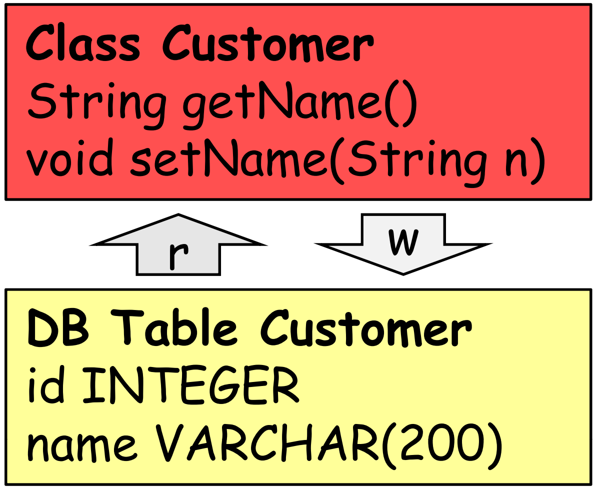
Data Access Layers

Data Access Layer (DAL)

- Application layer that provides functionality for convenient DB access
- Enables the use of OO classes to read and write from/to the DB (instead of having to use SQL)
- Advantages:
  - Higher level of abstraction
  - Separation of concerns makes application easier to maintain
  - Application is independent of particular DB management system (DBMS)



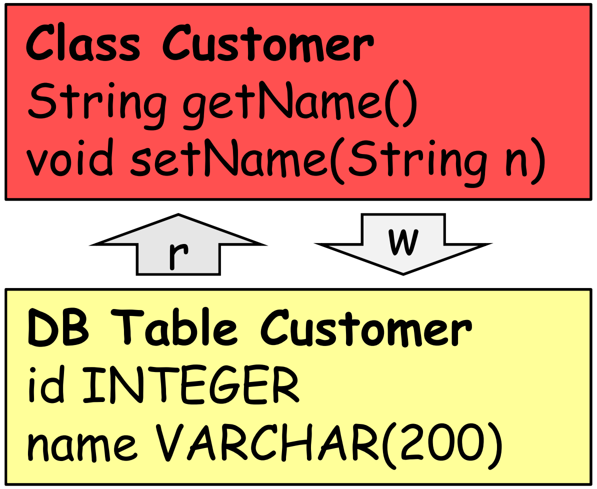


Class Customer
String getName()
void setName(String n)

DB Table Customer
id INTEGER
name VARCHAR(200)

Developing DALs: Writing DALs Manually

- Use a DB access API such as JDBC
- Low level: need to deal with SQL
- and possibly with DB specific code
- Tedious: writing SQL for getters/setters is very repetitive
- Maintenance problem when data model specification changes (DAL needs to be changed as well)



Class Customer
String getName()
void setName(String n)

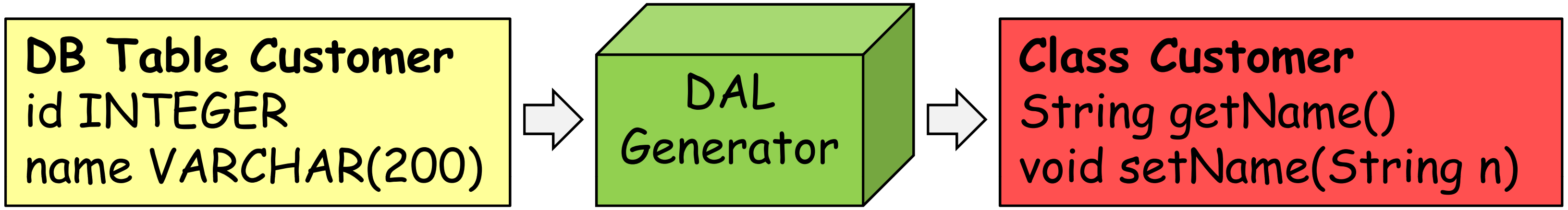
DB Table Customer
id INTEGER
name VARCHAR(200)

```
String getName() throws SQLException {
  Statement s = connection.createStatement();
  ResultSet r = s.executeQuery(
    "SELECT name FROM Customer WHERE id=" + id);
  String name = null;
  if (r.next())  name = r.getString(1);
  s.close();  return name;
}
```

Developing DALs: Generating DALs

Generating the DAL from data model specification

- For each data type, a class with getters and setters is generated by a DAL generator
- Getters/setters read from and write to the DB
- Generator may support different DBMS
- When the data model specification changes, simply re-run the generator to get an updated DAL
- Example: PDStore



DB Table Customer
id INTEGER
name VARCHAR(200)

DAL Generator

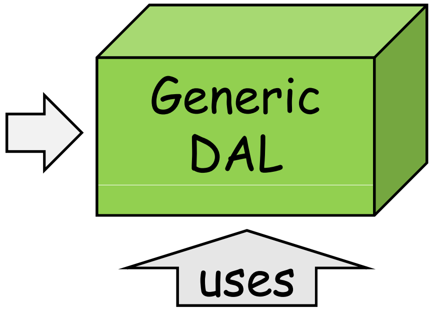
Class Customer
String getName()
void setName(String n)

