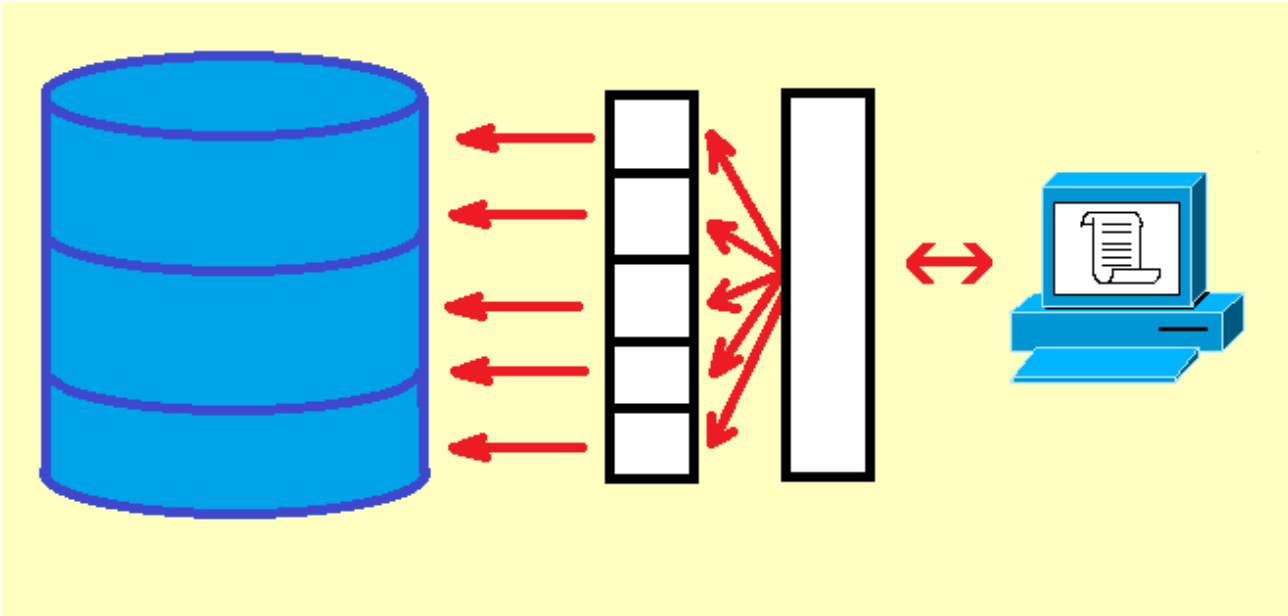
CFGS Desenvolupament d'Aplicacions Multiplataforma (DAM)
Mòdul 6 – Accés a dades

UF 4. Components d'accés a dades

Ajut exercici exemple UF4



Índex

Exercici 1

Classe AplicacioBDImpl

Implementa la interfície façana de la capa de persistència. Això significa que quan el programa necessiti realitzar alguna operació relacionada amb la persistència (com, per exemple, gravar objectes a la base de dades o realitzar-hi consultes o modificacions) cridarà als mètodes d'aquesta classe. Aquests mètodes seran els que realitzaran l'operació sobre la base de dades.

Podem dir que aquests mètodes són la «façana» que veu l'aplicació del mecanisme de persistència.

Normalment, les operacions d'accés a la base de dades són molt nombroses. Posar-ho tot directament en una classe produiria un mòdul massa gran i complex. Per aquest motiu, es situa un nou nivell d'abstracció: els objectes DAO (*Data Access Object*). Cadascun d'aquests s'encarrega d'un subconjunt de les operacions d'accés a la base de dades, normalment relacionades amb una de les classes del model.

Per tant, la classe *AplicacioBDImpl* es limitarà a crear els objectes DAO que necessiti per a poder proporcionar l'accés a les dades que es demana i a utilitzar els seus mètodes.