Developing DALs: Using a Generic DAL

Using a generic DAL with a mapping specification

- Use your own classes for the persistent data types
- Specify how the classes should be mapped to the DB
- Use generic functions to begin/commit transactions and load/save objects from and to the DB
- Example: Hibernate

```
<hibernate-mapping>
  <class name="Customer" table="Customer">
    <id name="id" column="ID"> ... </id>
    <property name="name" type="string"
       column="NAME"/>
</class> </hibernate-mapping>
```

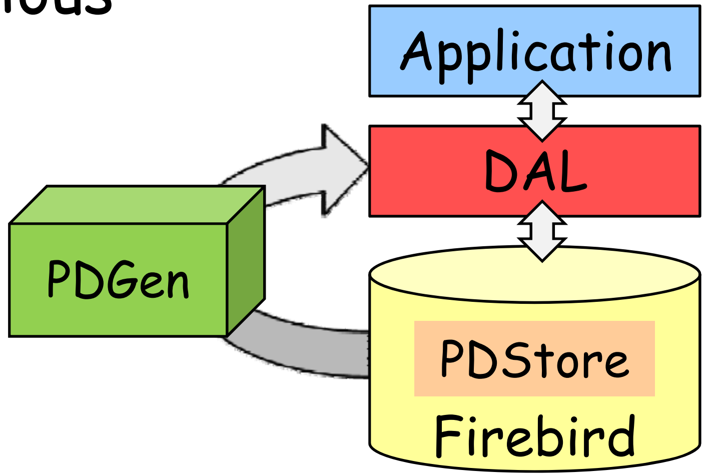


```
session.beginTransaction();
Customer c = new Customer();  c.setName("Joe");
session.save(c);  session.getTransaction().commit();
```

PDStore

PDStore

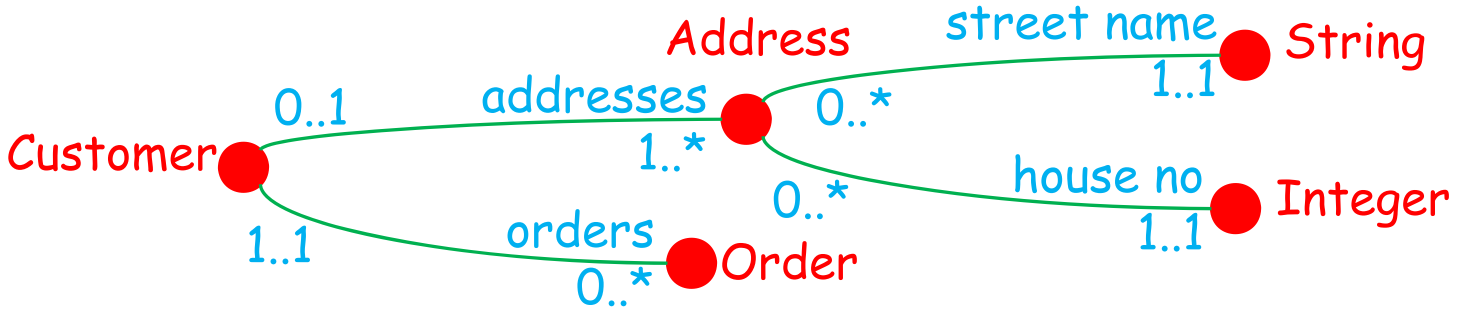
- DB system based the parsimonious data model (PD model)
- Implemented on a on a relational DBMS (Firebird)
- Provides a DAL generator for Java (PDGen)



Advantages:

- All data elements are indentified with GUIDs → data from different DBs can easily be merged
- All changes to the data are logged → changes can be undone/redone, versioning
- Support for change notification → applications can react to changes immediately

Parsimonious Data Model (PD Model)



- Types, Relations and Roles with Multiplicities
- Types are sets of elements
  - Primitive types contain values like strings, ints
  - Complex types contain GUIDs (e.g. for customers)
- Each relation has exactly two roles (one each end)
- Roles may have a name, e.g. “orders”, but need not
- Each role has a minimum and a maximum multiplicity
- Types contain instances, relations contain links

Globally Unique Identifiers (GUIDs)

- A identifier that is globally unique (nothing else in the world has the same identifier)
- Consists of 16 bytes
- GUIDs can be generated using the network card (MAC) address of a computer and a timestamp

In PDStore:

- We can get GUIDs by using the GUIDGen class (just run it and it spits out a list of new GUIDs)
- GUIDs are represented as 32 hex digits, e.g. `66bf14821704dc11b933e6037c01b18f`
- All instances of complex types have GUIDs as IDs

Creating a PDStore Data Model

Create an SQL script in a text file with the following:

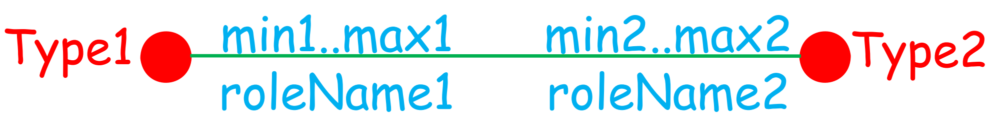
1. Connect to the database:

```
CONNECT 'pdstore.fdb' user 'sysdba' password 'masterkey';
```

2. Create a model first:

```
execute procedure create_model('model guid', 'model name');
```

Now go through the elements of the model:



3. For each type:

```
execute procedure create_type('type guid',
'model guid', 'type name', null);
```

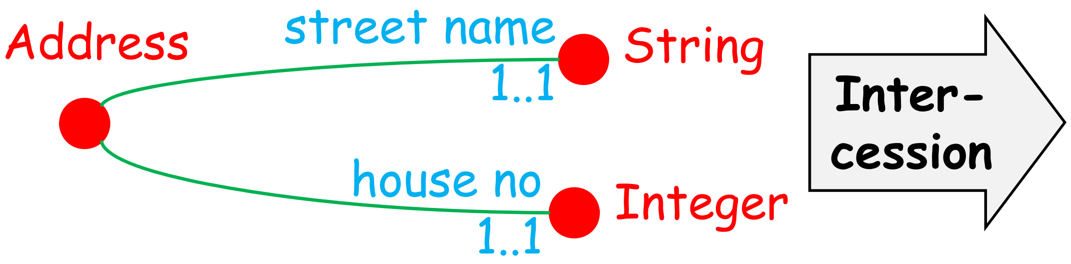
4. For each relation:

```
execute procedure create_relation(
'role1 guid','type1 guid',min1,max1,'roleName1',
'role2 guid','type2 guid',min2,max2,'roleName2');
```

5. Add a commit;

Creating a PDStore Data Model Cont.

6. After creating a model with types and relations, tell PDStore to create all the corresponding DB tables:
   `execute procedure intercession('model guid');`



DB Table **Address**
GUID <u>id</u>
VARCHAR street_name
INTEGER house_no

7. Add another `commit;`
8. Add the SQL script (e.g. `mymodel.sql`) to `reset-pdstore.bat`

```
del .\pdstore.fdb
fsql\fsql -i pdstore.sql 2> pdstore-errors.txt
fsql\fsql -i mymodel.sql 2> mymodel-errors.txt
```

9. Run `reset-pdstore.bat` and you get your model in a fresh database in file `pdstore.fdb`

Example Model: archivista.sql

```
CONNECT 'pdstore.fdb' user 'sysdba' password 'masterkey';
execute procedure
   create_model('4ef3e2dab0b9dd11b1bff11d9e19f111',
   'Archivista');
execute procedure
   create_type('09ca301f191edd11ad8da2fa74ba0698',
   '4ef3e2dab0b9dd11b1bff11d9e19f111', 'Building', null);
execute procedure create_relation(
   '57f3e2dab0b9dd11b1bff11d9e19f111',
   '09ca301f191edd11ad8da2fa74ba0698', 0, null, null,
   '58f3e2dab0b9dd11b1bff11d9e19f111',
   '4a8a986c4062db11afc0b95b08f50e2f', 0, 1, 'building
   name');
execute procedure create_relation(
   '59f3e2dab0b9dd11b1bff11d9e19f111',
   '09ca301f191edd11ad8da2fa74ba0698', 0, null, null,
   '5af3e2dab0b9dd11b1bff11d9e19f111',
   '4a8a986c4062db11afc0b95b08f50e2f', 0, 1, 'road name');
```

Generating a DAL with PDGen

- Run PDGen with the following arguments:
  1. Model name (`"Archivista"`)
  2. Package name (`archivista.dal`)
  3. Source root (`src`)

  e.g. `java PDGen "Archivista" archivista.dal src`
- PDGen will go through all the types `x` in the model and generate a Java class with name `PDx`
- The DAL classes will have getters and setters for all the named accessible roles,
  e.g. class PDBuilding will have

```
String getBuildingName()
void setBuildingName(String buildingName)
```

Using the Generated DAL

```
// load every DAL class you want to use, e.g.
Class.forName("archivista.dal.PDBuilding");

// create a new cache that is connected to the DB
PDCache cache = new
   PDCache("jdbc:firebirdsql:local:.\\pdstore.fdb",
     "sysdba", "masterkey");

// load an instance into memory
PDBuilding b = (PDBuilding) cache.load(PDBuilding.typeId,
   "My House");

b.setBuildingName("Grand Central");
System.out.println(b.getBuildingName());

PDBuilding b2 = (PDBuilding)   // create a new instance
   cache.newInstance(PDBuilding.typeId);


cache.commit();  // make changes permanent
```