D'aquesta manera, la solució a aquesta part de l'exercici és la següent (destacat el que s'ha afegit a l'enunciat):

```
public class AplicacioBDImpl extends AplicacioBDJdbc
                                    implements AplicacioBD{

    CompanyDao companyDao;

    EmployeeDao employeeDao;

   public void obrir() throws UtilitatPersistenciaException {

        super.obrir();
        companyDao = new CompanyDao(con);

        employeeDao = new EmployeeDao(con);

    }
    public void emmagatzemar(Company company)
                        throws UtilitatPersistenciaException{
        companyDao.emmagatzemar(company);
    }
```

Es declaren els dos objectes DAO que s'utilitzaran.

Es crida al mètode de la superclasse i, després, s'inicialitzen els objectes DAO.

La resta de mètodes d'aquesta classe són similars: es limiten a cridar al mètode de l'objecte DAO que proporciona la funció que s'està implementant.

Classes DAO

Les classes DAO són les que realitzen realment les operacions sobre la base de dades, treballant directament sobre JDBC, però amb l'ajut de les classes del paquet `ioc.dam.m6.persistencia`.

En aquest apartat es tractaran alguns dels mètodes d'aquestes classes. S'han triat de manera que hi hagi un exemple significatiu de cada tipus d'operació. Cal destacar també que, normalment (encara que no sempre), utilitzaran *queries* SQL parametritzades.

Els exemples, si no es diu el contrari, són de la classe `EmployeeDao`.

Actualiltzació

El mètode `inserir` dona d'alta a la base de dades els objectes de la classe `EmployeeDao`. Crea i utilitza un objecte que implementa la interfície `JdbcPreparedDao`. Els objectes que implementen aquesta interfície representen una sentència SQL d'actualització que pot contenir paràmetres.

A continuació, es presenta aquest mètode dividit en fragments per a poder exposar millor el seu funcionament.

L'objecte `JdbcPreparedDao` resultant s'assigna a la variable `jdbcPreparedDao`, que s'utilitzarà més endavant.

```
protected void inserir(final Employee entitat)
                                        throws UtilitatPersistenciaException {

    JdbcPreparedDao jdbcPreparedDao = new JdbcPreparedDao(){
```

A continuació, es van implementant els diferents mètodes d'aquesta classe que cal sobreescriure.

El primer és `setParameter`, que dona els valors requerits per a cada paràmetre. La variable `field` indica el número del paràmetre (començant per 1 - fixeu-vos que abans de cridar als mètodes `pstm.set...` la variable s'autoincrementa). A cada paràmetre se li assigna la dada corresponent de l'objecte `entitat`.

Un cas especial és el de la dada `Company` (al codi és la part de l'`if`). És una referència a un altre objecte. Això, en una base de dades relacional es tradueix per una clau forana. Per tant, el que es fa és que, si aquesta referència és null o l'objecte referenciat té la seva clau a `null`, s'assigna NULL (java.sql.**Types**.NULL) al paràmetre. En cas contrari, s'assigna la dada clau de l'objecte referenciat.

```
        @Override
        public void setParameter(PreparedStatement pstm) throws SQLException {
            int field=0;
            Company co=entitat.getCompany();

            pstm.setString(++field, entitat.getNumSS());
            pstm.setString(++field, entitat.getNom());
            pstm.setDouble(++field,entitat.getSalary());

            if(co==null || co.getRef()==null){
               pstm.setNull(++field,java.sql.Types.NULL);

            }else{
               pstm.setString(++field, entitat.getCompany().getRef());
            }

        }
```

A continuació es sobrecarrega el mètode getStatement. Aquest mètode es limita a retornar un text amb la sentència SQL. Fixeu-vos que la sentència porta incorporats els paràmetres.

```
        @Override
        public String getStatement() {
            return "insert into Employee(numSS, nom,
                                         salary, company) values(?,?, ?, ?)";
      }
     };
```

Per últim, es crida el mètode estàtic `executar` de la classe `UtilitatJdbcPlus`, tot passant-li `jdbcPreparedDao`. Aquest mètode executa una sentència SQL d'actualització, com la continguda a l'objecte `jdbcPreparedDao` de la següent manera: crea un sentència el text de la qual obté cridant a getStatement, omple els seus paràmetres cridant a setParameter i, per últim, executa la *query*. Com la sentència és d'actualització, no retorna cap valor.

```
    UtilitatJdbcPlus.executar(con, jdbcPreparedDao);
 }
```

Els mètodes modificar i eliminar funcionen de manera anàloga:

```
protected void modificar(final Employee entitat) throws
                                                 UtilitatPersistenciaException {

   JdbcPreparedDao jdbcPreparedDao = new JdbcPreparedDao() {

       @Override
       public void setParameter(PreparedStatement pstm) throws SQLException {
           int field=0;
           pstm.setString(++field, entitat.getNom());
           pstm.setString(++field, entitat.getCompany().getRef());
           pstm.setFloat(++field,entitat.getSalary());
           pstm.setString(++field,entitat.getNumSS());
       }

       @Override
       public String getStatement() {
         return "update Employee set nom=?, company=?, salary=? where numSS=?";
       }
   };
   UtilitatJdbcPlus.executar(con, jdbcPreparedDao);
}
```

```
@Override
public void eliminar(final Employee entitat) throws
                                                 UtilitatPersistenciaException {
  JdbcPreparedDao jdbcDao = new JdbcPreparedDao() {

      @Override
      public void setParameter(PreparedStatement pstm) throws SQLException {
          int field=0;
          pstm.setString(++field, entitat.getNumSS());
      }
```

```
      @Override
      public String getStatement() {
          return "delete from Employee where numSS = ?";
      }
  };
  UtilitatJdbcPlus.executar(con, jdbcDao);
}
```

Consultes

En primer lloc trobareu l'exposició del mètode `refrescar` de la classe `CompanyDao`. S'ha triat un mètode de la classe `Company` perquè il·lustra prou bé com cal realitzar les consultes.

A l'inici es crea un objecte que implementi la interfície `JdbcPreparedQueryDao` . Els objectes d'aquesta interfície representen consultes SQL amb paràmetres i que retornen un resultat.

```
public Company refrescar(final Company entitat)
                                          throws UtilitatPersistenciaException {
  Company ret= null;
  JdbcPreparedQueryDao jdbcDao = new JdbcPreparedQueryDao() {
```

A continuació, es van implementant els diferents mètodes d'aquesta classe que cal sobreescriure.

El primer és `writeObject`, que escriu cadascun dels objectes que formaran part del resultat, cadascun a partir d'una fila del *ResultSet* resultat de fer la consulta SQL.

```
   @Override
   public Object writeObject(ResultSet rs) throws SQLException {
      int field=0;
      Company ret = new Company();
      ret.setRef(entitat.getRef());
      ret.setCompanyName(rs.getString(++field));
      ret.setCity(rs.getString(++field));

      ret.getEmployeeList().clear();
      try {
            EmployeeDao ed = new EmployeeDao(con);
            ret.getEmployeeList().addAll(ed.employeesOfACompany(entitat));
          } catch (UtilitatPersistenciaException ex) {
      Logger.getLogger(CompanyDao.class.getName()).log(Level.SEVERE, null, ex);
      }
      return ret;
   }
```

El següent és `getStatement`, que es limita a retornar una cadena de caràcters amb la consulta SQL. Cal fixar-se que la consulta conté un paràmetre:

```
    @Override
    public String getStatement() {
         return "select nom, ciutat from Company where referencia=?";
    }
```

L'últim mètode és `setParameter`, que assigna valor a l'únic paràmetre de la consulta SQL, la clau de la companyia, l'atribut *ref*:

```
    @Override
    public void setParameter(PreparedStatement pstm) throws SQLException {
         int field=0;
         pstm.setString(++field, entitat.getRef());
    }
```

Per últim, es crida al mètode `obtenirObjecte` i es retorna el resultat.

El mètode `obtenirObjecte` crea una consulta SQL tot cridant al mètode `jdbcDao.getStatement`, omple els paràmetres d'aquesta consulta tot cridant al mètode `setParameter`, executa la consulta i retorna el resultat.

```
  };
  ret = (Company) UtilitatJdbcPlus.obtenirObjecte(con, jdbcDao);

  return ret;

}
```

Els propers mètodes són tots de la classe `EmployeeDao`. Podem considerar-los variants del mètode que acabem de veure.

Mètode esPersistent (retorna cert si troba a la base de dades un empleat amb el mateix número de seguretat social que el paràmetre i fals en cas contrari):

```
protected boolean esPersistent(final Employee entitat) throws
                                                  UtilitatPersistenciaException {
   boolean ret= false;
   if(entitat.getNumSS()==null){
       return ret;
   }

   JdbcPreparedQueryDao jdbcDao = new JdbcPreparedQueryDao() {
     @Override
     public Object writeObject(ResultSet rs) throws SQLException {
              return rs.getInt(1);
     }

     @Override
     public String getStatement() {
             return "select count(numSS) from Employee where numSS=?";
     }


   @Override
   public void setParameter(PreparedStatement pstm) throws SQLException {
       int field=0;
       pstm.setString(++field, entitat.getNumSS());
   }
  };
  ret = ((Integer)UtilitatJdbcPlus.obtenirObjecte(con, jdbcDao))>=1;
  return ret;
}
```

Les diferències principals respecte l'anterior són:

- Al mètode `getStatement` no es consulta per les dades d'un objecte, sinó que es demana que la consulta recompti objectes (`select count`).
- Al mètode `writeObject` es retorna un objecte de tipus enter, que és el comptador que s'obté a la consulta.
- No es retorna directament el resultat de la consulta, sinó que es retorna si aquest és major o igual que 1.

Mètode `refrescar` (actualitza les dades d'un objecte amb les dades emmagatzemades a la base de dades):

```
    public Employee refrescar(final Employee entitat) throws
                                                UtilitatPersistenciaException {
        Employee ret= null;
        JdbcPreparedQueryDao jdbcDao = new JdbcPreparedQueryDao() {

            @Override
            public Object writeObject(ResultSet rs) throws SQLException {
                int field=0;
                String ref;

                Employee ret = new Employee();
                ret.setNumSS(rs.getString(++field));
                ret.setNom(rs.getString(++field));
                ret.setSalary(rs.getFloat(++field));
                ref=rs.getString(++field);

                if(ref!=null){
                    Company aux= new Company();
                    aux.setRef(ref);
                    aux.setCompanyName(rs.getString(++field));
                    aux.setCity(rs.getString(++field));
                    aux.getEmployeeList().clear();
                    try {
                        aux.getEmployeeList().addAll(employeesOfACompany(aux));
                    } catch (UtilitatPersistenciaException ex) {
                    Logger.getLogger(
                      EmployeeDao.class.getName()).log(Level.SEVERE, null, ex);
                    }
                    ret.setCompany(aux);
                }
                return ret;
            }

    @Override
    public String getStatement() {
     return "SELECT e.numSS, e.nom, e.salary, e.company, "+"c.nom, c.ciutat "
            + "FROM Company c RIGHT JOIN Employee e ON e.company=c.referencia"
                                                        + " WHERE e.numSS=?";
    }
```

```
     @Override
     public void setParameter(PreparedStatement pstm) throws SQLException {
        int field=0;
        pstm.setString(++field, entitat.getNumSS());

     @Override
     public String getStatement() {
      return "SELECT e.numSS, e.nom, e.salary, e.company, "+"c.nom, c.ciutat "
             + "FROM Company c RIGHT JOIN Employee e ON e.company=c.referencia"
                                                          + " WHERE e.numSS=?";
     }

   ret = (Employee) UtilitatJdbcPlus.obtenirObjecte(con, jdbcDao);

   return ret;
}
```

Les diferències principals són als mètodes `getStatement` i `writeObject` i de l'objecte jdbcDao. El primer retorna una consulta consistent en un *join* de l'empleat que refresquem amb la companyia a la qual pertany. El segon, a l'hora de crear l'empleat, ha de crear l'objecte de la classe *Company* al qual pertany aquest. Com les companyies tenen una llista dels seus empleats, a part de copiar els valors del resultat de la consutla, crida al mètode `employeesOfACompany` per omplir aquesta llista. El mètode `setParameter` és similar al del mètode `refrescar` de la mateixa classe.

Mètode `employeesOfACompany` (consulta a la base de dades els empleats d'una companyia i retorna una llista amb el resultat):

```
public List<Employee> employeesOfACompany(final Company c) throws
                                                    UtilitatPersistenciaException {
    JdbcPreparedQueryDao jdbcDao = new JdbcPreparedQueryDao() {
      @Override
      public Object writeObject(ResultSet rs) throws SQLException {
         int field=0;
         Employee ret = new Employee();
         ret.setNumSS(rs.getString(++field));
         ret.setNom(rs.getString(++field));
         ret.setSalary(rs.getFloat(++field));
         ret.setCompany(c);
         return ret;
      }

      @Override
      public String getStatement() {
         return "SELECT numSS, nom, salary FROM Employee WHERE company = ?";
      }

      @Override
      public void setParameter(PreparedStatement pstm) throws SQLException {
           pstm.setString(1, c.getRef());
      }
    };
    List<Employee> ret = UtilitatJdbcPlus.obtenirLlista(con, jdbcDao);
    return ret;
}
```

Aquest mètode utilitza també un objecte de la classe `JdbcPreparedQueryDao` i el mètode setParameter és igual al del mètode `refrescar` de la classe `CompanyDao`.

Les diferències remarcables respecte el vist fins ara són:

- El mètode `getStatement` retorna una consulta que fa un join entre companyies i empleats, però, en aquest cas, aquesta consulta retorna més d'una fila.
- En conseqüència La instrucció de retorn crida a `UtilitatJdbcPlus.obtenirLlista`, que retorna una llista com a resultat de la consulta (en lloc de cridar a `UtilitatJdbcPlus.obtenirObjecte`, que només retorna un objecte -obtingut d'una fila- com a resultat de la consulta).
- El mètode `writeObject` retorna un sol objecte, en aquest cas de la classe `Employee`; aquest mètode serà cridat tantes vegades com files tingui el resultat de la consulta SQL.

Mètode `obtenirTot` (retorna tots els empleats de la base de dades):

Podem comparar aquest mètode amb els anteriors:

- El mètode `getStatement` retorna un *join* entre empleats i companyies; aquesta consulta SQL retorna més d'una fila.
- El mètode `writeObject` és idèntic al utilitzat a `refrescar` de la classe `Employee`. Se'l cridarà un cop per cada fila resultat de la consulta SQL.
- Com al mètode `employeesOfACompany`, com el resultat és una llista, es retorna el resultat de cridar a `UtilitatJdbcPlus.obtenirLlista`.

```
public List<Employee> obtenirTot() throws UtilitatPersistenciaException {
  JdbcQueryDao jdbcDao = new JdbcQueryDao() {

    @Override
    public Object writeObject(ResultSet rs) throws SQLException {
       int field=0;
       String ref;

       Employee ret = new Employee();
       ret.setNumSS(rs.getString(++field));
       ret.setNom(rs.getString(++field));
       ret.setSalary(rs.getFloat(++field));
       ref=rs.getString(++field);

       if(ref!=null){
         Company aux= new Company();
         aux.setRef(ref);
         aux.setCompanyName(rs.getString(++field));
         aux.setCity(rs.getString(++field));
         aux.getEmployeeList().clear();
         try {
              aux.getEmployeeList().addAll(employeesOfACompany(aux));
          } catch (UtilitatPersistenciaException ex) {
          Logger.getLogger(EmployeeDao.class.getName()).log(Level.SEVERE,
                                                              null, ex);
       }
       ret.setCompany(aux);
       }
       return ret;
    }

    @Override
    public String getStatement() {
       return "SELECT e.numSS, e.nom, e.salary, e.company, c.nom, c.ciutat "
           + "FROM Company c RIGHT JOIN Employee e ON e.company=c.referencia";
    }
  };
```

```
  List<Employee> ret = UtilitatJdbcPlus.obtenirLlista(con, jdbcDao);
  return ret;
}
```

Exercici 2

En aquest exercici cal utilitzar el patró *Publicació – subscripció*. En aquest patró hi ha els següents elements:

- Un o més objectes **observador**, que "observen" determinades accions d'un altre objecte, que anomenarem objecte **observat**, i que actuen en funció d'aquestes accions observades.
- La **subscripció**. És el mecanisme que permet connectar els objectes observats amb els objectes observadors. El funcionament és el següent:
  - Si un objecte admet observadors de determinades accions, **publica** (és a dir, declara) una funció pública perquè aquests observadors s'hi subscriguin.
  - Quan es produeix aquesta acció, l'objecte observat avisa a tots els observadors subscrits perquè aquests actuïn en conseqüència.

En aquest exercici, la classe observada és `Temperatura`. Aquesta classe admet dos tipus d'observadors que en Java s'anomenen respectivament `PropertyChangeListener` i `VetoableChangeListener` (quelcom així com escoltadors -és a dir, que "escolten"- canvis a les propietats i que "escolten" canvis a l'objecte susceptible de ser vetat -en principi, per incorrectes-, respectivament). La finalitat de cada tipus d'escoltador és la següent:

- Els `PropertyChangeListener` són avisats cada cop que hi ha un canvi en una determinada propietat i, en ser-ho, realitzen una determinada acció (al nostre cas, omplir un quadre de text).
- Els `VetoableChangeListener` són també avisats cada cop que hi ha un canvi en una determinada propietat, però, en aquest cas, el que fan en ser avisats és veure si el canvi és acceptable; si ho és, no fan res més; si no ho és, llencen una excepció (del tipus `PropertyVetoException` per evitar que s'acabi de realitzar l'acció -és a dir, per vetar-la). Al nostre cas, es llença aquesta excepció quan la temperatura que es vol assignar és massa baixa.

Al segon exercici, tot això s'implementa de la següent manera:

Classe observada

És la classe `Temperatura.` Aquesta classe admet observadors dels tipus `PropertyChangeListener` i `VetoableChangeListener;` per tant, a més de l'atribut `ultimaLectura` i el *setter* i el *getter* associats ha de tenir un mecanisme que permeti subscriure-s'hi als observadors i mecanismes per avisar-los.

Implementació del mecanisme de subscripció

Consta de:

- Un objecte de la classe `PropertyChangeSupport` i un altre de `PropertyChangeSupport`:

```
private final VetoableChangeSupport vcs=new VetoableChangeSupport(this);
                                                    // paràmetre no rellevant

private final PropertyChangeSupport pcs=new PropertyChangeSupport(this);
                                                    // paràmetre no rellevant
```

  Cadascun d'aquests objectes permet mantenir la seva llista d'observadors i avisar de cop a tots els observadors de la llista quan es produeix el canvi que estan observant.

- Mètodes que permetin afegir i suprimir observadors a cadascuna de les llistes que mantenen els objectes anteriors:

```
public void addVetoableChangeListener(VetoableChangeListener listener) {
        this.vcs.addVetoableChangeListener(listener);
}
```

```
public void removeVetoableChangeListener(VetoableChangeListener listener) {
        this.vcs.removeVetoableChangeListener(listener);
}

public void addPropertyChangeListener(PropertyChangeListener listener) {
        this.pcs.addPropertyChangeListener(listener);
}

public void removePropertyChangeListener(PropertyChangeListener listener) {
        this.pcs.removePropertyChangeListener(listener);
}
```

- Mètodes que avisin als subscriptors quan hi ha un canvi a la propietat que observen. Aquests avisos es fan mitjantçant crides als mètodes `fireVetoableChange` -avisa als `VetoableChangeListener`- i `firePropertyChange` -avisa als `PropertyChangeListener`. En aquest cas només hi ha una propietat, `ultimaLectura`, i els observadors de tots dos tipus l'observen. Per tant, caldrà cridar a tots dos mètodes dins del *setter* corresponent a aquesta propietat:

```
public void setUltimaLectura(int ultimaLectura) throws PropertyVetoException
{
   int lecturaAngiga=this.ultimaLectura;

   this.vcs.fireVetoableChange("ultima lectura", this.ultimaLectura,
                                                          ultimaLectura);
   this.ultimaLectura = ultimaLectura;

   this.pcs.firePropertyChange("ultima lectura", lecturaAntiga,
                                                          ultimaLectura);

}
```

A totes dues crides, el primer paràmetre és el nom de la propietat, el segon és el valor que té aquesta propietat abans de realitzar-se el canvi i el tercer és el valor que té o tindrà un cop realitzat.

El mètode `fireVetoableChange` és cridat abans de canviar el valor de la propietat perquè així podrà, si s'escau, vetar el canvi llençant l'excepció `PropertyVetoException`.

El mètode `firePropertyChange` és cridat quan ja s'ha realitzat el canvi, per estar segurs que el canvi és real. En aquest cas és evident que el canvi es realitzarà i, si s'hagués posat abans de canviar la variable, el resultat hauria estat el mateix; no obstant, a vegades els canvis no s'acaben de realitzar (per exemple, si assignem el resultat d'una divisió, aquesta no es realitza quan el divisor val zero, perquè abans es llença una excepció aritmètica).

Classe observadora

En aquest cas només hi ha una classe observadora: `Observador`, que fa a la vegada les funcions de `PropertyChangeListener` i `VetoableChangeListener`. Per tant, implementa aquestes dues interfícies. Podria haver-hi més classes observadores, cadascuna de les quals implementés una o totes dues interfícies.

Els mètodes on realitza les accions pròpies de l'observador (o subscriptor) són:

- Com a PropertyChangeListener:

```
public void propertyChange(PropertyChangeEvent evt) {
   Date data= new Date();
   text.setText(DateFormat.getTimeInstance().format(data)+"  "
                                    +DateFormat.getDateInstance().format(data));
}
```

El que fa és el tractament associat al canvi; en aquest cas, escriure la data i l'hora actuals en un quadre de text; el quadre de text es coneix perquè s'ha passat com a paràmetre al constructor en crear l'objecte.

- Com a VetoableChangeListener:

```
public void vetoableChange(PropertyChangeEvent evt)
                                                    throws PropertyVetoException {
   int nouValor=(Integer)evt.getNewValue();

   if(nouValor<-273){
       throw new PropertyVetoException("Massa freda",evt);
   }
}
```

Si el valor que es vol assignar és inferior a -273 (-273,15 és la temperatura mínima possible), llença una PropertyVetoException per evitar que s'assigni un valor incorrecte.

Creació dels objectes observador i observat

Es realitza a la classe Pantalla. En ella es defineixen els objectes observador i observat i es crida als mètodes addPropertyChangeListener i addVetoableChangeListener per subscriure l'objecte observador com a *PropertyChangeListener* i com a *VetoableChangeListener.*

```
//creació de l'objecte observat

private Temperatura temperatura = new Temperatura();
private final Observador observador;  // declaració de l'observador

    public Pantalla() {

        initComponents();
// creació de l'observador; el paràmetre és el quadre de text on escriurà
        observador=new Observador(jTxtCanvi);

// subscripció de l'observador a l'objecte observat

    // com a PropertyChangeListener
        temperatura.addPropertyChangeListener(observador);

    // com a VetoableChangeListener
        temperatura.addVetoableChangeListener(observador);

    }
...
```